**PROCESO DE RECONSTRUCCIÓN INVERSA DE UNA IMAGEN RGB A PARTIR DE TRANSFORMACIONES BINARIAS Y ENMASCARAMIENTO**

*Adriana Camila Erazo Mora -María Angélica Osorio Rincón*

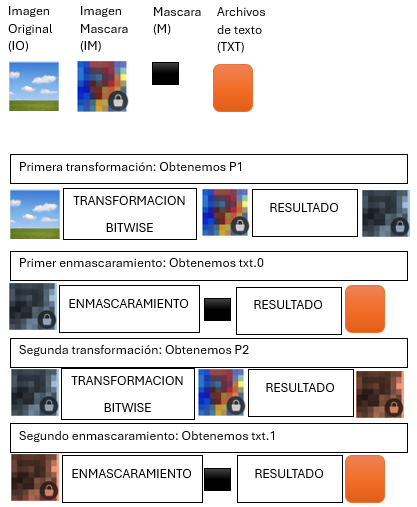
1. **Análisis del problema y consideraciones para la alternativa de solución propuesta:**
   1. ***Análisis del problema.***

El desafío consiste en recuperar una imagen original *IO* (RGB en formato BMP), la cual ha sido sometida a una cadena secuencial de transformaciones binarias y procesos de enmascaramiento parcial. El proceso inicia desde una imagen final distorsionada *ID,* junto a una imagen máscara *IM*, una máscara parcial *M* y una serie de archivos de textos que representan los resultados del enmascaramiento.

**Anexo de análisis visual del proceso a partir de *IO* a *ID*:**

La siguiente imagen representa gráficamente las transformaciones que se realizan:

(Si hay más de dos pasos de transformación y enmascaramiento, se irían anexando en el mismo orden, hasta llegar a la imagen distorsionada final)

  
