DESCRIPCIÓN DE LOS ALGORITMOS

El proyecto se encuentra dividido en dos paquetes:

1. View: La cual contiene un JFrame 'UploadFile' el cual como su nombre lo indica permite seleccionar el archivo con los casos de prueba, además, muestra una breve instrucción de cómo realizar el proceso planteado anteriormente.

2. Business: contiene 3 clases:
    a. Scanner: contiene los métodos necesarios para leer el archivo.
    b. Kmp: contiene los métodos que usa el programa para procesar y validar los casos de prueba.
    c. Main: contiene el código del programa (método principal).

A continuación, se define una descripción y el costo de los métodos implementados por el programa.

Método *split* de la clase Kmp

@param pattern - Cadena de texto a buscar
@return Retorna un ArrayList de tipo String el cual contiene en cada posición una subcadena de la cadena original (la que fue recibida como parámetro) debido a que fue dividida según los '*' presentes en ella (funcionamiento similar al método de Java).

Descripción a nivel de ejecución:

1. Se recibe un objeto de tipo String el cual corresponde al patrón que se desea buscar.
2. Se crea un ArrayList de tipo String.
3. Se crea un objeto de tipo String y se inicializa vació.
4. Se crea un variable de tipo Int ('m') y se inicializa con el tamaño del patrón (cadena recibida).
5. Se recorrerá toda la cadena, mientras se recorre se pregunta si en la posición en que va se encuentra un '*', si es así, se guardara en el ArrayList los caracteres anteriores a esta posición y la subcadena tomará el valor de vacía; si no, en la *subcadena* se concatenara cada carácter diferente a un '*' y en caso de que se llegue al final del patrón se guardara esta *subcadena*.

```
static ArrayList<String> split(String pattern) {
    ArrayList<String> split = new ArrayList<>();
    String subCadena = "";
    int m = pattern.length(), i=0;
    while (i < m) {
        if (pattern.charAt(i) == '*') {
            split.add(subCadena);
            subCadena = "";
        } else {
            subCadena += pattern.charAt(i);
            if (i == pattern.length() - 1) {
                split.add(subCadena);
            }
        }
        i++;
    }
    return split;
}
```

Costo:

Algoritmo	**OE**
ArrayList<String> split = new ArrayList<>();	1
String subCadena = "";	1
int m = pattern.length(), i=0;	3
while (i < m) {	1
if (pattern.charAt(i) == '') {*	3
split.add(subCadena);	2
subCadena = "";	1
} else {	
subCadena += pattern.charAt(i);	4
if (i == pattern.length() - 1) {	3
split.add(subCadena);	2
}	
}	
i++;	2
}	
return split;	1

- El while se repite *m* veces donde *m corresponde al tamaño del patrón*, por lo tanto, tenemos:

$$T(m) = 6 + \left(m * (1 + 14)\right)$$

$$T(m) = 6 + 15m$$

$$\boldsymbol{T(m) = \theta(m)}$$

Método *table* de la clase Kmp

El objetivo de la tabla (precalculada) de fallo es no permitir que cada carácter de la cadena 'texto' sea examinado más de 1 vez. El método clave para lograr esto, consiste en haber comprobado algún fragmento de la cadena **donde se busca** con algún fragmento de la cadena **que se busca**, lo que nos proporciona en qué sitios potenciales puede existir una nueva coincidencia, sobre el sector analizado que indica fallo.

Dicho de otro modo, partiendo del texto a buscar, elaboramos una lista con todas las posiciones, de salto atrás que señalen cuanto se retrocede desde la posición actual del texto a buscar.

Por tanto, esta tabla se confecciona con la distancia que existe desde un punto en la palabra a la última ocurrencia (de la 0ª letra de la palabra) distinta de la primera vez que aparece, y mientras sigan coincidiendo, se marca la distancia, cuando haya una ruptura de coincidencia se marca 0 o un valor previo ya calculado anteriormente, y así sucesivamente hasta terminar con el texto.

La tabla tiene su primer valor es fijo (0), de modo que la función de fallo empieza siempre examinando el 2 carácter del texto.

@param pattern- Cadena de texto a buscar.
@return Retorna un ArrayList de tipo Integer que representa la tabla de función de fallo.

