Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Puebla

TC2007B

Análisis y diseño de algoritmos avanzados

(Gpo 401)

E1. Actividad Integradora 1

Profesor

David Augusto Céspedes Hernández

Integrantes

Francisco Rocha Juárez A01730560

Juan Carlos Llanos Ordóñez A01734916

Iker Guerrero González A00830026

Contexto de la problemática:

Transmisiones de datos comprometidos.

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 ?

Reflexión y soluciones aplicadas:

Las problemáticas principales que teníamos que resolver era encontrar una substring dentro de una cadena de carácteres más largos, el segundo problema era encontrar el palíndromo más largo dentro de un texto y el tercero encontrar el substring común más largo entre dos textos. Adicionalmente a las soluciones queda como código comentado la posibilidad de poder encontrar la substring volteada en el texto, esto debido a que no entendimos bien la diferencia entre código espejeado y palindromo, llegando a ser algo ambiguo.

Para solucionar la primer problemática pensamos en ocupar el algoritmo de “Naive” que sería el más intuitivo a simple vista, este algoritmo consiste en ir comparando los caracteres del “pattern” uno por uno contra los del “text”, pero en el caso de que algún carácter no sea igual la comparación empieza de la siguiente posición del carácter del texto, dándonos una complejidad de O(mn). Sin embargo existen algoritmos que nos dan una complejidad menor y son más eficientes a la hora de resolver esta problemática, esto usando el algoritmo KMP

(Knuth-Morris-Pratt), la diferencia de este algoritmo con el previamente mencionado es que este lleva un preprocesamiento y hay algo llamado lista de pi o lps (longest prefix that is also a sufix), este nos permite saber cuántos caracteres debemos de saltarnos al comparar para no volver a analizar los que ya sabemos que coinciden, ahorrandonos tiempo, este algoritmo tiene una complejidad de O(m+n).

Para la solución de código espejeado que sería la adicional, se ocupó el mismo algoritmo de KMP de complejidad O(m+n), solamente que la búsqueda del pattern(substring) se hizo aplicando una función de reverse al vector de caracteres y fue lo que se buscó en el texto.

Y para los dos algoritmos siguientes ambos tienen una complejidad O(n^2) ya que para encontrar las substrings y palíndromos se ocuparon algoritmos similares que involucran ciclos for anidados, como los que se muestran a continuación, como ejemplo ilustrativo:

```
// O(n²) algorithm over myGrid[][]
for(i in myGrid)
    for(j in myGrid[i])
        yieldAction(myGrid[i][j])
```