ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE CIENCIAS

REALIZACIÓN DE CONCEPTOS ABSTRACTOS DE LA TEORÍA DE GRUPOS POR MEDIO DE UN MODELO COMPUTACIONAL

DESCOMPOSICIÓN CÍCLICA DE GRUPOS DE ENTEROS MULTIPLICATIVOS Y VISUALIZACIÓN DE PROPIEDADES PARA GRUPOS NO ABELIANOS D4 Y Q8.

TRABAJO DE INTEGRACIÓN CURRICULAR PRESENTADO COMO REQUISITO PARA LA OBTENCIÓN DEL TÍTULO DE MATEMÁTICA

CAMILA BELÉN GUEVARA FRANCO

camila-guevara@hotmail.es

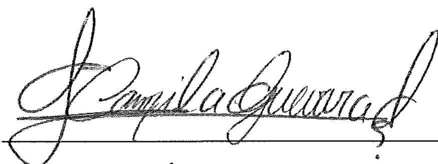
DIRECTOR: EURO DE JESÚS LUCENA DELGADO

euro.lucena@epn.edu.ec

DMQ, SEPTIEMBRE 2022

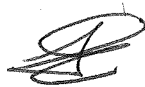
CERTIFICACIONES

Yo, CAMILA BELÉN GUEVARA FRANCO, declaro que el trabajo de integración curricular aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.



CAMILA BELÉN GUEVARA FRANCO

Certifico que el presente trabajo de integración curricular fue desarrollado por CAMILA BELÉN GUEVARA FRANCO, bajo mi supervisión.



EURO DE JESÚS LUCENA DELGADO

DIRECTOR

DECLARACIÓN DE AUTORÍA

A través de la presente declaración, afirmamos que el trabajo de integración curricular aquí descrito, así como el producto resultante del mismo, es público y estará a disposición de la comunidad a través del repositorio institucional de la Escuela Politécnica Nacional; sin embargo, la titularidad de los derechos patrimoniales nos corresponde a los autores que hemos contribuido en el desarrollo del presente trabajo; observando para el efecto las disposiciones establecidas por el órgano competente en propiedad intelectual, la normativa interna y demás normas.

CAMILA BELÉN GUEVARA FRANCO

EURO DE JESÚS LUCENA DELGADO

RESUMEN

Varios de los resultados y aplicaciones más importantes de la teoría de grupos subyacen en los grupos finitos, por esta razón existe singular interés en estas estructuras y aún permanece vigente el estudio y exploración de sus propiedades. Los grupos finitos se subdividen en grupos abelianos y no abelianos. En el presente trabajo se tomará esta distinción, y se estudiará por separado cada una, a través de un modelo computacional programado en lenguaje C++.
Para los grupos abelianos finitos se trabajará con uno de los lemas más importantes de la teoría, atribuído al matemático Kronecker en 1870, conocido como Teorema Fundamental de Grupos Abelianos Finitos, el cual afirma que cada uno de estos grupos puede ser descompuesto mediante grupos cíclicos. En base al teorema fundamental, el modelo contará con una función, que obtenga la descomposición cíclica de los grupos de enteros multiplicativos módulo n.
Para grupos no abelianos, debido a que no se cuenta con un resultado general que permita caracterizarlos, se han escogido dos grupos particulares con el fin de observar su comportamiento y extraer conclusiones que expongan los contrastes con los grupos abelianos; por ejemplo, el concepto subgrupo normal se puede apreciar de mejor forma en grupos no abelianos ya que es trivial en el caso de los abelianos.
Con ese enfoque, el modelo también incluye implementaciones correspondientes a los grupos D_4 y Q_8, y conceptos diversos como isomorfismo o subgrupo normal. Todo lo anterior se consolida en un programa que interactúa con el usuario y obtiene resultados según el modelo realizado, y las opciones disponibles escogidas.

Palabras clave: grupos, abelianos, finitos, teorema fundamental, modelo computacional, descomposición cíclica, isomorfismo, subgrupo, normal.

ABSTRACT

Several of the most important results and applications of group theory underlie finite groups, for this reason there is singular interest in these structures and the study and exploration of their properties is still in force. Finite groups are subdivided into abelian and non-abelian groups. In the present work this distinction will be taken, and each one will be studied separately, through a computational model programmed in C++ language.
For finite abelian groups we will work with one of the most important lemmas of the theory, attributed to the mathematician Kronecker in 1870, known as the Fundamental Theorem of Finite Abelian Groups, which states that each of these groups can be decomposed by cyclic groups. Based on the fundamental theorem, the model will have a function that obtains the cyclic decomposition of the groups of multiplicative integers modulo n.
For non-abelian groups, due to the lack of a general result that allows characterizing them, two particular groups have been chosen in order to observe their behavior and draw conclusions that expose the contrasts with the abelian groups; for example, the normal subgroup concept can be better appreciated in non-abelian groups since it is trivial in the abelian case.
With this approach, the model also includes implementations corresponding to the groups D_4 and Q_8 , and various concepts such as isomorphism or normal subgroup. All of the above is consolidated in a program that interacts with the user and obtains results according to the model made, and the available options chosen.

Keywords: groups, abelian, finite, computational model, cyclic decomposition, isomorphism, subgroup, normal

Índice general

1. Descripción del componente desarrollado **1**

1.1. Objetivo general 1

1.2. Objetivos específicos 1

1.3. Alcance 2

1.4. Marco teórico 3

1.4.1. Preliminares de la teoría de números 3

1.4.2. Definiciones y propiedades de la teoría de Grupos . . 4

1.4.3. Grupos abelianos finitos 11

1.4.4. Programación genérica 16

2. Metodología **18**

2.0.1. Implementaciones grupos abelianos 18

2.0.2. Implementaciones para grupos no abelianos 21

2.0.3. Descomposición de grupos U_n 25

2.0.4. Isomorfismos entre grupos no abelianos 30

3. Resultados, conclusiones y recomendaciones **36**

3.1. Resultados 36

3.1.1. Grupos U_n 36

3.2. Conclusiones y recomendaciones 43

3.2.1. Conclusiones 43
3.2.2. Recomendaciones 43

A. Anexos **45**

Bibliografía **63**

Capítulo 1

Descripción del componente desarrollado

Creación de un modelo computacional en C++ que permita descomponer los grupos de enteros multiplicativos módulo n como suma directa de grupos cíclicos. Además, visualizar características de grupos no abelianos como $D4$ y $Q8$, tales como subgrupo e isomorfismos.

1.1. Objetivo general

Aplicar en conjunto el teorema fundamental de grupos abelianos finitos, varios conceptos de teoría de grupos y recursos computacionales, para crear un modelo computacional que permita descomponer los grupos U_n como suma directa de grupos cíclicos; obtener características de los grupos no abeliano D_4 y Q_8, y visualizarlas mediante un programa realizado en lenguaje C++.

1.2. Objetivos específicos

1. Utilizar la programación orientada a objetos para implementar las estructuras de grupos de enteros multiplicativos U_n, dos representaciones del grupo D_4 y una representación del grupo Q_8, cada una con varios conceptos, métodos o atributos.

2. Crear una función en el programa en base a la demostración del

teorema fundamental de grupos abelianos finitos que interactúe con con grupos U_n y obtenga su descomposición cíclica.

3. Crear una función basada en el concepto de isomorfismo aplicado a las estructuras D_4 y Q_8.
4. Utilizar el modelo consolidado bajo el cumplimiento de los objetivos anteriores, para visualizar diversas características de los grupos en interés, así como los conceptos subgrupo o subgrupo normal.

1.3. Alcance

Se comienza por crear las funciones necesarias y particularizadas para los objetivos del presente trabajo. Por ejemplo, una función que calcule el máximo común divisor de dos números. Posteriormente se utilizarán las clases en C++ para implementar los grupos U_n, D_4 (dos implementaciones distintas) y Q_8. Cada clase contará con sus métodos y atributos. Se creará una función especial permita descomponer las estructuras U_n, anteriormente implementadas, como suma de grupos cíclicos. Esta función implícitamente permite conocer si dos grupos U_n son isomorfos mediante su descomposición cíclica. Por otra parte, como no se pueden aplicar los mismos teoremas para grupos no abelianos se crea una función que trabaja particularmente con $D4$ y $Q8$ y buscará responder varias preguntas de estos grupos. Cada función o estructura implementada tiene una base en algún concepto o teorema de la teoría de grupos.
Las diversas propiedades que se explorarán en común para las tres estructuras U_n ,D_4 y Q_8 son verificación de subgrupos, órdenes e inversos de elementos, tablas de multiplicar. Para D_4 se implementó adicionalmente el concepto de subgrupo normal, el cual ha permito ser un contraste con los grupos abelianos finitos. Finalizada la implementación del modelo, se realizan y anallizan varias pruebas, y se extraerán conclusiones de cada ensayo.

1.4. Marco teórico

Se abordará brevemente los conceptos y propiedades acerca de la teoría de números, teoría de grupos y programación, necesarios para una adecuada comprensión del componente y su desarrollo.
Se incluye una demostración solo para los resultados más importantes en los que se basa el presente trabajo. Para los demás resultados, se sugiere consultar su demostración en la fuente bibliográfica [1].

1.4.1. Preliminares de la teoría de números

Teorema 1.4.1. Sean $a, n \in \mathbb{Z}$ y $n \neq 0$. Existen únicos $m, r \in \mathbb{Z}$, tales que $0 \leq r \leq |n|$ y $a = mn + r$.

Definición 1.4.1. Bajo el contexto del teorema 1.4.1, se denomina división euclídea al hecho de obtener los enteros m y r. Adicionalmente, a se conoce como dividendo y n divisor. Los enteros m y r se conocen como cociente y residuo de la división de a entre n respectivamente.

Notación: El residuo de la división euclídea será ampliamente utilizado a lo largo del trabajo. Con dicha motivación, se denotará como $a\,\%n$ al residuo de la división de a entre n.

Definición 1.4.2. Sean $a, b \in \mathbb{Z}$, $a \neq 0$. Se dice que a divide a b, o a es un divisor de b, cuando existe un entero n tal que $b = n \cdot a$. Equivalentemente, a divide a b si $b\,\%\,a = 0$.

Definición 1.4.3. Un número entero positivo p se dice número primo si sus únicos divisores son 1 y p. En otras palabras, si $a|p$ entonces $a = 1$ o $a = p$.

Definición 1.4.4. El máximo común divisor de dos números enteros a y b, denotado como (a, b) se define como el entero que cumple las siguientes propiedades

1. $(a, b)|a$ y $(a, b)|b$.

2. Si un entero c es tal que $c|a$ y $c|b$ entonces $c|(a, b)$.

Proposición 1.4.1. Sean dos enteros a_1 y a_2. Existen x_1 y x_2 tales que $x_1a_1 + x_2a_2 = (a_1, a_2)$.

Definición 1.4.5. Dos números enteros positivos a y b se dicen primos relativos o coprimos si $(a, b) = \pm 1$

Teorema 1.4.2 (Teorema fundamental de la aritmética)**.** Todo número entero positivo tiene una única factorización de números primos.

Teorema 1.4.3 (Fermat)**.** Si p es un número primo, entonces para todo primo relativo a de p, se verifica $a^{p-1}\%p = 1$.

1.4.2. Definiciones y propiedades de la teoría de Grupos

Definición 1.4.6. Un grupo es un conjunto G no vacío en el cual se ha definido una operación binaria $(\cdot)$ conocida como producto, tal que verifica las siguientes propiedades

- Clausura: Para todo par de elementos $a, b \in G$ su producto está en G es decir, $a \cdot b \in G$.
- Existencia del elemento identidad: Existe $e \in G$ tal que para todo $a \in G$, $a \cdot e = e \cdot a = a$. El elemento e es conocido como neutro o identidad del grupo G
- Existencia de inversos: Para todo elemento $a \in G$ existe un $a^{-1} \in G$ (conocido como el inverso de a) tal que $a \cdot a^{-1} \cdot a = a \cdot a^{-1} = e$.
- Asociatividad: Para cualesquier $a, b, c \in G$ se cumple que $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.

Notaciones: Se usará ab en reemplazo de $a \cdot b$ es decir $ab = a \cdot b$. Se usará $(G, \cdot)$ cuando se quiera especificar un grupo y su producto.

Ejemplos

- **Grupo** $D4$ **de simetrías del cuadrado**: Considerar un cuadrado con sus esquinas numeradas. Las acciones que preservan su simetría son las siguientes:

- No realizar nada: La acción de dejar al cuadrado como en su estado inicial, evidentemente preserva su simetría. A esta acción se la llamará id.
- Rotaciones de $90°$: LLamaremos r_{90}, r_{180} y r_{270} a las acciones de rotar $90°$ en sentido antihorario alrededor del centro, una, dos y tres veces respectivamente.
- Giros por ejes de simetría: Girar a través del eje horizontal, vertical, la primera diagonal y la segunda diagonal se representarán como h, v, $d1$ y $d2$ respectivamente.

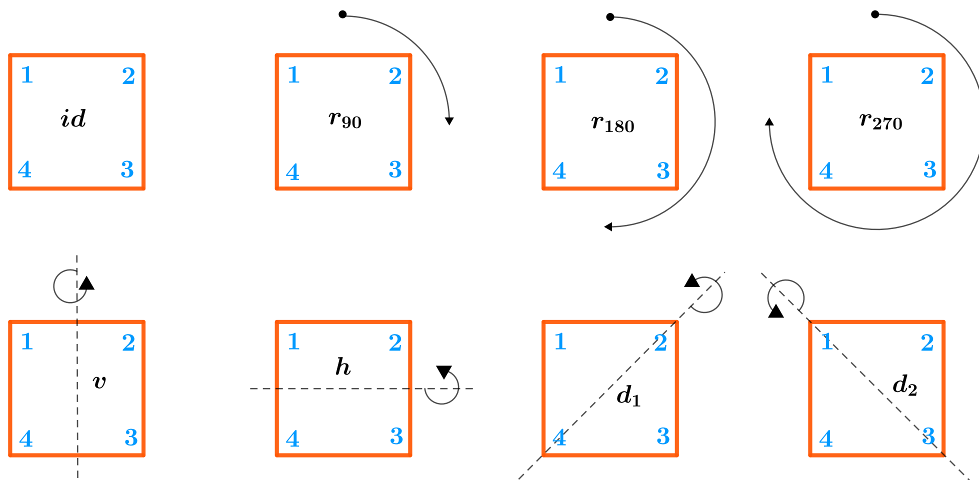


Figura 1.1: Acciones que preservan simetría en un cuadrado

El siguiente conjunto

$$D_4 = \{id,\ r_{90},\ r_{180},\ r_{270},\ v,\ h,\ d_1,\ d_2\}$$

es un grupo, junto con la operación binaria que consiste en aplicar las acciones de simetría al cuadrado. Su tabla de multiplicar se puede observar en 3.3.

- D_4 **como permutaciones**: El grupo D_4 tiene también otra perspectiva que se describe a continuación.
  Las permutaciones son maneras de reordenar un conjunto finito A o equivalentemente, son biyecciones de A en A.
  Para cada $n \in \mathbb{Z}^+$ se define $S(n)$ como el conjunto de todas las permutaciones de $\{1, 2, \ldots, n\}$ el cual forma un grupo, en donde si $P_1, P_2 \in S(n)$, el producto $P_1 * P_2 = P_2 \circ P_1$.

