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

*Adriana Camila Erazo Mora*

*María Angélica Osorio Rincón*

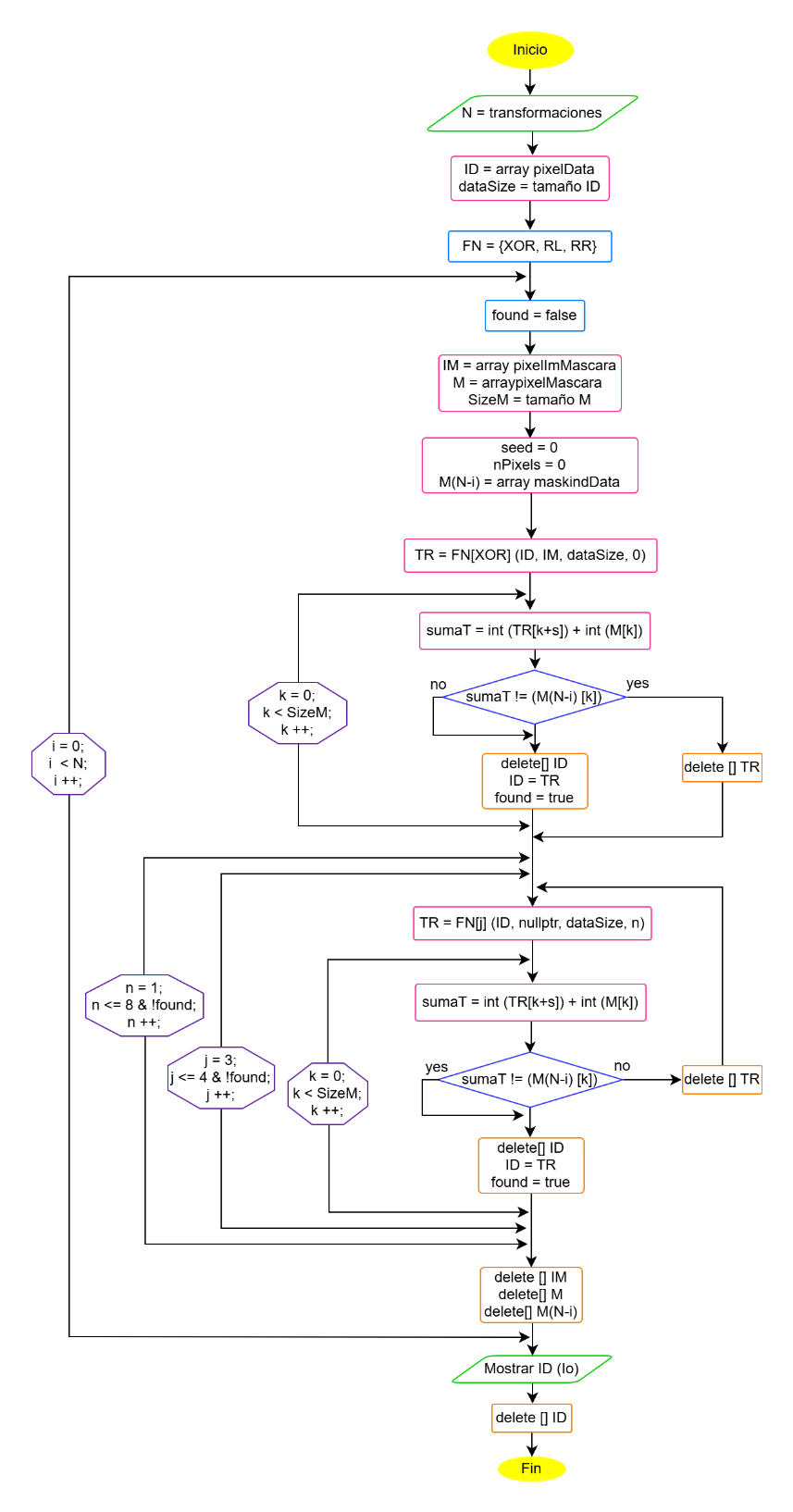
1. **Análisis del problema y consideraciones para la alternativa de solución propuesta:**
   1. ***Análisis del problema.***

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.