Descripción a nivel de ejecución:

1. Se crea una variable de tipo Int (‘*m*’) y se inicializa con el tamaño del patrón.
2. Se crea un ArrayList de tipo Integer (‘*tablePos*’).
3. Se agrega siempre en la primera posición cero (0) el valor cero (0).
4. Se crean dos variables de tipo Int:
   - ✓ *temp*: Se usará como puntero en el patrón y para asignar valores en el ArrayList.
   - ✓ *i*: Se usará como puntero en el patrón.
5. Se recorrerá el patrón desde la posición uno (1) hasta *m.*
   - ✓ Se agrega en la posición ‘i’ lo que contiene la posición ‘i’-1.
   - ✓ *temp* se inicializa con el valor de la posición ‘i’.

   - ✓ Mientras no exista un patrón (igualdad) entre los caracteres que se están comparando y ‘*temp*’ sea mayor a cero y menor a (*i+1)*, se procederá a calcular un valor nuevo para *temp* a partir de los valores ya establecidos en la tabla con anterioridad hasta que no se cumpla la condición que inicio este proceso. Esto se realiza con el fin de indicar al método kmp hacia qué posición del *patrón* debe

empezar a comparar después de haber encontrado una diferencia de caracteres en las cadenas *texto* y *patron*.

- ✓ Si los caracteres de la posición '*temp*' y la posición 'i' son iguales, se actualiza el valor de la '*tablaPos*' en la posición 'i' con '*temp*'+1.

```
static ArrayList<Integer> table(String pattern) {
    int m = pattern.length();
    ArrayList<Integer> tablePos = new ArrayList<>();
    tablePos.add(0);
    int temp, i = 1;
    while (i < m) {
        tablePos.add(tablePos.get(i - 1));
        temp = tablePos.get(i);
        while (temp > 0 && pattern.charAt(i) != pattern.charAt(temp)) {
            if (temp <= i + 1) {
                tablePos.set(i, tablePos.get(temp - 1));
                temp = tablePos.get(i);
            }
        }
        if (pattern.charAt(i) == pattern.charAt(temp)) {
            tablePos.set(i, temp + 1);
        }
        i++;
    }
    return tablePos;
}
```

Costo:

Algoritmo	OE
int m = pattren.length();	1
ArrayList<Integer> tablePos = new ArrayList<Integer>();	1
tablePos.add(0);	2
int temp, i = 1;	1
while (i < m) {	1
tablePos.add(tablePos.get(i - 1));	3
temp = tablePos.get(i);	2
while (temp > 0 && pattren.charAt(i) != pattren.charAt(temp)) {	7
if (temp <= i + 1) {	2
tablePos.set(i, tablePos.get(temp - 1));	5
temp = tablePos.get(i);	2
}	
}	

if (patron.charAt(i) == pattren.charAt(temp)) {	5
tablePos.set(i, temp + 1);	4
}	
i++;	2
}	
return tablePos;	1

Nota: El número total de veces que el ciclo interior es ejecutado es menor o igual al número de veces que se puede reducir temp.

En el peor caso *temp* llegara a tener el valor de *m-2,* y el ciclo se ejecutará m-3 veces (porque el valor se obtiene es con *temp-1)* con el fin de guardar en la posición *i (índice de ciclo externo)* el valor contenido en la posición cero de la tabla de fallos, es decir, esto indica que en el carácter *i* no hay un "patrón" con respecto al resto de la cadena, por tanto, esto obligaría al método kmp empezar a buscar desde el inicio del patrón nuevamente.

Ejemplo:

Patrón: xxxxxxxxxxy

Tabla de fallo: [0][1][2][3][4][5][6][7][8][9][0]

temp llega a valer 9

Para calcular el valor de la última posición, el ciclo ira disminuyendo *temp* de uno en uno, hasta que la operación tablePos.get*(temp – 1)* dentro del if retorne cero (indicando la primera posición de la tabla el cual tendrá siempre el valor de 0).

