

**Actividad Integradora 1**

Santiago Rodriguez Murialdo

David René Langarica Hernandez

Juan Pablo Cabrera Quiroga

1 de Octubre del 2023

Análisis y diseño de algoritmos avanzados

**Problemática**

Cuando se transmite información de un dispositivo a otro, se transmite una serie sucesiva de bits, que llevan una cabecera, datos y cola. Existe mucha gente mal intencionada, que puede interceptar estas transmisiones, modificar estas partes del envío, y enviarlas al destinatario, incrustando sus propios scripts o pequeños programas que pueden tomar cierto control del dispositivo que recibe la información

Suponiendo que conocemos secuencias de bits de código malintencionado:

* ¿Serías capaz de identificarlo dentro del flujo de bits de una transmisión?
* ¿Podremos identificar si el inicio de los datos se encuentra más adelante en el flujo de bits?
* Si tuviéramos dos transmisiones de información y sospechamos que en ambas han sido intervenidas y que traen el mismo código malicioso, ¿podríamos dar propuestas del código mal intencionado ?

**Instrucciones**

*Parte 1:*

El programa debe analizar si el contenido de los archivos mcode1.txt, mcode2.txt y mcode3.txt están contenidos en los archivos transmission1.txt y transmission2.txt y desplegar un true o false si es que las secuencias de chars están contenidas o no. En caso de ser true, muestra true, seguido de exactamente un espacio, seguido de la posición en el archivo de transmissiónX.txt donde inicia el código de mcodeY.txt

*Parte 2:*

Suponiendo que el código malicioso tiene siempre código "espejeado" (palíndromos de chars), sería buena idea buscar este tipo de código en una transmisión. El programa después debe buscar si hay código "espejeado" dentro de los archivos de transmisión. (palíndromo a nivel chars, no meterse a nivel bits). El programa muestra en una sola línea

dos enteros separados por un espacio correspondiente a la posición (iniciando en 1) en donde inicia y termina el código "espejeado" más largo (palíndromo) para cada archivo de transmisión. Puede asumirse que siempre se encontrará este tipo de código.

*Parte 3:*

Finalmente, el programa analiza que tan similares son los archivos de transmisión, y debe mostrar la posición inicial y la posición final (iniciando en 1) del primer archivo en donde se encuentra el substring más largo común entre ambos archivos de transmisión.

**Explicación de los algoritmos utilizados**

Parte 1 - Encontrar Código Malicioso

Para la parte 1, que consiste en encontrar los fragmentos en donde se encuentra el código malicioso (mcodeY.txt) como un substring de el archivo de transmisión (transmisionX.txt) aplicamos la técnica de manejo de strings de ***Suffix Array***. Esta técnica nos ayuda a construir un arreglo con todos los sufijos posibles de nuestro string principal y los ordena usando la técnica integrada en python de quicksort. Al final nos retorna un arreglo con los índices de cómo terminan siendo ordenados.

Realizamos después una búsqueda iterando por todo el suffix array construido, buscando si el mcode está en un rango de [suffix\_array\_start : suffix\_array\_start + len (mcode)] dentro del texto de transmisión. Si es así, guardamos ese rango dentro de un arreglo y retornamos el arreglo al final de la función.

Finalizamos imprimiendo el arreglo con las posiciones, indicando cuáles son las iniciales y cuáles son las finales.

Esta primera parte tiene una complejidad de *O(n log n)*, ya que la parte que es más compleja es la construcción y ordenamiento del suffix array. Todas las demás funciones de este algoritmo son de tipo *O(n)*.

Parte 2 - Palíndromo más largo

En esta segunda parte, utilizamos la función find\_longest\_palindromela cual se basa en un enfoque de **programación dinámica** para detectar el palíndromo más largo presente en el texto de transmisión. Este algoritmo utiliza una tabla dinámica, denominada table, para almacenar información que permite determinar si una subcadena es un palíndromo o no.

Inicialmente, el algoritmo inicia variables críticas, como maxLength para rastrear la longitud del palíndromo más largo encontrado hasta el momento, start para mantener la posición de inicio del palíndromo actualmente evaluado y i para iterar a través del texto.

El enfoque dinámico se inicia detectando palíndromos de longitud 1 y 2. El algoritmo recorre el texto y verifica si dos caracteres adyacentes son iguales. Si se encuentra un palíndromo de longitud 1 o 2, se actualiza start y maxLength según corresponda.

Luego, el algoritmo entra en un ciclo mientras i avanza a través del texto. Dentro de este ciclo, examina todas las subcadenas posibles de longitud 3 o más para determinar si son palíndromos. Para cada subcadena, verifica si el primer y último carácter son iguales y si la subcadena interior es un palíndromo según la información almacenada en la tabla table. Si estas condiciones se cumplen, se marca la subcadena como un palíndromo potencial y se actualizan start y maxLength si corresponde.