* 1. **Consideraciones clave**.
* Las operaciones binarias modifican la imagen mediante operaciones a nivel de bit (XOR, rotaciones, etc.).
* El enmascaramiento suma la máscara M desplazada a una porción de la imagen, 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.
* Las posiciones de enmascaramiento están desplazadas con una semilla *s* y operan sobre secciones 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.***

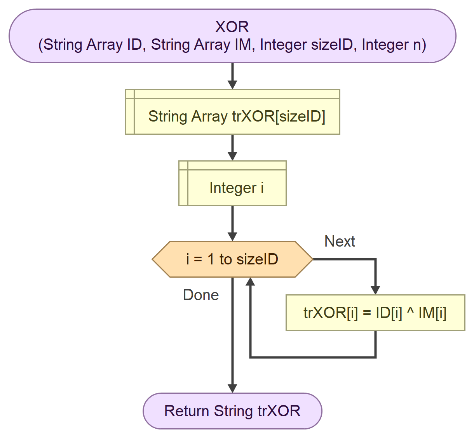
La reconstrucción se plantea de forma inversa:

1. Se parte de la imagen final *ID*.
2. Se intenta revertir cada transformación aplicando posibles operaciones binarias con la máscara *IM*.
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*.
6. **Esquema de tareas en el desarrollo de los algoritmos:**

****Figura 1. Diagrama de flujo del programa principal

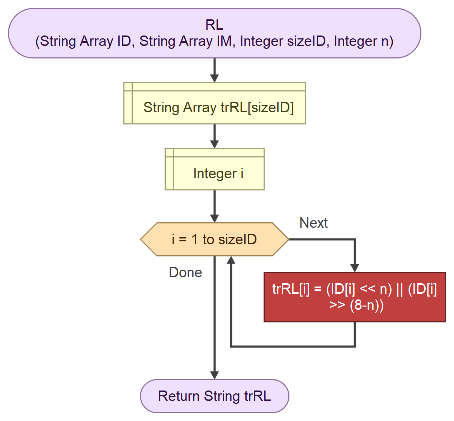
El diagrama de flujo del programa principal refleja la secuencia de tareas que el sistema sigue para aplicar y verificar las transformaciones, desde el inicio hasta la validación final de la imagen. La figura 1 representa la lógica central del proyecto, donde se coordinan las transformaciones y verificaciones, 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 2. Diagrama de flujo función XOR

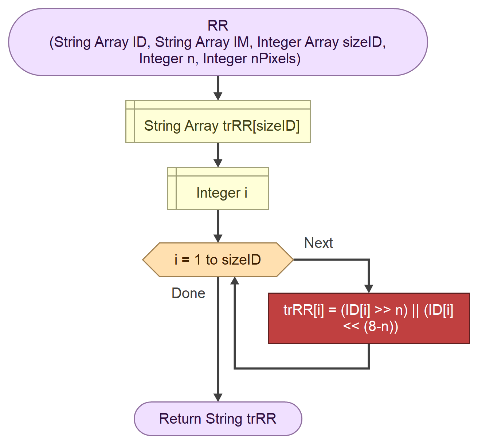
* 1. ***Rotación izquierda (RL).***

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 3. Diagrama de flujo función RL