Se utilizará la siguiente notación vectorial para detallar una permutación P

$$P = (P(1), P(2), \ldots, P(n)). \tag{1.1}$$

En este contexto, la enumeración de las esquinas del cuadrado inducen a que cada acción de simetría corresponda a una permutación ya que al aplicar una, se cambia la disposición de la numeración, y por esta razón se puede considerar a D_4 como un subconjunto de $S(4)$. Los elementos de D_4 como permutaciones y bajo la notación (1.1) se exponen a continuación:

$id = (1,2,3,4)$	$r_{90} = (2,3,4,1)$	$r_{180} = (3,4,1,2)$	$r_{270} = (4,1,2,3)$
$v = (2,1,4,3)$	$h = (4,3,2,1)$	$d_1 = (3,2,1,4)$	$d_2 = (1,4,3,2)$

Cuadro 1.1: Grupo D_4 como permutaciones

Observación
Si $p_1, p_2 \in D_4$ tal que $p_1 \neq p_2$ entonces $p_1(1) \neq p_2(1)$ o $p_1(2) \neq p_2(2)$. Más adelante esta observación permitirá identificar más fácilmente un elemento de D_4.

- **Cuaterniones**: Los cuaterniones son una extensión de los números reales con tres unidades imaginarias i, j y k, es decir son objetos de la forma

$$a = a_1 + a_2 i + a_3 j + a_4 k, \qquad a_1, a_2, a_3, a_e \in \mathbb{R}$$

También cuentan con un producto, si $a = a_1 + a_2 i + a_3 j + a_4 k$ y $b = b_1 + b_2 i + b_3 j + b_4 k$ son cuaterniones, entonces

$$\begin{aligned} ab = & (a_1b_1 - a_2b_2 - a_3b_3 - a_4b_4) + (a_1b_2 + a_2b_1 + a_3b_4 - a_4b_3)i \\ & + (a_1b_3 - a_2b_4 + a_3b_1 - a_4b_2)j + (a_1b_4 + a_2b_3 - a_3b_2 + a_4b_3)k \end{aligned} \tag{1.2}$$

En base a (1.2) se cumple la ecuación $i^2 = j^2 = k^2 = ijk = -1$ lo que implica que el conjunto $Q_8 = \{1, -1, i, -i, j, -j, k, -k\}$ es un grupo. Su tabla de multiplicar se puede observar en 3.4.

Definición 1.4.7. Un grupo G se dice abeliano si para todo par de elementos $a, b \in G$ se cumple $ab = ba$.

Ejemplo: Dado un entero n fijo, se considera el conjunto de todos sus primos relativos menores, el cual se denominará U_n, y sobre el cual se

define la siguiente operación binaria: Si $a, b \in U_n$ entonces $a \cdot b = (ab)\,\%n$. La dupla $(U_n, \cdot)$ conforma un grupo abeliano gracias a la conmutatividad de la multiplicación de enteros, y es conocido como grupo multiplicativo de enteros módulo n.
Para observar un caso particular, se considera $n = 8$, de esta manera $U_8 = \{1, 3, 5, 7\}$.
Si $a = 3$ y $b = 5$ entonces $a \cdot b = 5$, pues $ab\,\%n = 15\,\%8 = 7$.

Definición 1.4.8. Sea G un grupo. El número de elementos de G se denomina orden del grupo, y se denota como $|G|$.
Si $|G|$ es finito, se dice que G es un grupo finito caso contrario se dice que es un grupo infinito.

Definición 1.4.9. Un subconjunto H de un grupo $(G, \cdot)$ se dice que es un subgrupo si $(H, \cdot)$ es un grupo.

Teorema 1.4.4. Sean un grupo finito G y $H \subseteq G$. Se tiene que H es subgrupo si y solo sí H es cerrado con respecto al producto de G.

Definición 1.4.10. Sean $n \in \mathbb{Z}$ y un elemento a de un grupo G, se define

$$a^n = \begin{cases} e & \text{si } n = 0 \\ \underbrace{a \cdot a \dots a}_{n \text{ veces}} & \text{si } n > 0 \\ (a^{-1})^{|n|} & \text{si } n < 0. \end{cases}$$

Definición 1.4.11. Sea A subconjunto de un grupo G. Al subgrupo mínimo de G que contiene a A se denomina subgrupo generado por A y se denota por $\langle A \rangle$.

Definición 1.4.12. Un subconjunto A de un grupo G se dice que es un conjunto generador, o que G es generado por A, si $\langle A \rangle = G$.

Definición 1.4.13. Un grupo G se dice cíclico si $G = \langle a \rangle$ para algún $a \in G$.

Lema 1.4.1. Todo grupo cíclico es abeliano.

Ejemplo: Considerar el conjunto todos los residuos que resultan de la división euclídea de enteros entre un entero fijo n, al cual denotaremos por $\mathbb{Z}_n$, Por el teorema 1.4.1, se sabe que $\mathbb{Z}_n = \{0, 1, 2, \dots, n-1\}$, ahora si se define la operación que toma dos elementos $a, b \in \mathbb{Z}_n$ tal que $a \cdot b = (a +$

$b)\%n$, se obtiene que $(\mathbb{Z}_n, \cdot)$ es un grupo cíclico de orden n cuyo elemento generador es 1, el cual también se conoce como grupo de enteros aditivos módulo n.
De manera particular, considerar $n = 5$, así $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$. Si se toma $a = 3$ y $b = 4$ se tiene que $a \cdot b = 2$.

•	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Cuadro 1.2: tabla de multiplicar grupo $\mathbb{Z}_5$

Proposición 1.4.2. Sean G un grupo abeliano y $a, b \in G$. Se tiene que $(ab)^n = a^n b^n$ para todo $n \in \mathbb{Z}$.

Definición 1.4.14 (Orden de un elemento)**.** Sea un grupo G. El orden de un elemento $a \in G$ se define como el menor entero positivo α, tal que $a^\alpha = e$, y se lo denota como $|a|$.

Teorema 1.4.5. Si el orden de un grupo es primo, entonces el grupo es cíclico.

Teorema 1.4.6 (Lagrange)**.** Si H es un subgrupo de G, entonces $|H|$ divide a G.

Lema 1.4.2. Dado un elemento a de un grupo G, se tiene lo siguiente

- $|\langle a \rangle| = |a|$.
- $|a|$ divide a $|G|$.
- Si $n \in \mathbb{Z}$ es tal que $a^n = e$ entonces $|a|$ divide a n.

Teorema 1.4.7 (Cauchy)**.** Dados un grupo finito G y p un número primo tales que p divide a $|G|$, entonces existe un elemento $a \in G$ tal que $|a| = p$.

Definición 1.4.15. Dado un grupo G, se define el producto de dos subconjuntos H y K de G como $HK = \{hk : \ h \in H, \, k \in K\}$.

Notación: Cuando sea necesario representar el producto de varios subconjuntos $H_1, H_2, \ldots, H_n$ de un grupo G, se lo representará mediante $\prod_{i=1}^{n} H_i$.

Definición 1.4.16. Dado N un subgrupo de G. Se dice que N es un subgrupo normal de G si para todo $g \in G$ y para todo $n \in N$ se cumple que $gng^{-1} \in N$.

Teorema 1.4.8. El producto de dos subgrupos normales, es un subgrupo normal.

Definición 1.4.17. Un homomorfismo entre dos grupos $(G, \cdot)$ y $(H, *)$ se define como una función $\varphi : G \to H$ tal que para todo par de elementos $g_1, g_2 \in G$ se cumple $\varphi(g_1 \cdot g_2) = \varphi(g_1) * \varphi(g_2)$.

Definición 1.4.18. Sea $\varphi : G_1 \to G_2$ un homomorfismo entre los grupos G_1 y G_2, el kernel o núcleo de φ se define como el siguiente subconjunto de G_1

$$\mathcal{K}_\varphi = \{x \in G_1 : \varphi(x) = \bar{e}\}$$

donde $\bar{e}$ es la identidad de G_2.

Proposición 1.4.3. Un homomorfismo es inyectivo si y solo si su kernel es el conjunto cuyo único elemento es la identidad.

Definición 1.4.19. Dos grupos G_1 y G_2 se dicen isomorfos si existe un homomorfismo $\varphi : G_1 \to G_2$ que es biyectivo. Se denota como $G_1 \cong G_2$; además, φ en este caso es conocido como isomorfismo.

Definición 1.4.20. El producto directo de dos grupos G y H, denotado por $G \times H$, se define como el siguiente conjunto de pares ordenados

$$G \times H = \{(g, h) : \, g \in G, \, h \in H\}.$$

Adicionalmente se define una operación binaria $*$ sobre $G \times H$ de la siguiente forma: Si $(g_1, h_1), (g_2, h_2) \in H \times K$ entonces $(g_1, h_1) * (g_2, h_2) = (g_1 g_2, h_1 h_2)$.

Proposición 1.4.4. $(G \times H, *)$ es un grupo.

Lema 1.4.3. Dados un grupo H y dos grupos isomorfos, $G \cong \bar{G}$ se tiene que $G \times H \cong \bar{G} \times H$.

Notación: Para hacer referencia al producto directo de varios grupos, $G_1 \times G_2 \times \cdots \times G_n$ se usará la representación simplificada $\bigotimes_{i=1}^{n} G_i$.

Teorema 1.4.9. Sean G un grupo y dos subgrupos normales H y K tales que $H \cap K = \{e\}$ y $HK = G$ entonces

1. Para todo $h \in H$ y $k \in K$: $hk = kh$.

2. Para todo $a \in G$ existen únicos elementos $h \in H$ y $k \in K$ tales que $a = hk$.

3. $G \cong H \times K$.

Demostración.

1. Sean h, k cualquier par de elementos en H y K respectivamente, por la normalidad de H y K, se tiene que $h^{-1}(khk^{-1}) \in H \cap K$. Gracias a la hipótesis, $h^{-1}(khk^{-1}) = e$, lo que implica que $kh = hk$.

2. Debido a que $HK = G$, existen h_1, k_1 en H y K respectivamente tales que $a = h_1k_1$. Lo siguiente es suponer que existen otros elementos $h_2 \in H$ y $k_2 \in K$ tales que $a = h_2k_2$. Entonces $h_1k_1 = h_2k_2$ lo que implica que $(k_2k_1^{-1}), (h_2^{-1}h_1) \in K \cap H = \{e\}$. En consecuencia $h_2 = h_1$ y $k_2 = k_1$. Por tanto h_1 y k_1 son únicos.

3. Se define la siguiente función $\varphi : H \times K \to G$, donde $\varphi(h, k) = hk$. Gracias a la conclusión 1. se obtiene que φ es un homomorfismo. Ahora, dado un elemento $(h, k) \in K_\varphi$ que como tal $hk = e$. Así $h, k \in H \cap K$, es decir $(h, k) = (e, e)$. Gracias la proposición 1.4.3, se infiere que φ es inyectivo. Además φ es sobreyectivo pues $Im(\varphi) = HK = G$. Por ello φ es un isomorfismo y así $G \cong H \times K$.

□

Teorema 1.4.10. Sean G un grupo y $H_1, H_2, \ldots, H_n$ subgrupos normales, tales que $G = \prod_{i=1}^{n} H_i$ y $K_i = H_i \cap \prod_{j \neq i}^{n} H_j = \{e\}$ para todo $i \in \{1, 2, \ldots, n\}$, entonces $G \cong \bigotimes_{i=1}^{n} H_1$.

Demostración. La demostración de este resultado se realizará por inducción sobre n.

- Para $n = 2$, el resultado es equivalente al numeral 3. del teorema 1.4.9.

- Hipótesis de inducción: Dado un grupo $\bar{G} = \prod_{i=1}^{n-1} H_i$ donde $H_1, H_2, \ldots, H_{n-1}$ son normales y $K_i = \{e\}$ para todo $i \in \{1, 2, \ldots, n-1\}$, cumple $\bar{G} \cong \bigotimes_{i=1}^{n-1} H_i$.

- Demostración: Considerar un grupo G tal que $G = \prod_{i=1}^{n} H_i$ donde H_i es subgrupo de G y $K_i = \{e\}$ para todo $i \in \{1, 2, \ldots, n\}$. Se puede reescribir G de la siguiente forma $G = \bar{G}H_n$, donde el conjunto $\bar{G} = \prod_{i=1}^{n-1} H_i$ es un subgrupo normal de G debido al teorema 1.4.8. Además, por hipótesis $\bar{G} \cap H_n = K_n = \{e\}$. Todas estas observaciones convergen al caso $n = 2$, por lo tanto $G \cong \bar{G} \times H_n$.
  Es importante tener en cuenta que H_i es un subgrupo normal de $\bar{G}$ para todo $i \in \{1, 2, \ldots, n-1\}$. De esta forma se puede aplicar la hipótesis de inducción a $\bar{G}$ y en consecuencia $\bar{G} \cong \bigotimes_{i=1}^{n-1} H_i$.
  Aplicando el teorema 1.4.3 se obtiene que $G \cong \bigotimes_{i=1}^{n} H_i$.

□

Definición 1.4.21. Sea p un número primo. Un grupo G se dice $p-$grupo si para todo $g \in G$, $|g|$ es una potencia de p.

Definición 1.4.22. Sea G un grupo. Un elemento $g \in G$ se dice de orden máximo si $|g| \geq |a|$ para todo $a \in G$.

Proposición 1.4.5. En un grupo finito, siempre existe el elemento de orden máximo.

1.4.3. Grupos abelianos finitos

A partir de este momento se trabajará únicamente con grupos abelianos finitos. Para dicho propósito será útil contar con una notación especial, que permita distinguirlos de los grupos en general.

Notaciones y su significado

Para empezar, se hará referencia a los grupos abelianos como grupos aditivos y a su operación binaria de grupo como adición, representada

con el símbolo de suma $+$.
Se consideran un grupo aditivo finito G, dos elementos $a, b \in G$, dos subconjuntos $H, K \subseteq G$ y un entero $n \in \mathbb{Z}$. Bajo este contexto, se dispone de la siguiente tabla de notaciones, donde se reflejan los cambios.

grupos generales	ab	a^{-1}	e	ab^{-1}	HK	H_a	$\prod_{i=1}^{n}$	$H \times K$	$\bigotimes_{i=1}^{n}$	a^n
grupos aditivos	$a+b$	$-a$	0	$a-b$	$H+K$	$H+a$	$\sum_{i=1}^{n}$	$H \oplus K$	$\bigoplus_{i=1}^{n}$	na

Cuadro 1.3: notaciones para grupos abelianos

Teorema 1.4.11. Sean G un grupo aditivo finito y p un número primo. G es un $p-$grupo si y solo si $|G|$ es una potencia de p.

Demostración. $\Rightarrow$)
Sea G un $p-$grupo. Gracias al teorema fundamental de la aritmética 1.4.2, existen $p_1, p_2, \ldots, p_n$ números primos distintos entre sí, y $k_1, k_2, \ldots, k_n \in \mathbb{Z}^+$ tales que

$$|G| = p_1^{k_1} p_2^{k_2} \ldots p_n^{k_n}.$$

Por contradicción. Suponer que $|G|$ no es una potencia de p, lo cual implica que existe $i \in \{1, \ldots, n\}$, tal que $p_i \neq p$. Como p_i es primo y divide a $|G|$, se puede aplicar el teorema de Cauchy, 1.4.7, el cual afirma la existencia de un elemento $a \in G$ tal que $|a| = p_i$. Puesto que G es un $p-$grupo, se tiene que existe $k \in \mathbb{Z}^+$ tal que $|a| = p^k$, de este modo $p_i = p^k$, así p_i divide a p, lo cual es una contradicción pues p es primo y $p_i \neq p$.
Se concluye que $|G|$ es una potencia de p.
$\Leftarrow$)
Ahora, suponer que $|G|$ es una potencia de p, por ello, existe $\alpha \in \mathbb{Z}^+$ tal que $|G| = p^\alpha$. Sea $a \in G$. Por el lema 1.4.2, se sabe que $|a|$ divide a $|G|$, y dado que p es primo, entonces $|a| = p^\beta$ para algún $\beta \in \mathbb{Z}^+$ tal que $\beta \leq \alpha$. Lo que significa que $|a|$ es una potencia de p.
Se concluye que G es un $p-$grupo. □

Definición 1.4.23. Sea G un grupo aditivo finito cuya descomposición primaria de su orden es $|G| = p_1^{k_1} p_2^{k_2} \ldots p_n^{k_n}$. Para cada $i \in \{1, 2, \ldots, n\}$ se define el conjunto, G_i conformado por todos los elementos de G cuyo orden es una potencia de p_i. Los conjuntos $G_1, G_2, \ldots, G_n$ se conocen como las componentes primarias de G.

