**Informe Final Desafío 1**

**Integrantes:** Sara Echeverri Giraldo, Luis Miguel Alzate

**Curso:** Informática II

**Link del video:** https://youtu.be/\_hBMUY5sxXs

**Introducción**

El presente informe documenta de forma detallada el desarrollo técnico del Desafío 1 de Informática II. El objetivo del desafío fue recuperar un texto original que había sido comprimido (con RLE o LZ78) y posteriormente encriptado (rotación de bits y XOR). Se implementó una solución en C++ usando punteros y arreglos dinámicos, se diseñaron pruebas unitarias y se documentaron problemas y decisiones de diseño. A continuación, se presenta el análisis, el diseño, la implementación paso a paso, la explicación de cada función, las pruebas realizadas, las dificultades enfrentadas y el desarrollo final.

1. **Análisis del problema y consideraciones de la solución**

Se recibe un archivo que contiene un texto comprimido y luego encriptado. El reto consiste en:

* Determinar si la compresión fue RLE o LZ78.
* Encontrar el párametro n (rotación circular de bytes) y la clave K del XOR.
* Desencriptar y descomprimir para recuperar el texto original.

***Operaciones conocidas sobre cada byte:***

Primero se aplica rotación circular a la izquierda por n posiciones (1 ≤ n ≤ 7) y luego XOR con una clave K (un byte). Dado que XOR es su propio inverso, la desencriptación es XOR con K seguido de rotación a la derecha por n.

***Consideraciones clave:***

* La pista (fragmento en claro) permite detectar n y K buscando su aparición en el texto comprimido y encriptado.
* La búsqueda puede hacerse por fuerza bruta (probar K y n) o usando un método determinista que reduzca el espacio de búsqueda aprovechando las propiedades del XOR.
* Debe cuidarse la gestión de memoria (no usar STL) y evitar fugas y desbordamientos al manipular índices y concatenaciones.

1. **Esquema de tareas y flujo de trabajo**

Se definió el siguiente flujo de trabajo, que también sirvió como plan de desarrollo de las tareas para los integrantes del equipo:

* ***Tarea 1:*** Lectura de entradas

Leer en modo binario el archivo cifrado y leer la pista (fragmento conocido) en modo texto. Se implementaron funciones dedicadas para cada lectura.

* ***Tarea 2:*** Módulo de rotaciones y XOR

Implementar rotaciones circulares (izquierda y derecha) y funciones que aplican XOR por un byte a todo un buffer.

* ***Tarea 3:*** Búsqueda de parámetros n y K

Implementar dos métodos: **(a)** un método determinista que prueba n (1..7) y calcula K de un solo byte para verificar coincidencia (método pensado con análisis del algoritmo y búsqueda de patrones); **(b)** un método 'bruto' que además itera sobre K (0..255) en caso de necesitar verificación exhaustiva considerando cada posibilidad (método aplicado).

* ***Tarea 4:*** Desencriptado y pruebas unitarias

Una vez detectados n y K, aplicar la desencriptación al archivo completo y verificar mediante pruebas que la pista aparece en el resultado desencriptado.

* ***Tarea 5:*** Descompresión

Implementar descompresión RLE y LZ78 acorde al formato usado en la guía del desafío (bloques de 3 bytes por entrada). Probar ambas y elegir la que produzca texto legible.

* ***Tarea 6:*** Integración y validación final

Integrar el flujo: lectura → detección → desencriptado → descompresión → verificación. Añadir manejo de errores y liberar memoria.

1. **Implementación – paso a paso**

A continuación, se describe detalladamente el proceso de implementación y las decisiones técnicas en cada etapa.

* ***Paso 1:*** Lectura de archivos

***Funciones:*** leerEncriptado(ruta), leerPista(ruta)

* leerEncriptado abre el archivo en modo binario, calcula su tamaño, reserva un buffer dinámico y lee los bytes en memoria.
* leerPista abre el archivo en modo texto, determina el número de caracteres leyendo carácter por carácter y luego rellena un buffer de unsigned char.