Figura 1. Representación gráfica y textual de las operaciones realizadas para obtener *ID*

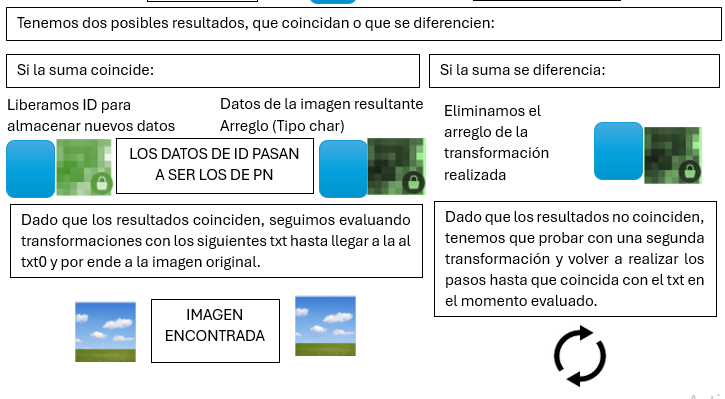
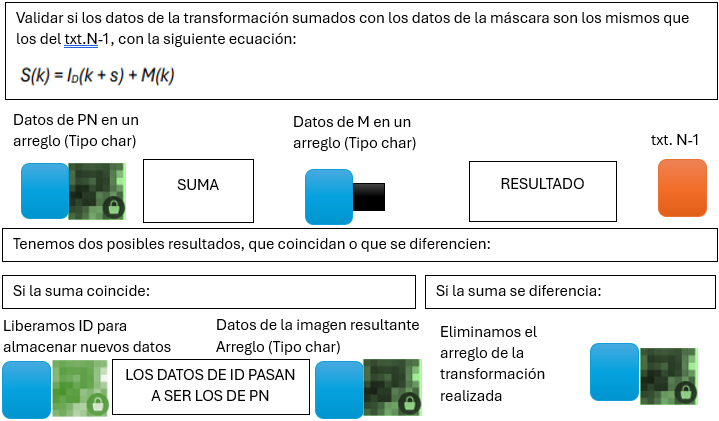
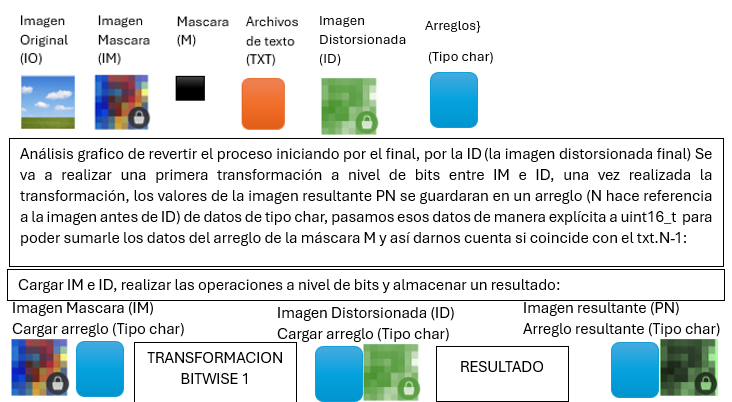
* 1. **Consideraciones clave**.
* Las operaciones binarias modifican la imagen mediante operaciones a nivel de bit (XOR y rotaciones).
* El enmascaramiento suma los valores del arreglo de la máscara *M* con una porción de la imagen dada una posición en específico, sin modificar directamente la imagen final, pero dejando evidencia en el archivo .txt.
  1. ***Dificultades clave.***
* La operación binaria aplicada en cada transformación es desconocida y debe determinarse mediante validación con el archivo txt. N-1.
* Las posiciones de enmascaramiento están desplazadas con una semilla *s* y operan a partir de una posición en específico del arreglo de la imagen.
* El formato BMP implica una estructura lineal RGB, lo que requiere precisión en el manejo de bytes (R, G, B).
  1. ***Solución propuesta.***