Proposición 1.4.6. Las componentes primarias de un grupo abeliano finito son subgrupos.

Demostración. Sea G un grupo abeliano finito cuya descomposición primaria de su orden es $|G| = p_1^{k_1} p_2^{k_2} \dots p_n^{k_n}$. Dado un índice $i = 1, 2, \dots, n$ se considera cualquier par de elementos $a, b \in G_i$ lo que significa que existen $\alpha_1, \alpha_2 \in \mathbb{Z}^+$ tales que $|a| = p_i^{\alpha_1}$ y $|b| = p_i^{\alpha_2}$. En virtud del teorema 1.4.2 se tiene que $p_i^{\alpha}(a + b) = p_i^{\alpha} \cdot a + p_i^{\alpha} \cdot b = 0$ con $\alpha = \text{máx}\{\alpha_1, \alpha_2\}$. Por el lema 1.4.2 se deduce que $|a + b|$ divide a p_i^{α}, pero al ser p_i primo, cualquier divisor de alguna de sus potencias debe ser necesariamente otra potencia de p_i. Por lo tanto $(a + b) \in G_i$. Gracias al teorema 1.4.4 se concluye que G_i es un subgrupo de G. □

Lema 1.4.4. Dado un grupo abeliano finito G con descomposición primaria de su orden, $|G| = p_1^{k_1} p_2^{k_2} \dots p_n^{k_n}$, se tiene que G es la suma directa de sus componentes primarias, es decir $G \cong \bigoplus_{i=1}^{n} G_i$.

Demostración. Se aplicará el teorema 1.4.10, para ello se debe probar que $G_i \cap \left(\sum_{j \neq i} G_j \right) = \{0\}$ para todo $i \in \{1, 2, \dots, n\}$ y además $G = \sum_{i=1}^{n} G_i$. La condición que exige a los subgrupos G_i ser normales, se cumple gracias a que G es abeliano.

Considerar cualquier $i \in \{1, 2, \dots, n\}$. Por contradicción. Suponer que existe un elemento $a \in G_i \cap \left(\sum_{j \neq j} G_j \right)$ tal que $a \neq 0$. Como $a \in G_i$, $|a| = p_i^{\alpha}$ para algún $\alpha \in \mathbb{Z}$, $\alpha \geq 1$. Por otro lado, $a \in \left(\sum_{j \neq i} G_j \right)$ así existen $a_j \in G_j$ tales que $a = \sum_{j \neq i} a_j$, con ello, también existen $\alpha_j \in \mathbb{Z}$ tales que $|a_j| = p_j^{\alpha_j}$, para todo $j \neq i$. De este modo $|a| = p_i^{\alpha}$ divide a $\prod_{j \neq i} p_j^{\alpha_j}$, lo cual es una contradicción, ya que $p_1, p_2, \dots, p_n$ son primos y distintos entre si. Por lo tanto $a = 0$.

Ahora considerar un elemento $a \in G$. Se definen los números $\alpha_i = |G| / p_i^{k_i}$ para todo $i \in \{1, 2, \dots, n\}$. Debido a que $p_1, \dots, p_n$ son primos y distintos entre si, es fácil evidenciar que $(\alpha_1, \alpha_2, \dots, \alpha_n) = 1$, con ello se puede aplicar la proposición 1.4.1 y afirmar que existen $x_1, x_2, \dots, x_n$ tales que $\sum_{i=1}^{n} x_i \alpha_i = 1$ lo cual implica que $\sum_{i=1}^{n} x_i \alpha_i a = a$. De esa forma definimos $a_i = x_i \alpha_i a$, para todo $i \in \{1, 2, \dots, n\}$, y en consecuencia $a = \sum_{i=1}^{n} a_i$. El objetivo es probar que $a_i \in G_i$. Notar que $p_i^{k_1} a_i = x_i(na) = 0$ lo que significa

que $|a_i|$ divide a $p_i^{k_i}$, es decir $|a_i|$ es una potencia de p_i. En resumen $a_i \in G_i$. Finalmente, aplicando el teorema 1.4.10 se concluye que $G \cong \bigoplus_{i=1}^{n} G_i$. □

Lema 1.4.5. Sean p un número primo y G un $p-$grupo abeliano, entonces G es isomorfo a la suma directa de grupos cíclicos.

Demostración. Por el teorema 1.4.11, existe $n \in \mathbb{Z}^+$ tal que $|G| = p^n$. A continuación se procede por inducción sobre n.

- Para $k = 1$: Se cumple gracias al teorema 1.4.5.
- Hipótesis de inducción: Todo $p-$grupo G, de orden p^k con $k \leq n-1$, es la suma directa de grupos cíclicos.
- Demostración: Suponer que $|G| = p^n$. Tomar el elemento $g \in G$ con orden máximo, es decir $|a| \leq |g|$ para todo $a \in G$. Como $|g|$ divide a p^n, existe $m \leq n$ tal que $|g| = p^m$. El primer paso será demostrar que

$$(p^m)a = 0 \text{ para todo } a \in G \tag{1.3}$$

Sea $a \in G$, se sabe que $|a| = p^r$ con $r \leq m$, así existe $z \in \mathbb{Z}$ tal que $m = z + r$, de este modo $(p^m)a = p^z(p^r\, a) = 0$. Con ello se verifica (1.3). Considerar H el máximo subgrupo de G tal que $\langle g \rangle \cap H = \{0\}$, el cual existe pues al menos para $H = \{0\}$ se cumple. El segundo paso consiste en demostrar que

$$\langle g \rangle + H = G \tag{1.4}$$

Por contradicción. Suponer que existe $b \in G$ tal que $b \notin \langle g \rangle + H$. Gracias a 1.3, existe el mínimo $r \in \mathbb{Z}$ tal que $(p^r)b \in \langle g \rangle + H$ ya que a lo más para $r = m$ se cumple. Con ello se define $c = (p^{r-1})b$, así $c \notin \langle g \rangle + H$ y $pc \in \langle g \rangle + H$ y por ende, existen $t \in \mathbb{Z}$ y $h \in H$ tales que $pc = tg + h$. Haciendo uso nuevamente de (1.3), se infiere que $p^{m-1}(pc) = 0$, lo que implica $p^{m-1}(tg) \in H$, además de estar en $\langle g \rangle$. Como $\langle g \rangle \cap H = \{0\}$ se tiene que $p^{m-1}(tg) = 0$, y de este modo $|tg|$ divide a p^{m-1}, entonces $|tg| < |g|$ lo cual provoca que $\langle tg \rangle \neq \langle g \rangle$. Por el lema 1.4.2 se infiere que p^m divide a tp^{m-1}, es decir, existe $\alpha \in \mathbb{Z}$ tal que $tp^{m-1} = \alpha p^m$, así $t = \alpha p$.
Se definen $s = -\alpha$ y con ello $K = \langle c + sg \rangle + H$, el cual es un subgrupo de

G. Notar que H es un subgrupo de K, sin embargo es un subgrupo propio, pues $c+sg \notin H$ debido a que $c \notin \langle g \rangle + H$. El objetivo es probar que $\langle g \rangle \cap K = \{0\}$ porque de esta manera se contradecería que H es el máximo subgrupo con esta propiedad, y se verificaría (1.4). Primero notar que todo elemento $y \in K$ es de la forma, $y = l(c+sg)+h_2$ donde $l \in \mathbb{Z}$ y $h_2 \in H$.
Ahora se analiza el elemento $p(c+sg)$

$$\begin{aligned} p(c+sg) &= pc - (\alpha p)g \\ &= pc - tg \\ &= h. \end{aligned}$$

Lo anterior tiene por consecuencia que el elemento $y = l(c+sg)+h_2$ pertenezca a H si l es un múltiplo de p. En resumen, si l es múltiplo de p entonces $y = 0$ o $y \notin \langle g \rangle$. Ahora se analiza el caso cuando l no es múltiplo de p. Como p es primo entonces $(l,p) = 1$, por la proposición 1.4.1 se sabe que existen $\gamma, \beta \in \mathbb{Z}$ tales que $\gamma l = 1 + \beta p$. Así

$$\begin{aligned} \gamma y = \gamma(l(c+sg)+h_2) = \gamma l(c+sg) + \gamma h_2 = \\ (1+\beta p)(c+sg) + \gamma h_2 = (c+sg) + \beta h + \gamma h_2 \\ \Rightarrow \gamma y \in K \end{aligned}$$

En este caso, no es posible que $y \in \langle g \rangle$, pues si lo fuera, $y = \lambda g$ y entonces

$$\begin{aligned} &\gamma y = (\gamma\lambda)g = c + sg + \beta h + \gamma h \\ &\Rightarrow c = (\gamma\lambda - s)g + (\beta h + \lambda h_2) \\ &\Rightarrow c \in \langle g \rangle + H \quad \rightarrow\leftarrow \end{aligned}$$

Entonces la única posibilidad es que $y = 0$, de esa manera se verifica que $\langle g \rangle \cap K = \{0\}$, lo que contradice el hecho que H es el único subgrupo con esta propiedad. Por lo tanto no existe dicho elemento $b \in G$. En resumen $G = \langle g \rangle H$. Aplicando el teorema 1.4.9, se tiene que $G \cong \langle g \rangle \oplus H$. Finalmente, como $|H| < p^n$, se puede aplicar la hipótesis de inducción a H, y se concluye el resultado.

$\square$

Este resultado tiene más formas de ser demostrado, sin embargo la que se ha llevado a cabo en el presente trabajo, está basada en la demostración disponible en [4].

Teorema 1.4.12 (Teorema fundamental de grupos abelianos finitos)**.** Todo grupo abeliano finito es la suma directa de grupos cíclicos

Demostración. Consecuencia de juntar los lemas 1.4.4, 1.4.5 y 1.4.3. □

1.4.4. Programación genérica

Varios lenguajes de programación cuentan con un versátil paradigma conocido como programación genérica o programación orientada a objetos. Su objetivo es trabajar con entidades abstractas o conceptos en el ámbito computacional, permitiendo al usuario definir nuevos tipos de datos o generalizar los ya existentes[3]. C++ cuenta con este recurso y es por ello que se lo ha escogido para la realización de este proyecto. Alguna de las herramientas de la programación orientada a objetos que se utilizarán son Clases y Plantillas.

Clases: Una clase es un tipo de dato, creado por el usuario en representación a un concepto, dentro del código del programa. A los elementos que cumplan las condiciones del concepto, dentro de las clases, son conocidos como objetos. Una clase incluye las características o atributos de los objetos y además permite crear las funciones o acciones que pueden ejecutarse entre ellos conocidos como métodos. Tanto métodos como atributos son conocidos de manera general como miembros de la clase.

Plantillas Sirven para generalizar tipos. En C++ cada variable pertenece a un tipo específico y cuando se crea una función, sus parámetros también deben cumplir esta formalidad, lo que implica que solo para ese tipo de datos, se ejecutará correctamente las acciones de la función. El problema empieza al momento de observar que las acciones de una función se podrían emplear con parámetros de diferentes tipos a los que se especificaron en su prototipo. En el lenguaje C, antecesor de C++, la única solución a este problema era crear una nueva función con los nuevos

parámetros. Afortunadamente se crearon las plantillas o *templates* que permiten crear funciones e inclusive clases de tipos generales, los cuales se especificarán al momento de llamar a la función o instanciar un objeto y más no al momento de declarar.

Tanto las clases como plantillas, son recursos que permitirán llevar a cabo los objetivos del presente componente. En el capítulo anterior se ejemplificaron varios grupos, los cuales se busca implementar computacionalmente, dicha tarea se ajusta a las bondades de las clases. Además se incluyeron conceptos en un nivel superior, es decir que trabajan con los grupos de manera general, por ejemplo los isomorfismos. Ya se puede deducir que la herramienta adecuada para reflejar estos conceptos en el programa, son las plantillas. En definitiva, uno de los pilares mas fuertes del modelo que resultará de este trabajo, tiene sus bases en la programación genérica.

Capítulo 2

Metodología

Este capítulo detallará el proceso de implementación del modelo computacional, explicando cómo funciona cada una de las partes del código y cómo guardan armonía con la teoría anteriormente presentada.

El software utilizado ha sido el IDE CodeBlocks versión 17.12 y el lenguaje de programación C++. Las librerias requeridas han sido *iostream*, *algorithm* y *cmath*.

La dinámica a usar en la metodología será describir cada una de las implementaciones del programa. Se incluirá el código únicamente de las partes más relevantes que necesiten una explicación más detallada. Sin embargo el código entero de cada archivo creado se incluirá como anexo al final del documento al cual se realizarán referencias a lo largo del capítulo.

2.0.1. Implementaciones grupos abelianos

Funciones globales

- *int mcd(int a, int b)*: La primera función del código es referente al cálculo del máximo común divisor entre dos números enteros, a y b, basado en el algoritmo de Euclides [citar] y utilizando la recursividad.

- $bool\ verificacion_primo(\ int\, p)$: En varias ocasiones se querrá utilizar el teorema 1.4.5, que utiliza el concepto de número primo. Debido a esta situación ha sido útil implementar una función que permita verificar si un entero p es un número primo.

  El algoritmo verifica que el número a analizar no sea divisible para 2. Luego, que no sea divisible entre los números impares. Además el criterio de parada usa el resultado que indica que todo número compuesto (no primo) posee un divisor primo menor o igual que su raíz cuadrada.

- $void\ factores_primos(\ int\, n,\ int^*k\,\&,\ int\, t\&)$. Las componentes primarias serán utilizadas posteriormente para la descomposición de los grupos U_n. Recordando que este concepto aplica descomponer en factores primos al orden del grupo, será necesaria una función que se encargue de esta tarea. La estrategia es similar a la de la anterior función. Va dividiendo primero para dos y luego para los impares hasta donde sea posible. Usa nuevamente el hecho que todo compuesto tiene un factor primo menor a su raíz. Lo importante de esta función es que va almacenando cada divisor diferente en un arreglo dinámico k y contando el número de factores diferentes, t.

Clase para Grupos U_n

Como se vio en el ejemplo de la definición 1.4.7, cada entero n induce un grupo U_n. Si se quiere estudiar a estos grupos de manera general, se debe implementar el concepto de U_n y más no un grupo U_n con n específico. Tal motivación llevó a crear en el programa la clase U_n. A continuación se detalla como esta clase permite acceder y trabajar con cuaquiera de estos grupos.

Miebros de la clase

- Constructor vacio $Un(\,)$: Para crear objetos de la clase sin argumentos previos.
- $int\ n$: Referente al entero que define la operación modular del grupo.
- $int\ p$: Corresponde a un primo relativo de n. Permite trabajar con un elemento de U_n de manera individual.