El uso de programación dinámica implica que el algoritmo utiliza información previamente calculada (almacenada en la tabla table) para optimizar la detección de palíndromos. Esto reduce la cantidad de cálculos redundantes y mejora la eficiencia general del algoritmo.

En términos de complejidad, este enfoque tiene una complejidad de tiempo de , donde 'n' es la longitud del texto de transmisión. Esto se debe a que se deben considerar todas las combinaciones posibles de subcadenas en el texto para determinar si son palíndromos. Aunque es un enfoque cuadrático en la teoría, es altamente eficiente en la práctica para textos de longitud razonable y permite encontrar eficazmente el palíndromo más largo en el texto de transmisión.

Parte 3 - Substring más largo común

Para la última parte del programa, era preciso reconocer cuál es el substring más largo que tienen en común ambos archivos de transmisión (comparación de archivos transmisionX.txt y transmisionY.txt).

Para este caso se optó por utilizar programación dinámica partiendo de la idea de ser una alternativa para problemas de optimización al descomponer un problema en problemas más pequeños. Para este caso, precisamente el problema se descompone en encontrar el substring más largo común entre los dos archivos. A grandes rasgos, la tabla de programación dinámica llena cada celda [i][j] con la longitud del substring común más largo e itera a través del texto de ambos archivos, si los caracteres son iguales entonces se incrementa en 1 la longitud y si es mayor que el “longest\_common\_substring” entonces se actualiza el valor anterior.

Por este proceso es que la función para encontrar el substring más largo común entre los dos archivos es de , pues existe la necesidad de tener que llenar la tabla para cada combinación de caracteres en ambos archivos de transmisión. Si bien esta complejidad es cuadrática, se considera que para el tipo de problema es justificable, ya que tiene un enfoque más óptimo en comparación con otro en el que se tengan que comparar todos los posibles substrings; al igual que tiene una eficiencia práctica tomando en consideración el tamaño de los archivos.

Ahora bien, para encontrar las posiciones de dicho substring en los dos archivos de transmisión, era necesario cambiar el enfoque, pues en este caso la necesidad ya era encontrar las ocurrencias del patrón. Por dicha necesidad, el algoritmo KMP era una mejor solución.

Con esta idea en mente, la solución en general genera un “preprocesamiento” (función get\_LPS\_array) para controlar que caracteres deben de ser omitidos, por ello tiene una complejidad *O(n)* en donde n es la longitud del substring encontrado previamente.

La función anterior es parte de la función principal find\_substring\_positions, la cual recorre la cadena de texto (i) y el substring común más largo encontrado anteriormente (j). Por esta razón, esta función también tiene una complejidad de *O(n).* Se considera que esta eficiencia satisface muy bien las necesidades del problema porque no se hacen comparaciones innecesarias, sino que se aprovecha la información ya almacenada en el lps (para no tener que retroceder) con el fin de optimizar la busqueda de las diferentes ocurrencias del susbtring en los archivos. Siendo así como se tiene la complejidad del algoritmo KMP de *O(n+m).*

**Conclusiones**

En este proyecto de análisis y diseño de algoritmos avanzados, hemos abordado la problemática de identificar código malintencionado en transmisiones de datos, detectar palíndromos en los textos de transmisión y encontrar el substring más largo común entre dos archivos de transmisión. Cada una de estas partes se ha abordado con algoritmos específicos y técnicas adecuadas.

Para la detección de código malicioso, hemos empleado la técnica del Suffix Array, que nos permite buscar secuencias de caracteres maliciosos en el flujo de bits de una transmisión. Esta técnica se basa en la construcción y ordenamiento de un arreglo de sufijos, lo que nos permite realizar búsquedas eficientes en el texto de transmisión. La complejidad de este enfoque es razonable, con un rendimiento eficiente incluso para archivos de gran tamaño.

La detección de palíndromos se abordó utilizando programación dinámica, lo que nos permitió encontrar el palíndromo más largo presente en los textos de transmisión. A pesar de su complejidad teórica de O(n^2), este enfoque demostró ser eficiente en la práctica y proporciona resultados precisos.

La última parte se centró en encontrar el substring más largo común entre dos archivos de transmisión utilizando programación dinámica y el algoritmo KMP para la búsqueda de ocurrencias. Este enfoque es efectivo y ofrece una solución óptima para este problema particular.

Para concluir, el proyecto aborda con éxito las problemáticas planteadas y proporciona soluciones sólidas basadas en algoritmos y técnicas de análisis de datos. La combinación de enfoques como Suffix Array, programación dinámica y el algoritmo KMP demuestra la versatilidad y eficiencia de las técnicas de diseño de algoritmos en la resolución de problemas complejos relacionados con la seguridad de la información y el análisis de datos.

**Bibliografía**

* Langmead, B. (n.a). Suffix Array. <https://www.mimuw.edu.pl/~szczurek/TSG2/04_suffix_arrays.pdf>
* GeeksforGeeks. (2023). KMP algorithm for pattern searching. *GeeksforGeeks*. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/