**Informe Final Desafío 1 (Informática 2)**

**Estudiantes:**

Jose Eduardo Valverde Alvarez ([josee.valverdea@udea.edu.co](mailto:josee.valverdea@udea.edu.co))

Andrés Felipe Lafaurie Rincon ([andres.lafaurie@udea.edu.co](mailto:andres.lafaurie@udea.edu.co))

**Profesor:**

Anibal Jose Guerra Soler

Universidad de Antioquia

Facultad de Ingeniería

Ingeniería en Telecomunicaciones e Ingeniería electrónica

Medellín

28 de Septiembre de 2025

1. **Análisis y diseño de la solución**

1.1. Contextualización

Un cliente nos entrega un mensaje original hecho en texto plano que pasó por dos tipos de codificación, primero pasó a ser codificado (o en este caso, comprimido) en uno de los dos siguientes formatos de compresión: RLE o Lempel-Ziv (LZ78), luego de pasar por el proceso de comprensión en uno de los dos formatos, el texto comprimido pasa a binario, dando lugar al proceso de rotación de entre 1 y 7 bits a la izquierda, y finalmente se le asignó una clave que fue proporcionada por el cliente la cual somete a un enmascaramiento usando el operador XOR a los bytes rotados, la clave ocupa el tamaño de un solo byte y está en hexadecimal, al hacer la conversión de cada uno de estos caracteres en binario rotados y enmascarados a char (ASCII/UTF-8) veremos lo que sería el conjunto de caracteres extraños en el archivo de texto que nos proporcionó el cliente.

1.2. Análisis detallado del desafío

Sabemos que el mensaje fue comprimido y encriptado utilizando dos posibles métodos para cada caso, para el caso de la comprensión el texto podría estar comprimido en los métodos RLE o LZ78 y finalmente para la encriptación se utilizarían rotaciones de n bits a la izquierda junto con un XOR aplicado con una de 256 posibles claves (todas las combinaciones posibles en un bit de 8 bytes), todos estos procedimientos se aplicaron una sola vez.

Recibiendo el mensaje encriptado, y sabiendo que se sometió a alguno de los dos métodos de comprensión anteriormente mencionados, y después se le aplicaron dichos procedimientos de encriptación en los bits de los caracteres del archivo, deducimos que al texto encriptado fue sometido a la siguiente secuencia:

Texto Original (*EncriptadoX.txt*) > Compresión (RLE/LZ78) > Aplicación de máscara (o clave) con XOR > Rotación de n-bits a la izquierda (n-bytes a la izquierda, or (|) con bit 8-n bytes rotados a la derecha) > Texto encriptado.

Conociendo este proceso sabemos que para desencriptar el texto debemos hacer el proceso inverso con el texto que tenemos encriptado y compararlo con la pista que nos dieron para comprobar que el proceso realizado sea el correcto.

1.3. Diseño del Código

Dentro de la función Main se inicializa el programa, se le pide al usuario que ingrese el número de textos que desea descifrar, tras de esto se ingresa a un while que itere tantas veces como textos se desean descifrar, se procede a llamar a una función que leerá los archivos que se necesitan para empezar el descifrado utilizando operadores de la librería fstream, al leer el archivo correspondiente al punto del ciclo y almacenar su contenido en una arreglo dinámico cuyo tamaño se define en el conteo de caracteres que ocupa el archivo, luego se procede a almacenar el contenido de los archivos en un arreglo dinámico caracter por caracter.

La primera función del desencriptado es la aplicación de la máscara 1 de 256, para después llamar una función que va a aplicar la rotación de n bytes correspondiente para desencriptar el texto utilizando operadores de bits que trabajaran directamente con los bytes que conforman cada carácter en el texto encriptado. Después de los pasos de encriptación se comprobará si el texto está comprimido en RLE o LZ78 y (...)

