TECNOLÓGICO DE MONTERREY
Campus Querétaro

PROFESOR ENCARGADO:

Análisis y diseño de algoritmos avanzados (Gpo 602)

Actividad integradora 1

Presenta Equipo 2
Daniel Gutiérrez Gómez A01068056
Juan Manuel González Ascencio A00572003
Julio César Rodríguez Pérez A01705763

FECHA DE ENTREGA:
01/10/2023

Parte 1

El programa debe analizar si el contenido de los archivos mcode1.txt, mcode2.txt y mcode3.txt están contenidos en los archivos transmission1.txt y transmission2.txt y desplegar un true o false si es que las secuencias de chars están contenidas o no. En caso de ser true, muestra true, seguido de exactamente un espacio, seguido de la posición en el archivo de transmisiónX.txt donde inicia el código de mcodeY.txt

Pasos para hacer primera parte:
Para la primera parte se pensó en realizar un algoritmo que pudiera leer la cantidad de archivo de casos prueba que se nos ha proporcionado, de la manera que se nos propuso, que en este caso es la de analizar si el patrón de cualquier archivo mcode (son 3) se encuentran en cada uno de los archivos de transmisión (son 2), entonces tendríamos la proporción de aplicación de una comparación de que por cada patrón se hará la búsqueda de este patrón para cada 2 de transmisión. Podemos pensarlo como seis iteraciones totales. Antes de pasar con la solución a la relación de 3 a 2, proseguimos a plantear una solución de una relación 1 a 1. Ahora, para la relación 1 a 1, investigamos la mejor manera de hacer la comparación de encontrar un patrón en un string de búsqueda, pero sin antes repasar los temas vistos en clase en los cuales se ven involucrados los strings. En un principio se había optado por la solución en la que se usa un trie, generando el trie por medio del patrón para después pasar a la comparación de los prefijos dentro del string, sin embargo se pensó que la implementación sería algo complicada y que en el peor de los casos la complejidad sería de $O(n^2)$, por lo que decidimos utilizar el algoritmo de KMP en honor a Knuth Morris Pratt. Dicho algoritmo lo que utiliza es una "tabulación" que es preprocesada antes de aplicar la parte del algoritmo de KMP. Esta tabulación es llamada LPS, Longest Preffix Suffix, el cual se encarga de generar patrones que ya han sido encontrados y generar "apuntadores" a las posiciones a las que se requiera mover en caso de llegar al carácter de la posición de la tabulación.

La tabla LPS como dije, funciona de la mejor manera cuando hay muchos patrones repetidos dentro de un patrón de búsqueda, puesto que cuando pasa esto, y se hacen comparaciones, no es necesario regresar al inicio del patrón cuando no se encuentran "matcheos" sino que ya la tabla te indica a qué posición ir. Como sabemos, los mcode son patrones palíndromos en su mayoría, lo cual afecta el algoritmo de KMP y de LPS, puesto que la tabulación no es del todo eficiente, amenos de que se encuentren caracteres repetidos dentro del mismo patrón en un rango, por ejemplo: "aaabc" nos dejaría con la tabulación de [0, 1, 2, 0, 0] que nos servirá bastante en caso de buscar en un string ya que en KMP, si encontramos un caracter que no es match, después de haber encontrado "aa" nos llevaría a una posición anterior en LPS a donde apunta el apuntador de LPS en la posición anterior, en cambio "comoc" no del todo porque tenemos una tabla LPS [0, 0, 0, 0, 1] lo cual casi siempre terminaríamos regresando al inicio del patrón cada vez que hay mismatch, haciendo el algoritmo KMP no tan eficiente. Al final, optamos de todas maneras por realizar el algoritmo de KMP, puesto que es más sencillo de aplicar y tiene una complejidad de O(n+m) en el peor de los casos.

Entonces, después de la realización de la LPS, realizar el algoritmo de KMP, ahora entonces aplicando el algoritmo obtendremos una complejidad en el peor de los casos de O(n+m). Dicho esto, proseguimos a resolver toda la situación pero ahora pensando en que tenemos la relación 3 a 2. El flujo propuesto fue el siguiente: hacer un vector de pares de vectores de string en el que se guardarían por pares los files, es decir el primer par tendría: <[mcode1,txt, mcode2.txt, mcode3.txt], [transmission1.txt, transmission2.txt]> así para todos los casos prueba. Una vez obtenido esto, el siguiente paso fue para cada mcode, abrir el file en ese "path" guardar el patrón y generar su tabla LPS, para después aplicar el algoritmo de KMP con el vector de transmisión (ya leídos los files), las tablas LPS de los 3 mcode, y los 3 mcode, además la posición del apuntador de la iteración en la que nos encontramos. Y ahora, en la función de aplicación de KMP, se itera para cada transmisión y para cada string de mcode, aplicar el KMP. Al final, como sabemos el número de files y casos prueba la complejidad es de: O(6*(n*m)) porque sabemos cantidad de files, sino, sería O(x*y(n*m)) donde "x" y "y" son la cantidad de files de casos prueba.

