Informe Final Desafio 1

1st Julián Sánchez Ceballos
Facultad de ingeniería
Universidad de Antioquia
Medellín, Colombia
julian.sanchez6@udea.edu.co

2nd Jaider Bedoya Carmona
Facultad de ingeniería
Universidad de Antioquia
Medellín, Colombia
jaider.bedoya@udea.edu.co

***Abstract*—Este informe presenta la solución a un problema de descompresión y desencriptación de archivos de texto, diseñado como ejercicio práctico para afianzar el uso de punteros, memoria dinámica y modularización en C++. El mensaje original fue comprimido mediante los algoritmos RLE o LZ78 y posteriormente encriptado mediante rotación de bits y operación XOR. Se propone una estrategia de ingeniería inversa basada en fuerza bruta, que evalúa todas las combinaciones posibles de parámetros y valida los resultados mediante un fragmento de texto conocido. Para optimizar el proceso, se incorporaron heurísticas que reducen el costo computacional descartando candidatos inválidos antes de la descompresión completa. La implementación final permitió recuperar de manera fiable los parámetros de cifrado y el mensaje original, demostrando la eficacia de un enfoque sistemático y modular en la resolución de problemas de compresión y criptografía básica.**

***Index Terms*—Compresión, encriptación, ingeniería inversa, desafío, memoria dinámica, punteros, C++**

INTRODUCCIÓN

Como parte de la evaluación del curso de Informática 2, se plantea un desafío de ingeniería inversa en C++ orientado al uso de punteros y memoria dinámica. El objetivo es implementar un programa capaz de descifrar archivos comprimidos y encriptados, demostrando habilidades en análisis de problemas, diseño algorítmico y dominio del lenguaje.

El reto consiste en partir de un texto comprimido y encriptado, junto con un fragmento conocido del mensaje original, para determinar los parámetros utilizados en el proceso (algoritmo de compresión, valor de rotación y clave XOR) y, con ellos, reconstruir la totalidad del mensaje original.

Para lograrlo, se definen y analizan los pasos necesarios, priorizando un enfoque modular y un flujo de validación que minimice resultados erróneos.

PROPUESTA DE SOLUCIÓN

A. *Condiciones y restricciones*

Se conoce que:

- El mensaje fue comprimido utilizando RLE o LZ78.
- Una vez comprimido, se aplicaron dos operaciones de cifrado:
  - Rotación a la izquierda de cada byte en n bits, con $0 < n < 8$.
  - Operación XOR con una clave de un solo byte K.
- Se dispone de un fragmento en texto plano del mensaje original, el cual servirá como referencia para identificar:
  - El método de compresión utilizado.
  - El valor de la rotación n.
  - El valor de la clave XOR K.
  - El mensaje original completo.

B. *Operaciones*

Los métodos de compresión considerados en el desafío son:

*1) **Compresión RLE**:* Es un algoritmo de compresión sin pérdida que funciona mejor en datos con repeticiones consecutivas. Por ejemplo, la secuencia `AAAAABBCCCDDDD` se codifica como `5A2B3C4D`, representando el número de repeticiones seguido del carácter.

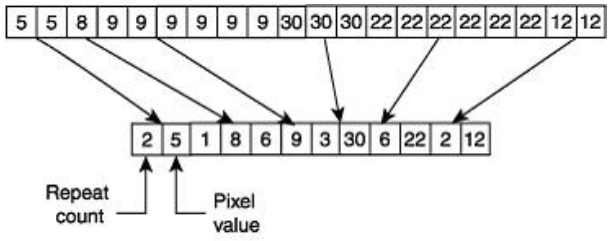

Fig. 1. Ejemplo compresión RLE

*2) **Compresión LZ78**:* A diferencia de RLE, LZ78 construye dinámicamente un diccionario de secuencias vistas. Cada nueva cadena encontrada se codifica como un par `(índice, carácter)`, donde el índice refiere a la secuencia más larga ya presente en el diccionario, y el carácter corresponde al siguiente símbolo en el texto. Esto genera una representación más compacta en casos con patrones recurrentes.

Las operaciones de encriptación aplicadas tras la compresión fueron:

*3) **Rotación de bits**:* Consiste en desplazar los bits de un byte hacia la izquierda o derecha, de forma circular. A diferencia de un corrimiento, los bits que salen por un extremo entran de nuevo por el otro.