La reconstrucción se plantea de forma inversa:

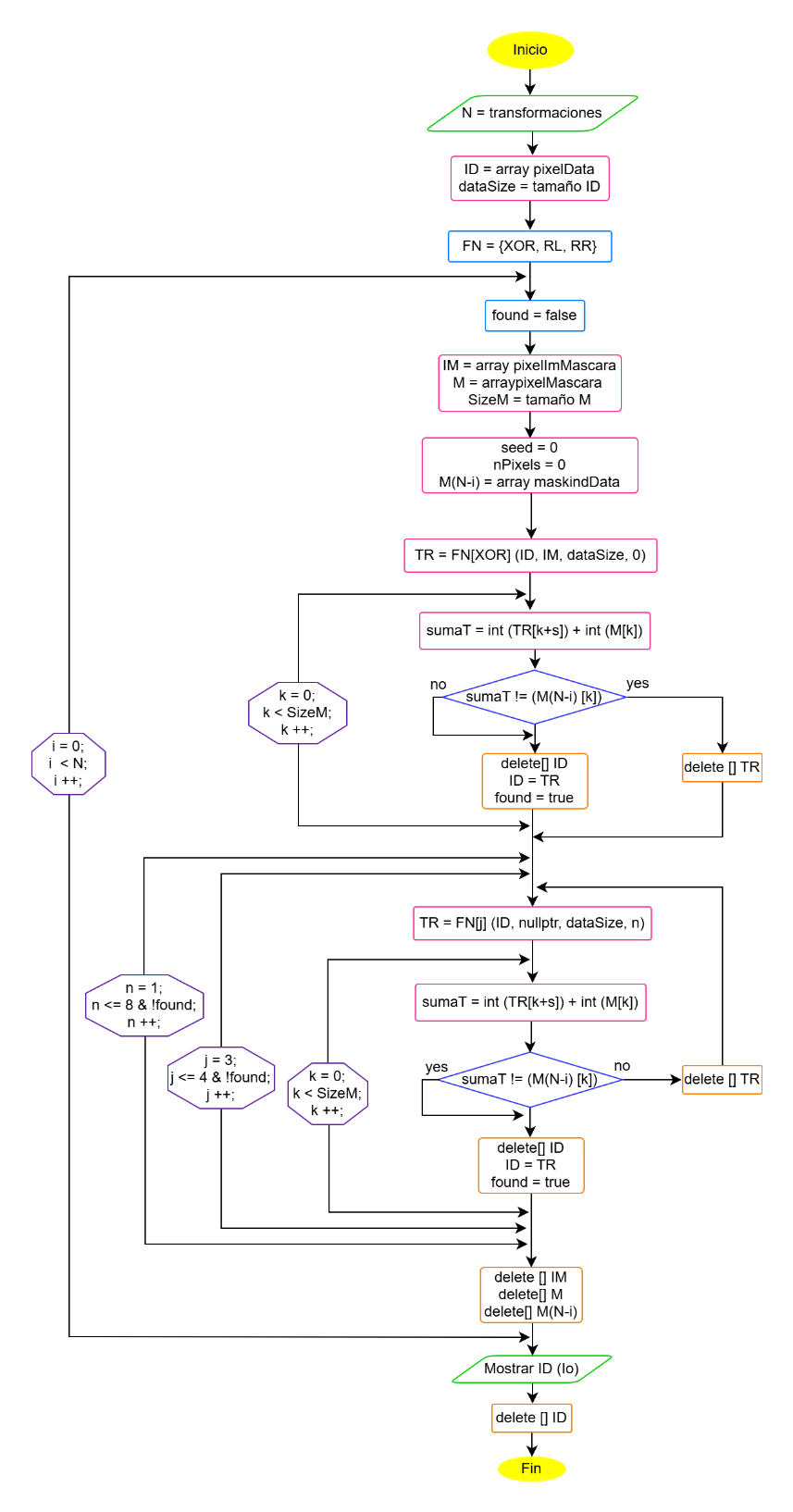
1. Se parte de la imagen final *ID*.
2. Se intenta revertir cada transformación aplicando posibles operaciones binarias con la máscara *IM* o rotaciones en su defecto.
3. Se valida cada operación aplicando el enmascaramiento (con *M* y la semilla *s*) y comparando el resultado con los archivos .txt.
4. Si la comparación es exitosa, se acepta la operación y se procede al paso anterior en la cadena.
5. Se repite hasta obtener la imagen original *IO*.

**Anexo de análisis visual del proceso a partir de *ID* a *IO*:**

La siguiente imagen representa la solución aplicada al análisis planteado, antes de iniciar la codificación del programa en Qt :

