Tarea #3
Algoritmos de búsqueda en texto

Algoritmos y Estructuras de Avanzadas/ Magíster en Ciencias de la Computación
Departamento Ciencias de la Computación y
Tecnologías de la Información

Universidad del Bío-Bío

Profesor: Gilberto Gutiérrez

Diego Arteaga
Rodrigo Sandoval

Otoño 2025

1. Introducción

2. Implementación de algoritmos de búsqueda

En la presente sección se muestra como se implemento y como se compara el rendimiento de ejecución de los algoritmos de búsqueda en texto `KMP` y `RABIN-KARP`, para los cuales se tuvieron las siguientes consideraciones previas.

Un alfabeto $\sum$ {0,1,2,3,4,5,6,7,8,9}.

Casos en que el texto (T) tenga diferentes largos.

Casos en que el patrón (P) tenga diferentes largos.

2.1. Algoritmo KMP

Para la implementación del algoritmo `KMP` se utiliza una función de pre-procesamiento denominada `computeLPS`, la cual construye la tabla de prefijos-sufijos más largos (LPS).

El algoritmo presenta una complejidad O(n+m) en el peor caso, por lo cual es de un carácter eficiente para patrones largos y textos grandes. De igual manera, presenta un mejor funcionamiento en patrones con alta repetición.

A continuación se presenta el algoritmo `KMP` implementado.

```java
public static List<Integer> kmpSearch(String text, String pattern) {
        List<Integer> matches = new ArrayList<>();
        int n = text.length();
        int m = pattern.length();

        if (m == 0) return matches;

        int[] lps = computeLPS(pattern);
        int i = 0; // Índice para text
        int j = 0; // Índice para pattern

        while (i < n) {
            if (text.charAt(i) == pattern.charAt(j)) {
                i++;
                j++;
            }
            if (j == m) {
                matches.add(i - j);
                j = lps[j - 1];
            } else if (i < n && text.charAt(i) != pattern.charAt(j)) {
                if (j != 0) {
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }
        return matches;
    }
```

Figura 1: Implementación KMP

A continuación se presentan los tiempos de ejecución del algoritmo `KMP` tanto con un patrón de tamaño fijo (5) y un texto variables, como un texto fijo (100.000) y un patrón variable.

Tamaño Texto	**KMP (ns)**
1000	147080
10000	651180
50000	1049900
100000	610380
500000	3249730
1000000	6435360

Cuadro 1: Tiempos de ejecución del algoritmo `KMP` con texto variable y patrón de tamaño 5

Tamaño Patrón	**KMP (ns)**
2	711810
5	756330
10	881170
20	672370
50	679420
100	765440
200	648120
500	647670
1000	663190

Cuadro 2: Tiempos de ejecución del algoritmo `KMP` con texto fijo (100.000) y patrón variable

2.2. Algoritmo Rabin-Karp

Para la implementación del algoritmo `Rabin-Karp` se utilizo un hashing rodante que permite comparar eficientemente subcadenas, ademas de considerar una base numérica de 10, apta para dígitos del 0 al 9, así mismo, un número primo de 101 para reducir colisiones.

El algoritmo posee una complejidad promedio de O(n+m), con un peor caso de O(nm), por lo cual es muy eficiente para patrones pequeños y para la detección de múltiples patrones.

A continuación se presenta el algoritmo `Rabin-Karp` implementado.

```java
public static List<Integer> rabinKarpSearch(String text, String pattern, int prime) {
        List<Integer> matches = new ArrayList<>();
        int n = text.length();
        int m = pattern.length();

        if (n < m || m == 0) return matches;

        int base = 10; // Tamaño del alfabeto
        int h = 1;
        int patternHash = 0;
        int windowHash = 0;

        // Calcular h = base^(m-1) mod prime
        for (int i = 0; i < m - 1; i++) {
            h = (h * base) % prime;
        }

        // Calcular hash del patrón y primera ventana
        for (int i = 0; i < m; i++) {
            patternHash = (base * patternHash + (pattern.charAt(i) - '0')) % prime;
            windowHash = (base * windowHash + (text.charAt(i) - '0')) % prime;
        }

        for (int i = 0; i <= n - m; i++) {
            // Verificar si los hashes coinciden
            if (patternHash == windowHash) {
                // Verificar caracter por caracter
                boolean match = true;
                for (int j = 0; j < m; j++) {
                    if (text.charAt(i + j) != pattern.charAt(j)) {
                        match = false;
                        break;
                    }
                }
                if (match) {
                    matches.add(i);
                }
            }

            // Calcular hash para la siguiente ventana
            if (i < n - m) {
                windowHash = (base * (windowHash - (text.charAt(i) - '0') * h) +
                              (text.charAt(i + m) - '0')) % prime;

                if (windowHash < 0) {
                    windowHash += prime;
                }
            }
        }
        return matches;
    }
```