- Constructor $Un(int\,, int)$: Su primer argumento transmite información a n y el segundo a p. Tendrá sentido siempre que $(n,p)=1$, caso contrario se retorna $Un(1,1)$ precedido por un mensaje de error. Por ejemplo el objeto $Un(7,4)$ hace referencia al elemento 4 de U_7.
- *bool* **elementos*: Este atributo permite trabajar con subconjuntos de U_n y no únicamente con elementos individuales, es decir un objeto de la clase puede tener dos puntos de vista, como elemento o subconjunto. Por tanto siempre que se mencione que un elemento pertenece a un objeto de la clase, es porque se lo está considerando desde la perspectiva de conjunto.
  Su funcionamiento es de la siguiente manera, si un elemento i del grupo, pertenece al objeto, entonces la $i-$ésima posición de *elementos* es verdadera, y falsa en caso contrario.
- *void agregar*(int): Permite agregar elementos a un objeto. Si el parámetro toma un valor m, primero se verifica que sea un primo relativo de n, y si lo es, el valor de la $(m\,\%n)-$ésima posición de *elementos* se vuelve verdadero.
- *void eliminar*(int): Sirve para sacar elementos de un objeto mediante un proceso análogo al de *agregar*.
- *Un operator* $+$ $(const\ Un\,\&)const$: Es la representación de la operación binaria del grupo, la cual es una adición pues U_n son grupos abelianos finitos (contexto en 1.3). Tiene sentido siempre que el atributo n de ambos objetos a multiplicar sean iguales.
- *bool operator* $==$ $(const\ Un\,\&)const$: Verifica que dos objetos vistos como elementos, sean iguales.
- *int obtener_orden*( ) Retorna el orden del grupo. Con la ayuda de la funcion *mcd* se cuenta los primos relativos menores de n.
- *int orden_elemento*( ): Devuelve el orden de un objeto visto como elemento.
- *Un inverso*( ): Obtiene elemento inverso del objeto original.
- *int cardinalidad*( ) Brinda información de la cardinalidad del objeto, es decir el número de valores verdaderos en su atributo *elementos*.
- *Un G*( ): Crea un objeto que contiene a todos los elementos del grupo.
- *void tabla*( ) Imprime la tabla de multiplicar del grupo U_n. Utiliza el objeto que resulta del método $G(\,)$.

- Un *suma_conjuntos* (Un, Un): Sus argumentos corresponderán a objetos de la clase y lo que ejecuta es la suma de los objetos vistos como subconjuntos.
- *bool subgrupo*( ) Verifica que un objeto visto como subconjunto, sea un subgrupo. Hace uso de la proposición 1.4.4 y el hecho que los grupos U_n son abelianos.

La implementación de las funciones globales, de la clase Un y sus miembros, se encuentran disponibles en el código A.1.

2.0.2. Implementaciones para grupos no abelianos

A continuación se detalla como se crearon tres nuevas clases en representación a los grupos no abelianos D_4 como simetrías, D_4 como permutación y el grupo de los cuaterniones, Q_8. Las tres clases siguen una implementación similar y los nombres de sus métodos son iguales, ésto con el propósito de utilizar plantillas posteriormente.
La construcción de objetos de estas clases están en base a parámetros de tipos variados, no únicamente de tipo entero como en el caso de U_n, lo que provoca que sea más complicado crear objetos de la clase. Lo ideal sería poder instanciar objetos por medio de variables de tipo entera. Con esa motivación, para cada grupo se ha creado un arreglo global del mismo tipo que los parámetros de sus constructores, con la finalidad de acceder y crear objetos a través de índices, pasando primero por los valores de estos arreglos externos.

Clase para D_4 como permutaciones

El arreglo global correspondiente a la clase $D4$, que permitirá construir objetos a partir de sus valores, está declarado de la siguiente manera:

```
string GrupoD4[8]={"id","f4","r1","f3","r3","f1","f2","r2"};
```

Sus valores corresponden a los nombres de los elementos de este grupo y la equivalencias con los elementos de D_4 como simetrías son de la

siguiente forma:

$$r1 = r_{90}, \quad r2 = r_{180}, r3 = r_{270},$$
$$f1 = v, \quad f2 = h, \quad f3 = d_1, \quad f4 = d_2.$$

En principio, no tiene relevancia la posición de los nombres, pero más adelante se observarán comportamientos interesantes a partir de realizar cambios a las posiciones. Sin embargo, varios métodos están diseñados para que el primer elemento sea siempre la identidad del grupo, por lo que se recomienda conservar la de posición a este elemento.

Una particularidad del grupo $D4$ es que se puede generar con tan solo dos elementos, una rotación y un giro por ejes de simetría. Se puede aprovechar esta propiedad, declarando a los dos elementos generadores como arreglos globales de tamaño cuatro, y dentro de la clase generar todos los demás elementos del grupo a partir de estos arreglos. Por tanto como preámbulo de la clase, se definen adicionalmente los siguientes arreglos

```
int R1[4]={2,3,4,1};//Rotacion de 90 grados
int F4[4]={1,4,3,2};//Reflexion por la primera diagonal
```

Miembros de la clase

- Nombre de la clase: $D4$.
- $string\ nombre$: Representa el nombre del objeto.
- $int\ a[4]$: Atributo basado en la notación vectorial de permutación vista en (1.1).
- Constructor $D4(\ int\, m[4])$: Los valores del arreglo m se transmitirán a a, luego se asigna el nombre de la función por medio de conocer cuales son los dos primeros elementos de m, en base a la observación del cuadro 1.1.
- Constructor $D4(\ string)$: El parámetro pasa información al atributo nombre. Luego se realiza una verificación de igualdad sobre el nombre y según ello se asigna los valores de a. En este punto son de utilidad los arreglos $R1$ y $F4$. Por ejemplo si el nombre es $r2$, se refiere a la rotación de $180°$ y se genera a partir de aplicar dos veces $R1$. Consultar anexo A.2 lineas 399-458.

- $bool\,{}^*elementos$ Análogo al de la clase Un.
- $void\ agregar(\,string)$: Agrega elementos por su nombre. Busca la posición del nombre en $GrupoD4$, y luego asigna verdadero a esa posición en el arreglo $elementos$.
- $void\ eliminar(\,string)$ Elimina elementos de un objeto, siguiendo la misma estrategia que para agregarlos.
- $D4\ operator * \ (const\ D4\,\&)const$ Representa a la operación del grupo. Al tratarse de permutaciones, lo que realiza el operador, es pasar posiciones mediante el atributo a.
- $bool\ operator\ ==\ (\,const\ D4\,\&)const$ Operador de comparación de igualdad para verificar si dos objetos de la clase son iguales mediante comparar que sus nombres coincidan.
- $bool\ subgrupo(\,)$ Verifica que un objeto visto como subconjunto sea un subgrupo, mediante comprobar que se cumple la clausura, pero primero comprobando que la identidad se encuentre en el conjunto.
- $bool\ subgrupo_normal(\,)$ Verifica que un objeto sea un subgrupo normal. Primero comprueba que sea subgrupo y luego la definición 1.4.16.
- $D4\ transmutar(\,int)$ Permite que un objeto se transforme en cualquier otro objeto de la clase, donde su parámetro indica la posición en el arreglo $GrupoD4$ del nombre del elemento al cual se va a transmutar.
- $bool\ pertenece(\,string)$ Verifica si un elemento del grupo pertenece al objeto, por medio de su nombre.

Cabe mencionar que algunos métodos no se han mencionado, debido a que su implementación tiene un procedimiento análogo al de los métodos de Un. Para el detalle de los miembros omitidos se sugiere consultar A.2.

Clase para D_4 como acciones de simetría

El modelo cuenta con otra clase llamada $d4$ la cual es la representación del grupo D_4 visto como simetrías. Dentro del código, se ha correspondido a las acciones de simetría con funciones, pero para darle un carácter especial que las diferencie de las funciones comunes, se ha asignado un nuevo nombre, $fd4$, al tipo función entera mediante el operador $typedef$. La sentencia que realiza este cambio es la siguiente:

```
typedef int (*fd4)(int)
```

Por lo tanto cada vez que se instancie una variable del tipo $fd4$, se hará referencia a que es una función de simetría del cuadrado. Cada función de simetría debe ser declarada, según la acción que realice. Por lo tanto, un preámbulo de la clase necesita de la declaración de las ocho acciones de simetría.

```
int id(int);
int r90(int);
int r180(int);
int r270(int);
int h(int);
int v(int);
int d1(int);
int d2(int);
```

En el código A.2, en las líneas 15 a 52 está incluída la implementación de cada una de estas funciones.
Adicionalmente, para esta clase también existe un arreglo que permitirá crear objetos mediante variables de tipo entero.

```
fd4 Grupod4[8]={id,r90,r180,r270,v,h,d2,d1};
```

Esta clase guarda bastante similitud con la anterior $D4$, por ello se detallarán únicamente los miembros de la clase que impliquen algo novedoso en su implementación.

Miembros de la clase

- $fd4\ funcion$ Representa a la función de simetría la cual puede ser id, $r90$, $r180$, $r270$, v, h, $d1$ o $d2$.
- Constructor $d4\ (fd4)$. Construye el objeto a partir de la función de simetría.
- $string\ nombre$: Será de utilidad para imprimir objetos y saber de cual se trata. Se le asigna un valor desde el constructor $d4(fd4)$, dependiendo de cual sea la función tipo $fd4$.
- $d4\ operator * (const\ d4\&)const$ Representa el producto del grupo. Su dinámica consiste en encontrar la función de $Grupod4$, que sea igual a la composición de los atributos $funcion$ de los objetos sobre los cuales se está multiplicando mediante verificar que las imágenes de las funciones coincidan en al menos dos elementos.
- $bool\ operator == (constd4\&)const$ Comparacion de igualdad entre objetos. Se lleva a cabo por medio de verificar que las imágenes de los atributos $funcion$ coincidan en al menos dos elementos.

La implementación de la clase y de todos sus miembros se puede consultar en el código A.2 entre las líneas 55 hasta 254.

Clase para grupo Q_8

El arreglo correspondiente a esta clase para crear objetos mediante variables enteras se declaró de la siguiente forma

```
string GrupoQ8[8]={"1","-1","i","-i","j","-j","k","-k"};
```

Miembros de la clase

- $string\ nombre$: Correspondiente al nombre del elemento que puede ser 1, -1, i, $-i$, j, $-j$, k y $-k$.
- $int\ a, b, c, d$: Corresponden a los coeficientes del cuaternion, por ejemplo $i = 0 + (1)i + 0j + 0k$, entonces para este elemento se tiene que $a = 0$, $b = 1$, $c = 0$ y $d = 0$.
- Constructor $Q8\,(int,\ int,\ int,\ int)$: Sus parámetros asignan valores para a, b, c y d.
- $Q8\ operator * \ (const\ Q8\&)const$: Respresenta el producto del grupo. Está basado en la ecuación (1.2).

Los demás miembros de la clase y su implementación se encuentran en el código A.2.

2.0.3. Descomposición de grupos U_n

En esta sección se verá el detalle de la implementación de la función para descomponer como sumas directas de grupos cíclicos a los grupos U_n. La base teórica está conformada por el teorema fundamental de grupos abelianos finitos el cual a su vez tiene su respaldo los lemas 1.4.4 y 1.4.5. El procedimiento de la función va de la siguiente forma: Por el lema 1.4.4 se asegura que todo grupo abeliano es la suma directa de sus componentes primarias y luego cada componente a su vez, debe descomponerse en grupos cíclicos como lo asegura el lema 1.4.5. La cuestión es cómo descomponer cada componente primaria. La respuesta se encuentra en la demostración de1.4.5.

Para entrar en contexto, suponer que se busca descomponer en grupos

cíclicos a un $p-$grupo G (recordar que las componentes primarias son $p-$grupos, p primo). La demostración del lema, indica que $G = \langle g_1 \rangle \oplus H_1$ donde g_1 es el elemento de orden máximo de G y H_1 un subgrupo, tales que $\langle g_1 \rangle \cap H = \{0\}$. H es el subgrupo al cual precisamente se le aplica la hipótesis inductiva. El efecto de dominó de la inducción matemática conduce a pensar que H_1 también va a ser descompuesto como $H_1 = \langle g_2 \rangle \oplus H_2$ donde H_2 es un subgrupo de H_1 y g_2 su elemento de orden máximo, tales que $\langle g_2 \rangle \cap H_2 = \{0\}$, y así se repite este proceso una cantidad finita de pasos, suponer m pasos, y precisamente se detiene pues G llega a ser igual a la suma de los subgrupos cíclicos generados por $g, g_1, \ldots, g_m$, que deben cumplir las siguientes propiedades

$$\langle g_i \rangle \cap \sum_{j>i} \langle g_j \rangle = \{0\} \qquad \forall i \in \{1, 2, \ldots, m-1\} \tag{2.1}$$

$$\sum_{j=1}^{m} \langle g_j \rangle = G. \tag{2.2}$$

Debido a que $0 \in \langle g_i \rangle$ para todo $i \in \{1, 2, \ldots, n\}$, la condición (2.1) es equivalente a

$$\langle g_i \rangle \cap \sum_{j \neq i} \langle g_j \rangle = \{0\} \qquad \forall i \in \{1, 2, \ldots, m\}. \tag{2.3}$$

Por otro lado, se definen los conjuntos $D_k = \sum_{i=1}^{k} \langle g_i \rangle$, con $k \in \{1, 2, 3, \ldots\}$. Gracias a la propiedad (2.2), se infiere que el proceso termina cuando

$$D_k = G. \tag{2.4}$$

En este ejemplo, terminaría cuando $k = m$.
En resumen, la descomposición de cada componente primaria se basa en buscar los elementos g_i que cumplan la condición (2.3) e ir actualizando los conjuntos D_k, hasta que se verifique (2.4).
Además cada vez que se quiera encontrar un nuevo elemento g_i no se deben de tomar en cuenta los anteriores, ni los elementos que pertenecen a sus subgrupos cíclicos, entonces

$$g_i \in G \setminus D_{i-1} \qquad \forall\, i \in \{1, 2, \ldots, m\}. \tag{2.5}$$

Teniendo en cuenta la dinámica que existe detrás de la demostración del teorema 1.4.12 se procede a implantarla dentro de la función de descomposición para grupos U_n. A continuación se detalla su implementación.

Función de descomposición en grupos cíclicos para grupos U_n

El prototipo de la función es el siguiente

```
int* descomposicionUn(int n ,int & indice)
```

La función recibe un entero positivo n para devolver un arreglo dinámico que almacene los *órdenes* de los grupos cíclicos en los que se descompone el grupo U_n. El segundo parámetro es referencial de tipo entero con el objetivo de ir contando la cantidad de grupos cíclicos en los que se descompone el grupo.