***Consideraciones:***

* leerEncriptado usa lectura en bloque para eficiencia.
* leerPista hace lectura carácter a carácter.
* ***Paso 2:*** Rotaciones y XOR

***Funciones:*** rotarIzquierda(valor, n); rotarDerecha(valor, n); encriptarArreglo(…); desencriptarTexto(…).

***Detalles:***

* rotarIzquierda: realiza (valor << n) | (valor >> (8 - n)) y retorna unsigned char. Debe manejar solo 1 ≤ n ≤ 7 (rotación por 0 no cambia nada).
* rotarDerecha: operación inversa: (valor >> n) | (valor << (8 - n)).
* encriptarArreglo: recorre el arreglo de entrada, hace rotarIzquierda(byte,n) y luego XOR con K, guardando el resultado en salida[].
* desencriptarTexto: recorre el texto comprimido y encriptado, XOR con K, y luego rotarDerecha para recuperar el byte original.
* ***Paso 3:*** Detección de n y K

***Funciones:*** encontrarParametros(S, M, C, N, nDetectado, KDetectado, pos); encontrarParametrosPorFormato(...).

***Algoritmo bruto (implementado en encontrarParametros):*** Para cada desplazamiento i razonable en C (0 ≤ i ≤ N − M):

Para cada n en 1..7:

*1.* Calcular K candidato como K = rotarIzquierda(S[0], n) XOR C[i].

*2.* Verificar que para todo j (0..M−1) se cumple rotarIzquierda(S[j], n) XOR K == C[i + j].

Si se cumple, retornar n, K y pos = i.

* ***Paso 4:*** Descompresión

***Funciones:*** descompresionRLE(desencriptado, longitud, &totalSalida), descompresionLZ78(desencriptado, longitud, &tamDinamico).

***Formato manejado:***

* Ambos módulos asumen que el buffer desencriptado está organizado en bloques de 3 bytes por entrada.
* Para RLE: cada bloque contiene cnt, ch donde cnt se toma de la posición i+1 y ch de i+2; el código actual itera i += 3 y expande cnt veces el carácter ch.
* Para LZ78: cada bloque de 3 bytes contiene b1,b2,ascii. Se calcula posicion = b1\*256 + b2 que es un índice al diccionario de entradas previas; valores[actual] = ascii; el algoritmo reconstruye la secuencia siguiendo la cadena de referencias hacia atrás y copiando los caracteres referenciados en orden.

***Consideraciones sobre LZ78 implementado:***

* El algoritmo reserva arreglos posiciones[] y valores[] con tamaño estimado (longitud/3 + 2).
* Para cada entrada se construye temporalmente una lista de índices siguiendo las referencias y después se vuelcan los valores en el buffer de salida dinámico.
* La descompresión puede crecer en memoria; actualizacionArreglo gestiona el crecimiento del buffer de salida.
* ***Paso 5:*** Integración final y verificación

***Flujo integrador propuesto para main:***

* Leer archivo encriptado y pista conocida.
* Generar la versión comprimida de la pista (tanto RLE como LZ78) o asumir que la pista ya tiene formato compatible con las entradas del texto comprimido y encriptado.
* Ejecutar encontrarParametros con la pista comprimida y el ciphertext para obtener n, K y pos.
* Si no se detecta con el método determinista, usar encontrarParametrosPorFormato.
* Desencriptar el texto completo con desencriptarTexto usando n y K detectados.
* Intentar descompresionRLE; si el resultado es legible y coherente, aceptar; de lo contrario, intentar descompresionLZ78.
* Guardar el texto final en archivo y validar que la pista aparezca.

1. **Explicación detallada de funciones**

### rotarIzquierda(unsigned char valor, int n)

Realiza una rotación circular del byte “valor” hacia la izquierda “n” posiciones. Implementación: (valor << n) | (valor >> (8 - n)). Devuelve unsigned char.

Precauciones: n debe estar en 1..7; rotar por 0 no cambia el valor.

### rotarDerecha(unsigned char valor, int n)

Realiza la rotación circular inversa: (valor >> n) | (valor << (8 - n)). Se usa para deshacer la rotación aplicada en la encriptación.

### encriptarArreglo(unsigned char\* entrada, unsigned char\* salida, int tam, int n, unsigned char K)

Aplica la operación de encriptado usada en el desafío: para cada byte realiza rotarIzquierda(byte, n) y luego XOR con K. Parámetros: “entrada” buffer original, “salida” buffer donde se almacena el texto comprimido y encriptadp (pre-asignado), “tam” tamaño, “n” rotación, “K” clave.

Notas: usada principalmente en pruebas unitarias (función probarCaso).

### desencriptarTexto(unsigned char\* textoCifrado, int tamañoCifrado, unsigned char\* textoDesencriptado, int n, unsigned char K)

Invierte la encriptación: para cada byte hace XOR con K y luego rotarDerecha(..., n) para recuperar el byte original. El buffer “textoDesencriptado” debe estar previamente reservado en un bloque de memoria con tamaño “tamañoCifrado”.