Figura 2: Implementación Rabin-Karp

A continuación se presentan los tiempos de ejecución del algoritmo `Rabin-Karp` tanto con un patrón de tamaño fijo (5) y un texto variables, como un texto fijo (100.000) y un patrón variable.

Tamaño Texto	Rabin-Karp (ns)
1000	118310
10000	613050
50000	1321960
100000	678410
500000	3486370
1000000	7104310

Cuadro 3: Tiempos de ejecución del algoritmo `Rabin-Karp` con texto variable y patrón de tamaño 5

Tamaño Patrón	Rabin-Karp (ns)
2	668720
5	716260
10	1429650
20	855830
50	1351530
100	858680
200	837580
500	832910
1000	851460

Cuadro 4: Tiempos de ejecución del algoritmo `Rabin-Karp` con texto fijo (100.000) y patrón variable

Durante el apartado de resultados experimentales 4 se llevará a cabo una comparativa de ambos algoritmos.

3. Descripción del algoritmo

4. Resultados experimentales

En esta sección se comparara el rendimiento de los algoritmos `KMP` y `Rabin-Karp`, para los cuales se responderán las siguientes preguntas:

i Que ocurre si varía el largo del texto manteniendo fijo el tamaño del patrón: ¿Cual presenta mejor rendimiento?.

ii Que pasa si se mantiene fijo el tamaño del texto y se varía el tamaño del patrón.

Escenario 1: Texto variable, patrón fijo (tamaño 5)			Escenario 2: Texto fijo (100,000), patrón variable		
Tamaño Texto	**KMP (ns)**	**Rabin-Karp (ns)**	**Tamaño Patrón**	**KMP (ns)**	**Rabin-Karp (ns)**
1000	147080	118310	2	711810	668720
10000	651180	613050	5	756330	716260
50000	1049900	1321960	10	881170	1429650
100000	610380	678410	20	672370	855830
500000	3249730	3486370	50	679420	1351530
1000000	6435360	7104310	100	765440	858680
			200	648120	837580
			500	647670	832910
			1000	663190	851460

Cuadro 5: Tiempos de ejecución de los algoritmos KMP y Rabin-Karp para dos escenarios distintos.

Para el escenario 1 se puede considerar que en textos pequeños entre mil y diez mil, `Rabin-Karp` es más rápido, ya que presenta un menor tiempo de ejecución. A partir de los cincuenta mil caracteres, `KMP` presenta una superación a `Rabin-Karp`, finalmente con textos muy grandes entre quinientos mil y un millón, `KMP` mantiene una ventaja.

Para el escenario 2 se puede indicar que `KMP` es consistentemente más rápido que `Rabin-Karp` cuando el patrón crece. Por otro lado, **Rabin-Karp** empeora significativamente con patrones más grandes, esto para diez, cincuenta y 100 caracteres. Sin embargo, `KMP` mantiene un tiempo estable, independientemente del tamaño del patrón.

En respuesta a la pregunta `i` se puede indicar que para textos pequeños entre mil y diez mil, **Rabin-Karp** es más eficiente, mientras que para textos mas grandes entre cincuenta mil y un millón, `KMP` es más rápido. Con esto se puede indicar que `Rabin-Karp` es óptimo apra búsquedas con patrones cortos de texto y no muy extensos.

En respuesta a la pregunta `ii` se puede indicar que `KMP` siempre es más eficiente, especialmente con patrones grandes. Mientras que `Rabin-Karp` empeora cuando el patrón crece, debido a la sobrecarga del hash rodante. Por lo cual es posible indicar que `KMP` es la mejor opción si el patron varía en tamaño.

5. Conclusiones

6. Anexos

6.1. Código implementado

GitHub