1. **Esquema de la solución al desafío**

Para darle solución a este problema planteamos entonces una estructura de archivo basada en ciclos anidados que permiten probar todas las combinaciones de parámetros posibles y las dos posibles descompresiones. almente se inicializa un ciclo que preguntará al usuario cuantos archivos se evaluaran, después se inicia el proceso de desencriptación, donde un primer ciclo podrá tomar todos los

El código se ejecuta dentro del main, princip valores de máscaras posibles, seguidamente otro ciclo donde por cada valor de máscara se podrá evaluar cada valor posible para la rotación de bits, después de ambos procesos de encriptación, el texto ya desencriptado se somete a descompresión, probando con ambos posibles casos, RE y LZ78, donde luego de cada proceso de descompresión de evalua si el texto obtenido, el cual se supone debería ser el completamente descifrado contiene dentro la pista, lo que indicaría que en efecto se hizo el proceso correcto. El programa además viene acompañado de ciertas funciones necesarias para realizar estos procesos.

en aspectos generales la arquitectura del programa consta de los siguientes módulos:

**Módulo de entrada**: se encarga de abrir los archivos encriptados y las pistas.

**Módulo de desencriptado**: aplica la máscara XOR y rotación de bits para obtener datos legibles.

**Módulo de filtrado y validación**: elimina caracteres nulos y verifica que el texto corresponda a un formato válido.

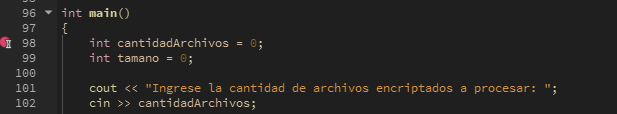
**Módulos de descompresión**: implementa los algoritmos RLE y LZ78.

**Módulo de búsqueda de pista**: verifica si el texto contiene la pista correspondiente.

**Salida de resultados**: muestra en pantalla los parámetros usados (máscara, rotación, formato) y el texto original.

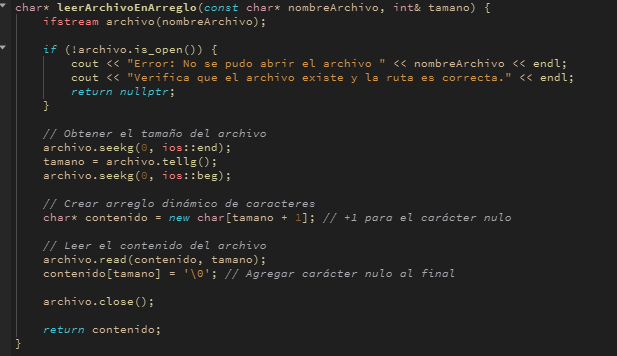
1. **Algoritmos Implementados.**

3.1. **Lectura del archivo**

En las primeras líneas de main(), se inicializan dos variables, una a la cantidad de archivos que se van a leer, y otra para el tamaño del arreglo en donde se va a guardar el contenido del texto encriptado.

Al tomar de la terminal la cantidad de archivos que se van a desencriptar en el archivo se entra al ciclo for en donde se cuenta del 1 a la n cantidad de archivos que se van a desencriptar (empezando por Encriptado1.txt), se declaran las memorias estáticas para las rutas del archivo y luego se hace lectura de archivos (sprintf) guardandolo en la variable nombreArchivo (y para la pista, nombrePista).

Más abajo se va a declarar una variable tipo apuntador que invoca a la funcion leerArchivoEnArreglo en donde se guardará el contenido del archivo en un arreglo dinámico tipo char (variada según la cantidad de caracteres que tiene el archivo, dentro de la función hay operadores para contar los caracteres del archivo que definirían el tamaño del arreglo)



3.2. **Desencriptación con operadores de bits**.

Al momento de entrar al ciclo de desencriptacion, se asegura que no hubo algun error al leer el archivo que haya causado que la funcion de leerArchivoenArreglo haya retornado un puntero nulo (nullptr), de ser así, se empieza el proceso de desencriptación de la siguiente manera:

Se va a inicializar un ciclo for con la variable máscara, que será la encargada de proporcionar una de las 256 combinaciones posibles en un solo bit para el XOR, luego otro ciclo con otra variable inicializada que es la cantidad de rotaciones a la derecha que se le aplicarán al bit, y por último un proceso adicional en donde se pasan los caracteres de entero a numérico imprimible y por último el filtrado de caracteres nulos (ver punto 4. para más detalles)