El arreglo estático $grupos ciclicos$ será el que se retornará, el cual se inicia reservando 30 espacios de memoria. A continuación se verifica si n es primo y si lo es se finaliza la función con $indice = 1$ y $grupos ciclicos[0] = n - 1$, ya que si n es primo U_n es cíclico, por consecuencia del teorema de Fermat.

```
if(verificacion_primo(n))
{
    gruposciclicos[0]=n-1;
    indice=1;
}
```

Si n no es primo, comienza un proceso de obtención de componentes primarias del grupo U_n.
Se instancia el siguiente objeto

```
Un GRUPO=Un(n,1).G();
```

para trabajar con todos los elementos del grupo, en virtud del método $G(\,)$ de la clase Un.
Lo siguiente es utilizar la función $factores_primos$ para obtener un arreglo dinámico con los factores primos del orden del grupo y una variable de tipo entera cuyo valor es la cantidad de factores primos diferentes que hay.

```
int orden_grupo=GRUPO.obtener_orden();
int *factores_orden_grupo;
int numero_factores_orden_grupo;
```

```
factores_primos(orden_grupo,factores_orden_grupo,
                numero_factores_orden_grupo);
```

La variable $numero_factores_orden_grupo$ permite saber cuantas componentes primarias existen, por ello se crea un arreglo de objetos Un donde cada uno de sus elementos es una componente primaria. Más adelante se encontrarán y agregarán los elementos de cada componente, sin embargo al ser subgrupos se las instancia en principio como el conjunto unitario de la unidad

```
Un COMPONENTES[numero_factores_orden_grupo];
        for(int i=0;i<numero_factores_orden_grupo;i++)
            COMPONENTES[i]=Un(n,1);
```

Luego, se crea un arreglo dinámico $vector_ordenes$ con $n-1$ espacios de memoria reservados, donde se almacenarán los órdenes de cada elemento del grupo, con la ayuda del método $orden_elemento$ y el objeto *GRUPO* a fin de descomponer en factores primos a cada orden.

```
  int *vector_ordenes=new int [n-1];
        for(int i=0;i<n-1;i++)
            if(GRUPO.elementos[i])
            vector_ordenes[i]=Un(n,i+1).orden_elemento();
            else
                vector_ordenes[i]=0;
```

Ahora se procederá a asignar los elementos de cada componente primaria. Para ello, se crean un arreglo dinámico $factores_orden_elementos$ y una variable de tipo entero $cantidad_factores$ los cuales, por medio de la función $factores_primos$, se encargan de obtener los factores primos y la cantidad de los mismos para el orden de cada elemento del grupo. Los elementos para los cuales la variable $cantidad_factores$ tome el valor de 1, significará que su orden es potencia de un primo y por ende, son parte de alguna componente. Para estos elementos, se hace una búsqueda de la componente a la que pertenecen y se les añade a la misma. El proceso de búsqueda y asignación de los elementos de cada componente se lo realiza mediante un bucle for, como se observa a continuación.

```
for(int i=0;i<n-1;i++)
{
    if(vector_ordenes[i]!=0)//si es cero, i no es parte del grupo
    {
         buscador_componente=0;
         factores_primos(vector_ordenes[i],factores_orden_elementos,
         cantidad_factores);
        if( cantidad_factores==1)
        {
        while(factores_orden_grupo[buscador_componente]!=
```

```
        factores_orden_elementos[0])
            buscador_componente++;
        COMPONENTES[buscador_componente].agregar(i+1);
        }
    }
}
```

Ya disponiendo de las componentes primarias, se procede finalmente a descomponer cada una. Al analizar una componente, se verifica primero si su orden es primo, en ese caso, por el teorema 1.4.5, la componente ya sería cíclica, y no hay nada más que descomponer. En caso de que el orden no sea primo, se realiza lo siguiente. Se declaran cuatro nuevas variables de tipo entero, la primera se llama $OrdenComponente$ y almacena el orden de la componente en análisis, la segunda, g, y hace referencia a los elementos g_i de la propiedad 2.3, la tercera se referirá al orden del elemento g encontrado y la cuarta tiene el propósito de transformarse en los elementos del grupo cíclico generado por g.

```
int OrdenComponente,g,Orden_g,elemento_ciclico;
```

También se instancian dos objeto de la clase Un. El primero es $grupo_ciclico$ que como su nombre lo indica, visto como conjunto será el subgrupo cíclico de g, donde justamente va adquiriendo sus elementos con la ayuda de $elemento_ciclico$. El segundo objeto, D, es en representación a los conjuntos D_k del criterio de parada de (2.4), y se inicia siendo el conjunto unitario del neutro del grupo. En virtud de la propiedad (2.5), se deben eliminar todos los elementos de D de la componente para hacer una adecuada búsqueda de los g_i, por ello se realizan los siguientes pasos

```
for(int i=1;i<n-1;i++)
    if(D.elementos[i])//si pertenece a D, se elimina de la componente
    COMPONENTES[l].eliminar(i+1);
```

Se comienza la búsqueda de los g_i por medio de g. Una observación adicional es que g deberá satisfacer que su orden multiplicado por la cardinalidad de D no sea mayor al orden de la componente, caso contrario si se actualiza $D = \langle g \rangle + D$ (computacionalmente), se estaría violando la propiedad (2.3).

Entre los elementos que cumplen (2.3), se buscan los candidatos de g y se empieza por tomar los de orden máximo, si bien se mencionó que esa característica no garantiza que tales elementos sean alguno de los g_i buscados, tienen más potencial de generar que otros.

```
for(int i=1;i<n-1;i++){
    if(COMPONENTES[l].elementos[i]){
        if(vector_ordenes[i]*D.cardinalidad()<=OrdenComponente){
            if(vector_ordenes[i]>Orden_g){
                    Orden_g=vector_ordenes[i];
                    g=i+1;
                }
            }
        }
    }
```

Ya encontrado un nuevo g con las condiciones anteriores, se procede a crear su subgrupo cíclico respectivo para luego verificar que se trate en realidad de uno de los g_i.

```
elemento_ciclico=g;
grupo_ciclico=Un(n,1);
for(int i=2;i<=Orden_g;i++)
    {
        grupo_ciclico.agregar(elemento_ciclico);
        elemento_ciclico=(elemento_ciclico*g)%n;
    }
```

Por último, para actualizar D, se debe verificar que el elemento g cumpla la condición (2.3). Si cumple, g es uno de los g_i buscados y D se actualiza acorde a como se definieron los conjuntos D_k.

```
D=D.suma_conjuntos(D,grupo_ciclico);
```

Para alimentar el arreglo que retorna de la función, se debe añadir el orden de g a $gruposciclicos$

```
gruposciclicos[indice]=vector_ordenes[g-1];
indice++;
```

En caso que g no verifique (2.3), únicamente se lo descarta y se elimina de la componente para no volverlo a tener en cuenta. Este proceso se repite hasta que la cardinalidad de D sea igual al orden componente (condición equivalente a que sean iguales); y para todas las componentes. La implementación completa de esta función se encuentra en el anexo A.3.

2.0.4. Isomorfismos entre grupos no abelianos

La última función del modelo tiene el objetivo de hacer exploraciones con los grupos no abelianos implementados anteriormente, Q_8 y D_4. Específicamente la función determinará isomorfismos entre estos grupos. No cabe duda en el hecho que los grupos D_4 como simetrías y D_4 como

permutaciones son isomorfos, ya que simplemente son perspectivas diferentes del mismo grupo. También se puede verificar rápidamente que Q_8 y D_4 no son isomorfos, una razón se debe a que D_4 cuenta con cuatro elementos de orden 2 mientras que Q_8 tiene solo uno.
Dicho lo anterior, los resultados de la función serán bastante predecibles para nosotros, sin embargo, para la computadora, son en absoluto conocidos, por muy evidentes que sean, y lo interesante será observar la manera en que un computador puede llegar a obtener conclusiones sobre entidades abstractas mediante relacionar los recursos computacionales con conceptos, y teoremas con algoritmos.

Función isomorfismo

También se incluye un preámbulo de la implementación de esta función, explicando la estrategia que sigue y su base teórica. Es importante mencionar que la función está implementada particularmente para las tres clases $Q8$ $D4$ y $d4$ creadas anteriormente. Sin embargo si se llegara a implementar más clases en representación a otros grupos, el algoritmo de la función serviría todavía. La razón se debe a la forma en que la función determina si dos grupos son isomorfos, y no es más que utilizar el concepto.

Dos grupos son isomorfos cuando existe un homomorfismo biyectivo entre ellos. Con base a la definición, la función busca una biyección entre los grupos, que cumpla la propiedad de homomorfismo.

La cuestión es como trabajar con biyecciones entre los grupos y la respuesta es utilizar permutaciones. La idea es la siguiente: Suponer que G_1 y G_2 son dos grupos tales que $|G_1| = |G_2| = n$, que por extensión, son

$$G_1 = \{x_1, x_2, \ldots, x_n\}, \quad G_2 = \{y_1, y_2, \ldots, y_n\}.$$

Si $P \in S(n)$, entonces

$$P = \{p_1, p_2, \ldots, p_n\}, \quad p_i \in \{1, 2, .., n\}, \ \forall\, i \in \{1, 2, \ldots, n\}.$$

Se define la función $F : G_1 \to G_2$, tal que

$$F(x_i) = y_{p_i} \qquad \forall i \in \{1, 2, .., n\} \tag{2.6}$$

la cual es biyectiva.
De esa forma se pueden obtener todas las biyecciones entre G_1 y G_2 a través de permutaciones. Ahora para que alguna de estas biyecciones inducidas por permutaciones sea un homomorfismo, se debe verificar lo siguiente

$$\begin{aligned} &F(x_i x_j) = F(x_i)F(x_j) = y_{p_i} y_{p_j} \\ &\Rightarrow F(x_k) = y_{p_i} y_{p_j} \text{ con } \ x_k = x_i x_j \qquad \forall i \in \{1, 2, ..., n\} \end{aligned} \tag{2.7}$$

Para poder reflejar esta idea en el programa, serán de gran ayuda los arreglos externos $GrupoD4$, $Grupod4$ y $GrupoQ8$, pues equivalen a tener un orden sobre los elementos de los grupos, o en este caso, de las clases, tal como lo tienen G_1 y G_2. Se detalla el proceso de la función, a continuación.

Primero se debe mencionar que la función utiliza dos recursos de la biblioteca $algorithm$, los cuales son la función $sort$, para ordenar arreglos, y $next_permutation$ que sirve para obtener permutaciones de un arreglo.
El prototipo de la función es el siguiente

```
template <class P, class Q> bool isomorfismo (P grupo1, Q grupo2)
```

Los argumentos de la función serán objetos de las clases $D4$, $d4$ y $Q8$, lo que significa que cada parámetro puede ser de tres tipos distintos. Por tanto es necesario generalizar el tipo de cada parámetro y es aquí donde el uso de plantillas es primordial ya que hace posible esta tarea. Esa es la razón por la cual la función se encuentra declarada mediante plantillas con dos tipos de datos generalizados, uno para cada parámetro.
Luego se crean dos arreglos de ocho elementos que almacenarán el orden de cada elemento de los grupos a analizar. A estos arreglos se los ordena de menor a mayor, y si no son iguales implica que los grupos en análisis no son isomorfos, en caso de ser iguales, no se puede concluir nada aún y se deberá proceder a usar la idea antes descrita. Sin embargo este criterio es de utilidad sobre todo cuando se analicen isomorfismo entre de D_4 y Q_8, pues no será necesario pasar por permutaciones para con-

cluir que no son isomorfos. Como se detallará más adelante, en el ámbito computacional, las permutaciones representan un alto costo.

```
int ordenes1[8];
int ordenes2[8];
for(int i=0;i<8;i++)
{
    ordenes1[i]=grupo1.transmutar(i).orden_elemento();
    ordenes2[i]=grupo2.transmutar(i).orden_elemento();
}
sort(ordenes1,ordenes1+8);
sort(ordenes2,ordenes2+8);
```

Notar que el método transmutar cumple su objetivo de permitir acceder a todos los elementos de su clase a través de un solo objeto. También es importante notar que los métodos para cada tipo de dato coinciden, y es lo que permite usar plantillas sin ocasionar errores de correspondencia de tipos, más adelante.

Si los arreglos son iguales, se deben utilizar permutaciones: Para ello, primero se crean el arreglo que permutará a lo largo del programa para generar isomorfismos y algunas variables más.

```
int permutacion[8]={0,1,2,3,4,5,6,7};
int k,h;
h=1;
P producto1;
bool verificar;
```

La variable h tiene el objetivo de contar cuantas permutaciones se analizan hasta que la función obtenga una respuesta. La variable k y tiene la finalidad de encontrar posiciones de elementos en los arreglos externos $GrupoD4$, $grupod4$ y $GrupoQ8$, y trabajará en conjunto con la variable $producto1$.

Lo que sigue es buscar entre las permutaciones del arreglo $permutacion$ alguna que induzca un homomorfismo, con la ayuda de $next_permutation$, dentro de un lazo $do - while$.

```
   do{
      //codigo
    }while((next_permutation(permutacion,permutacion+8)));
```

Las siguientes lineas de código corresponden a la implantación de (2.7) dentro del lazo $do - while$:

```
verificar=true;
for(int i=0;i<8;i++){
    for(int j=0;j<8;j++){
        k=0;
        producto1 = (grupo1.transmutar(i)*grupo1.transmutar(j));
        while(grupo1.transmutar(k).nombre!=producto1.nombre)
```

```
                k++;
            verificar=  verificar&( (grupo2.transmutar(permutacion[i])*
            grupo2.transmutar(permutacion[j]))==
            grupo2.transmutar(permutacion[k]) );
            if(!verificar)
            break;
        }
        if(!verificar)
        break;
}
if(verificar){
            cout<<" cuentas permutaciones ? "<<h<<endl;
            return true;
        }
}
```

Las correspondencias entre 2.7 y las variables del código se exponen a continuación

$$x_i = producto1.transmutar(i),$$

$$x_j = producto1.transmutar(j)$$

$$x_k = grupo1.transmutar(k)$$

$$y_{p_i} = grupo2.transmutar(permutacion[i]),$$

$$y_{p_j} = grupo2.transmutar(permutacion[j])$$

Observar que la variable de tipo lógica $verificar$ es de gran utilidad pues precisamente se actualiza según la permutación cumpla o no la propiedad de homomorfismo para cada par de elementos del primer grupo. Si en algún punto se vuelve falsa, significa que dicha permutación no induce un homomorfismo, y por tanto no tiene sentido completar los pasos de los lazos for, de ese modo, si $verificar$ es falsa, se rompen los lazos mediante los dos primeros $break$ y se continúa con la siguiente permutación. Igualmente, si una permutación ya induce un homomorfismo, los grupos son isomorfos y ya no es necesario seguir analizando las permutaciones restantes, por lo tanto el lazo $do - while$ se rompería mediante el tercer $break$. Finalmente si $verificar$ sale del lazo $do - while$ con valor falso significa que ninguna permutación induce un homomorfismo, por tanto, los grupos en análisis no serían isomorfos. La implementación completa de la función se encuentra disponible en el anexo A.4.
Las clases $D4$, $d4$ y $Q8$ tienen una dependencia en los arreglos externos $GrupoD4$, $Grupod4$ y $GrupoQ8$ respectivamente. El orden de cómo se disponen los elementos de cada uno de los arreglos tiene consecuencias en

varios métodos de las clases, el más sencillo de percibir sería en las tablas de multiplicar, ya que este método imprime las multiplicaciones acorde al orden de los elementos en los arreglos externos. La consecuencia más importante que tienen, es en la función isomorfismo debido a que un cambio en el orden de los elementos de estos arreglos, implica que hubo una permutación, por tanto, la permutación que induzca el homomorfismo para un orden no va a ser la misma para un orden diferente. Es interesante de observar esta situación, pues de hecho ayuda a verificar que la función trabaja adecuadamente al encontrar diferentes permutaciones según el orden de los elementos de los arreglos.

Capítulo 3

Resultados, conclusiones y recomendaciones

3.1. Resultados

Con el propósito de interactuar con el modelo y obtener sus resultados más fácilmente, se ha creado una interfaz de usuario a través de una aplicación de consola, la cual se encuentra disponible, junto con todos los archivos necesarios para su compilación, en el siguiente hipervínculo.
A continuación, se presentan varias exploraciones sobre grupos finitos que han sido factibles gracias al modelo computacional.