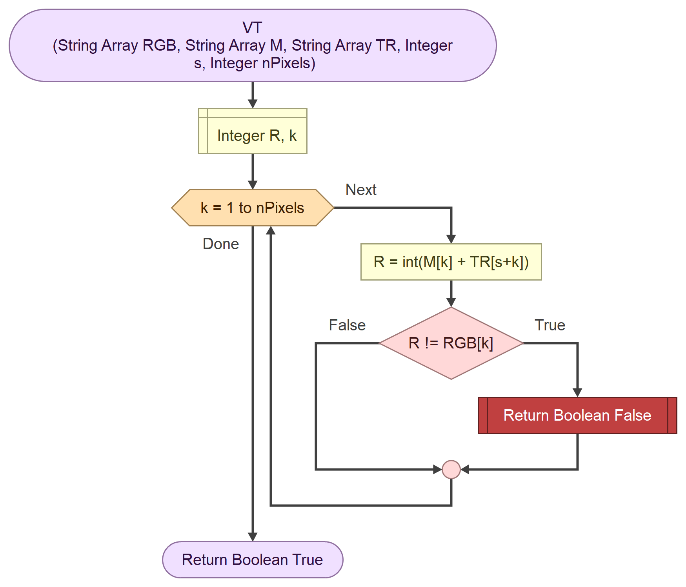
* 1. ***Rotación derecha (RR).***

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 4. 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 5. Diagrama de flujo función VT

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 ocultaba información en una imagen sin crear una nueva.

* 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.

* 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 decidió pedir al usuario la cantidad total de transformaciones N 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.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Conclusiones**

* Se logró implementar una solución que permite revertir cualquier secuencia de transformaciones binarias y enmascaramientos parciales sin conocer previamente el orden exacto ni los tipos.
* Las transformaciones más efectivas para el desafío fueron XOR, rotación izquierda (RL) y rotación derecha (RR). Las funciones AND, OR y NOT no aportaron valor al problema.
* El proceso fue cuidadosamente verificado contra los archivos .txt, garantizando la precisión del resultado.
* El enfoque modular y validado paso a paso permite que la solución sea escalable, mantenible y aplicable a otros problemas similares.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Estrategia de solución gráfica:**
   1. **Representación visual de las transformaciones aplicadas a la imagen original IO:**

La siguiente imagen representa gráficamente las transformaciones que se realizan dependiendo el caso:

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

Imagen que contiene colorido, azul, grande, calle

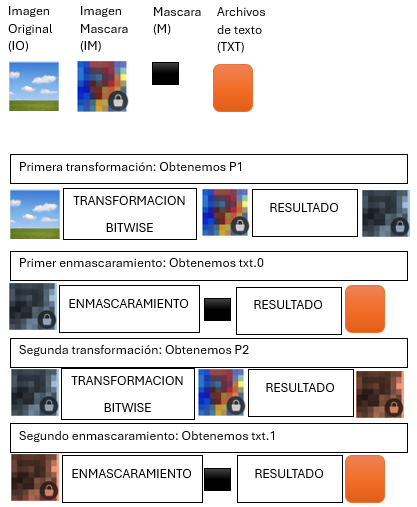
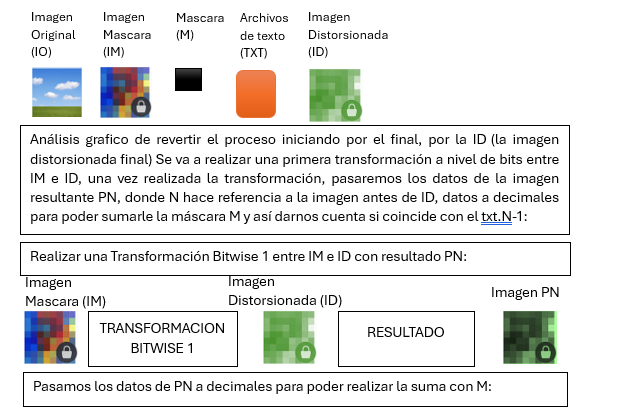
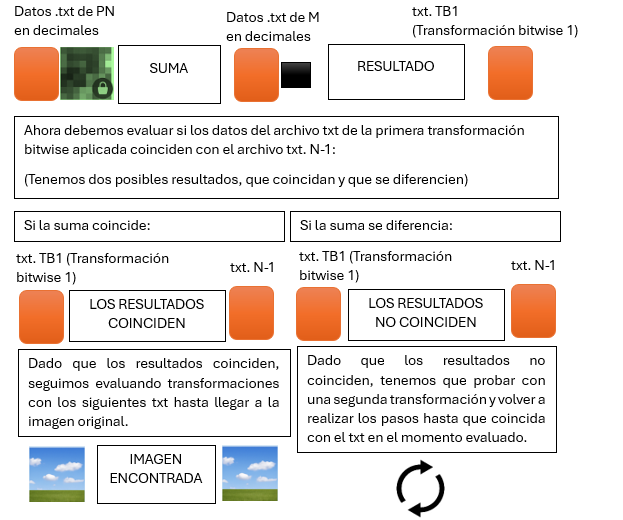
El contenido generado por IA puede ser incorrecto.