3.3 Comprobación de estructuras

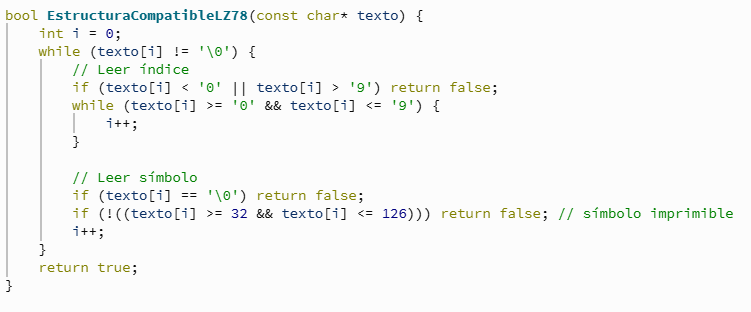


Para hacer el proceso de descompresión de texto tras haber desencriptado el mensaje dado, aplicando XOR y Rotación de bits a la inversa, se plantean dos condicionales que permiten comprobar si el texto resultante después de haber hecho estos procesos cumple la estructura estándar que debe tener un arreglo para poder ser descomprimido, estas validaciones se hacen con la intención de mejorar la eficiencia del código y evitar que se prueben por completo las 256\*7\*2 combinaciones posibles, de esta forma con las dos funciones que permiten comprobar las estructuras del texto descifrado para cada caso se evita realizar el proceso de desencriptación si se sabe que no va a arrojar un resultado probable.

Comprobacion RLE



La función EstructuraValida permite comprobar que un texto pueda ser descomprimido utilizando el método de desencriptación RLE.   
Esta función recibe un texto que va a analizar, comprueba caracter por caracter del texto siempre y cuando este no sea el carácter nulo que marca el final de la cadena, para cada carácter analiza que sea un numero, esta funcion tambien permite comprobar si los siguientes también son números, en caso de que el siguiente carácter no sea número se comprueba entonces que sea una letra mayuscula o minuscula, este mismo proceso para todo el código, de esta forma se controla que la estructura del código sea la adecuada par pasar a descomprimir.



La función EstructuraCompatibleLZ78 actúa de forma muy similar a la anterior, de igual forma recibe un texto que analiza y recorre caracter por caracter de este arreglo comprobando si el carácter que está leyendo es un número o no, y si es número, si los siguientes caracteres son números, la diferencia de esta función frente a la que comprueba RLE es que en esa función no se aceptan caracteres sueltos ni precedidos de un 0, cosa que es muy común en el caso de descompresión con LZ78.

3.4. Descompresion RLE y LZ78

Tras comprobar que las estructuras sean correctas para descomprimir, se ingresa a los condicionales donde se reservarán espacios de memoria suficientes para almacenar los textos descomrpimidos y se invican las funciones de descompresion respectivas para cada metodo.

DescompresionRLE



Esta función recibe el texto descifrado y con los caracteres nulos filtrados, texto ya listo para descomprimir, además este texto ya cuenta con al estructura necesaria por lo que el trabajo del algoritmo es recorrer caracter por caracter tomando el número anterior a una letra, que puede ser de más de una cifra, y después tomando el símbolo que le sigue a este número, imprimiendo este símbolo tantas veces como el número que le precede, así con todo el arreglo, esta funcion tambien recibe el espacio de memoria reservado que es donde se escribirá el resultado del descomprimido.