El costo de este while está representado en color azul.

$$T(m) = 7 + m - 1(1 + 23) + 7 + m - 3(7 + 16)$$

$$T(m) = 14 + m - 1(24) + m - 3(23)$$

$$T(m) = 14 + 24m - 24 + 23m - 69$$

$$T(m) = 47m - 79$$

$$T(m) = \theta(m)$$

El tiempo de ejecución para calcular la función de la tabla de fallos puede ser acotada por los incrementos de la variable *i,* que es $\theta(m)$.

Método *kmp* de la clase Kmp

@param text - Cadena de texto en la cual se va a buscar.
@param pattern- Cadena de texto a buscar.
@param pos – Entero que indica la posición en la cual empezará a buscar.
@return Si se encuentra el patrón se retorna la posición de la primera coincidencia en la cadena ´texto´ de la sección donde fue encontrado, en caso contrario retorna -1.

Descripción a nivel de ejecución:

- ✓ Se crean dos variables de tipo Int, una se inicializa con el tamaño de la cadena en la que se buscará (*'n'*) y la otra con el tamaño de la cadena a buscar ('*m*').
- ✓ Se crea un variable de tipo boolean y se inicializa en true, esta servirá para saber cuándo empezó existir coincidencia entre las cadenas, con el final de almacenar esta posición.
- ✓ Se crea un ArrayList de tipo Integer y se inicializa con la tabla de saltos (conocida como tabla de fallos) que después al examinar entre si las cadenas se utilizan para hacer saltos cuando se localiza un fallo.
- ✓ Se crean dos variables de tipo Int:
  - ✓ '*seen*' corresponde al puntero que se moverá por la cadena '*patron*' y se inicializa en 0.
  - ✓ '*i*' corresponde al puntero que se moverá por la cadena '*texto*' y se inicializa con el valor de '*pos*' (la primera vez *pos* vale 0, la segunda un valor mayor que cero '*x*' y la tercera un valor mayor que *x* y así sucesivamente… todo en caso de que el patrón contenga '*' y cada sub-patrón se haya encontrado*).*

- ✓ Se recorre la cadena 'texto' desde la posición 0 hasta n.

  Entonces ambas cadenas comienzan a compararse usando el puntero de avance de cada una, si ocurre un fallo (*la variable seen es mayor a cero y los caracteres que se están comparando de ambas cadenas son diferentes*) en vez de volver a la posición siguiente a la primera coincidencia, se actualizara el valor de seen con el valor de una posición especifica de la tabla de fallos hasta que la condición que inicio este proceso no se cumpla.

  El objetivo es determinar si se puede seguir comparando los caracteres de *texto* con los de *patrón* desde una posición no tan lejana en la que se iba dentro del patrón, o en el peor caso se deba volver a empezar.

Mientras existan coincidencias el puntero de avance (*seen*) del 'patrón', se va incrementando y pueden pasar dos cosas:

- ✓ Se actualiza la variable *stratInt* con el valor de la primera coincidencia siempre y cuando la variable boolean '*start*' se encuentre en true.
- ✓ Si *seen* llega al final de la cadena '*patrón*' se retorna la posición donde se dio la primera coincidencia en el proceso valido (cuando se encontró el patrón).

- ✓ En caso de que el valor del puntero de avance de la cadena '*texto*' llegue al final de esta y no se haya encontrado el patrón, se devuelve -1.

```
static int kmp(String text, String pattern, int pos) {
    int n = text.length(), m = pattern.length();
    boolean start = true;
    int startInt = 0;
    ArrayList<Integer> tab = table(pattern);
    int seen = 0, i = pos;
    while (i < n) {

        while(seen>0 && text.charAt(i)≠pattern.charAt(seen)){
            seen = tab.get(seen - 1);
            start = true;
        }
        if (text.charAt(i) == pattern.charAt(seen)) {
            if (start) {
                start = false;
                startInt = i - seen;
            }
            seen++;
        }
        if (seen == m) {
            return startInt;
        }
        i++;
    }
    return -1;
}
```

Costo:

Algoritmo	**OE**
int n = text.length(), m = pattren.length();	4
boolean start = true;	1
int startInt = 0;	1
ArrayList<Integer> tab = table(pattren);	$\theta(m)$
int seen = 0, i = pos;	2
while (i < n) {	1
while (seen > 0 && text.charAt(i) != pattren.charAt(seen)) {	7
seen = tab.get(seen - 1);	3
start = true;	1
}	
if (text.charAt(i) == pattren.charAt(seen)) {	5
if (start) {	1
start = false;	1
startInt = i - seen;	2
}	
seen++;	2
}	
if (seen == m) {	1
return startInt;	1
}	
i++;	2
}	
return -1;	1