Algoritmo de KMP con tabla preprocesada

Iterar string de transmisionX.txt hasta que se termine, con tabla LPS

- En caso de que se haga match transmision[i] lpsMcode[j + 1] guardar índice i, avanzar en i y j
- En caso de que el apuntador j sea igual al tamaño del mcode, quiere decir que hay un match: j = 0
- En caso de que no se haga match con LPS del caracter en i, j + 1, se hace un recorrido del índice de j hacía atrás (accediendo a j = lps[j - 1]) solo si j > 0 sino, i se aumenta en uno. Repetir paso 5.3

Ejemplo de entradas y salidas:

Entradas:

- Transmission 01:
  “70616e206465206e6172716e6a6150616e206465206e6172616e6a61”
- Transmission 02:
  “70616e206465206e6172716e6a61
  70616e206465206e6172716e6a61
  70616e206465206e6172716e6a61
  70616e206465206e6172716e6a61
  70616e206465206e6172716e6a61
  70616e206465206e6172716e6a61”
- Mcode 01: “16e61”
- Mcode 02: “737”
- Mcode 03: “17271”

Con estas entradas el resultado que nos da es el siguiente:

```
----- PARTE 1 ------
(False) El archivo transmission01.txt no contiene el codigo contenido en el archivo: mcode01.txt
(False) El archivo transmission01.txt no contiene el codigo contenido en el archivo: mcode02.txt
(True) En la posicion inicial: 18 y final: 22 en el archivo de transmission01.txt en archivo: mcode03.txt
(False) El archivo transmission02.txt no contiene el codigo contenido en el archivo: mcode01.txt
(False) El archivo transmission02.txt no contiene el codigo contenido en el archivo: mcode02.txt
(True) En la posicion inicial: 18 y final: 22 en el archivo de transmission02.txt en archivo: mcode03.txt
(True) En la posicion inicial: 46 y final: 50 en el archivo de transmission02.txt en archivo: mcode03.txt
(True) En la posicion inicial: 74 y final: 78 en el archivo de transmission02.txt en archivo: mcode03.txt
(True) En la posicion inicial: 102 y final: 106 en el archivo de transmission02.txt en archivo: mcode03.txt
(True) En la posicion inicial: 130 y final: 134 en el archivo de transmission02.txt en archivo: mcode03.txt
(True) En la posicion inicial: 158 y final: 162 en el archivo de transmission02.txt en archivo: mcode03.txt
```

Parte 2

Suponiendo que el código malicioso tiene siempre código "espejeado" (palíndromos de chars), sería buena idea buscar este tipo de código en una transmisión. El programa después debe buscar si hay código "espejeado" dentro de los archivos de transmisión. (palíndromo a nivel chars, no meterse a nivel bits). El programa muestra en una sola línea dos enteros separados por un espacio correspondientes a la posición (iniciando en 1) en donde inicia y termina el código "espejeado" más largo (palíndromo) para cada archivo de transmisión. Puede asumirse que siempre se encontrará este tipo de código.

Pasos para realizar la segunda parte:
Para esta actividad teníamos que obtener el palíndromo más largo en cada archivo de transmisión por lo cual para realizarla primero buscamos las diferentes opciones que teníamos para cumplir el objetivo.
La primera opción que encontramos fue empezar checando cada carácter y comparar las posiciones que tenía adelante y atrás para compararlos y comprobar si es palíndromo y de qué tamaño, esta idea tiene un problema, ya que deberíamos tener una función para los palíndromos nones y otra para los pares. Esta solución tiene una complejidad de O(n^2) por lo cual consideramos que tomaría un tiempo elevado cuando fueran cantidades más grandes de archivos.
La segunda solución que encontramos fue implementar el algoritmo de Manacher, el cual es un algoritmo utilizado para encontrar todas las subcadenas palindrómicas de una cadena de caracteres en tiempo lineal, lo cual quiere decir que tiene una complejidad de O(n), por esta razón escogimos usar esta solución. Los pasos que seguimos para implementar el algoritmo fueron los siguientes:

1. Creamos una nueva cadena de caracteres a partir de la cadena original, insertando un carácter especial entre cada par de caracteres. Por ejemplo, si la cadena original es "abba", la nueva cadena sería "#a#b#b#a#".
2. Creamos un arreglo llamado "P" de la misma longitud que la nueva cadena, inicializado con ceros.
3. Definimos una variable "C" que representa el centro de la subcadena palindrómica más a la derecha que se ha encontrado hasta el momento, y una variable "R" que representa el borde derecho de dicha subcadena.
4. Recorrimos la nueva cadena de izquierda a derecha, calculando el valor de "P[i]" para cada posición "i" de la siguiente manera:

a. Si "i" está dentro de la subcadena palindrómica más a la derecha encontrada hasta el momento, se utiliza el valor de "P[j]" para la posición simétrica "j" de "i" respecto a "C". Es decir, "P[i] = P[j]" si "i + P[j] < R", y "P[i] = R - i" en caso contrario.
b. Si "i" está fuera de la subcadena palindrómica más a la derecha encontrada hasta el momento, se utiliza el valor cero.
c. Se expande la subcadena palindrómica centrada en "i" todo lo posible, actualizando los valores de "C" y "R" si se encuentra una subcadena palindrómica más a la derecha que la actual.

5. Finalmente, recorrimos el arreglo "P" buscando el valor mayor en el cual se encontraría la cadena más grande para luego mandar las posiciones en las que se está el palíndromo, eliminando los caracteres especiales.

Ejemplo de entradas y salidas:

En este caso únicamente tenemos como entradas los archivos de transmission, en los cuales se encuentra el substring palíndromo más largo.
Ejemplos de transmission:

- Transmission 01: “70616e206465206e6172716e6a6150616e206465206e6172616e6a61”
- Transmission 02: “70616e206465206e6172716e6a61”

Ejemplo de salidas esperadas:

```
----- PARTE 2 ------
15 25 (para archivo de transmission01)
15 25 (para archivo de transmission02)
15 17 (para archivo de transmission11)
16 19 (para archivo de transmission12)
35 52 (para archivo de transmission21)
33 39 (para archivo de transmission22)
1 7 (para archivo de transmission31)
49 55 (para archivo de transmission32)
6 8 (para archivo de transmission41)
9 11 (para archivo de transmission42)
11 13 (para archivo de transmission51)
23 25 (para archivo de transmission52)
1 11 (para archivo de transmission61)
7 17 (para archivo de transmission62)
```

Recordemos que imprime la posición inicial y final de donde se encuentra el palíndromo más largo.

Parte 3

Finalmente el programa analiza qué tan similares son los archivos de transmisión, y debe mostrar la posición inicial y la posición final (iniciando en 1) del primer archivo en donde se encuentra el substring más largo común entre ambos archivos de transmisión.

Para esta parte 3, se nos pide una tarea un poco más sencilla, la cual es realizar la comparación de dos strings, pero en este caso se trata de encontrar la subcadena más larga común entre el par de archivos de transmisión. Al igual que las otras dos partes, analizamos y pensamos que lo mejor era llevar un flujo de aplicación de lectura para después llegar a la aplicación del algoritmo seleccionado. Similarmente como en la primera parte, se optó por realizar un vector con pares, pero en este caso los pares son solo strings, "transmission1.txt" y "transmission2.txt". Una vez creado esto, se crearon todos los pares de transmisiones, para después leer los documentos, para después convertir en strings separados y luego se aplicaría para cada par de substrings la comparación de strings y se nos regresaría la subcadena común más larga.

Ahora, una vez llegado a este punto, como equipo decidimos buscar maneras de realizar esta comparación, y llegamos a la conclusión de que lo más sencillo de implementar y que es muy eficiente es la programación dinámica, la cual, va comparando caracter por caracter, y al final se van rellenando las posiciones donde sí existían un "matcheo" con un contador, además de que se va revisando una casilla anterior y una diagonal izquierda anterior, con el objetivo de regresar la subcadena más larga. Entonces, una vez hallada la propuesta, seguimos haciendo este algoritmo, sin embargo, con la tabulación únicamente obtenemos la longitud de la subcadena con longitud máxima. Como es necesario encontrar la posición inicial y la posición final, se optó por el uso de vector de pares con enteros, en los que se insertarán pares de posiciones cada vez que se encuentre un nuevo máximo de subcadena. Al final, obtendremos un vector de pares con las posiciones iniciales y finales donde se hallaron subcadenas iguales. Ahora, con este vector de pares de posiciones, se prosiguió a calcular la máxima diferencia que pueda existir entre posición inicial y posición final, quedando así al final, la posición inicial y posición final de la subcadena encontrada en los dos archivos de mayor longitud. La complejidad de este algoritmo es de O(n*m) donde "n" es longitud del primer string y "m" es la longitud del segundo string.