*4) **Operación XOR**:* Es una operación bit a bit con la siguiente regla: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$. Una de sus propiedades es que, aplicando la misma clave dos veces, se recupera el valor original.

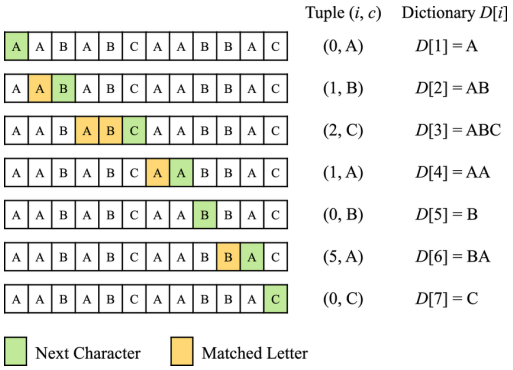

Fig. 2. Ejemplo compresión LZ78

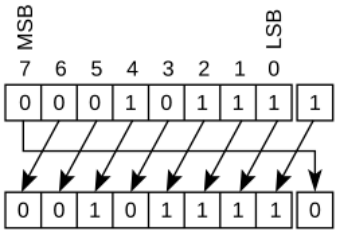

Fig. 3. Ejemplo rotación de un bit

C. *Estrategia mediante fuerza bruta*

Conociendo la naturaleza de las transformaciones, se plantea un algoritmo de fuerza bruta que recorra todas las combinaciones posibles de (n, K). Para cada par:

1) Se aplica la operación inversa (XOR y rotación).
2) Se intenta descomprimir con ambos algoritmos (RLE y LZ78).
3) Se valida si el resultado contiene el fragmento conocido.

Este enfoque garantiza encontrar los parámetros correctos al verificar contra un ancla confiable (la pista).

TABLE I
PROCEDIMIENTO DE CIFRADO Y DESCIFRADO

Paso	Cifrado (original)	Descifrado / Verificación
1	Texto plano	Leer fichero encriptado
2	ROL_n sobre cada byte	XOR inverso: $b' = b \oplus K$
3	XOR con clave K	ROR_n sobre cada byte
4	Flujo comprimido (RLE/LZ78)	Comprobar ratio de caracteres imprimibles
5	–	Descompresión LZ78 (si válida)
6	–	Descompresión RLE (si válida)
7	–	Buscar fragmento conocido en el texto
8	–	Si aparece, aceptar (n, K) y método

D. *Criterio de validación*

Se considera que un resultado es correcto si:

- El flujo desencriptado es interpretable por RLE o LZ78 sin errores de formato.
- El texto descomprimido contiene de manera íntegra el fragmento conocido.

```
1011 XOR 0011 = 1000
1000 XOR 0011 = 1011
```

Fig. 4. Ejemplo de una XOR

Este criterio evita falsos positivos al no depender únicamente de obtener texto legible.

E. *Diagrama de flujo de la solución*

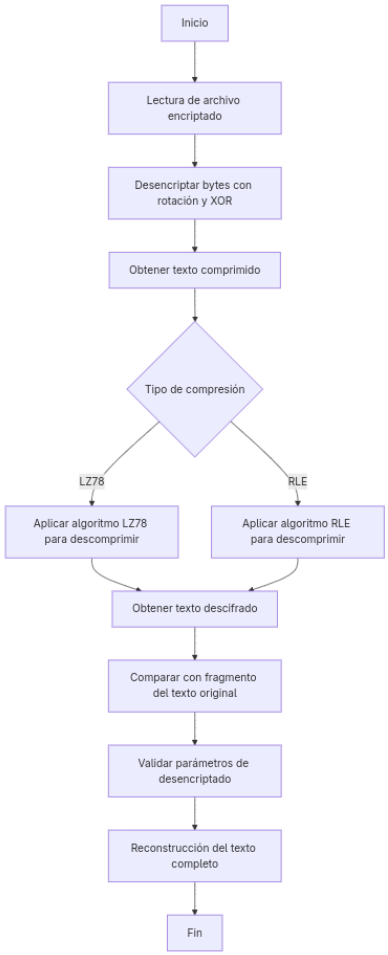

Fig. 5. Diagrama de flujo de la solución

F. *Arquitectura de la solución*

El sistema se organiza en módulos independientes, empleando memoria dinámica y tipos primitivos en lugar de STL.

