Análisis de Algoritmos
Tarea 2

Alonso Bustos, Gabriela Escalona, Constanza Cristinich

9 de Junio, 2025

1. Casos de Prueba (Gabriela)

Propiedad importante:

- **Simetría:** La distancia de edición es simétrica, es decir:

$$d(S,T) = d(T,S)$$

  Esto se cumple porque cada `Insert` desde S a T equivale a una `Delete` desde T a S, y viceversa.

Cadenas seleccionadas: `CAT`, `MAP`, `CUT` y `CUP`.
La matriz de distancia es simétrica, porque $d(S,T) = d(T,S)$, pero se consideran todos los valores como casos de pruebas del código:

	CAT	MAP	CUT	CUP
CAT	0	4	2	4
MAP	4	0	6	4
CUT	2	6	0	2
CUP	4	4	2	0

- $d(\texttt{CAT}, \texttt{MAP}) = d(\texttt{MAP}, \texttt{CAT}) = 4$: Se mantiene la letra `'A'` del centro, pero `'C'` se reemplaza por `'M'` y `'T'` por `'P'`. Cada cambio implica 2 operaciones (`Delete` + `Insert`), totalizando 4.
- $d(\texttt{CAT}, \texttt{CUT}) = d(\texttt{CUT}, \texttt{CAT}) = 2$: Solo hay una letra diferente: `'A'` frente a `'U'`. Se reemplaza con una `Delete` y una `Insert` (2 operaciones).
- $d(\texttt{CAT}, \texttt{CUP}) = d(\texttt{CUP}, \texttt{CAT}) = 4$: Se conserva la `'C'`, pero cambian las otras dos letras: `'A'` por `'U'` y `'T'` por `'P'` (2 diferencias $\times$ 2 operaciones = 4).
- $d(\texttt{MAP}, \texttt{CUT}) = d(\texttt{CUT}, \texttt{MAP}) = 6$: No hay coincidencias de letras en posiciones equivalentes. Se eliminan las 3 letras de `MAP` e insertan las de `CUT`, es decir, 3 `Delete` + 3 `Insert` = 6.
- $d(\texttt{MAP}, \texttt{CUP}) = d(\texttt{CUP}, \texttt{MAP}) = 4$: Solo coincide la letra `'P'` al final. Las otras letras requieren 2 cambios, lo que da un total de 4 operaciones.
- $d(\texttt{CUT}, \texttt{CUP}) = d(\texttt{CUP}, \texttt{CUT}) = 2$: Coinciden las letras `'C'` y `'U'`. Solo cambia `'T'` por `'P'` $\rightarrow$ 1 `Delete` + 1 `Insert` = 2.

2. Fórmula Recursiva (Alonso)

Para calcular la distancia de edición mínima entre dos cadenas S y T usando solo los operadores de `Delete` e `Insert`, definimos la función recursiva $D(S, T, i, j)$ donde:

- D es la función de distancia de edición, solo utilizando inserciones y eliminaciones
- S es la cadena de texto de origen
- T es la cadena de texto de destino
- i es un indice de la cadena de origen (empieza desde el final)
- j es un indice de la cadena de destino (empieza desde el final)

Caso base

Los casos base ocurren cuando hemos procesado completamente una de las dos strings:

$$\begin{aligned} D(S,T,-1,j) &= j+1 && \text{se insertan los } j+1 \text{ caracteres restantes en T} \\ D(S,T,i,-1) &= i+1 && \text{se eliminan los } i+1 \text{ caracteres restantes en S} \end{aligned}$$

Caso recursivo

El caso recursivo/general donde ambas strings tienen caracteres por procesar:

- Si $S[i] = T[j]$, entonces no se requiere operación y se retrocede un carácter en ambas:

$$D(S,T,i,j) = D(S,T,i-1,j-1)$$

- Si $S[i] \neq T[j]$, elegimos el mínimo entre la recursión que corresponde a eliminar $S[i]$ de S, o insertar el carácter $T[j]$ en S:

$$D(S,T,i,j) = 1 + \text{mín}(D(S,T,i-1,j), D(S,T,i,j-1))$$

Caso general:

Finalmente, la función recursiva seria:

$$D(S,T,i,j) = \begin{cases} j+1 & \text{si } i < 0 \\ i+1 & \text{si } j < 0 \\ D(S,T,i-1,j-1) & \text{si } S[i] = T[j] \\ 1 + \text{mín} \begin{cases} D(S,T,i-1,j) \\ D(S,T,i,j-1) \end{cases} & \text{si } S[i] \neq T[j] \end{cases}$$