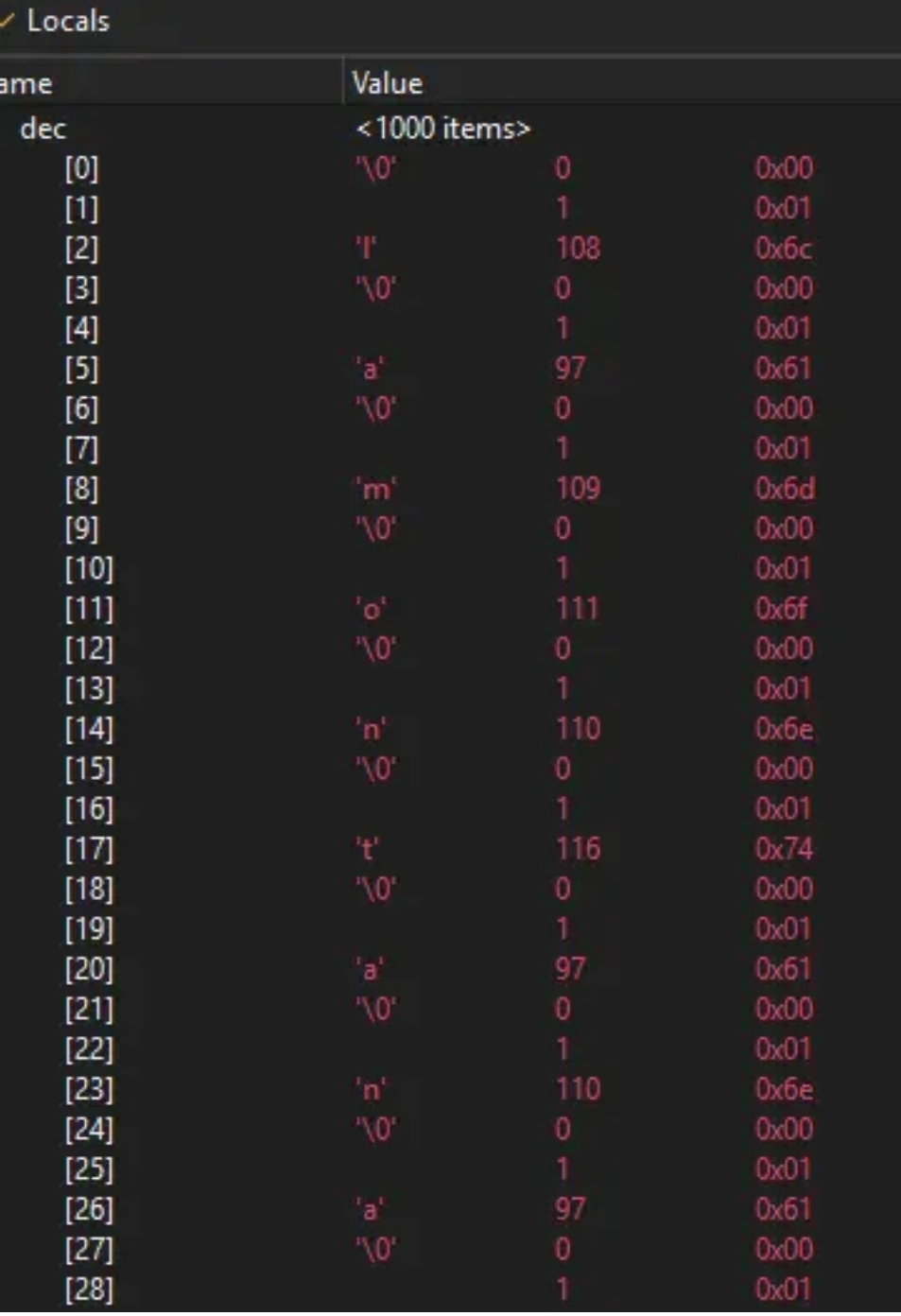


Para descomprimir archivos por método LZ78 se hace uso de un diccionario, que consta de tuplas donde el primer dígito es un índice, que indica una posición dentro del diccionario, y el segundo dígito es un símbolo que se agrega al final de los digitos que estan en la posición de diccionario apuntada por el índice de la tupla, este algoritmo permite hacer ese proceso donde siempre que se encuentre un índice 0 v a ser una nueva entrada al diccionario y se va a almacenar según el paso en el que vaya, si un índice de una tupla próxima apunta a este símbolo entonces se imprimirá y se agregara al final de este el símbolo de la tupla, este algoritmo también evalúa que los índices no excedan el orden ni la cantidad de pasos.

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

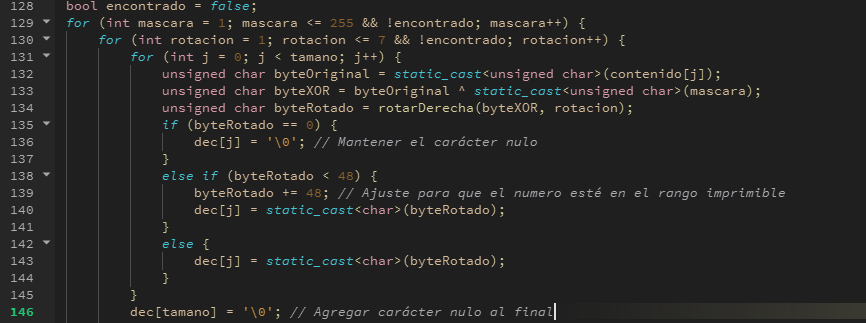
4.1. Caracteres no imprimibles inicialmente al desencriptar el texto.

Al hacer debug para probar la rotación y XOR en uno de los archivos del dataset de prueba (con la clave y cantidad de bytes rotados a la derecha especificados en el readme) se encontró con un pequeño detalle en la manera con la que está estructurado el texto comprimido.



Identificamos que, en este caso que el archivo es un texto comprimido en RLE, cada tres caracteres están formados sucesivamente por: primero, un caracter nulo, luego el carácter que representaría el número en que el carácter se repite consecutivamente (representado en un valor entero hexadecimal) y por último, el carácter. El carácter que representaría al número de veces que el carácter se repite en el texto plano descomprimido, representa en los valores de la tabla ASCII a un carácter no imprimible, diferente al carácter numérico esperado.

