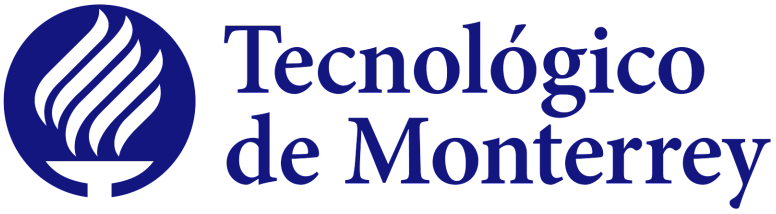
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Santa Fe



Análisis y diseño de algoritmos avanzados
TC2038B, Grupo 601

E1. Actividad Integradora 1

Equipo:

Nombre	Matricula:
Mauricio Tumalán Castillo	A01369288
Oswaldo Ilhuicatzi Mendizábal	A01781988
Santiago Benitez Pérez	A01782813

Noviembre 2023

1. Knut-Morris Pratt: Buscar si el código malicioso está en el transmission

El algoritmo Knut-Morris Pratt es utilizado dentro de nuestra solución para la búsqueda de patrones en las cadenas de texto dadas, siendo en este caso la búsqueda de código malicioso dentro de los archivos de transmisión. Este funciona declarando la función, siendo KPM, la cual toma las cadenas de entrada, una siendo el texto a analizar y la otra siendo el patrón que se desea encontrar.

El algoritmo Knut-Morris Pratt hace un preprocesamiento del patrón a buscar dentro del texto. Este preprocesamiento del patrón consiste en generar un vector, llamémosle V, que contiene información sobre los prefijos sufijos más largos en el patrón. Este vector nos ayuda a saber a qué posición del patrón regresar para seguir comparando el patrón con el texto. Esto nos ayuda a optimizar la búsqueda del patrón en el texto pues con este vector no tenemos que retroceder la búsqueda en el texto.

Una vez que se obtiene el vector V (preprocesamiento del patrón), se utilizan dos variables, **‘i’** y **‘j’**, inicializadas en 0. La variable **‘i’** hace referencia al carácter actual del texto y **‘j’** al carácter actual del patrón. Se inicializa entonces un bucle **while** que recorre el texto de izquierda a derecha, comparando los caracteres correspondientes del texto con los caracteres del patrón. Si hay una coincidencia entre los caracteres, se incrementan tanto ‘i’ y ‘j’ para seguir comparando los caracteres del texto y del patrón.

Si no hay coincidencia en **‘i’** y **‘j’**, se toman dos acciones: Si **‘j’** es 0, implica que estamos en el primer carácter del patrón y no hay coincidencia, aquí solo se incrementa **i**; si **‘j’** no es 0, se utiliza el vector de preprocesamiento **‘V’** para determinar el nuevo valor de **‘j’** que representa el índice del patrón al que hace sentido regresar para empezar a comparar nuevamente (esto ayuda a evitar que se empiece a buscar el patrón en el texto desde el inicio si es que el patrón contiene un sufijo que también sea prefijo), asignando así a **‘j=V[j-i]**’. El ciclo continúa hasta que se recorra todo el texto en búsqueda de coincidencias.

El hecho de que **‘j’** sea igual a la longitud del patrón implica que se ha encontrado una coincidencia dentro del texto, regresando la posición inicial, la cual viene siendo **‘i - j’**. Por otro lado, si **‘i’** llega a ser igual a la longitud del texto, quiere decir que el patrón no se ha encontrado en el código.

La complejidad del algoritmo KMP es **O(n)**, donde 'n' es la longitud de la cadena de texto en la que se está buscando el patrón. Esto se debe a que en el peor caso, cada carácter de la cadena de texto se compara solo una vez, y no hay retrocesos innecesarios. Esto lo convierte en una opción muy eficiente para la búsqueda de patrones en cadenas de texto largas.