3. Implementación (Alonso y Constanza)

3.1. Programa recursivo

```
int editDistanceRecursive(string &s1, string &s2, int i, int j){
    if (i < 0) return j + 1;
    if (j < 0) return i + 1;

    if (s1[i] == s2[j])
        return editDistanceRecursive(s1, s2, i - 1, j - 1);
    else
        return 1 + min(editDistanceRecursive(s1, s2, i - 1, j),
        editDistanceRecursive(s1, s2, i, j - 1));
}
```

3.2. Programa recursivo con Memoización

```
int editDistanceMemo(string &s1, string &s2, int i, int j, vector<vector<int>> &memo){
    if (i < 0) return j + 1;
    if (j < 0) return i + 1;

    if (memo[i][j] != -1) return memo[i][j];

    if (s1[i] == s2[j])
        return memo[i][j] = editDistanceMemo(s1, s2, i-1, j-1, memo);
    else
        return memo[i][j] = 1 + min(editDistanceMemo(s1, s2, i-1, j, memo),
                                    editDistanceMemo(s1, s2, i, j-1, memo));
}
```

3.3. Programación dinámica

```
int editDistanceDP(const string& s1, const string& s2) {
    int n = s1.length(), m = s2.length();
    vector<vector<int>> M(n + 1, vector<int>(m + 1));

    for (int i = 0; i <= n; ++i) M[i][0] = i;
    for (int j = 0; j <= m; ++j) M[0][j] = j;

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (s1[i - 1] == s2[j - 1]) {
                M[i][j] = min({M[i - 1][j] + 1,
                               M[i][j - 1] + 1,
                               M[i - 1][j - 1]});
            } else {
                M[i][j] = min(M[i - 1][j] + 1,
                               M[i][j - 1] + 1);
            }
        }
    }
    return M[n][m];
}
```

3.4. Programación dinámica optimizada

```
int editDistanceDPOptimized(const string& s1, const string& s2) {
    int n = s1.length(), m = s2.length();
    if (m < n) return editDistanceDPOptimized(s2, s1); // intercambiamos

    vector<int> anterior(m + 1), actual(m + 1);
    for (int j = 0; j <= m; ++j) anterior[j] = j;

    for (int i = 1; i <= n; ++i) {
        actual[0] = i;

        for (int j = 1; j <= m; ++j) {
            if (s1[i - 1] == s2[j - 1]) {
                    actual[j] = min({anterior[j] + 1,
                    actual[j - 1] + 1,
                    anterior[j - 1]});
            } else {
                    actual[j] = min(anterior[j] + 1,
                                actual[j - 1] + 1);
            }
        }
        swap(anterior, actual);
```

```
    }
    int resultado = anterior[m];
    return resultado;
}
```

4. Complejidad (Alonso y Constanza)

4.1. Programa recursivo

La función recursiva pura utilizada genera un árbol de llamadas donde cada nivel corresponde a un paso en el procesamiento de las strings. Además, en el peor caso ($S[i] \neq T[j]$), cada llamada generará 2 llamadas recursivas adicionales de forma que el tamaño máximo del árbol será de $n + m$.

- **Complejidad temporal:** $O(2^{n+m})$, donde n y m corresponden a los tamaños de las strings. Esto es así porque en el peor caso habrá un árbol recursivo de tamaño $n + m$ con 2 hijos por cada nodo.
- **Complejidad espacial:** $O(n + m)$, ocurre cuando las string son recorridas completamente.

4.2. Programa recursivo con Memoización

Al utilizar memoización con ayuda de una matriz, evitamos re-computar subproblemas ya solucionados.

- **Complejidad temporal:** $O(n \cdot m)$, ya que hay $n + m$ subproblemas únicos/distintos y cada uno se soluciona en tiempo constante.
- **Complejidad espacial:** $O(n \cdot m)$, que corresponde al tamaño de la matriz de memoización y además esta el stack de llamadas recursivas de $O(n+m)$ (pero esta es dominada por la memoria de la matriz).

4.3. Programación dinámica

La función implementa el algoritmo de distancia de edición mediante programación dinámica, utilizando una matriz de tamaño $(n + 1) \cdot (m + 1)$.

- **Complejidad temporal:** $O(n \cdot m)$, se recorre toda la matriz y en cada celda se realiza una cantidad constante de operaciones.
- **Complejidad espacial:** $O(n \cdot m)$, se almacena toda la matriz de subproblemas.

4.4. Programación dinámica optimizada

La función usa la misma lógica que la anterior, pero en vez de una matriz, usa solo dos filas de la matriz en memoria, aprovechando que solo se necesita la fila anterior para calcular la actual.

- **Complejidad temporal:** $O(n \cdot m)$, igual que en la versión no optimizada.
- **Complejidad espacial:** $O(\text{mín}(n, m))$, ya que solo se almacenan dos vectores del tamaño de la cadena más corta.