3.1.1. Grupos U_n

Con el objetivo de visualizar los resultados que el modelo puede brindar referentes a los grupos U_n se ha realizado una exploración que aprovechó la mayoría de recursos disponibles para estos grupos.
Mediante un lazo for y la función $factores_primos$ se ha buscado un entero n, tal que el orden de U_n tenga al menos cinco factores primos, ya que de esa forma hay más posibilidades que U_n tenga varias componentes primarias y la visualización sea más llamativa.

```
U17162 con orden 8580 Factores primos del orden =2,3,5,11,13
U17554 con orden 8580 Factores primos del orden =2,3,5,11,13
U17558 con orden 8778 Factores primos del orden =2,3,7,11,19
```

Figura 3.1: Búsqueda de grupos con varias componentes primarias

Entre los enteros n tales que el orden de U_n cumplían tener un orden con al menos cinco factores primos, se escogió $n = 17554$. Inmediatamente se descompuso este grupo gracias a la función *descomposicionUn*, y se obtuvieron los siguientes resultados:

```
Grupo U17554 con orden 8580
COMPONENTES
COMPONENTE_2 orden=4:
1, 5763, 11791, 17553,
COMPONENTE_3 orden=3:
1, 10481, 15983,
COMPONENTE_5 orden=5:
1, 2547, 2681, 8175, 9783,
COMPONENTE_11 orden=11:
1, 263, 2359, 5241, 5503, 6027, 7075, 7861, 9171, 13625, 16507,
COMPONENTE_13 orden=13:
1, 2279, 4959, 8577, 9381, 10587, 10855, 13401, 13669, 14339, 14473, 15411, 16081,


U17554 '=' Z2+Z2+Z3+Z5+Z11+Z13
```

Se utilizó el método *subgrupo* con la información de la componente más grande, G_{13}, y se ha verificado que es en efecto un subgrupo tal como lo indica el teorema 1.4.6.

```
Ingrese los 13 elementos del conjunto
1
2279
4959
8577
9381
10587
10855
13401
13669
14339
14473
15411
16081

El conjunto si es un subgrupo de U17554
```

Otra razón de la elección de $n = 17554$ se debe a que es un número aleatorio y grande, y al haber obtenido resultados válidos, se ha evidenciado que el modelo trabaja correcta y adecuadamente para los diversos valores de n. Sin embargo por la gran cantidad de elementos que tiene este grupo, no se han utilizado algunos recursos más de Un, como por ejemplo visualizar la tabla de multiplicar, o sumar conjuntos. Por tal razón, se aplicaron estos métodos al grupo U_{22} para visualizar la manera en la que actúan.

```
Ingrese n para ver la tabla de Un:
22
 X|| 1  | 3  | 5  | 7  | 9  | 13 | 15 | 17 | 19 | 21 |
1  | 1  | 3  | 5  | 7  | 9  | 13 | 15 | 17 | 19 | 21 |
3  | 3  | 9  | 15 | 21 | 5  | 17 | 1  | 7  | 13 | 19 |
5  | 5  | 15 | 3  | 13 | 1  | 21 | 9  | 19 | 7  | 17 |
7  | 7  | 21 | 13 | 5  | 19 | 3  | 17 | 9  | 1  | 15 |
9  | 9  | 5  | 1  | 19 | 15 | 7  | 3  | 21 | 17 | 13 |
13 | 13 | 17 | 21 | 3  | 7  | 15 | 19 | 1  | 5  | 9  |
15 | 15 | 1  | 9  | 17 | 3  | 19 | 5  | 13 | 21 | 7  |
17 | 17 | 7  | 19 | 9  | 21 | 1  | 13 | 3  | 15 | 5  |
19 | 19 | 13 | 7  | 1  | 17 | 5  | 21 | 15 | 9  | 3  |
21 | 21 | 19 | 17 | 15 | 13 | 9  | 7  | 5  | 3  | 1  |
```

Figura 3.2: Tabla de multiplicar de U_{22} obtenida mediante el modelo.

```
Componente G2 = 1, 21,
Componente G5 = 1, 3, 5, 9, 15,
La suma de los conjuntos anteriores es
1, 3, 5, 7, 9, 13, 15, 17, 19, 21,
Si es todo el grupo U22
```

Figura 3.3: Suma de componentes primarias de U_{22}

Por último, se presenta una tabla con la descomposición primaria de 46 grupos U_n, donde el n para los 20 primeros grupos ha tiene un valor aleatorio entre 10 y 120, y los 26 grupos restantes tienen un valor de n entre 300 y 40000. E

Observación:

- Para todo entero positivo n, todos los grupos cíclicos de orden n son isomorfos entre sí, por tal motivo, toda descomposición cíclica puede expresarse mediante los grupos $\mathbb{Z}_n$.

- El algoritmo para la descomposición de grupos U_n es computacionalmente limitado. Después de realizar varias pruebas, se determinó que descomponer grupos U_n con n mayor a 40000 se vuelve una tarea inestable. Por ello, se escogieron valores aleatorios de n hasta 40000.

U_n	Orden	Descomposición	U_n	Orden	Descomposición
U_{109}	108	$\mathbb{Z}_{108}$	U_{30}	8	$\mathbb{Z}_2 + \mathbb{Z}_4$
U_{40}	16	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_2$	U_{74}	36	$\mathbb{Z}_4 + \mathbb{Z}_9$
U_{105}	48	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3$	U_{55}	40	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_5$
U_{14}	6	$\mathbb{Z}_2 + \mathbb{Z}_3$	U_{73}	72	$\mathbb{Z}_{72}$
U_{87}	56	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_7$	U_{116}	56	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_7$
U_{92}	44	$\mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_{11}$	U_{16}	8	$\mathbb{Z}_4 + \mathbb{Z}_2$
U_{10}	4	$\mathbb{Z}_4$	U_{31}	30	$\mathbb{Z}_{30}$
U_{51}	32	$\mathbb{Z}_{16} + \mathbb{Z}_2$	U_{55}	40	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_5$
U_{104}	48	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3$	U_{94}	46	$\mathbb{Z}_2 + \mathbb{Z}_{23}$
U_{22}	10	$\mathbb{Z}_2 + \mathbb{Z}_5$	U_{46}	22	$\mathbb{Z}_2 + \mathbb{Z}_{11}$
U_{7041}	4692	$\mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3 + \mathbb{Z}_{17} + \mathbb{Z}_{23}$	$U1286$	642	$\mathbb{Z}_2 + \mathbb{Z}_3 + \mathbb{Z}_{107}$
U_{647}	646	$\mathbb{Z}_{646} +$	U_{2954}	1260	$\mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3 + \mathbb{Z}_3 + \mathbb{Z}_5 + \mathbb{Z}_7$
U_{6682}	3120	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3 + \mathbb{Z}_5 + \mathbb{Z}_{13}$	U_{2961}	1656	$\mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3 + \mathbb{Z}_3 + \mathbb{Z}_{23}$
U_{7077}	4032	$\mathbb{Z}_{16} + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3 + \mathbb{Z}_3 + \mathbb{Z}_7$	U_{1346}	672	$\mathbb{Z}_{32} + \mathbb{Z}_3 + \mathbb{Z}_{27}$
U_{5914}	2956	$\mathbb{Z}_4 + \mathbb{Z}_{739}$	U_{7638}	2376	$\mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_9 + \mathbb{Z}_3 + \mathbb{Z}_{11}$
U_{1007}	936	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_9 + \mathbb{Z}_{13}$	U_{1357}	1276	$\mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_{11} + \mathbb{Z}_{29}$
U_{9502}	4750	$\mathbb{Z}_2 + \mathbb{Z}_{125} + \mathbb{Z}_{19}$	U_{6425}	5120	$\mathbb{Z}_{256} + \mathbb{Z}_4 + \mathbb{Z}_5$
U_{4960}	1920	$\mathbb{Z}_8 + \mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3 + \mathbb{Z}_5$	U_{9308}	4272	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3 + \mathbb{Z}_{89}$
U_{1553}	1552	$\mathbb{Z}_{1552}$	U_{2197}	2028	$\mathbb{Z}_4 + \mathbb{Z}_3 + \mathbb{Z}_{169}$
U_{1551}	920	$\mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_5 + \mathbb{Z}_{23}$	U_{9100}	2880	$\mathbb{Z}_4 + \mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3 + \mathbb{Z}_3 + \mathbb{Z}_5$
U_{31484}	14784	$\mathbb{Z}_{16} + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_3 + \mathbb{Z}_7 + \mathbb{Z}_{11}$	U_{36265}	29008	$\mathbb{Z}_4 + \mathbb{Z}_4 + \mathbb{Z}_{49} + \mathbb{Z}_{37}$
U_{32952}	10976	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_{343}$	U_{18552}	5616	$\mathbb{Z}_4 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_9 + \mathbb{Z}_3 + \mathbb{Z}_{13}$
U_{35017}	32832	$\mathbb{Z}_{32} + \mathbb{Z}_2 + \mathbb{Z}_9 + \mathbb{Z}_3 + \mathbb{Z}_{19}$	U_{13812}	4600	$\mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_2 + \mathbb{Z}_{25} + \mathbb{Z}_{23}$

Cuadro 3.1: Descomposición de varios grupos U_n.

Grupos $D4$ **y** $Q8$

Gracias a los métodos *inverso* y *orden_elemento* se obtuvieron el inverso y orden de cada elemento de lo grupos $D4$ y $Q8$. Los resultados se resumen en la siguiente tabla:

D_4			Q_8		
Elemento	Orden	Inverso	Elemento	Orden	Inverso
id	1	id	1	1	1
r_{90}	4	r_{270}	-1	2	-1
r_{180}	2	r_{180}	i	4	$-i$
r_{270}	4	r_{90}	$-i$	4	i
v	2	v	j	4	$-j$
h	2	h	$-j$	4	j
d_1	2	$d1$	k	4	$-k$
d_2	2	$d2$	$-k$	4	k

Cuadro 3.2: Ordenes e inversos de los elementos de D_4 y Q_8

Además gracias al método *tabla* se obtuvieron las tablas de multiplicar de ambos grupos.

$\bullet$	id	r_{90}	r_{180}	r_{270}	v	h	d_1	d_2
id	id	r_{90}	r_{180}	r_{270}	v	h	d_1	d_2
r_{90}	r_{90}	r_{180}	r_{270}	id	d_2	d_1	v	h
r_{180}	r_{180}	r_{270}	id	r_{90}	h	v	d_2	d_1
r_{270}	r_{270}	id	r_{90}	r_{180}	d_1	d_2	h	v
v	v	d_1	h	d_2	id	r_{90}	r_{270}	r_{180}
h	h	d_2	v	d_1	r_{180}	id	r_{270}	r_{90}
d_1	d_1	h	d_2	v	r_{90}	r_{270}	id	r_{180}
d_2	d_2	v	d_1	h	r_{90}	r_{270}	r_{180}	id

Cuadro 3.3: Tabla de multiplicar grupo D_4

$\bullet$	1	-1	i	$-i$	j	$-j$	k	$-k$
1	1	-1	i	$-i$	j	$-j$	k	$-k$
-1	-1	1	$-i$	i	$-j$	j	$-k$	k
i	i	$-i$	-1	1	k	$-k$	$-j$	j
$-i$	$-i$	i	1	-1	$-k$	k	j	$-j$
j	j	$-j$	$-k$	k	-1	1	i	$-i$
$-j$	$-j$	j	k	$-k$	1	-1	$-i$	i
k	k	$-k$	j	$-j$	$-i$	i	1	-1
$-k$	$-k$	k	$-j$	j	i	$-i$	1	-1

Cuadro 3.4: Tabla de multiplicar grupo Q_8

La información de la tabla 3.2 permite inferir que los conjuntos $\{id, r_{90}, r_{180}, r_{270}\}$, $\{id, r_{90}\}$, $\{id, v\}$, $\{id, h\}$, $\{id, d_1\}$ y $\{id, d_2\}$ son subgrupos, pero no se tiene conocimiento de si son normales. Para resolver esta inquietud, se utilizará el métodos *subgrupo_normal* de la clase *d*4. Las respuestas que arroja el modelo son:

```
El conjunto {id r180 r90 r270 } Si es un subgrupo normal de d4
El conjunto {id v } no es un subgrupo normal
El conjunto {id h } no es un subgrupo normal
El conjunto {id d1 } no es un subgrupo normal
El conjunto {id d2 } no es un subgrupo normal
El conjunto {id r180 } Si es un subgrupo normal de d4
```

Figura 3.4: Verificación de subgrupos normales de D_4

Para finalizar con los resultados de grupos no abelianos se visualizarán varias pruebas de isomorfismo entre $D4$, $d4$ y $Q8$.
Se comienza por verificar mediante el modelo que efectivamente D_4 y Q_8 no son isomorfos

```
Escoja el grupo 1
1) D4
2) d4
3) Q8
1

Escoja el grupo 2
1) D4
2) d4
3) Q8
3
Verificacion de isomorfismo entre D4 y Q8
Los grupos no son isomorfos
```

Figura 3.5: D_4 y Q_8 no son isomorfos

Por el hecho que la función $isomorfismo$ obtiene resultados según varíe el orden de los elementos de los arreglos $GrupoD4$, $Grupod4$ y $GrupoQ8$. Se ha realizado pruebas de isomorfismos para $D4$ y $d4$ con varios cambios de orden en los arreglos $GrupoD4$ y $Grupod4$. Los resultados se exponen a continuación:

```
string GrupoD4[8]={"id","f4","f3","r1","f2","r3","r2","f1"};
fd4 Grupod4[8]={id,r270,r90,v,d1,d2,h,r180};
```

```
Isomorfismo entre d4 y D4
 cuentas permutaciones ? 1811
Isomorfismo a traves de la permutacion: {0, 3, 5, 1, 4, 7, 2, 6}
```

```
string GrupoD4[8]={"id","f1","f2","r2","f4","r3","r1","f3"};
fd4 Grupod4[8]={id,r180,v,r90,d2,d1,h,r270};
```

```
Isomorfismo entre D4 y d4
 cuentas permutaciones ? 1211
Isomorfismo a traves de la permutacion: {0, 2, 6, 1, 4, 7, 3, 5}
```

```
string GrupoD4[8]={"id","r1","r3","f4","f1","r2","f2","f3"};
fd4 Grupod4[8]={id,r180,v,r90,d2,d1,h,r270};
```

```
Isomorfismo entre d4 y D4
 cuentas permutaciones ? 3136
Isomorfismo a traves de la permutacion: {0, 5, 3, 1, 6, 4, 7, 2}
```

La linea *cuantas permutaciones* se refiere a el número de permutaciones que se evaluaron antes de encontrar la permutación que induce el isomorfismo, por lo tanto el número que se refleja puede ser hasta $8!$ que

corresponde al número de permutaciones de 8 elementos que existen.

3.2. Conclusiones y recomendaciones

3.2.1. Conclusiones

1. Los objetivos planteados se han cumplido y el resultado obtenido es la interfaz en base al modelo computacional implementado. El modelo permite obtener la descomposición cíclica de los grupos U_n, verificar isomorfismos entre Q_8 y D_4, visualizar tablas de multiplicación, conocer los órdenes e inversos de sus elementos y verificar si subconjuntos particulares son subgrupos o subgrupos normales, para las tres estructuras.

2. Al obtener la descomposición cíclica de los grupos U_n mediante el modelo computacional, toda pregunta referente a U_n se reduce a una pregunta acerca de grupos cíclicos que serán más sencillas de responder o comprender, gracias a las propiedades básicas de estos grupos.

3. El modelo permite visualizar dos niveles de correspondencias. El más evidente, son los isomorfismos entre grupos. El otro nivel es corresponder los conceptos u objetos abstractos con algoritmos y recursos computacionales. En conclusión, concretar entidades abstractas conlleva una estructura compleja de varios niveles y recursos para su realización.

3.2.2. Recomendaciones

1. La función para descomponer en grupos cíclicos a los grupos U_n tiene ciertas limitaciones. Una de las más importantes es con respecto al arreglo $elementos$ cuya asignación de memoria se realiza en función lineal de n. Si bien esto evita que se reasigne espacio, lo cual implica un alto costo computacional, también desperdicia recursos debido a que el orden del grupo suele ser mucho menor a n. Por tal

razón se recomienda indagar nuevas estrategias que permitan trabajar con subconjuntos de U_n tal como se lo ha logrado a traves de $elementos$.