Imagen Distorsionada

(ID)

* 1. **Representación visual de la solución planteada para revertir el proceso:**



****

1. **Docstrings de funciones para cada transformación (operaciones a nivel de bits):**
   1. **Función AND (Operación &&):**

@brief Aplica la operación AND bit a bit entre los bytes de dos imágenes RGB. Esta función realiza una operación binaria AND (‘&’) entre cada byte correspondiente de dos imágenes de igual tamaño: la imagen distorsionada (ID) y la imagen máscara (IM). Cada byte representa un canal de color (R, G o B) y está almacenado en un arreglo lineal de tipo ‘unsigned char\*’ en formato RGB secuencial (R, G, B, R, G, B, ...). La operación AND se realiza byte a byte, es decir, por cada canal de cada píxel: resultado[i] = ID[i] | IM[i]. El resultado de esta transformación se almacena en un nuevo arreglo dinámico con el mismo formato RGB lineal.

@param ID Arreglo de bytes de la imagen distorsionada.

@param IM Arreglo de bytes de la imagen máscara.

@param dataSize Cantidad total de bytes (ancho × alto × 3 canales).

@return Puntero a un arreglo dinámico de tipo ‘unsigned char\*’ que contiene los resultados de la operación AND por byte.

*Ejemplo de la función AND:*

ID = 11001100 (204 en entero)

IM = 10101010 (170 en entero)

Resultado = 10001000 (136 en entero)

* 1. **Función OR (Operación ||):**

@brief Aplica la operación OR bit a bit entre los bytes de dos imágenes RGB. Esta función realiza una operación binaria OR (‘|’) entre cada byte correspondiente de dos imágenes de igual tamaño: la imagen distorsionada (ID) y la imagen máscara (IM). Cada byte representa un canal de color (R, G o B) y está almacenado en un arreglo lineal de tipo ‘unsigned char\*’ en formato RGB secuencial (R, G, B, R, G, B, ...). La operación OR se realiza byte a byte, es decir, por cada canal de cada píxel: resultado[i] = ID[i] | IM[i]. El resultado de esta transformación se almacena en un nuevo arreglo dinámico con el mismo formato RGB lineal.

@param ID Arreglo de bytes de la imagen distorsionada.

@param IM Arreglo de bytes de la imagen máscara.

@param dataSize Cantidad total de bytes (ancho × alto × 3 canales).

@return Puntero a un arreglo dinámico de tipo ‘unsigned char\*’ que contiene los resultados de la operación OR por byte.

*Ejemplo de la función OR:*

ID = 11001100 (204 en entero)

IM = 10101010 (170 en entero)

Resultado = 10001000 (238 en entero)

* 1. **Función NOT (Operación ~):**

@brief Aplica la operación NOT bit a bit a los bytes de una imagen RGB.

Esta función realiza una operación binaria NOT (‘~’) sobre cada byte de una imagen: la imagen distorsionada (ID).

Cada byte representa un canal de color (R, G o B) y está almacenado en un arreglo lineal de tipo ‘unsigned char\*’ en formato RGB secuencial (R, G, B, R, G, B, ...). La operación NOT invierte todos los bits de cada byte:

resultado[i] = ~ID[i].

El resultado de esta transformación se almacena en un nuevo arreglo dinámico con el mismo formato RGB lineal.