5. Protocolo Experimental (Gabriela)

Con el objetivo de validar el comportamiento de distintas implementaciones del algoritmo `EditDistanceDelete` se diseña el siguiente protocolo experimental en el cual se busca comparar el tiempo de ejecución y el uso de memoria entre las distintas versiones de este.

Implementaciones a comparar

Se consideran cuatro implementaciones para el problema EditDistance:

1. `editDistanceRecursive` – versión recursiva sin mejoras.
2. `editDistanceMemo` – versión recursiva con memoización.
3. `editDistanceDP` – versión con programación dinámica clásica.
4. `editDistanceDPOptimized` – versión con uso de espacio optimizado.

Parámetros del experimento

- Se seleccionan cuatro fragmentos de texto de longitud variable
- A partir de estos textos, se generan todas las combinaciones ordenadas de pares distintos (12 en total).
- Para cada implementación, se mide el **tiempo de ejecución** y el **uso de memoria** al calcular la distancia de edición entre cada par de cadenas seleccionadas. Debido a que la versión `Recursive` presenta una complejidad exponencial, su ejecución se limita exclusivamente a los pares de textos más breves. Para los pares que involucran cadenas más largas (como frases, párrafos o capítulos), la versión recursiva no se ejecuta por razones prácticas de tiempo y consumo de recursos. Las otras implementaciones (`Memo`, `DP` y `DPOptimized`) se aplican a todos los pares, incluyendo los más extensos, y permiten observar con mayor claridad las diferencias de eficiencia en escenarios más realistas.

Métricas evaluadas

- **Tiempo de ejecución** (en milisegundos), medido con funciones de reloj del sistema o bibliotecas estándar (`chrono`).
- **Uso de memoria**, especialmente para comparar la versión dinámica estándar frente a la optimizada.

Hipótesis a validar

- La versión recursiva pura será la más lenta, con complejidad exponencial.
- La versión con memoización y las versiones dinámicas tendrán un rendimiento significativamente mejor.
- La versión dinámica optimizada usará menos memoria manteniendo un rendimiento similar en tiempo.

6. Experimentación (Gabriela)

Textos escogidos: Los extractos utilizados para el text3 y text4 fueron generados mediante inteligencia artificial. A continuación, se describen los textos empleados:

- **text1:** Una palabra simple: `experimento` (11 bytes).
- **text2:** Una frase de tres palabras: `problema edit distance` (23bytes).
- **text3:** Un párrafo técnico simulado (232 bytes).
- **text4:** Un texto extenso de estilo académico (1.8 kB).

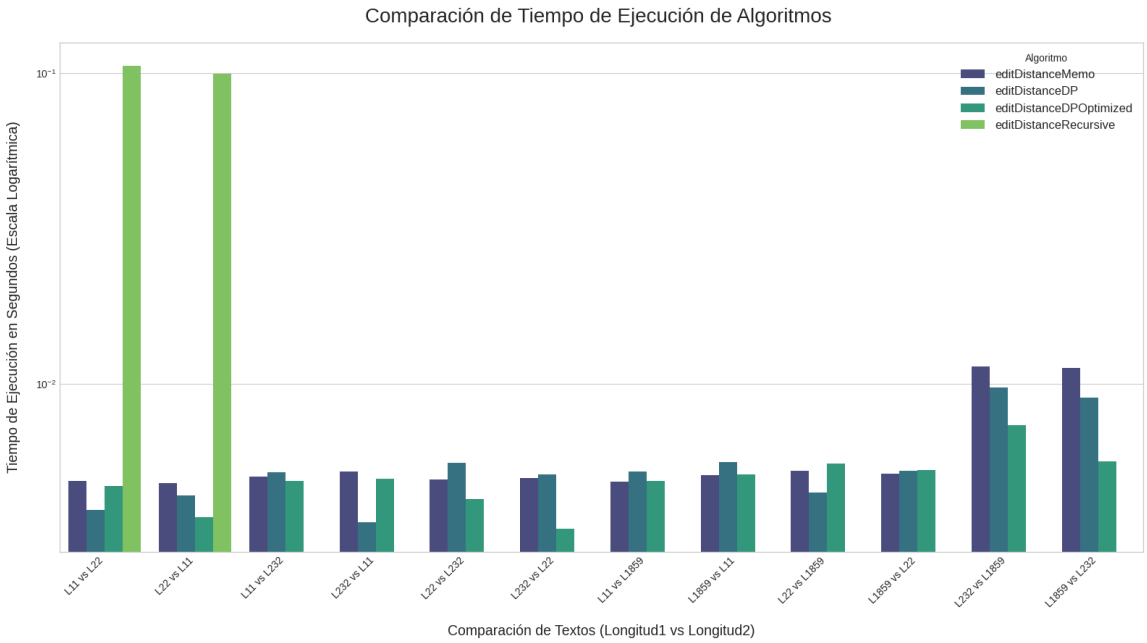

Figura 1: Comparación del tiempo de ejecución de los algoritmos.