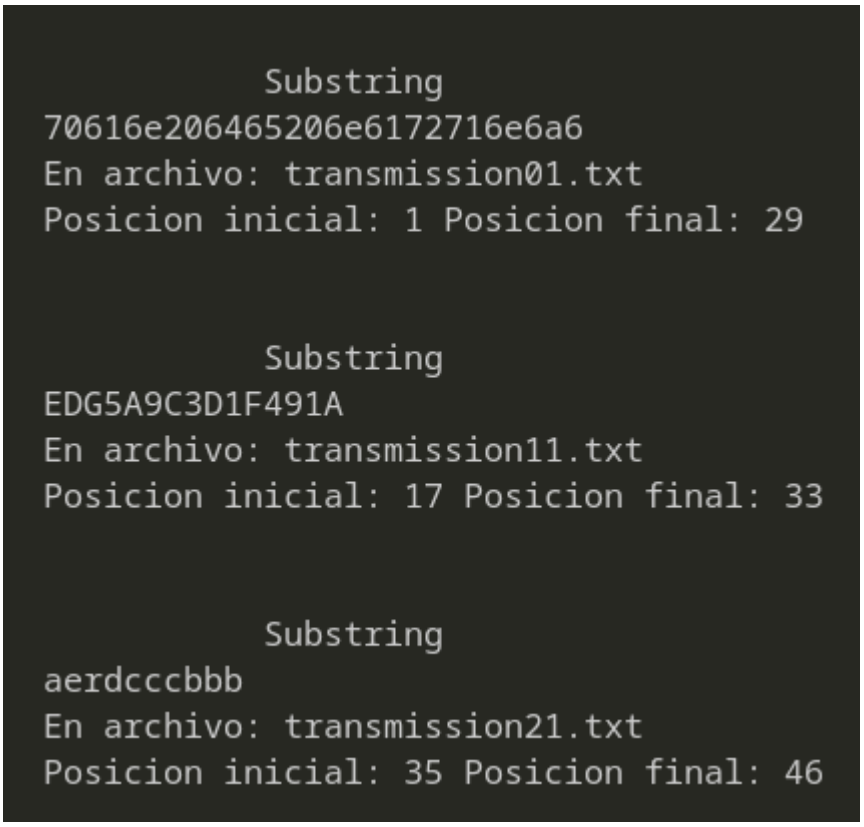
Ejemplo de entradas y salidas:

En este caso únicamente tenemos como entradas los archivos de transmission, en los cuales se encuentra el substring común más largo.

Ejemplos de transmission:

- Transmission 01: “70616e206465206e6172716e6a6150616e206465206e6172616e6a61”
- Transmission 02: “70616e206465206e6172716e6a61”

Ejemplo de salidas esperadas:



Conclusiones

Daniel Gutiérrez Gómez:

Me quedo con la parte del análisis del problema, para poder pasar al diseño del algoritmo para después pasar a una implementación eficiente y rápida, no sin antes pasar por la validación del equipo y llegar a un consenso. Como parte final, puedo decir que existe una gran variedad de soluciones para un problema, sin embargo, siempre existirán algoritmos más eficientes que son los mejores, puesto que el tiempo de ejecución para gran cantidad de datos puede llegar a ser la gran diferencia en los beneficios de tiempos de ejecución. Además, la importancia del saber el manejo de strings mediante el uso de algoritmos avanzados, con temas como la programación dinámica, algoritmo de Manacher y algoritmo de KMP, entre otros.

Julio César Pérez Rodríguez:

La actividad nos hizo como equipo buscar la mejor forma de solucionar la problemática presentada, me ayudó a entender mejor el KMP y como implementarlo para resolver diversas problemática, también me hizo aprender a detalle sobre el algoritmo de Manacher para tener una solución más eficiente para resolver el problema, me di cuenta que hay diferentes maneras para resolver un problema y la más correcta depende de la situación en la que nos encontramos y la disponibilidad de recursos.

Juan Manuel González Ascencio.
En esta actividad logré aprender que es importante saber cómo resolver problemas para diseñar algoritmos, ya que aunque existan miles de maneras de resolver problemas, siempre se puede encontrar e implementar la mejor. También hemos aprendido la importancia de abordar problemas complejos con un enfoque estructurado y sistemático.

Referencias bibliográficas

GeeksforGeeks. (2023). KMP algorithm for pattern searching. GeeksforGeeks. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

GeeksforGeeks. (2021). Manacher s Algorithm Linear Time Longest Palindromic Substring part 1. GeeksforGeeks. https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/

GeeksforGeeks. (2023a). Trie insert and search. GeeksforGeeks. https://www.geeksforgeeks.org/trie-insert-and-search/