Nota (while interno): *El número total de veces que el ciclo interior es ejecutado es menor o igual al número de veces que se puede reducir seen; pero seen comienza desde cero y es siempre mayor o igual a cero, por lo que dicho número es menor o igual al número de veces que seen es incrementado, el cual es menor o igual a n (tamaño del texto).*

Se pudo comprobar imprimiendo un contador de repeticiones dentro de este *while* en función de la *i* y en el peor caso se ejecuta *n* veces.

Demostración:

Texto: xxxxxxxx… x (tamaño: 357)
Patrón: xy

```
seen: 1 i: 355 pos: 0 start: 354
Contador del while interno: 355 valor de la i: 355
seen: 1 i: 356 pos: 0 start: 355
Contador del while interno: 356 valor de la i: 356
seen: 1 i: 357 pos: 0 start: 356
Contador del while interno: 357 valor de la i: 357
xy NO
```

El costo de este while está representado en color azul.

$$T(n,m) = 10 + n(1+21) + 7 + n(7+4) + m$$

$$T(n,m) = 10 + 22n + 7 + n(11) + m$$

$$T(n,m) = 10 + 22n + 7 + 11n + m$$

$$T(n,m) = 33n + m + 17$$

$$T(n,m) = \theta(n+m)$$

Donde *n* es el tamaño de la cadena donde se buscará y *m* el tamaño del patrón.

Método searchPatterns de la clase Main

@param *pathFile*– Cadena de texto que representa la ruta del archivo a procesar.
@return Imprime el patrón más un 'SI' si ese patrón se encuentra en la cadena 'texto' o un 'NO' para el caso contrario.

Clases implementadas:

PrintWriter: Esta clase implementa todos los métodos de impresión encontrados en PrintStream.

- ✓ Println(""): Es útil para mostrar texto por pantalla y luego deja un salto de línea (como si presionara enter). Se caracteriza por ser más rápido que el System.out.println().
- ✓ Flush(): Vacía el buffer de salida permitiendo imprimir su contenido en pantalla.

Scanner: Esta clase implementa todos los métodos que permiten realizar la lectura de archivos.

- ✓ HasNext(): Este método devuelve un tipo de dato booleano que sirve como un indicador para saber si todavía hay tokens para iterar.
- ✓ Next(): Lee hasta donde encuentra un espacio vacío y retorna la cadena (funcionamiento por filas).

Descripción a nivel de ejecución:

1. Se procederá a leer todo el archivo, mediante el uso de un while y la implementación del método hasNext().
2. Se crea una variable de tipo Int ('*entradas'*) y se inicializa con la cantidad de líneas a leer para un conjunto de casos de prueba.
3. Se captura la cadena en la que se buscara y se guarda en un objeto de tipo String llamado '*texto*'.
4. Se proceden a leer cuantos casos de prueba (patrones a buscar) se indicaron con la variable '*entradas*' (serian entradas-1). Para esto se implementó un while que se ejecutara hasta que '*entradas*' sea igual a cero.
5. Se crea un objeto de tipo String '*patron*' y se inicializa con un patrón a buscar.

---------- *COMIENZA PROCESO DE BUSQUEDA* ----------

6. Se crea un ArrayList de tipo String y se inicializa con el retorno del método Split de la clase *Kmp*.

7. Se crea una variable de tipo Int (‘*indice*’) y se inicializa en 0. Esta variable será usada para indicar las posiciones en las que se debe empezar la búsqueda de cada sub-patrón presente en el ArrayList.
8. Se crea una variable de tipo Boolean (‘*esta*’) y se inicializa en true, esta nos indicara si ‘*patrón*’ se encuentra en ‘*texto*’.
   - ✓ En true -> esta
   - ✓ En false -> no esta
9. Se recorre todo el ArrayList (‘*split*’) y se determina si el sub-patrón (en la posición ‘*i*’) no es vacío, se procede a ser buscado con el método kmp de la clase Kmp:

   En caso de encontrarlo el método retorna la posición del primer carácter del sub-patrón en la cadena ‘*texto*’ y se calcula el nuevo índice donde se empezará a buscar el siguiente sub-patrón (siguiente posición del ArrayList). Si no lo encuentra, el método retornara menos uno (-1) provocando que la variable booleana ‘*esta*’ cambie su valor a *false* y se rompa el ciclo (break).