@param ID Arreglo de bytes de la imagen distorsionada.

@param dataSize Cantidad total de bytes (ancho × alto × 3 canales).

@return Puntero a un arreglo dinámico de tipo ‘unsigned char\*’ que contiene los resultados de la operación NOT por byte.

*Ejemplo de la función NOT:*

ID = 11001100 (204 en entero)

Resultado = 00110011 (51 en entero)

* 1. **Función XOR (Operación ^):**

@brief Aplica la operación XOR bit a bit entre los bytes de dos imágenes RGB.

Esta función realiza una operación binaria XOR (‘^’) entre cada byte correspondiente de dos imágenes de igual tamaño: la imagen distorsionada (ID) y la imagen máscara (IM).

Cada byte representa un canal de color (R, G o B) y está almacenado en un arreglo lineal de tipo ‘unsigned char\*’ en formato RGB secuencial (R, G, B, R, G, B, ...). La operación XOR se realiza byte a byte, es decir, por cada canal de cada píxel:

resultado[i] = ID[i] ^ IM[i].

El resultado de esta transformación se almacena en un nuevo arreglo dinámico con el mismo formato RGB lineal.

@param ID Arreglo de bytes de la imagen distorsionada.

@param IM Arreglo de bytes de la imagen máscara.

@param dataSize Cantidad total de bytes (ancho × alto × 3 canales).

@return Puntero a un arreglo dinámico de tipo ‘unsigned char\*’ que contiene los resultados de la operación XOR por byte.

*Ejemplo de la función XOR:*

ID = 11001100 (204 en entero)

Resultado = 01100110 (102 en entero)

* 1. **Función rotación izquierda:**

@brief Aplica una rotación a la izquierda de n (1 ≥ n ≤ 8) bits sobre cada byte de una imagen RGB. Esta función realiza una operación binaria de rotación a la izquierda sobre n bits de cada byte de la imagen distorsionada (ID). Cada byte representa un canal de color (R, G o B) y está almacenado en un arreglo lineal de tipo ‘unsigned char\*’ en formato RGB secuencial (R, G, B, R, G, B, ...).

La operación se realiza byte a byte, es decir, por cada canal de cada píxel:

resultado[i] = (ID[i] << n) | (ID[i] >> (8 - n))

El resultado de esta transformación se almacena en un nuevo arreglo dinámico con el mismo formato RGB lineal.

@param ID Arreglo de bytes de la imagen distorsionada.

@param dataSize Cantidad total de bytes (ancho × alto × 3 canales).

@param n Número de bits a rotar a la izquierda.

@return Puntero a un arreglo dinámico de tipo ‘unsigned char\*’ que contiene los resultados de la rotación.

*Ejemplo de la función rotación izquierda (3 bits):*

ID = 11001100 (204 en entero)

n = 3

*Resultado*

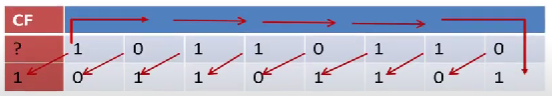
*Inicio*: 11001100

*Rotación 1:* 10011001

*Rotación 2:* 00110011

*Rotación 3:* 01100110

01100110 (102 en entero)

*Aplicación visual de cada rotación:*

*Bajo la fórmula:*

resultado[i] = (ID[i] << n) | (ID[i] >> (8 - n))

ID = 11001100 (204 en entero)

n = 3

*Resultado*

ID[i] << 3 = 01100000

ID [i] >> (8-3) = ID[i] >> 5 = 00000110

Aplicamos OR:

01100000

00000110

OR 01100110 (102 en entero)

* 1. **Función rotación derecha:**

@brief Aplica una rotación a la derecha de n (1 ≤ n ≤ 8) bits sobre cada byte de una imagen RGB. Esta función realiza una operación binaria de rotación a la derecha sobre n bits de cada byte de la imagen distorsionada (ID). Cada byte representa un canal de color (R, G o B) y está almacenado en un arreglo lineal de tipo ‘unsigned char\*’ en formato RGB secuencial (R, G, B, R, G, B, ...).