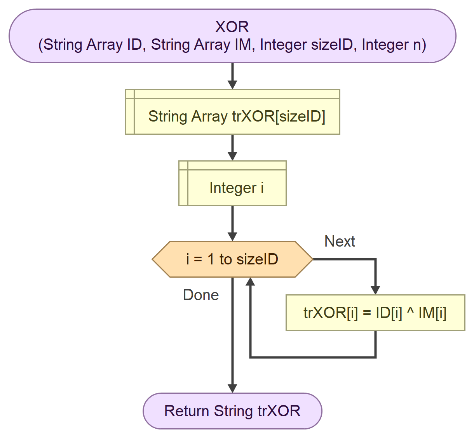
********

1. **Esquema de tareas en el desarrollo de los algoritmos:**

****Figura 2. Diagrama de flujo del programa principal

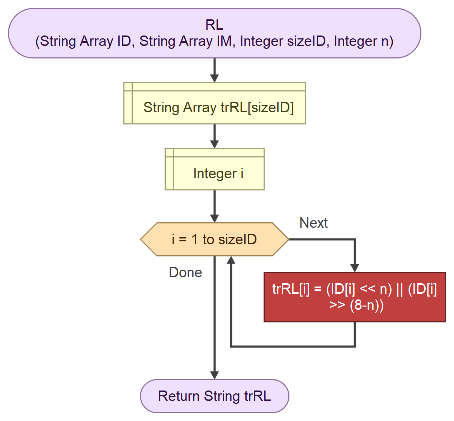
El diagrama de flujo del programa principal refleja la secuencia de tareas que el sistema sigue para aplicar y verificar las transformaciones, desde el inicio hasta la validación final de la imagen. La figura 2 representa la lógica central del proyecto, donde se coordinan las transformaciones y verificaciones, permitiendo visualizar el flujo de decisiones (XOR → RL → RR → verificación → siguiente paso). Las tareas principales, como se describe a continuación, se desarrollan de la siguiente manera:

1. Cargar la imagen distorsionada: al inicio del proceso, se cargan las imágenes necesarias, incluida la imagen distorsionada (*ID*) y la imagen de la máscara (*IM*), con sus dimensiones. Esto es crucial para realizar las transformaciones y compararlas contra las imágenes de referencia.
2. Aplicación de transformaciones (XOR, RL y RR): el sistema aplica las transformaciones en el siguiente orden: primero XOR, luego RL (rotación a la izquierda) y, si es necesario, RR (rotación a la derecha). Este flujo es optimizado ya que XOR es la operación más eficiente para la verificación inicial. Las transformaciones se aplican sobre el arreglo dinámico de datos de la imagen distorsionada en cada iteración.
3. Verificación de la transformación: después de aplicar cada transformación, el sistema verifica si la imagen transformada corresponde a la imagen esperada. Para ello, utiliza los datos de la máscara (*IM*). Si la verificación es exitosa, se guarda el resultado y se detiene el proceso. Si la verificación falla, se pasa a la siguiente transformación.
4. Iteración y terminación: el proceso sigue iterando con nuevas transformaciones hasta encontrar la correcta o hasta que se agoten todas las posibles transformaciones definidas (según el número de iteraciones y transformaciones permitidas). Si se encuentra la transformación correcta, el sistema avanza y se termina la operación; de lo contrario, el algoritmo continúa probando otras opciones.
5. Liberación de memoria: se libera la memoria de los arreglos de datos de las imágenes procesadas después de cada transformación para evitar fugas de memoria y garantizar la eficiencia del algoritmo a lo largo del proceso.
6. **Algoritmos implementados:**
   1. ***XOR.***La función XOR aplica una operación bit a bit entre la imagen transformada (*ID*) y la máscara (*IM*), procesando cada byte (RGB) de forma secuencial. Su objetivo es revertir una transformación previa del mismo tipo, siendo completamente reversible. Se prueba primero porque, a nivel de arquitectura, la operación XOR es computacionalmente más eficiente que las rotaciones (RL, RR), ya que requiere menos ciclos de CPU. Esto la convierte en la opción óptima para iniciar el proceso de validación y revertido.

****Figura 3. Diagrama de flujo función XOR

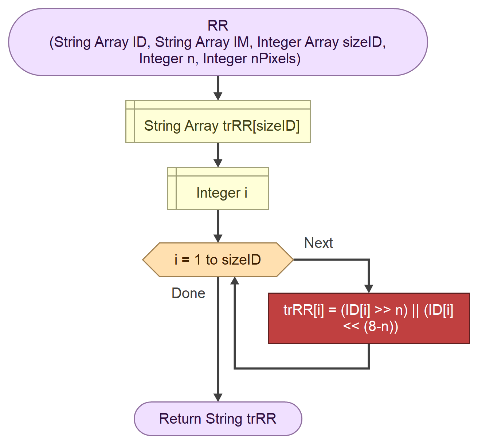
* 1. ***Rotación izquierda (RL).***

La función RL realiza una rotación circular a la izquierda de n bits sobre cada byte de la imagen (*ID*). Es reversible y se prueba después de XOR, ya que tiene mayor carga computacional. Se elige probar RL antes que RR porque en los dos casos planteados permite encontrar la imagen original con menos iteraciones, aunque esto puede variar según la transformación aplicada a *IO*.