*1) **Estructura de directorios**:*

```
/project
  include/
     app.hpp
     solver.hpp
     compress.hpp
  src/
```

```
    solver.cpp
    compress.cpp
    app.cpp
main.cpp
```

IMPLEMENTACIÓN DE LA SOLUCIÓN

Como se mencionó previamente, el desafío se solucionó mediante la implementación de unidades funcionales o módulos independientes que cumplen con una función dentro del pipeline de descifrado y descompresión, en esa sección se describen de manera especifica el comportamiento:

app.hpp

Este archivo contiene las definiciones y prototipos de funciones necesarias para el manejo de archivos, ejecución principal y control de errores de la aplicación, las funciones contenidas son:

- **my_error_t app_main():** Contiene la logica y el flujo de la aplicación, ejecutando los módulos principales, esta función retornara un OK si la ejecución del programa fue exitosa o un ERROR en caso de que ocurra un error durante la ejecución.
- **my_error_t read_file(const char* path, uint8_t** out_buf, size_t* out_len):** Abre un archivo de texto indicado en uno de sus argumentos de entrada (path), guarda su contenido en un buffer dinamico, reserva la memoria pero no la libera. Retorna OK si el archivo se leyó correctamente o ERROR en caso contrario.
- **char* get_frag(const char* path):** Función auxiliar para extraer la la información especifica de un archivo de texto. Retorna un puntero a una cadena de caracteres con el fragmento extraido. No libera la memoria reservada.

solver.hpp

Este archivo contiene las definiciones y prototipos para las función de fuerza bruta, además cuenta con funciones auxiliares para el descifrado, análisis del porcentaje imprimible en el texto y coincidencias entre el texto descomprimido y la pista.

- **[aux] int8_t decrypt_buffer(const uint8_t *in, size_t len, uint8_t n, uint8_t k):** Toma un buffer de entrada encriptado, aplica la operación XOR con la clave k y luego rota cada byte n posiciones a la derecha generando el buffer descifrado.
- **[aux] bool printable_ratio(const uint8_t buf, size_t len, double min_ratio):** Recorre el buffer y calcula la proporción de caracteres comprendidos en el rango ASCII imprimible (32-126) retorna verdadero si la proporción de caracteres imprimibles es mayor o igual a un umbral previamente definido.
- **[aux]bool contains_substr(const char *text, const char *part):** Implementa una búsqueda secuencia de la subcadena part dentro del texto text. Retorna un verdadero si encuentra la cadena y un false si no encuentra la cadena.
- **bool finder(const uint8_t* enc, size_t enc_len, const char* know_fragment, char** out_msg, char** out_method, uint8_t* out_n, uint8_t* out_k):** Esta funcion recorre todas las combinaciones de los valores n (1 a 7) y k (0 a 255), aplicando el descifrado mediante decrypt_buffer. Luego valida si el resultado contiene un número suficiente de caracteres imrpimibles usando printable_ratio si es así el texto descifrado se intenta descomprimir primero con RLE y después con LZ78 verificando si la salida contiene el fragmento conocido. Si se encuentra coincidencia: Retorna el mensaje descifrado, el método y usado y los parámetros n y k.

compress.hpp

Este archivo contiene las definiciones de las funciones de descompresión y operaciones a nivel de bit:

- **uint8_t ror_8 (uint8_t v, unsigned int n):** Realiza una rotación de bits a la derecha. Retorna el byte rotado n veces a la derecha.
- **uint8_t rol_8 (uint8_t v, unsigned int n):** Realiza una rotación de bits a la izquierda. Retorna el byte rotado n veces a la izquierda.
- **char *rle_decompress(const uint8_t *in, size_t len):** Descomprime una cadena usando Run-Length Encoding. Recibe el puntero a la cadena comprimida y la longitud de esta cadena y retorna un puntero a la cadena descomprimida o nullptr en caso de error, aunque reserva la memoria dinámica es el usuario que llame esta función el responsable de liberarla para evitar fugas de memoria.
- **char *lz78_decompress(const uint8_t *in, size_t len):** Descomprime una cadena usando el algoritmo de LZ78. Recibe un puntero a la cadena comprimida y la longitud de esta cadena y retorna un puntero a la cadena descomprimida o nullptr en caso de error, aunque reserva la memoria dinámica es el usuario que llame esta función el responsable de liberarla para evitar fugas de memoria.

