Actividad Integradora 1:

Tecnológico de Monterrey

Análisis y diseño de algoritmos avanzados
TC2038.601

Docente:

David Augusto Céspedes Hernández

Estudiante:

Carlos Alberto Vega Pérez ITC A01731416

Fecha de Entrega:
08 de septiembre de 2022

La actividad integradora consiste en implementar un programa que sea capaz de:

1) Determinar si en el archivo de transmisión X se encuentra el código malicioso del archivo code Y
2) Determinar el mismo proceso del punto anterior, pero espejeando el contenido de los archivos code Y.
3) Encontrar el substring común más largo entre los archivos de transmisión

Para la primera parte del problema, se optó por implementar dos funciones:

- *functionZ(string pattern, string text)*:

  Se utilizó para poder construir el arreglo Z que diría la coincidencia en cada posición del arreglo.

  El algoritmo utiliza una complejidad de O(n * m), donde *n* representa el tamaño del patrón y *m* el tamaño del texto donde se quiere buscar el patrón

- *checkMatch(vector<int> arrayZ, int patternSize)*

  Una vez contando con el arreglo Z, simplemente debe recorrerse y comprobar si el valor en el arreglo es igual al tamaño del patrón.

  Por lo anterior, se puso alcanzar una implementación de complejidad O(n), donde n es el tamaño del arreglo.

Para la segunda parte, se utilizaron dos funciones más

- *reverse()*

  Una función construida que nos ayuda a obtener la versión espejeada, y que de acuerdo a la documentación tiene complejidad O(n).

- *getPosition(vector<int> arrayZ, int patternSize)*

  Utiliza la misma base que checkMatch. Simplemente busca la coincidencia asumiendo que siempre la encontrará, y una vez que la encuentra, imprime la posición en arrayZ que corresponde a la inicial, y al sumarle el patternSize se obtiene la final

Para la última parte, se implementó una última función:

- *commonSubstr(string wordA, string wordB)*

  Se creó una matriz de tamaño de *wordA+1*, *wordB+1*. Se recorre toda la matriz y en caso de que haya un match en caracteres, se suma el valor del índice anterior más uno. De esta forma, si en el caracter anterior existía un match, se creará un match aún mayor. Finalmente, se guarda la posición del match más alto

  Al tener que recorrer simplemente la matriz, se tiene una complejidad de O(*n***m*), donde *n* y *m* corresponden al tamaño de las palabras.