Otra limitación es referente a los función para verificar isomorfismos entre los grupos no abelianos implementados. La función trabaja con permutaciones, que implican altos costos computacionales por su complejidad algorítmica. Se recomienda agregar más criterios, para que la función resulte más eficiente y deba de pasar por permutaciones en un menor número de ocasiones.

2. Se recomienda para futuros trabajos implementar grupos finitamente generados y su respectivo teorema fundamental, con el fin de lograr una descomposición cíclica de esta clase de estructuras también.

3. Los métodos que trabajan con el concepto de subgrupo y subgrupo normal, cumplen simplemente responder si un subconjunto que especifique el usuario, tiene tales cualidades. Sin embargo para los grupos Q_8 y D_4 sería factible implementar una función que trabaje con permutaciones para poder encontrar todos los subgrupos y subgrupos normales que existan.

4. La programación genérica ha sido el recurso que ha permitido el desarrollo de este modelo computacional, pero existen muchos más enfoques a diversas áreas de la matemática abstracta que se han pretendido estudiar y trabajar por medio de la programación genérica. Uno de ellos, es la implementación de conceptos por medio de *templates*. Se recomienda seguir la bibliografía [2] para tener una mejor comprensión de este tema.

Capítulo A

Anexos

Código A.1: Archivo abelianos.h

```
#ifndef ABELIANOS_H_INCLUDED
#define ABELIANOS_H_INCLUDED
#include <iostream>
#include<cmath>
using namespace std;
int mcd(int, int );
bool verificacion_primo(int);
void factores_primos(int, int & ,int &);
int mcd(int a, int b)
{
        if(b==0)
        return(a);
        return mcd(b,a%b);
}
bool verificacion_primo(int p)
{
    if((p!=1)&&(p%2!=0))
    {
        for(int i=3;i<=sqrt(p);i=i+2)
            if(p%i==0)
                return false;
    }
    else
        return false;
    return true;
}

void factores_primos(int n,int* &k,int &t)
{
    k=new int[15];
    t=1;
    if(n%2==0)
    {
        k[t-1]=2;
        t++;
    }
    while (n % 2 == 0)
        n = n/2;
    for (int i = 3; i <= sqrt(n); i = i + 2)
```

```
40      {
41          if(n%i==0)
42          {
43              k[t-1]=i;
44              t++;
45          }
46          while (n % i == 0)
47              n = n/i;
48      }
49      if (n > 2)
50          k[t-1]=n;
51      else
52          t--;
53  }
54
55  class Un
56  {
57  public:
58      Un(){};
59      Un(int,int);
60      int n;
61      int p;
62      bool *elementos;
63      void agregar(int);
64      void eliminar(int );
65      Un operator +(const Un&)const;
66      bool operator ==(const Un&)const;
67      int obtener_orden();
68      Un inverso();
69      int cardinalidad ();
70      int orden_elemento();
71      bool subgrupo();
72      void tabla();
73      Un G();
74      friend std::ostream& operator<<(std::ostream& os,const Un&);
75          Un suma_conjuntos(Un,Un);
76
77  };
78
79  int Un:: cardinalidad()
80  {
81      int x=0;
82      for(int i=0;i<n-1;i++)
83          if(elementos[i])
84              x++;
85
86      return x;
87  }
88  void Un::eliminar(int m)
89  {
90      if(mcd(m,n)==1)
91          elementos[(m%n)-1]=false;
92      else
93          cout<<"No es un elemento del grupo"<<endl;
94  }
95  Un Un::suma_conjuntos(Un a, Un b)
96  {
97      Un c;
98      if(a.n==b.n)
99      {
100         c=Un(a.n,1);
101         c.eliminar(1);
102        for(int j=0;j<a.n-1;j++)
103         if(a.elementos[j])
104             for(int k=0;k<a.n-1;k++)
```

```
105                 if(b.elementos[k])
106                     c.agregar((j+1)*(k+1));
107     }
108     else
109     {
110         cout<<"No son subconjuntos de un mismo grupo"<<endl;
111         return Un(1,1);
112     }
113 return c;
114 }
115 Un Un:: G()
116 {
117         Un b=Un(n,1);
118         for(int i=2;i<n;i++ )
119                 if(mcd(i,n)==1 )
120                         b.agregar(i);
121         return b;
122 }
123
124 void Un::tabla()
125 {
126     bool*A=new bool[n-1];
127     A=G().elementos;
128
129         cout.setf(ios::left);
130     cout.width(2);
131         cout<<" X|| ";
132         for(int k=0;k<n-1;k++)
133         {
134                 if(A[k])
135                 {
136                         cout.setf(ios::left);
137             cout.width(2);
138                         cout<<k+1<<" | ";
139                 }
140         }
141         cout<<endl;
142         for(int i=0;i<n-1;i++)
143         {
144                 if(A[i])
145                 {
146                         cout.setf(ios::left);
147             cout.width(2);
148                         cout<<i+1<<" | ";
149                         for(int j=0;j<n-1;j++)
150                         {
151                                 if(A[j])
152                                 {
153                                         cout.setf(ios::left);
154                         cout.width(2);
155                         cout<<(i+1)*(j+1)%n<<" | ";
156                                 }
157                         }
158                         cout<<endl;
159                 }
160         }
161 }
162
163 int Un::orden_elemento()
164 {
165         int x=p;
166         int j=1;
167         while(x!=1)
168         {
169                 x=(x*p)%n;
```

```
170                     j++;
171           }
172           return j;
173 }
174
175 bool Un::subgrupo()
176 {
177      if(elementos[0])
178      {
179           for(int i=1;i<n-1;i++)
180           {
181                     if(elementos[i])
182                     {
183                               for(int j=i;j<n-1;j++)
184                               {
185                                         if(elementos[j])
186                                                   if(!elementos[(i+1)*(j+1)%n-1])
187                                                             return false;
188                                         }
189                               }
190                     }
191      }
192      else
193           return false;
194      return true;
195 }
196
197 std::ostream& operator<<(std::ostream& os, const Un &q)
198 {
199      os<<q.p;
200      return os;
201 }
202
203 Un Un::inverso()
204 {
205      int k=1;
206      while(k*p%n!=1)
207           k=k*p%n;
208           return Un(n,k);
209 }
210 void Un::agregar(int m)
211 {
212      if(mcd(m,n)==1)
213            elementos[(m%n)-1]=true;
214      else
215           cout<<"No es elemento del grupo"<<endl;
216 }
217 int Un::obtener_orden()
218 {
219      int x=1;
220      for(int i=2;i<=n-1;i++)
221      {
222           if(mcd(i,n)==1)
223                x++;
224      }
225      return x;
226 }
227
228 bool Un::operator ==(const Un& a)const
229 {
230      if(n==a.n)
231           return(p==a.p);
232      else
233      {
234           cout<<"No son elementos de un mismo grupo"<<endl;
```

```
            return false;
        }
}
Un Un::operator +(const Un& a)const
{
    if(n==a.n)
        return Un(n,(p*a.p)%n);
    else
    {
        cout<<"No son elementos de un mismo grupo"<<endl;
        return Un(1,1);
    }
}

Un::Un(int _n,int _p)
{
    n=_n;
    elementos=new bool [n-1];
    if(mcd(_p,n)==1)
    {
        p=_p%n;
        for(int i=0;i<n-1;i++)
            elementos[i]=((i+1)==p);
    }
    else
        cout<<"No es un elemento del grupo"<<endl;
}
#endif // ABELIANOS_H_INCLUDED
```

Código A.2: Archivo no abelianos.h

```
#ifndef NO_ABELIANOS_H_INCLUDED
#define NO_ABELIANOS_H_INCLUDED
#include<algorithm>
using namespace std;
typedef int (*fd4)(int);
int id(int);
int r90(int);
int r180(int);
int r270(int);
int h(int);
int v(int);
int d1(int);
int d2(int);
fd4 Grupod4[8]={id,d2,d1,h,v,r180,r90,r270};
int id(int x)
{
    return x;
}
int r90(int x)
{
    if (x==3)
        return 4;
    else
    return(x+1)%4;
}
int r180(int x)
{
    return r90(r90(x));
}
int r270(int x)
{
    return r90(r180(x));
}
```

```
34 int h(int x)
35 {
36     if (x==1)
37         return 4;
38     else
39         return 4-(x-1)%4;
40 }
41 int v(int x)
42 {
43     return(r180(h(x)));
44 }
45 int d1(int x)
46 {
47     return(r270(h(x)));
48 }
49 int d2(int x)
50 {
51     return(r90(h(x)));
52 }
53
54 ///CLASE D4 simetrias
55 class d4
56 {
57 public:
58     d4(){};
59         d4 (fd4);
60         fd4 funcion;
61         d4 operator*(const d4&) const;
62     bool operator==(const d4&) const;
63     void tabla() ;
64     bool *elementos;
65     void agregar(fd4);
66     void eliminar(fd4);
67     bool subgrupo();
68     bool pertenece(d4);
69     bool subgrupo_normal();
70     d4 G();
71     d4 transmutar (int);
72     int orden_elemento();
73     d4 inverso ();
74     friend std::ostream& operator<<(std::ostream& os,const d4 & );
75     string nombre;
76 };
77
78 d4 d4::transmutar(int x)
79 {
80     return d4(Grupod4[x]);
81 }
82
83 d4 d4::G()
84 {
85     d4 grupo=d4(id);
86     for(int i =0;i<8;i++)
87         grupo.elementos[i]=true;
88
89 return grupo;
90 }
91
92
93 int d4::orden_elemento()
94 {
95 int k=1;
96 d4 T=d4(id);
97 while(!((T*d4(funcion))==d4(id)))
98 {
```

```
99      T=T*d4(funcion);
100     k++;
101 }
102 return k;
103 }
104
105
106
107 bool d4::subgrupo_normal()
108 {
109 if(subgrupo())
110     {
111         for(int i =1;i<8;i++)
112         if(elementos[i])
113           for(int j=1;j<8;j++)
114              if( !pertenece((d4(Grupod4[j]).inverso()*d4(Grupod4[i]))
115                                   *d4(Grupod4[j])))
116                      return false;
117     }
118 else
119     return false;
120
121 return true;
122 }
123
124 d4 d4::inverso(){
125 for(int i=0;i<8;i++)
126     if( d4(funcion)*d4(Grupod4[i])==d4(id) )
127         return d4(Grupod4[i]);
128 }
129 bool d4::pertenece(d4 x)
130 {
131     for(int i=0;i<8;i++)
132         if(elementos[i])
133             if(d4(Grupod4[i])==x)
134                 return true;
135     return false;
136 }
137
138 bool d4::subgrupo()
139 {
140     if(elementos[0])
141     {
142          for(int i=1;i<8;i++)
143          {
144              if(elementos[i])
145             {
146                 for(int j=1;j<8;j++)
147                 {
148                     if(elementos[j])
149                     {
150                         if(!( pertenece((d4(Grupod4[i])*
151                                          d4(Grupod4[j])).funcion) ))
152                             return false;
153                     }
154                 }
155             }
156          }
157     }
158     else
159         return false;
160
161 return true;
162 }
163
```

```
164
165
166 void d4::agregar(fd4 f1)
167 {
168     int x=0;
169     while( !((f1(1)==Grupod4[x](1))&    (f1(2)==Grupod4[x](2))) )
170         x++;
171
172     elementos[x]=true;
173 }
174
175
176 void d4::eliminar(fd4 f1)
177 {
178     int x=0;
179     while( !((f1(1)==Grupod4[x](1))&    (f1(2)==Grupod4[x](2))) )
180         x++;
181
182     elementos[x]=false;
183 }
184
185
186
187 d4::d4 (fd4 f)
188 {
189  elementos = new bool [8];
190     for(int i=0;i<8;i++ )
191         elementos [i]=0;
192
193     if(f==id)
194         nombre="id";
195     else if(f==r90)
196         nombre="r90";
197     else if(f==r180)
198         nombre="r180";
199     else if(f==r270)
200         nombre="r270";
201     else if(f==h)
202         nombre="h";
203     else if(f==v)
204         nombre="v";
205     else if(f==d1)
206         nombre="d1";
207     else if(f==d2)
208         nombre="d2";
209
210     int x=0;
211     while(!((f(1)==Grupod4[x](1))&(f(2)==Grupod4[x](2))))
212         x++;
213
214     elementos[x]=true;
215     funcion=f;
216 }
217
218 d4 d4::operator*(const d4 &f) const
219 {
220    int j=0;
221    while (!((f.funcion(funcion(1))==Grupod4[j](1)) &&
222              (f.funcion(funcion(2))==Grupod4[j](2))))
223         j++;
224    return d4(Grupod4[j]);
225 }
226
227
228 bool d4::operator==(const d4& f) const
```

```
229 {
230     return((funcion(1)==f.funcion(1))&&
231            funcion(2)==f.funcion(2));
232 }
233
234 void d4::tabla()
235 {
236  for(int i=0;i<8;i++)
237  {
238 cout<<"--------------------------------------------------------"<<endl;
239      for(int j=0;j<8;j++)
240      {
241          cout.setf(ios::left);
242          cout.width(4);
243          cout<<(d4(Grupod4[i])*d4(Grupod4[j]))<<" | ";
244      }
245      cout<<endl;
246  }
247 cout<<"--------------------------------------------------------"<<endl;
248 }
249 std::ostream& operator<<(std::ostream& os, const d4 &f)
250     {
251         os<<f.nombre;
252         return os;
253     }
254
255 int R1[4]={2,3,4,1};
256 int F4[4]={1,4,3,2};
257 string GrupoD4[8]={"id","r1","r2","r3","f1","f2","f3","f4"};
258
259 class D4
260 {
261 public:
262     D4(){};
263     string nombre;
264     int a[4];
265     D4(int [4]);
266     D4(string);
267     bool *elementos;
268     void agregar(string);
269     void eliminar(string);
270     D4 operator*(const D4&)const;
271     bool operator==(const D4&)const;
272     D4 inverso();
273     int orden_elemento();
274     bool subgrupo();
275     bool subgrupo_normal();
276     void tabla();//LISTO
277     void mostrar_funcion();
278     D4 transmutar(int);
279     bool pertenece(string);
280     bool pertenece(D4);
281
282    friend std::ostream& operator<<(std::ostream& os,const D4&);
283 };
284 D4 D4::transmutar(int x)
285 {
286     return D4(GrupoD4[x]);
287 }
288
289
290 int D4::orden_elemento()
291 {
292     int k=1;
293     D4 T=D4("id");
```

```
294      while( !( (T*D4(nombre))==D4("id") )  )
295      {
296          T=T*D4(nombre);
297          k++;
298      }
299
300      return k;
301  }
302
303  bool D4::pertenece(D4 x)
304  {
305      for(int i=0;i<8;i++)
306          if(elementos[i])
307              if(D4(GrupoD4[i])==x)
308                  return true;
309      return false;
310  }
311  bool D4::pertenece(string x)
312  {
313      for(int i=0;i<8;i++)
314          if(elementos[i])
315              if(GrupoD4[i]==x)
316                  return true;
317      return false;
318  }
319
320
321  D4 D4::inverso()
322  {
323      for(int i=0;i<8;i++)
324          if( D4(nombre)*D4(GrupoD4[i])== D4("id") )
325            return D4(GrupoD4[i]);
326  }
327
328  bool D4::subgrupo_normal()
329  {
330      if(subgrupo())
331          {
332              for(int i =1;i<8;i++)
333              if(elementos[i])
334                  for(int j=1;j<8;j++)
335                      if(!pertenece((D4(GrupoD4[j]).inverso()*
336                      D4(GrupoD4[i])*D4(GrupoD4[j])).nombre ))
337                          return false;
338          }
339      else
340          return false;
341
342  return true;
343  }
344
345
346  bool D4::subgrupo()
347  {
348      int x;
349      if(elementos[0])
350      {
351          for(int i=1;i<8;i++)
352          {
353              if(elementos[i])
354              {
355                  for(int j=1;j<8;j++)
356                  {
357                      if(elementos[j])
358                      {
```

```
359                             x=0;
360                             while(GrupoD4[x]!=(D4(GrupoD4[i])*
361                                 D4(GrupoD4[j])).nombre)
362                                 x++;
363                             if(!elementos[x])
364                                 return false;
365                         }
366                     }
367                 }
368             }
369     }
370     else
371         return false;
372 
373 return true;
374 }
375 std::ostream& operator<<(std::ostream& os, const D4 &a)
376 {
377     os<<a.nombre;
378     return os;
379 }
380 
381 
382 void D4::agregar(string t)
383 {
384    int i=0;
385     while(GrupoD4[i]!=t)
386         i++;
387 
388         elementos[i]=true;
389 
390 }
391 
392 void D4::eliminar(string t)
393 {
394     for(int i=0;i<8;i++)
395     if(GrupoD4[i]==t)
396         elementos[i]=false;
397 }
398 
399 D4::D4(int c[4])
400 {
401 elementos=new bool[8];
402 
403     if((c[0]==1)&&(c[1]==2))
404         nombre="id";
405     else if((c[0]==2)&&(c[1]==3))
406         nombre="r1";
407     else if((c[0]==3)&&(c[1]==4))
408         nombre="r2";
409     else if((c[0]==4)&&(c[1]==1))
410         nombre="r3";
411     else if((c[0]==1)&&(c[1]==4))
412         nombre="f4";
413     else if((c[0]==2)&&(c[1]==1))
414         nombre="f1";
415     else if((c[0]==3)&&(c[1]==2))
416         nombre="f3";
417     else if((c[0]==4)&&(c[1]==3))
418             nombre="f2";
419     for(int i=0;i<=3;i++)
420         a[i]=c[i];
421     for(int i=0;i<8;i++)
422             elementos[i]=(nombre==GrupoD4[i]);
423 }
```

```
424
425 D4::D4(string nnombre)
426 {
427 elementos =new bool[8];
428 for(int i=0;i<8;i++)
429     elementos[i]=(nnombre==GrupoD4[i]);
430         if(nnombre=="id")
431             for(int i=0;i<=3;i++)
432             a[i]=i+1;
433         else if(nnombre=="r1")
434             for(int i=0;i<=3;i++)
435             a[i]=R1[i];
436         else if(nnombre=="r2")
437             for(int i=0;i<=3;i++)
438             a[i]=R1[R1[i]-1];
439         else if(nnombre=="r3")
440             for(int i=0;i<=3;i++)
441             a[i]=R1[R1[R1[i]-1]-1];
442         else if(nnombre=="f2")
443             for(int i=0;i<=3;i++)
444             a[i]=F4[R1[i]-1];
445         else if(nnombre=="f3")
446             for(int i=0;i<=3;i++)
447             a[i]=F4[R1[R1[i]-1]-1];
448         else if(nnombre=="f1")
449             for(int i=0;i<=3;i++)
450             a[i]=F4[R1[R1[R1[i]-1]-1]-1];
451
452         else if(nnombre=="f4")
453             for(int i=0;i<=3;i++)
454             a[i]=F4[i];
455
456     nombre=nnombre;
457
458 }
459
460 D4 D4:: operator*(const D4 &x)const
461 {
462     int z[4];
463     for(int i=0;i<=3;i++)
464         z[i]=x.a[a[i]-1];
465     return D4(z);
466 }
467 bool D4:: operator==(const D4 &c)const
468 {
469         return (nombre==c.nombre);
470 }
471
472 void D4::mostrar_funcion()
473 {
474     cout<<"la funcion  "<<nombre<<"  es"<<endl;
475     for (int i=0;i<=3;i++)
476         cout<<i+1<<"  -->  "<<a[i]<<endl;
477 }
478 void D4::tabla()
479 {
480     for(int i=0;i<8;i++)
481     {
482 cout<<"--------------------------------------------------------"<<endl;
483         for(int j=0;j<8;j++)
484         {
485             cout.setf(ios::left);
486             cout.width(4);
487             cout<<(D4(GrupoD4[i])*D4(GrupoD4[j]))<<" | ";
488         }
```

```
489            cout<<endl;
490        }
491    cout<<"---------------------------------------------------------"<<endl;
492    }
493
494    /// CUATERNIONES
495    string GrupoQ8[8]={"1","-1","i","-i","j","-j","k","-k"};
496    class Q8
497    {
498    public:
499        Q8(){};
500        Q8(string);
501        Q8(int,int,int,int);
502        string nombre;
503        bool operator == (const Q8&)const;
504        Q8 operator *(const Q8&)const;
505        void tabla();
506        int a,b,c,d;
507        int orden_elemento();
508        Q8 inverso();
509        Q8 transmutar(int);
510        friend std::ostream& operator<<(std::ostream& os, const Q8 & );
511    };
512
513    Q8 Q8::inverso()
514    {
515        Q8 uno=Q8("1");
516        for(int i=0;i<8;i++)
517        {
518            if(uno*Q8(nombre)==Q8("1"))
519                return uno;
520            else
521               uno=uno*Q8(nombre);
522        }
523    }
524
525    Q8 Q8::transmutar (int z){
526    return Q8(GrupoQ8[z]);
527    }
528
529    int Q8::orden_elemento(){
530    int x=1;
531    Q8 q=Q8("1");
532    while(!((Q8(nombre)*q)==Q8("1")))
533    {
534        q=Q8(nombre)*q;
535        x++;
536    }
537    return x;
538    }
539    Q8::Q8(int aa,int bb,int cc,int dd){
540        a=aa;
541        b=bb;
542        c=cc;
543        d=dd;
544        if(a==1)
545        nombre="1";
546        else if (a==-1)
547        nombre="-1";
548        else if (b==1)
549        nombre="i";
550        else if (b==-1)
551        nombre="-i";
552        else if (c==1)
553        nombre="j";
```

```
554         else if (c==-1)
555         nombre="-j";
556         else if (d==1)
557         nombre="k";
558         else if (d==-1)
559         nombre="-k";
560     }
561     Q8::Q8(string g)
562     {
563         nombre=g;
564         a=0;
565         b=0;
566         c=0;
567         d=0;
568         if(g=="1")
569         a=1;
570         else if(g=="-1")
571         a=-1;
572         else if(g=="i")
573         b=1;
574         else if(g=="-i")
575         b=-1;
576         else if(g=="j")
577         c=1;
578         else if(g=="-j")
579         c=-1;
580         else if(g=="k")
581         d=1;
582         else if(g=="-k")
583         d=-1;
584     }
585     Q8 Q8:: operator * (const Q8 &x)const
586     {
587         return Q8(
588         (a*x.a)-(b*x.b)-(c*x.c)-(d*x.d),
589         (a*x.b)+(b*x.a)+(c*x.d)-(d*x.c),
590         (a*x.c)-(b*x.d)+(c*x.a)+(d*x.b),
591         (a*x.d)+(b*x.c)-(c*x.b)+(d*x.a) );
592     }
593     bool Q8:: operator ==(const Q8& x)const
594     {
595         return ((a==x.a)&(b==x.b)&(c==x.c)&(d==x.d));
596     }
597
598     std::ostream& operator<<(std::ostream& os, const Q8 &x)
599         {
600             os<<x.nombre;
601             return os;
602         }
603
604     void Q8::tabla()
605     {
606         for(int i=0;i<8;i++)
607         {
608     cout<<"-------------------------------------------------------"<<endl;
609             for(int j=0;j<8;j++)
610             {
611                 cout.setf(ios::left);
612                 cout.width(4);
613                 cout<< Q8(GrupoQ8[i])*Q8(GrupoQ8[j]) <<" | ";
614             }
615             cout<<endl;
616         }
617     cout<<"-------------------------------------------------------"<<endl;
618     }
```

```
619
620 #endif // NO_ABELIANOS_H_INCLUDED
```