2. Manacher: Buscar palíndromo más largo en un transmission

El algoritmo de Manacher es implementado dentro de nuestro código para encontrar la subcadena palindrómica más grande dentro del texto. Se define una función llamada **‘manacher’** que toma una cadena de texto como entrada y regresa una tupla con el índice de inicio y final de la subcadena palindrómica más larga encontrada dentro de la cadena de texto dada. Se realiza un preprocesamiento de texto para transformar la cadena de texto en una con delimitadores de caracteres, estos delimitadores facilitan la identificación de palíndromos dentro de la cadena. De igual manera, se inicializa el vector **P** de enteros, utilizado para almacenar las longitudes de palíndromos encontrados en la cadena.

Dentro de la solución, se inicializan varias variables, dentro de las cuales se incluye **‘maximumLenghtP’**, la cual indica la longitud máxima del palíndromo encontrado hasta el momento, **‘centerIndex’** la cual indica el centro del palíndromo más largo, **‘center’** indicando el centro del palíndromo actual y **‘rightBoundary’**, la cual viene siendo el centro del palíndromo actual.

Se busca correr la cadena **‘T’** en un bucle **‘for’**. En cada iteración se estarán buscando palíndromos centrados con el index **‘i’**. De igual manera se calculará el índice espejo **‘mirror’** del índice actual **‘i’**, reflejando la posición opuesta del mismo con respecto al centro actual.

Si el índice **‘i’** está dentro del límite del palíndromo actual **‘i < rightBoundary**’, se utiliza información del índice espejo y el límite derecho para determinar la longitud del palíndromo en ‘P[i]’. Se realiza una comparación de caracteres para expandir el palíndromo actual comparando los caracteres a la izquierda y derecha de **‘i’**. Si el límite derecho del palíndromo actual se extiende más allá del límite derecho del palíndromo anterior, se actualizan las variables **center** y **rightBoundary** para reflejar el nuevo palíndromo actual. Además, se verifica si el palíndromo actual es el más largo encontrado hasta ahora, y si es así, se actualizan las variables **maximumLengthP** y **centerIndex**.

Después de recorrer todo el texto y encontrar palíndromos, se determina el índice de inicio y final del palíndromo más largo encontrado en el texto. El índice de inicio se calcula como **‘(centerIndex - maximumLengthP) / 2’**, y el índice de final se calcula como **‘(startIndex + maximumLengthP) - 1’.** Finalmente, se devuelve una tupla que contiene estos dos índices.

3. Dinámica: Buscar el *substring* más largo común

El algoritmo de búsqueda del substring común más largo trabaja conforme a la entrada de dos textos. Así, este se basa en la programación dinámica y utiliza una matriz para almacenar información sobre las subcadenas comunes entre ambos archivos. Se emplea la creación de matrices para trabajar sobre este algoritmo: una matriz 2D llamada **matriz** con dimensiones **(m + 1)** x **(n + 1)**, donde **m** y **n** son las longitudes de las cadenas **texto1** y **texto2**, respectivamente. Por otro lado, se utilizan dos bucles anidados **for** para recorrer ambos textos, comparando así los caracteres en las posiciones correspondientes

entre ambos. Si se encuentra una coincidencia, se incrementa el valor en la matriz en la posición **(i, j)** en uno más el valor en la posición diagonal superior izquierda **(i-1, j-1).** Esto indica que se encontró una subcadena común en ambos textos de longitud **matriz[i][j]**.

Utilizando una variable llamada **longitudMax,** se emplea para saber si el valor en la matriz es mayor a la longitud máxima, y si es así, este valor se actualiza.

Una vez que se completan los bucles, se construye la cadena a partir del **texto1** a través de la función **substr**, la cual toma el índice donde comienza la subcadena (**finMax - longitudMax**) y la longitud máxima (**longitudMax**).

Finalmente, este algoritmo tiene una complejidad de tiempo de **O(m * n)**, donde **m** y **n** son las longitudes de las cadenas en los textos.