La solución al problema planteado fue primero, agregar tres condicionales en main, después de aplicar todas las operaciones de bits al carácter que se está desencriptando.



Entre las líneas 135 y 143 los condicionales que se implementaron al código son los siguientes:

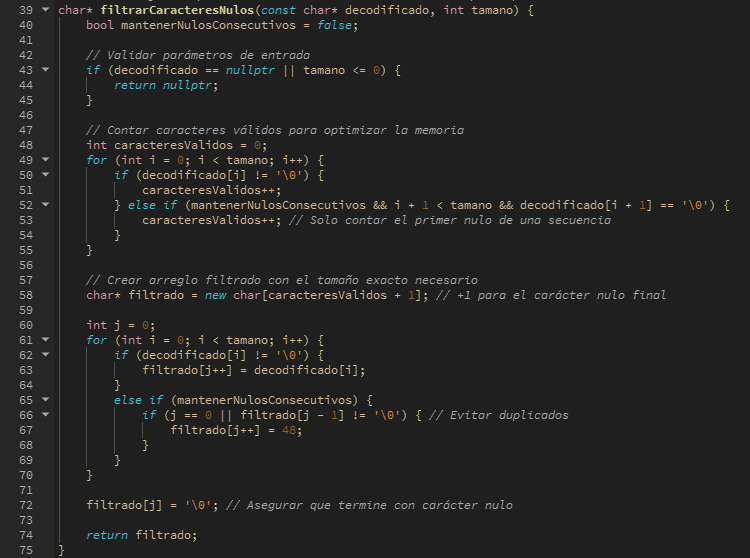
1. Si el byte después de aplicar rotación es el carácter ASCII de valor 0 (es decir, carácter nulo), se mantiene el caracter nulo,
2. Si el caracter es un carácter diferente de 0 pero su valor es menor de 48, pasa a ser imprimible alfanumérico sumándole el valor del carácter a 48, llevándolo al rango de la tabla de valores ASCII por donde están los caracteres de dicho tipo.

Caracter 0x01 (Inicio de encabezado) -> 0x01 + 0x30 -> Caracter ‘1’ (0x31)

1. Si el carácter no cumple con las otras dos condiciones anteriores que se plantearon, el caracter se mantiene en el arreglo.

Después de la implementación de estas tres condiciones, notamos que cada índice en el arreglo final tenía caracteres nulos que se repetían cada 3 caracteres, que separaban el índice y el caracter (ej. /01l/01a/01m…).

Para este detalle se implementó la siguiente función para filtrar los caracteres nulos y así conseguir el texto comprimido en su notación correcta.



La función *filtrarCaracteresNulos()*, consiste de la siguiente lógica:

1. En main(), después de la decodificación del arreglo, se declara un puntero en donde se guardará el arreglo de caracteres comprimido en su notación correcta, que invocará la función en donde se hará dicho filtrado de caracteres nulos



La función se encargará de recibir dec, que sería el arreglo de caracteres que se decodificó inicialmente en el proceso anterior, y el respectivo tamaño de este.

1. Luego de verificar si el arreglo de caracteres decodificado que recibe la función si contenga el texto y no sea nulo, se procede contar los caracteres a considerar para la reserva de la memoria dinámica en donde se guardará el arreglo de caracteres filtrado unas lineas del codigo más abajo (línea 58), la lógica para la preparación de la memoria dinámica es la siguiente:
   * Si el bit que se está tomando es de carácter nulo, no se le reservará el espacio para el arreglo.
   * Si en el bit siguiente al bit tomado es otro carácter nulo, implicando que hay dos caracteres nulos consecutivos, se dejará el espacio solo para solo uno de los dos caracteres nulos.

**Esta parte de la lógica se hizo pensando en los índices de los caracteres comprimidos en LZ78**.

1. Luego de reservar correctamente los espacios a reservar en la memoria dinámica, se procede a realizar el proceso de filtrado bajo la misma lógica de la reserva de caracteres explicado anteriormente.

Finalmente, probando la función con los archivos 1 y 3 que están comprimidos en RLE, conseguimos el texto comprimido en su notación correcta (e imprimible) dando paso a la implementación de la descompresión en dicho formato.



***\*Esta es una prueba con el texto Encriptado1.txt del dataset de prueba***