Análisis de Tiempo:

- `editDistanceRecursive` es la más lenta incluso en pares cortos y fue omitida en casos largos por ineficiencia.
- `editDistanceDPOptimized` es consistentemente la más rápida, especialmente en textos extensos.
- `editDistanceMemo` muestra tiempos más altos que las versiones dinámicas debido al acceso menos eficiente a memoria.

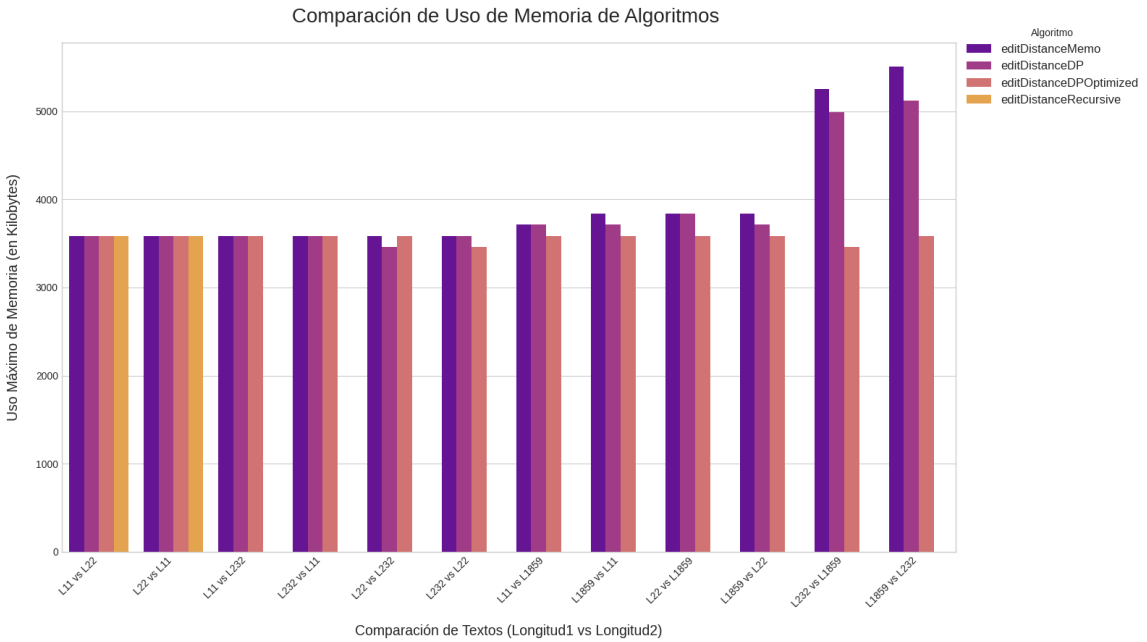

Figura 2: Comparación del uso de memoria de los algoritmos.

Análisis de Memoria:

- Las implementaciones `editDistanceMemo` y `editDistanceDP` utilizan más memoria en todos los casos, especialmente con textos largos, debido al almacenamiento de matrices completas.

- `editDistanceDPOptimized` mantiene un consumo bajo y constante de memoria, como se espera por su diseño que solo almacena dos filas.

7. Conclusión (Gabriela y Constanza)

Los resultados obtenidos durante la experimentación concuerdan con el análisis teórico de complejidad realizado para cada una de las implementaciones del algoritmo.

- La implementación **recursiva** (`editDistanceRecursive`) mostró un crecimiento exponencial en tiempo de ejecución, tal como se anticipa teóricamente con una complejidad de $O(2^{n+m})$. Esto la vuelve impráctica para textos de longitud media o grande, y explica su exclusión en algunas pruebas.
- La versión **memoizada** (`editDistanceMemo`) logró reducir el tiempo a una complejidad $O(n \cdot m)$, pero a costa de un mayor uso de memoria, ya que almacena todos los subproblemas en una matriz.
- La implementación mediante **programación dinámica clásica** (`editDistanceDP`) también presenta una complejidad $O(n \cdot m)$ en tiempo y espacio, mostrando tiempos de ejecución más eficientes que `Memo` debido al acceso más secuencial a la memoria.
- Finalmente, la versión **optimizada en espacio** (`editDistanceDPOptimized`) mantiene la eficiencia temporal, pero reduce el uso de memoria a $O(\text{mín}(n, m))$, siendo en la práctica la más eficiente globalmente en tiempo y espacio.

En resumen, la experimentación valida las predicciones teóricas: algoritmos con menor complejidad espacial y temporal escalan mucho mejor frente al aumento en la longitud de los textos. Por otro lado, se comprueba que los métodos recursivos básicos, aunque conceptualmente útiles, no son recomendables en aplicaciones reales debido a su alto costo computacional.