10. Una vez se termine el recorrido del for (ya sea porque ‘i’ es igual al tamaño del ArrayList o porque se haya roto el ciclo) se procede a revisar el valor de la variable booleana ‘*esta*’:
    - ✓ Si es true se imprime “patrón SI”.
    - ✓ Si es false se imprime “patrón NO”.

---------- *FINALIZA PROCESO DE BUSQUEDA* ----------

11. Se realizará todo lo anterior mencionado para cada caso de prueba.
12. Todo el programa se encuentra encerrado en un try catch con el fin de capturar posibles excepciones que puedan presentarse en la lectura del archivo.

```
public void searchPatterns(String pathFile) {
    PrintWriter so = new PrintWriter(System.out);
    try {
        Scanner scanner = new Scanner(pathFile);
        while (scanner.hasNext()) {
            int entradas = Integer.parseInt(scanner.next()) - 1;
            String texto = scanner.next();
            while (entradas > 0) {
                String patron = scanner.next();
                String[] split = patron.split( regex: "\\*");
                int indice = 0;
                boolean esta = true;
```

```
                    for (String s : split) {
                        if (s.length() > 0) {
                            int pos = Kmp.kmp(texto, s, indice);
                            if (pos < 0) {
                                esta = false;
                                break;
                            }
                            indice = (pos + s.length());
                        }
                    }

                    if (esta) {
                        so.println(patron + " SI");
                    } else {
                        so.println(patron + " NO");
                    }
                    entradas--;
                }
            }
        } catch (Exception e) {
            so.println("Error: ".concat(e.getMessage()));
        }
        so.flush();
    }
```

Costo:

Algoritmo	**OE**
PrintWriter so = new PrintWriter(System.out);	1
try {	
Lector lector = new Lector(rutaArchivo);	1
while (lector.hasNext()) {	1
int entradas = Integer.parseInt(lector.next()) - 1;	3
String texto = lector.next();	1
while (entradas > 0) {	1
String patron = lector.next();	1
ArrayList<String> split = Cadena.split(patron);	$\theta(m)$
int indice = 0;	1
boolean esta = true;	1
for (int i = 0; i < split.size(); i++) {	Inicialización - 1 Condición - 2 Incremento - 2

if(split.get(i).length()>0){	3
int pos = Cadena.kmp(texto,split.get(i), indice);	5
if (pos < 0) {	1
esta = false;	1
break;	1
}	
indice = (pos + split.get(i).length());	5
}	
}	
if (esta) {	1
so.println(patron + " SI");	1
} else {	
so.println(patron + " NO");	1
}	
entradas--;	2
}	
}	
} catch (FileNotFoundException fe) {	1
so.println("Error en Lectura del archivo");	1
so.println(fe);	1
} catch (IOException ie) {	1
so.println("Error en el Lector");	1
so.println(ie);	1
}	
so.flush();	1

Nota: El costo se concentra en la sección de código que analiza las cadenas, y no, en la lectura de archivo y captura de excepciones.

Funcionamiento del for

A continuación, se especifican 3 casos:

- **Caso 1:** Tendrá que buscar tantos sub-patrones como caracteres diferentes de un '*' en el patrón, cuyo número está representado por:

$$k = \frac{tamañodelpatron}{2}$$

Tenemos un ejemplo:

*v*o*y*a*s*a*c*a*r*c*i*n*c*o*

- Tamaño del patrón: 29
- *K* = Cantidad de caracteres diferentes de '*': 14

Para este caso: 14 sub-patrones, que al sumar sus tamaños dará la mitad del tamaño original del patrón.