De este modo al ejecutar el programa se obtiene la siguiente salida:

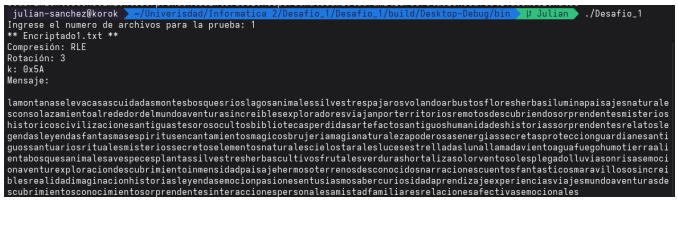

Fig. 6. Resultados para el texto de prueba 1

PROBLEMAS EN LA IMPLEMENTACIÓN Y POSIBLES MEJORAS

Durante la implementación se identificaron algunos problemas y oportunidades de mejora:

- **Formato de la compresión:** El estándar de compresión RLE, tal como se presenta en la literatura y en la propia guía del desafío, no contempla la inserción de caracteres

adicionales en el flujo comprimido. Sin embargo, en los textos de prueba se incluyó un byte extra que no corresponde a la especificación. Este valor puede interpretarse como ruido y genera una complejidad adicional al proceso. Una solución práctica consistió en descartarlo al inicio de la descompresión; no obstante, dicha estrategia resulta limitada desde el punto de vista de buenas prácticas y escalabilidad, ya que no sería aplicable frente a implementaciones estándar de RLE.

- **Complejidad del código:** La implementación basada en punteros crudos y gestión manual de memoria dinámica, prescindiendo de librerías estándar, permite que el código sea portable y aplicable en sistemas embebidos o entornos con recursos limitados. No obstante, esta aproximación incrementa la complejidad del código y disminuye su legibilidad. Como mejora futura, se propone la incorporación de estructuras de la STL, como `std::string` y `std::vector`, con el fin de simplificar la sintaxis, mejorar la claridad del código y fortalecer la seguridad en la gestión de memoria.
- **Gestión de memoria:** Si bien se controlaron fugas de memoria en la mayoría de las funciones, aún existen casos en los que la liberación depende explícitamente del usuario. Una mejora recomendable es encapsular la gestión de recursos mediante el uso del patrón RAII (*Resource Acquisition Is Initialization*), lo cual asegura la liberación automática de memoria del `heap` y reduce la probabilidad de errores por gestión manual, además de mitigar problemas de fragmentación.
- **Validación de entradas:** Actualmente se asume que los archivos existen y cumplen con el formato esperado. Se podría robustecer el programa agregando más validaciones y mensajes de error descriptivos,

CONCLUSIONES

El desarrollo de este desafío permitió aplicar de manera práctica conceptos avanzados de C++, incluyendo el uso de punteros, la gestión de memoria dinámica y la modularización del código.

El sistema implementado fue capaz de:

- Procesar archivos comprimidos y encriptados.
- Detectar el algoritmo de compresión utilizado (RLE o LZ78).
- Aplicar correctamente las transformaciones inversas de rotación y XOR.
- Reconstruir el mensaje original, verificando su validez a partir de un fragmento conocido.

El enfoque de fuerza bruta permitió recuperar los parámetros correctos, destacando la importancia de combinar técnicas algorítmicas con validaciones externas. Para reducir el costo computacional, se implementaron heurísticas que evalúan la coherencia de los textos candidatos antes de ejecutar operaciones de descompresión completas, evitando así procesamiento innecesario sobre datos inválidos.

En conclusión, el ejercicio no solo fortaleció competencias en programación de bajo nivel y análisis algorítmico, sino que también estableció una base sólida para afrontar proyectos de mayor complejidad, en los cuales aspectos como seguridad, eficiencia y claridad del código resultan igualmente críticos.

REFERENCES

[1] D. Salomon, *Data Compression: The Complete Reference*, 4th ed. Springer, 2004.

[2] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[3] W. Storer, "Data compression via textual substitution," *Journal of the ACM*, vol. 29, no. 4, pp. 928–951, 1982.

[4] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. Wiley, 1996.

[5] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley, 2013.

[6] H. Sutter and A. Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2005.

[7] R. Barry, *Mastering the FreeRTOS Real Time Kernel*, Real Time Engineers Ltd., 2021. [Online]. Available: https://www.freertos.org/