### encontrarParametros(unsigned char\* S, int M, unsigned char\* C, int N, int &nDetectado, unsigned char &KDetectado, int &pos)

Busca identificar los parámetros correctos (n y K) que permiten verificar si una secuencia S aparece dentro de un texto cifrado C. Recorre todas las posiciones posibles de inicio en C y, en cada una, prueba con distintos valores de rotación n (del 1 al 7). En cada intento, calcula el valor de la clave K a partir del primer carácter de la secuencia y el correspondiente en el cifrado, y luego comprueba que todos los caracteres de S, al aplicárseles la misma transformación, coincidan con la subsecuencia en C. Si la verificación es exitosa, guarda los parámetros encontrados (n, K y la posición pos) y devuelve true; si no encuentra ninguna coincidencia tras probar todas las combinaciones, devuelve false.

* bool encontrarParametrosPorFormato(unsigned char\* pista, int tamPista, unsigned char\* textoCifrado, int tamCifrado, int &nDetectado, unsigned char &KDetectado, int &pos, int &metodoDetectado)

### Se encarga de probar distintas combinaciones de parámetros de desencriptación (n y K) sobre un texto cifrado, con el fin de encontrar cuáles son los correctos. Para cada intento, primero desencripta el texto y verifica que el resultado tenga un formato válido. Luego, según los primeros bytes, determina si el método de compresión usado fue **RLE** o **LZ78**, y procede a descomprimir. Una vez obtiene el texto descomprimido, busca dentro de él la presencia de una pista específica. Si la encuentra, guarda los parámetros que hicieron posible esa coincidencia (n, K, el método y la posición) y retorna true. Si después de probar todas las combinaciones no se encuentra la pista, la función devuelve false.

### probarCaso(const char\* nombre, unsigned char\* S, int M, int nReal, unsigned char KReal)

Función de prueba que genera un texto comprimido y encriptado corto a partir del texto S usando nReal y KReal; la usa para validar que encontrarParametros detecte los parámetros correctamente. Imprime el texto generado y el resultado de la detección.

### leerEncriptado(const char\* ruta, int &tam)

Abre archivo en modo binario, determina su tamaño con seekg/tellg, aloca un buffer de tamaño “tam” y carga los bytes. Retorna puntero al buffer (dynamic allocation) o nullptr si hay error. Importante liberar la memoria tras su uso.

### leerPista(const char\* ruta, int &tam)

Lee el archivo de la pista en modo texto. Cuenta caracteres, reserva buffer y copia carácter a carácter. Retorna buffer dinámico con los bytes del texto y el tamaño en “tam”.

Limitación: lectura carácter a carácter; para pistas grandes es preferible lectura en bloque.

### imprimirUTF8(unsigned char\* texto, int tam, int limite = 200)

Imprime hasta “limite” bytes del buffer como caracteres (cast a char). Útil para depuración y verificación rápida del contenido desencriptado/descomprimido.

### actualizacionArreglo(unsigned char\*& buffer, int& tamaño, int& capacidad, unsigned char valor)

Implementa append con redimensionamiento automático: si tamaño >= capacidad duplica la capacidad (o establece 16 si era 0), copia el buffer antiguo y agrega el nuevo valor. Es la base para construir buffers dinámicos sin usar STL.

### descompresionRLE(unsigned char\* desencriptado, int longitud, int &totalSalida)

Interpreta el buffer desencriptado como bloques de 3 bytes. Para cada bloque lee “cnt” (desencriptado[i+1]) y “ch” (desencriptado[i+2]) y escribe “cnt” veces “ch” en el buffer de salida. Devuelve un buffer dinámico con la salida y asigna “totalSalida”.

Consideraciones: la función asume formato correcto; no realiza chequeos extensivos de límites y puede fallar si `longitud` no es múltiplo de 3.

### descompresionLZ78(const unsigned char\* desencriptado, int longitud, int &tamDinamico)

Interpreta cada tripleta (b1, b2, ascii) como una entrada del diccionario: posicion = b1\*256 + b2 y el caracter asociado = ascii. Construye arrays `posiciones[]` y `valores[]` y para cada entrada reconstruye la cadena siguiendo referencias hacia atrás (posiciones[posicion]...). La salida se construye con actualizacionArreglo en un buffer dinámico.

Limitación: la implementación reconstruye siguiendo referencias en tiempo lineal por referencia.