******Figura 4. Diagrama de flujo función RL

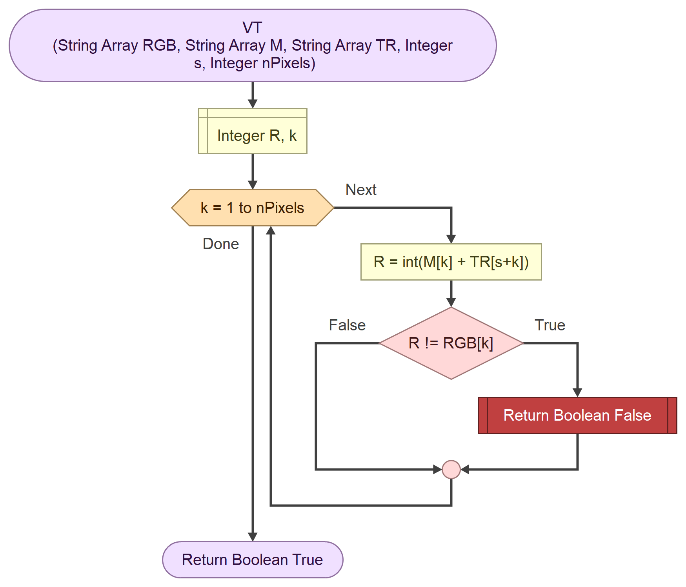
* 1. ***Rotación derecha (RR).***

La función RR aplica una rotación circular a la derecha de n bits en cada byte de la imagen (*ID*). Es complementaria a la rotación izquierda y también se usa para revertir posibles transformaciones anteriores. Se prueba de última debido a su complejidad relativa en comparación con XOR.

Figura 5. Diagrama de flujo función RR

* 1. ***Verificación de transformación (VT).***

Esta función valida si una transformación fue correcta comparando los valores resultantes del enmascaramiento (imagen + máscara desde una semilla s) con los datos almacenados en un archivo .txt. Si coinciden, se confirma que la operación aplicada fue la adecuada para ese paso del proceso inverso.

******Figura 6. Diagrama de flujo función VT

* 1. **Contador de archivos txt.**

Esta función cuenta los archivos que encuentra en el build que inician con una letra M y retorna su valor como entero. Primero, se usa QDir para acceder a la carpeta donde se está ejecutando el programa (build). Luego, se busca una lista de archivos que cumplan con el patrón "M\*.txt", lo que significa: archivos que empiezan con **M** y terminan en **.txt**. Esa lista se guarda en una variable llamada lista. Finalmente, se devuelve cuántos archivos hay en esa lista, usando lista.size().

1. **Problemas de desarrollo que se afrontaron:**

* Problema de lectura de archivos BMP y archivos de texto: durante el desarrollo, se presentó un problema al intentar leer las imágenes BMP (*ID, IM, M*) y los archivos de texto (*M(N-1)*). El programa no lograba acceder a estos archivos debido a que no se había especificado correctamente la ruta relativa o absoluta en el código. La solución fue mover las imágenes y archivos de texto a la carpeta *build* del proyecto de Qt, donde se encuentra el archivo ejecutable generado por el compilador.
* Liberación incorrecta de memoria: durante el proceso de codificación, se presentó un problema cuando una transformación no era la correcta. Se estaba liberando la memoria de los arreglos que contenían los valores RGB de las imágenes *ID* y *IM* de manera prematura, lo que provocaba que el programa fallara (crasheara). Esto se solucionó asegurando que la memoria solo se liberara cuando fuera apropiado, después de haber completado la transformación correcta y no antes de tiempo.
* Overflow debido a la definición incorrecta de la variable N: al analizar los tipos de datos adecuados para las variables, se definió inicialmente N como int. Sin embargo, más adelante, al calcular *IO*, se cambió N a uint8\_t para optimizar el uso de memoria. Este cambio provocó un overflow en la operación N = c - 1, ya que el valor de c podía superar el rango de uint8\_t. Finalmente, se corrigió este problema al volver a definir N como int, permitiendo que la operación N = c - 1 se realizara correctamente sin desbordamiento.