La operación se realiza byte a byte, es decir, por cada canal de cada píxel:

resultado[i] = (ID[i] >> n) | (ID[i] << (8 - n))

Esta operación asegura una rotación circular, es decir, los bits que "salen" por la derecha son reinsertados por la izquierda. El resultado de esta transformación se almacena en un nuevo arreglo dinámico con el mismo formato RGB lineal.

@param ID Arreglo de bytes de la imagen distorsionada.

@param dataSize Cantidad total de bytes (ancho × alto × 3 canales).

@param n Número de bits a rotar a la derecha.

@return Puntero a un arreglo dinámico de tipo ‘unsigned char\*’ que contiene los resultados de la rotación.

*Ejemplo de la función rotación derecha (3 bits):*

ID = 11001100 (204 en entero)

n = 3

*Resultado*

*Inicio:* 11001100

*Rotación 1:* 01100110

*Rotación 2:* 00110011

*Rotación 3:* 10011001

10011001 (153 en entero)

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.*Aplicación visual de cada rotación:*

*Bajo la fórmula:*

resultado[i] = (ID[i] >> n) | (ID[i] << (8 - n))

ID = 11001100 (204 en entero)

n = 3

*Resultado*

ID[i] >> 3 = 00011001

ID[i] << (8 - 3) = ID[i] << 5 = 10000000

*Aplicamos OR:*

00011001

10000000

OR10011001 (153 en entero)

**NOTA:** En la solución planteada no se tendrá en cuenta la operación binaria desplazamiento hacia la izquierda (<<) y desplazamiento hacia la derecha (>>), dado que son operaciones parcialmente irreversibles (al aplicarlas se puede “perder” información sobre la imagen BMP).

**Acerca de funciones de NOT:** Estamos analizando si realmente necesitamos esta función para la solución planteada, ya que esta función no requiere tomar datos de IM y una imagen PN (resultante de cada transformación) o ID, porque se realizan bajo una sola imagen, bajo unos datos específicos, ya sean de PN o de ID, pero no compuesta.

1. **Docstrings de funciones para procesar la información:**
   1. **Función de conversión unsigned char\* 🡪 int\***

@brief Convierte un arreglo dinámico de bytes (unsigned char\*) a enteros (int\*) manteniendo el orden RGB. Esta función toma un puntero a un arreglo de tipo ‘unsigned char\*’, que contiene los valores RGB de una imagen en formato lineal (R, G, B, R, G, B, ...), y lo convierte en un nuevo arreglo dinámico de tipo ‘int\*’. Esta conversión permite realizar operaciones posteriores (como enmascaramiento. Cada byte del arreglo original se convierte explícitamente a un entero utilizando ‘static\_cast<int>’, manteniendo la misma secuencia de datos RGB.

Ejemplo: salida[i] = static\_cast<int>(entrada[i]);

@param entrada Puntero a un arreglo dinámico de tipo ‘unsigned char\*’ que representa los datos RGB de una imagen.

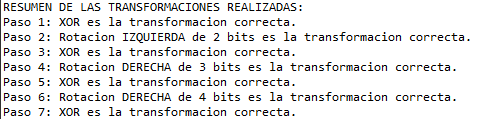
@param dataSize Tamaño total del arreglo (en bytes), equivalente a ancho × alto × 3 canales.

@return Puntero a un nuevo arreglo dinámico de tipo ‘int\*’ con los mismos valores convertidos a enteros, ordenados en formato RGB secuencial (R, G, B, R, G, B, ...).

1. **Conclusiones**

* Dada la posible solución lineal y obteniendo el resultado final de las transformaciones inversas a ID  y lograr llegar a la IO, nos hemos dado cuenta de que las funciones NOT, AND y OR, son transformaciones que no se usan:

En la siguiente imagen podemos observar la conclusión anterior:

****