Imagen que contiene Patrón de fondo

El contenido generado por IA puede ser incorrecto.

**Informe final de entrega**

**Informática II**

**Desafío 1**

**Estudiantes:**

**Jhon Tristancho Muñoz**

**Mateo Arbeláez Cardona**

**Profesor:**

**ANÍBAL JOSÉ GUERRA SOLER**

1. **Introdución**

Este desafío que nos plantearon consistió en recibir un archivo cuyo contenido se encuentra encriptado y posteriormente comprimido. Para resolverlo, mi compañero y yo decidimos que el programa debe ser capaz de leer el archivo en bloques de memoria dinámica, desencriptar su contenido aplicando diferentes métodos y claves posibles, intentar descomprimir el resultado utilizando algoritmos alternativos y finalmente validar si la información obtenida contiene la pista otorgada por el usuario que ejecutaría el programa, entender esta parte es de suma importancia porque nos plantea todo el proyecto.

La alternativa de solución adoptada se basó en un enfoque modular y exhaustivo. Por un lado, se implementaron funciones independientes para la lectura, desencriptación y descompresión, lo cual nos permitió un orden y claridad al manejar este proyecto. Por otro, se diseñó el sistema de prueba de combinaciones que recorre todas las claves y algoritmos posibles hasta dar con la solución correcta.

1. **Pasos que decidimos seguir:**

**Lectura en bloques:** dividir el archivo en fragmentos de memoria dinámica para evitar limitaciones de tamaño.

Abre el archivo con std::ifstream archivo; archivo.open(ruta, std::ios::in | std::ios::binary); . Esto es de tener en cuenta porque lo esta abriendo en modo binario.

Lee el archivo en bloques fijos de 8 KiB, reserva new unsigned char[TAM\_BLOQUE] y escribe en él los bytes leídos. Guarda el puntero a ese bloque en un arreglo dinámico unsigned char \*\*bloques.

El arreglo de punteros bloques y el arreglo tamanios se administran con capacidad inicial y se duplica si se necesita.

Devuelve por referencia (\*bloques\_out, \*tamanios\_out, num\_bloques\_out, total\_bytes\_out) para que el llamador pueda procesar y finalmente liberar con liberar\_bloques.

**Desencriptación**: aplicar los métodos XOR (con claves de 0 a 255) y rotación de bits (con desplazamientos de 0 a 7).

Xor, Crea nuevos bloques con new unsigned char[tamanios[i]] (o TAM\_BLOQUE manteniendo misma división) y aplica salida[j] = entrada[j] ^ clave.

Aplica rotación de bits n en cada byte (por ejemplo: (b >> n) | (b << (8-n)) & 0xFF), con n en 0..7.

**Descompresión:** probar los algoritmos RLE y LZ78 sobre la salida del desencriptado.

RLE, secuencia de dígitos decimales que forman un número n seguido de un símbolo c.

LZ78, Diccionario como dos arreglos paralelos: unsigned char \*\*dict : punteros a entradas (cada entrada es un arreglo dinámico contiguo). Size\_t \*dict\_lens : longitudes de cada entrada.

**Comparación de pista:** verificar si la salida contiene la cadena buscada, que valida la combinación correcta.

1. **Una breve descripción de los algoritmos:**

**Lectura en bloques dinámicos**: se gestionan punteros y memoria para manejar archivos grandes sin cargarlos por completo en un arreglo fijo.

**Desencriptación XOR**: aplica la operación binaria XOR entre cada byte y una clave en el rango 0–255.

**Desencriptación por rotación**: rota los bits de cada byte una cantidad determinada de posiciones, en el rango 0–7.

**Descompresión RLE**: interpreta secuencias en formato <n><caracter>, expandiendo cada carácter tantas veces como indica el número.

**Descompresión LZ78**: utiliza pares <índice, símbolo> para construir dinámicamente un diccionario de secuencias y reconstruir el mensaje.

**Comparador de pista**: recorre el resultado final y detecta la aparición de la palabra clave definida.

1. **Problemas de desarrollo**

**Gestión del Main:** Al trabajar con varios módulos en proyectos separados fue necesario unificar todo en un único archivo principal. Esto se nos complico bastante porque no podíamos unificarlos de una manera correcta y que a la vez pudiera leer los archivos de prueba correctamente.

**Conflictos en Git**: aparecieron problemas de sincronización debido a la longitud de rutas, además de eso tuvimos un poco de dificultad ya que no tuvimos muy claro como manejar bien algunas partes del git bash.

**Manejo de memoria dinámica**: Desde el principio teníamos miedo de que hubieran fugaz de memoria y en una parte del código nos pasó, en la parte del desencriptado al inicio no limpiaba bien todos los arreglos y tuvimos un poco de inconveniente.

**Depuración: Cuando tuvimos dificultad con el Main, tuvimos que depurar para encontrar en que parte no estaba entregando lo adecuado y tuvimos una dificultad porque se nos presentaron unos errores de alta complejidad, ya que hubieron unas validaciones que no se hicieron y corrimos con unos errores difícil de detectar.**

La solución que planteamos tuvo varios momentos, primero se desarrolló la lectura y un desencriptado básico con XOR, luego se añadió la rotación de bits, posteriormente se implementaron los algoritmos de compresión (RLE y LZ78), y finalmente se integró todo en un flujo único con búsqueda de la pista.

En la implementación final se tuvieron en cuenta consideraciones como el control de memoria, límites de tamaño para evitar desbordamientos, modularidad del código para permitir la ampliación futura y la posibilidad de automatizar pruebas exhaustivas sobre todas las combinaciones de claves y métodos. Esto fue muy necesario ya que en los conversatorios con el profesor nos especifico que era necesario.

1. **Resultados obtenidos:**

El programa demostró ser efectivo para resolver el desafío propuesto, encontrando la combinación correcta en un tiempo razonable. La arquitectura modular permitió probar exhaustivamente todas las posibilidades, mientras que las optimizaciones evitaron búsquedas innecesarias.

Finalmente, luego de poder abordar todos los problemas paso a paso. Usando el depurador y dándonos cuenta donde se encontraban las diferentes partes erróneas.

1. **Cambio respecto al preinforme**

nuestra propuesta inicial consistía en leer el archivo encriptado y comprimido y almacenarlo en un único arreglo dinámico de tamaño creciente. Sin embargo, al avanzar en el desarrollo detectamos que esta alternativa era ineficiente: cada vez que el archivo creciera se requerirían nuevas reservas y copias de memoria, lo que aumentaba la complejidad y el riesgo de fugas. Además, al no conocer de antemano el tamaño real del archivo, el uso de un arreglo único podía provocar sobrecostos en rendimiento y memoria.  
Por esta razón decidimos replantear la estrategia de lectura e implementar un esquema de lectura en bloques fijos de 8 KiB. Esta solución permitió manejar archivos de gran tamaño de forma más controlada, ya que cada bloque se reserva solo una vez y se administra a través de dos arreglos paralelos: uno con los punteros a los bloques y otro con los tamaños efectivos leídos. De esta manera, la memoria se usa de forma más eficiente, se facilita la liberación posterior y se asegura que el programa pueda escalar sin limitaciones.