INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Escuela de Ingeniería

Análisis y diseño de algoritmos avanzados (Gpo 2)

Reflexión individual de la “Actividad Integradora 1”

Presenta:

Carlos Noel Ojeda Angulo, Matrícula A01741085

Guadalajara, Jal., 17 de Septiembre de 2021

Reflexión individual de la "Actividad Integradora 1"

En el presente documento se encuentra mi reflexión acerca de la solución a la que mi compañera Ana Cristina Munguía Romero y yo (Carlos Noel Ojeda Angulo) llegamos para la entrega de la situación problema 1.

Al analizar la problemática planteada comprendimos que necesitamos leer y procesar lo que contienen los archivos *transmission1.txt*, *transmission2.txt*, *mcode1.txt*, *mcode2.txt* y *mcode3.txt* para resolver la partes que conforman la actividad integradora. Es por lo anterior que creamos una función cuyo objetivo es obtener caracteres de los archivos y almacenarlos en una *string* (cadena), dicha función tiene una complejidad temporal O(n), donde n es el número de caracteres que el archivo contiene.

Después de obtener lo que contiene cada archivo y asignarlo a una *string* (cadena), decidimos crear funciones para cada parte. La función *searchByKMP* resuelve la primera parte que consiste en determinar si una string se encuentra dentro de otra, y si es así, dónde. Después de analizar los diversos algoritmos que vimos en clase, concluimos que usar el algoritmo Knuth-Morris-Pratt es la mejor opción de acuerdo al tiempo que disponemos y la eficiencia requerida para procesar las cadenas que representan el contenido de los extensos archivos. Descartando así el *Naïve Algorithm* por tener una complejidad temporal $O(n \cdot m)$ en comparación a la complejidad temporal $O(n + m)$ del algoritmo *KMP*, donde n es el número de caracteres en la cadena que representa el contenido del archivo *mcode*, y m en *transmission*. La función *searchByKMP* necesita de un preprocesamiento para saber cuántos saltos dar, eso se logra con la ayuda de un arreglo obtenido a partir del sufijo cuyos caracteres sean también prefijo. Este preprocesamiento tiene una complejidad temporal $O(m)$ donde m es el número de caracteres en *mcode.* Es importante decir que mediante punteros evitamos el uso de memoria extra.

Para resolver la segunda parte fue necesario encontrar el palíndromo más extenso en *transmission1.txt* y *transmission2.txt.* Esto se podía hacer con un algoritmo de fuerza bruta, pero no es lo ideal debido a que se procesan cadenas extensas (que representan a los archivos), y su complejidad temporal es $O(n^3)$, donde n es el número de caracteres que contienen las cadenas anteriormente mencionadas. En cambio, vimos en clase el algoritmo de *Manacher,* el cual

aprovecha algunas propiedades de los palíndromos y lo resuelve con una complejidad temporal $O(n)$, por lo que implementamos este algoritmo en la función *findLongestPalindrome*. Es importante decir que para evitar problemas con aquellas cadenas pares, se hace un preprocesamiento a la cadena y se añaden *hash* (#) antes y después de cada caracter, este preprocesamiento tiene una complejidad temporal $O(n)$. Es de gran importancia decir que al llamar a la función *findLongestPalindrome* creamos una copia del contenido del parámetro (no se manejó con referencias o punteros) con la finalidad de no modificar la cadena original al añadir los *hash*.

La última parte consistía en encontrar la *substring* (subcadena) más larga en común entre los archivos *transmission1.txt* y *transmission2.txt.* Para resolverla hicimos uso de programación dinámica, muy similar a lo que vimos en clase con *subsecuencias*, usamos una matriz bidimensional e identificamos la diagonal más extensa donde los caracteres coincidían (el final de esta diagonal nos daría la posición del último carácter de la subcadena más larga en común). Es importante decir que nunca pensamos en una solución por fuerza bruta, debido a que creímos que sería más complejo implementarla, algo extraño desde mi punto de vista porque en teoría sería la forma común de pensarlo, sin embargo, es evidente que trasladamos el algoritmo que usa programación dinámica para subsecuencias a subcadenas. La complejidad temporal de la función que implementamos es $O(n \cdot m)$, donde n es la longitud de la cadena que representa *transmission1.txt* y m la que representa *transmission2.txt.*

Para finalizar la reflexión, puedo decir que lo primordial en la actividad fue considerar diversas variables y decidir qué algoritmos implementar. Ya que esto nos permitió desarrollar algoritmos eficientes, en el menor tiempo posible y disfrutar del proceso al hacerlo en *pair programming* (programación en pareja).