1. **Problemas detectados y correcciones aplicadas**

Durante el desarrollo y pruebas se presentaron los siguientes problemas y las acciones de corrección:

* Tipos y conversiones: hubo confusión entre unsigned char, unsigned int y unsigned short cuando se combinaban bytes para formar índices (b1\*256 + b2). Se normalizó el uso de unsigned int para lecturas temporales y unsigned short para índices para evitar overflow o resultados incorrectos.
* Off-by-one y cálculo de índices: en LZ78 se debe tener clara la relación entre la posición del bloque (i/3)+1 y el índice usado en posiciones[]. Se corrigieron cálculos que generaban referencias incorrectas.
* Gestión de memoria: se añadieron borrados de arrays (delete[]) y se verificó que no queden punteros colgantes. Se implementó actualizacionArreglo para controlar ampliaciones del buffer.
* Errores en rotaciones: se detectaron casos donde la rotación usada era incorrecta por uso de tipos que promovían a int y cambiaban el resultado al desplazar. Se corrigió forzando operaciones sobre unsigned char y usando parenthesis apropiados.
* Validación de formatos de entrada: se detectó que, si la longitud del buffer desencriptado no es múltiplo de 3, los módulos de descompresión pueden leer fuera de los límites. Se añadió validación previa y mensajes de error para estos casos.

1. **Consideraciones finales**

Observaciones finales:

* La solución propuesta y el código entregado permiten recuperar el texto original siempre que la pista sea representativa y el formato de compresión siga las convenciones asumidas.
* Documentar detalladamente el formato de compresión (qué contiene cada byte de la tripleta) es clave para robustecer la solución frente a variaciones.

1. **Solución planteada para encontrar parámetros (no aplicada)**

Se pensó un método determinista que prueba n (1..7) y calcula K de un sólo byte para verificar coincidencia.

Una estrategia alternativa que se planteó para hallar los parámetros consistía en aprovechar el hecho de que, sin importar el método de compresión utilizado (RLE o LZ78), el primer byte del archivo comprimido siempre sería un cero. Esto puede justificarse de la siguiente manera: en el caso del algoritmo LZ78, como el diccionario se encuentra vacío al inicio, el índice asociado a la primera letra siempre será cero. Por su parte, en RLE, el primer byte se considera irrelevante o “basura”, ya que el segundo byte es el que define cuántas veces se repite el carácter indicado en el tercer byte.

Partiendo de esta premisa, si sabemos que el primer byte antes del proceso de codificación era cero, podemos simplificar el hallazgo de la clave XOR. Esto se debe a que, al aplicar cualquier número de rotaciones de bits a un valor cero, siempre se obtiene nuevamente cero. En consecuencia, en la ecuación de codificación solo interviene la operación XOR. Recordemos que cualquier número x XOR 0 es igual a x, por lo que el resultado de esta primera posición en el archivo codificado corresponde directamente a la clave XOR utilizada.

De este modo, bastaría con tomar el primer byte del archivo cifrado, interpretarlo como un valor en ASCII, y reconocerlo como la clave XOR. Así, se evitaría realizar las 256 iteraciones posibles para buscar la clave entre todos los valores de un byte, ya que esta se obtiene de forma inmediata.

Una vez conocida la clave XOR, el único parámetro restante por determinar sería la cantidad de rotaciones de bits aplicadas. Para resolver esto, se procedería a descifrar el archivo con la clave hallada y se probarían las siete posibles rotaciones (de 1 a 7). Después de cada intento de decodificación, el resultado se descomprimiría y se verificaría si contiene la pista proporcionada. Si la pista aparece, se habría identificado correctamente el número de rotaciones utilizado en la codificación original.

1. **Enfoques descartados**

El primer enfoque que tuvimos en general fue hacer las operaciones de encriptación en el fragmento de prueba y a partir de ahí compararlo con el fragmento encriptado entero, así reduciendo el volumen de datos para poder hacer las pruebas con mucho menos uso de la memoria que haciendo todo directamente con el texto grande. El problema con este enfoque era la rotación de bits, ya que, al ser una rotación cuadrada, trabajar con cantidades de datos tan diferentes haría que los resultados de esta operación no fueran equivalentes

1. **Conclusiones**

El presente informe (versión final) documenta de forma técnica y detallada el desarrollo del desafío, explicando la lógica detrás de los algoritmos, las funciones implementadas, las pruebas requeridas y las decisiones de diseño. Con estas especificaciones se puede comprender la implementación actual, replicar las pruebas y proponer mejoras o ampliaciones para escenarios de producción.