1. **Evolución de la solución y consideraciones para implementación:**
   1. ***Análisis del problema y sus requisitos.***

Inicialmente, se realizó un análisis profundo del problema para comprender claramente qué se debía lograr. Este análisis incluyó la diferenciación entre "transformación" y "enmascaramiento", conceptos clave para el desarrollo. Se determinó que las transformaciones generaban nuevas imágenes, mientras que el enmascaramiento guardaba y modificaba información de una imagen sin crear una nueva, o sea, con más precisión, sus valores se editaban con una suma y se guardaban los resultados en un archivo txt.

* 1. ***Selección de operaciones a nivel de bits.***

Una vez comprendida la naturaleza del problema, se procedió a analizar las operaciones binarias permitidas. Durante este análisis, se descartaron las operaciones de desplazamiento a la izquierda (<<) y a la derecha (>>), ya que estas no son reversibles, lo que implica que al aplicarlas se perdería información importante. Además, se evaluó la operación NOT, pero se concluyó que no era pertinente incluirla, ya que no se mencionaba en las especificaciones del problema y no requería la interacción con la imagen máscara.  
Al principio, se incluyeron las operaciones AND y OR, pero al mostrar en consola las transformaciones realizadas al reconstruir la imagen *IO*, nunca aparecían estas operaciones. Además, en el documento proporcionado no se mencionaba que se debían usar, por lo que se eliminaron del código para evitar iteraciones innecesarias sobre funciones que no aportaban al proceso de transformación.

* 1. ***Prueba de operaciones binarias en papel.***

Antes de proceder con la implementación del código, se realizaron pruebas conceptuales de las funciones XOR, AND, OR, RL y RR en papel, para entender su comportamiento y asegurarse de que las transformaciones se realizaban correctamente.

* 1. ***Lectura de archivos y pruebas iniciales.***

Una vez que se comprendieron las operaciones, se implementaron las funciones para leer los archivos de imágenes y los archivos de enmascaramiento. Se probaron diferentes transformaciones con los datos obtenidos, específicamente comenzando con XOR, para asegurarse de que las operaciones binarias se realizaban correctamente.

* 1. ***Optimización en el manejo de memoria.***

Durante la implementación, se tomó la decisión de no convertir los arreglos de punteros a unsigned char (que contenían las imágenes *ID* e *IM*) a binarios o enteros, ya que se pudo verificar que las operaciones binarias podían realizarse directamente sobre estos punteros. Además, se optimizó el uso de memoria, evitando la creación de arreglos intermedios para almacenar las sumas de las transformaciones con la máscara. En su lugar, se compararon los resultados de las operaciones directamente con los valores en el archivo .txt.

Teniendo en cuenta que los valores del arreglo que lee los datos del archivo .txt nunca superarán los 510 —ya que en las imágenes BMP los canales RGB están codificados a 8 bits cada uno, lo que implica que toman valores entre 0 y 255—, podemos afirmar que, en el peor de los casos, la suma de dos canales (por ejemplo, R=255 + R=255) no excederá dicho valor. A partir de esta observación, se tomó la decisión de cambiar el tipo de dato de la función loadSeedMasking, que originalmente era un unsigned int, por un uint16\_t. Esto se debe a que reservar memoria para un tipo unsigned int representa un mayor costo computacional, mientras que con un uint16\_t —capaz de almacenar valores entre 0 y 65,535— es más que suficiente. Esta modificación no afectó el correcto funcionamiento del programa y permitió optimizar el uso de la memoria dinámica.