Código A.3: Archivo descomposicion Un.h

```
1  #ifndef DESCOMPOSICION_UN_H_INCLUDED
2  #define DESCOMPOSICION_UN_H_INCLUDED
3  #include "abelianos.h"
4  int* descomposicionUn(int ,int &);
5  int* descomposicionUn(int n, int &indice)
6  {
7      cout<<"\nGrupo U"<<n<<" con orden "<<
8      Un(n,1).obtener_orden()<<endl;
9      indice=0;
10     static int gruposciclicos[30];
11     if(verificacion_primo(n))
12     {
13         cout<<"U"<<n<<" es ciclico de orden "<<n-1<<endl;
14         gruposciclicos[indice]=n-1;
15         indice++;
16     }
17     else
18     {
19         Un GRUPO=Un(n,1).G();
20         int orden_grupo=GRUPO.obtener_orden();
21         int *factores_orden_grupo;
22         int numero_factores_orden_grupo;
23         factores_primos(orden_grupo,factores_orden_grupo,
24                         numero_factores_orden_grupo);
25
26         Un COMPONENTES[numero_factores_orden_grupo];
27         for(int i=0;i<numero_factores_orden_grupo;i++)
28             COMPONENTES[i]=Un(n,1);
29         int *vector_ordenes=new int [n-1];
30         for(int i=0;i<n-1;i++)
31             if(GRUPO.elementos[i])
32                 vector_ordenes[i]=Un(n,i+1).orden_elemento();
33             else
34                 vector_ordenes[i]=0;
35
36         int *factores_orden_elementos;
37         int cantidad_factores;
38         int buscador_componente;
39         for(int i=0;i<n-1;i++)
40         {
41             if(vector_ordenes[i]!=0)
42             {
43                 buscador_componente=0;
44                 factores_primos(vector_ordenes[i],
45                                 factores_orden_elementos,
46                                 cantidad_factores);
47                 if( cantidad_factores==1)
48                 {
49                     while(factores_orden_grupo[buscador_componente]!=
50                           factores_orden_elementos[0])
51                     {buscador_componente++;}
52                     COMPONENTES[buscador_componente].agregar(i+1);
53                 }
54             }
55         }
56         cout<<"COMPONENTES PRIMARIAS"<<endl;
57         for(int y=0;y<numero_factores_orden_grupo;y++)
58         {
59             cout<<"G_"<<factores_orden_grupo[y]<<" orden=" <<
```

```
60              COMPONENTES[y].cardinalidad()<<":\n";
61              cout<<"{ ";
62              for(int d=0;d<n-1;d++)
63                  if(COMPONENTES[y].elementos[d])
64                      cout<<d+1<<" ";
65                      cout<<"}"<<endl;
66          }
67
68          Un D;
69          int OrdenComponente,Orden_g,g,elemento_ciclico,z;
70          Un grupo_ciclico;
71          bool verificar;
72          for(int l=0;l<numero_factores_orden_grupo;l++)
73          {
74              OrdenComponente=COMPONENTES[l].cardinalidad();
75              if(OrdenComponente==factores_orden_grupo[l])
76              {
77                  gruposciclicos[indice]=OrdenComponente;
78                  indice++;
79              }
80              else
81              {
82                  D=Un(n,1);
83                  while((D.cardinalidad()!=OrdenComponente))
84                  {
85                      for(int i=1;i<n-1;i++)
86                          if(D.elementos[i])
87                              COMPONENTES[l].eliminar(i+1);
88                      grupo_ciclico=Un(n,1);
89                      Orden_g=1;
90                      for(int i=1;i<n-1;i++)
91                      {
92                          if(COMPONENTES[l].elementos[i])
93                          {
94                              if(vector_ordenes[i]*D.cardinalidad()<=
95                                 OrdenComponente)
96                              {
97                                  if(vector_ordenes[i]>Orden_g)
98                                  {
99                                      Orden_g=vector_ordenes[i];
100                                     g=i+1;
101                                 }
102                             }
103                         }
104                     }
105                     elemento_ciclico=g;
106                     grupo_ciclico=Un(n,1);
107                     for(int i=2;i<=Orden_g;i++)
108                     {
109                         grupo_ciclico.agregar(elemento_ciclico);
110                         elemento_ciclico=(elemento_ciclico*g)%n;
111                     }
112                     verificar=true;
113                     z=1;
114                     while((z<n-1)&(verificar))
115                     {
116                         if(grupo_ciclico.elementos[z])
117                             if(D.elementos[z])
118                                 verificar=false;
119                         z++;
120                     }
121
122                     if(verificar)
123                     {
124                         D=D.suma_conjuntos(D,grupo_ciclico);
```

```
125                         gruposciclicos[indice]=vector_ordenes[g-1];
126                         indice++;
127                     }
128                     else
129                         COMPONENTES[l].eliminar(g);
130                 }
131             }
132         }
133     }
134     return gruposciclicos;
135 }
136 
137 #endif // DESCOMPOSICION_UN_H_INCLUDED
```

Código A.4: Archivo isomorfismos no abelianos.h

```
1  #ifndef ISOMORFISMOS_NO_ABELIANOS_H_INCLUDED
2  #define ISOMORFISMOS_NO_ABELIANOS_H_INCLUDED
3  #include "no abelianos.h"
4  
5  template <class P, class Q> bool isomorfismo (P grupo1, Q grupo2){
6  int ordenes1[8];
7  int ordenes2[8];
8  for(int i=0;i<8;i++)
9  {
10     ordenes1[i]=grupo1.transmutar(i).orden_elemento();
11     ordenes2[i]=grupo2.transmutar(i).orden_elemento();
12 }
13 sort(ordenes1,ordenes1+8);
14 sort(ordenes2,ordenes2+8);
15 
16 int z=0;
17 while((z<8)&(ordenes1[z]==ordenes2[z]))
18     z++;
19 if(z<8)
20     {
21         cout<<"Los grupos no son isomorfos"<<endl;
22         return false;
23     }
24 
25 else
26 {
27     int permutacion[8]={0,1,2,3,4,5,6,7};
28     int k,h;
29     h=1;
30     P producto1;
31     bool verificar;
32     do
33     {
34         verificar=true;
35         for(int i=0;i<8;i++)
36         {
37             for(int j=0;j<8;j++)
38             {
39                 k=0;
40                 producto1 = (grupo1.transmutar(i)*grupo1.transmutar(j));
41                 while(grupo1.transmutar(k).nombre!=producto1.nombre)
42                     k++;
43                 ///Condicion de isomorfismo
44         verificar=  verificar&( (grupo2.transmutar(permutacion[i])*
45                                 grupo2.transmutar(permutacion[j]))==
46                                 grupo2.transmutar(permutacion[k]) );
47                  if(!verificar)
48                     break;
```

```
49                }
50                if(!verificar)
51                    break;
52
53            }
54            if(verificar){
55                cout<<" cuentas permutaciones ? "<<h<<endl;
56                cout<<"Isomorfismo a traves de la permutacion: {";
57                for (int j=0;j<7;j++)
58                    cout<<permutacion[j]<<", ";
59                    cout<<permutacion[7]<<"} "<<endl;
60                 return true;
61            }
62
63        h++;
64        }while((next_permutation(permutacion,permutacion+8)));
65
66 return false;
67 }
68 }
69 #endif // ISOMORFISMOS_NO_ABELIANOS_H_INCLUDED
```

Referencias bibliográficas

[1] Israel N Herstein. *Topics in algebra*. John Wiley & Sons, 2006.

[2] Alexander A Stepanov and Daniel E Rose. *From mathematics to generic programming*. Pearson Education, 2014.

[3] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[4] Amin Witno. Finite abelian groups. *WON Series in Discrete Mathematics and Modern Algebra*, 7:1–10, 2012.