# E1. Actividad Integradora 1

Análisis y diseño de algoritmos avanzados (Gpo 601)

1/11/2023

A01026364 - Joaquín Badillo Granillo

A01027384 - Gabriel Rodríguez De Los Reyes

A01782767 – Iker Garcia German

Profesor: Víctor Manuel de la Cueva Hernández

## Reflexión de los códigos

### Longest Substring

Este código nos ayuda a encontrar el substring más largo en común entre dos strings, usando programación dinámica. Usando una tabla 2D con nombre **`memo`**, donde **`memo[i][j]`** contiene la longitud del sufijo común más largo de los substrings **`s1[0…i-1]`** y '**`s2[0…j-1]`**'. Si **`s1[0…i-1]`** coincide **`s2[0…j-1]`** entonces tenemos un sufijo de longitud **`memo[i-1][j-1] + 1`**.

La idea principal es encontrar el valor máximo en esta tabla, porque entonces ese va a ser el largo del substring común más largo. La posición (en la tabla) donde encontramos

este valor máximo nos permite determinar el final del substring en los dos strings de entrada.

La complejidad computacional seria del tamaño de ambos strings de entrada ya que primero llenan la tabla y luego la populan depende del tamaño, por ende, encontramos que la complejidad es de **O(nm).**

## Palindromes

Este código implementa el algoritmo de Manacher para encontrar el palíndromo más largo en un string.

Al principio se prepara el string para que tenga **#** entre cada carácter. Luego se le da un vector a **`PalindromeArr`** para luego recorrer **`proString`** (que es nuestro string transformado) desde los posibles centros **`i`** y buscar los o el palíndromo. Si se encuentra uno lo almacena en **`PalindromeArr`**. Una vez finalizado se recorre **`PalindromeArr`** para encontrar el índice del palíndromo más largo, se calculan los índices de inicio **`start`** y **`end`** del palíndromo más largo en la cadena original a partir de los valores obtenidos en **`PalindromeArr`**.

La complejidad computacional es **O(n)** donde n es la longitud del string de entrada. Es la intención e innovación de Manacher, ya que un intento de fuerza bruta en el peor de los casos puede llegar a **O(n^2)**

## Substrings

Este código contiene la declaración de dos funciones en el espacio de nombres **`p1`**. Estas funciones están diseñadas para procesar strings y verificar si un string es un substrings de otra.

La primera función **`preprocessing`** crea una tabla para el pattern dado (llamada tabla de fallos en el algoritmo de KMP). Esta tabla se usa para saber hasta qué punto se debe de retroceder en la búsqueda si se encuentra una discrepancia entre el **`text`** y el **`pattern`**.

**`isSubstring`** utiliza la tabla creada anteriormente para verificar si el **`pattern`** es un substring del **`text`**. El objetivo es avanzar a través de **`text`** mientras se compara con el **`pattern`**, utilizando la tabla creada para evitar comparaciones redundantes (KMP)

Dado a que `n` es la longitud del texto en donde se busca el patrón y `m` es la longitud del patrón que se está buscando. Podemos decir que la complejidad computacional es de **O(n + m).**

## Main

El codigo de main funciona como helper para correr todos los algoritmos empezando por leer los archivos que contiene **`transmissions`** y **`malicious codes`** en vectores separados. Y luego va corriendo todos los algoritmos por partes.