* 1. ***Estrategia de verificación y reducción de iteraciones.***

Para evitar una cantidad excesiva de iteraciones durante el paso de verificación de la transformación, se adoptó la estrategia de comprobar si al menos un valor de la suma entre el arreglo de la transformación y la máscara era diferente del valor correspondiente en el archivo de texto. Si se detectaba una diferencia, se concluía que la transformación aplicada era incorrecta y se pasaba a intentar otra transformación. Esta estrategia permitió reducir considerablemente la cantidad de iteraciones innecesarias, optimizando el proceso.

* 1. ***Determinación del número de transformaciones (N).***

Para establecer cuántas transformaciones debían aplicarse en el ciclo principal, se optó por implementar una función que retornara la cantidad de transformaciones aplicadas c para el caso establecido, para definir N se tomó a c y restar 1 a ese valor, de manera que el proceso de verificación y enmascaramiento se realizara correctamente utilizando los archivos de texto correspondientes a M(N-1).

* 1. ***Implementación y desarrollo sin mayores complicaciones.***

Dado que se había realizado un análisis exhaustivo antes de comenzar a escribir el código, la implementación fue relativamente fluida y sin mayores problemas. La estructura del programa, con ciclos de iteración y verificaciones, fue diseñada de acuerdo con los requerimientos definidos, lo que permitió aplicar las transformaciones de manera eficiente y con pocos errores.

1. **Conclusiones:**

Se desarrolló e implementó una solución eficiente y modular para revertir secuencias de transformaciones binarias y enmascaramientos parciales aplicados a imágenes RGB en formato BMP, sin requerir conocimiento previo del orden ni del tipo de operaciones. Las transformaciones efectivas fueron XOR, rotación a la izquierda (RL) y rotación a la derecha (RR), mientras que las funciones AND, OR, NOT y los desplazamientos fueron descartados por su irreversibilidad o falta de aporte al proceso. Se validó que las operaciones bit a bit podían aplicarse directamente sobre datos tipo char, sin convertir a binario, optimizando así el uso de memoria y simplificando la implementación. Asimismo, se mejoró la eficiencia al cambiar el tipo de dato en la función loadSeedMasking a uint16\_t, en coherencia con los rangos máximos de los canales RGB (máximo 510), y al reducir iteraciones mediante una estrategia de verificación anticipada basada en comparación con archivos .txt. Implementar la función de contar los txt contenidos en el build facilita al usuario ya que el programa hará el numero de transformaciones precisas para el caso dado automáticamente.

En el momento de automatizar aún más el procedimiento de volver a la imagen original se llevó a cabo la implementación de una función que cuenta y retorna el número de archivos almacenados en la carpeta build, esto conllevó a realizar una investigación profunda sobre si esta operación tenía una carga computacional alta. En el código se utiliza memoria dinámica porque QStringList es un contenedor que almacena internamente una lista de cadenas (QString) cuyo tamaño puede variar en tiempo de ejecución. Esto significa que, cuando el programa ejecuta dir.entryList(...), Qt reserva espacio en el **heap** (espacio de memoria dinámica) para almacenar cada uno de los nombres de archivos que coincidan con el patrón. Esta memoria no está definida con un tamaño fijo como en un arreglo estático, sino que crece según la cantidad de archivos encontrados. Qt administra esta memoria automáticamente: cuando el objeto lista sale del alcance (por ejemplo, cuando termina la función), el sistema de objetos de Qt y su manejo de referencias aseguran que toda la memoria usada por QStringList y sus QString internos sea liberada correctamente. Esto lo logra gracias al uso de contadores de referencias y destructores bien definidos en sus clases (clases propias de Qt), evitando pérdidas de memoria sin que el programador tenga que intervenir directamente. La memoria se libera automáticamente al final de la función, justo después de return lista.size();, cuando lista sale de alcance y su destructor se ejecuta. Qt se encarga de liberar correctamente toda la memoria dinámica asociada.

El enfoque adoptado permitió obtener una solución precisa, escalable y adaptable a problemas similares.