- **Caso 2:** Cuando el patrón solo tiene un asterisco (*):

  Tenemos un ejemplo:

  voyasa*carcinco

  - Tamaño del patrón: 15
  - *K* = Cantidad de sub-patrones diferentes de '*': 2

  Para este caso: 2 sub-patrones, que al sumar sus tamaños dará (*tamaño del patrón – 1)* porque: *voyasa (**6** caracteres) y carinco (**8** caracteres)* donde un total de ***14***.

- **Caso 3:** Cuando el patrón no contiene asteriscos (*) la búsqueda se realizará de forma lineal respecto al tamaño del patrón

  Tenemos un ejemplo:

  voyasacarcinco

  - Tamaño del patrón: 15
  - *K = 0*

Se definen ciertos criterios:

- ✓ Cuando se busca en la cadena *texto* el índice nunca se regresará, si no, siempre avanzara de una posición menor a una mayor hasta el fin de la cadena.
- ✓ Cuando se encuentra un sub-patrón, el siguiente se busca a partir de una posición optima (posición de la primera coincidencia + el tamaño del sub-patrón), haciendo que la búsqueda en *texto* sea lineal.

Por tanto, la solución es lineal en la sumatoria de la longitud de los patrones + la longitud del texto.

$$\sum_{i=0}^{k} subpatron_i * 18 + n$$

Donde:

- ✓ ***subpatron$_i$*** *representa el tamaño de los sub-patrones.*
- ✓ ***k*** *es el número de subpatrones resultantes del método Split.*
- ✓ ***18*** *es el costo de operaciones elementales del ciclo.*
- ✓ ***n*** *es el tamaño de la cadena donde se va a buscar.*

Solución la sumatoria:

Al sumar los tamaños de los sub-patrones se pueden dar como resultado:

Caso 1: *k =* $\frac{m}{2}$

$$18 * \sum_{i=0}^{m/2} subpatron_i = \frac{18 * m}{2}$$

Caso 2: *k = 2*

$$18 * \sum_{i=0}^{2} subpatron_i = 18 * (m - 1)$$

Caso 3: *k = 0*

$$18 * \sum_{i=0}^{0} subpatron_i = 18 * m$$

Se puede concluir que la sumatoria se encuentra acotada por $\theta(m)$, se plantea la solución matemática:

$$T(n,m) = 11 + m + 18 \sum_{i=0}^{k} subpatron_i + n$$

$$T(n,m) = 11 + m + 18m + n$$

$$T(n,m) = 11 + 19m + n$$

$$\left(\sum_{i=0}^{k} subpatron_i + n \right) = \theta(m+n)$$

$$T(n,m) = \theta$$

CONCLUSIÓN

1. La implementación del algoritmo de búsqueda Knuth-Morris-Pratt (KMP) permite maximizar el rendimiento del programa debido a que utiliza información basada en los fallos previos, aprovechando la información que la propia palabra a buscar contiene de sí (sobre ella se pre-calcula una tabla de valores), para determinar donde podría darse la siguiente existencia, sin necesidad de analizar más de 1 vez los caracteres de la cadena donde se busca.

2. Al realizar la alteración en el KMP que permite retornar la posición optima en la cual se debe empezar a buscar el siguiente sub-patron (en caso de que el patrón contenga '*') garantiza que el costo en función de la cadena donde se busca sea lineal.

BIBLIOGRAFÍA

- Baase, Sara y Gelder Allen Van., Algoritmos computacionales. Introducción al análisis y diseño, PEARSON EDUCACION, México, 2002.

- Wikipedia. Algoritmo Knuth-Morris-Pratt [en línea]. Disponible en: http://es.wikipedia.org/wiki/Algoritmo_Knuth-Morris-Pratt

- Valenzuela Ruz, Víctor. Manual de análisis de algoritmos [en línea]. Disponible en: http://colabora.inacap.cl/sedes/ssur/Asignatura%20Indtroduccion%20a%20la%20Programacn/An%C3%A1lisis%20de%20Algoritmo/Manual-Analisis%20de%20Algoritmos_v1.pdf

- Tushar Roy, Knuth–Morris–Pratt(KMP), Pattern Matching(Substring search)[video en linea], https://www.youtube.com/watch?v=GTJr8OvyEVQ

- Uva Online Judge. 12785 - Emacs Plugin, Disponible en: https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=4650