# REPORTE TAREA 2 y 3
# ALGORITMOS Y COMPLEJIDAD

## *«Encontrando Diferencias entre dos Secuencias»*

Renato Ramírez

19 de junio de 2025

01:18

**Resumen**

*Este informe compara dos formas de resolver el problema de encontrar diferencias entre secuencias: una usando fuerza bruta y otra con programación dinámica. Para ello, se implementaron ambos algoritmos y se probaron con conjuntos de datos generados automáticamente, midiendo su tiempo de ejecución y uso de memoria.*

*Los resultados muestran que la programación dinámica es mucho más eficiente y escalable, especialmente cuando el número de casos crece, mientras que la fuerza bruta rápidamente se vuelve inviable. Esto demuestra la importancia de elegir el enfoque correcto al momento de diseñar soluciones algorítmicas. Además, el trabajo permitió validar en la práctica lo estudiado en teoría, y deja espacio para seguir explorando mejoras en este tipo de problemas.*

## Índice

# 1. Introducción

El análisis y diseño de algoritmos es una de las bases fundamentales en la formación de estudiantes en Ciencias de la Computación. Comprender cómo se comportan distintos algoritmos en la práctica permite tomar mejores decisiones sobre cuándo y por qué utilizar uno u otro. En este trabajo se busca evaluar el rendimiento de dos enfoques para resolver un problema clásico: encontrar las diferencias entre dos secuencias.

El problema de encontrar las diferencias entre dos secuencias consiste en identificar las partes no coincidentes entre ambas, resaltando al mismo tiempo las similitudes. Para lograr esto, se suele buscar la subsecuencia común más larga (LCS), y luego, con base en esa referencia, segmentar las porciones que no coinciden. Esta tarea es útil en áreas como el control de versiones, la comparación de textos o secuencias biológicas, donde es importante entender cómo y dónde dos cadenas difieren.

Para ello, se comparan dos estrategias distintas: una implementación por fuerza bruta, que explora todas las posibles subsecuencias para encontrar la subsecuencia común más larga, y otra basada en programación dinámica, que utiliza una solución eficiente construyendo una tabla de subproblemas. Ambos enfoques fueron implementados y puestos a prueba en diferentes casos.

El objetivo principal de este informe es analizar cuáles son las diferencias prácticas entre ambos algoritmos, en términos de tiempo de ejecución y uso de memoria. Para ello, se desarrolló una serie de experimentos que permiten observar hasta qué punto es viable usar fuerza bruta y cuál es el comportamiento real de la programación dinámica en escenarios más exigentes.

Este tipo de comparación no busca ser un aporte novedoso a la literatura científica, sino una herramienta de aprendizaje. A través de este informe, se espera poner en práctica los conocimientos adquiridos en el curso, como el análisis de eficiencia, la complejidad algorítmica y el diseño de experimentos.

# 2. Diseño y Análisis de Algoritmos

Para la formación del pseudocódigo y para ayudar con el informe se utilizaron prompts con IA.

## 2.1. Fuerza Bruta

La solución por fuerza bruta consiste en generar todas las posibles subsecuencias de la secuencia `s` y verificar para cada una si es una subsecuencia de `t`. De todas las subsecuencias válidas, se selecciona aquella de mayor longitud (la LCS). Luego, usando las posiciones de la LCS en ambas secuencias, se determinan las diferencias al segmentar las partes no comunes.

**Algoritmo 1:** Algoritmo de fuerza bruta para diferencias de secuencias

**Input:** Secuencias $s$, $t$

**Output:** Lista de diferencias entre $s$ y $t$

```
lcs ← cadena vacía;
for cada subsecuencia sub de s do
    if sub es subsecuencia de t y |sub| > |lcs| then
        lcs ← sub;
        Guardar posiciones de sub en s y t;
    end
end
Construir diferencias usando lcs y posiciones;
return lista de diferencias
```

### Complejidad

- Tiempo: $O(2^n \cdot m)$, donde $n = |s|$, porque se generan $2^n$ subsecuencias y cada una puede tomar hasta $O(m)$ para verificar si es subsecuencia de `t`.
- Espacio: $O(n+m)$, principalmente por almacenar posiciones y subsecuencias temporales.

## 2.2. Programación Dinámica

### 2.2.1. Descripción de la solución recursiva

La solución con programación dinámica implementa el clásico algoritmo para encontrar la LCS utilizando una matriz de subproblemas. Primero se construye la tabla de DP, y luego se reconstruye la LCS desde dicha tabla. A partir de ella, se identifican los segmentos diferentes entre las secuencias.

La solución original al problema puede modelarse recursivamente considerando que si los últimos caracteres de $s$ y $t$ coinciden, entonces se incluye ese carácter en la LCS y se reduce el problema a los prefijos anteriores. En caso contrario, se toma el máximo entre los resultados al omitir un carácter de una de las secuencias.

#### 2.2.2. Relación de recurrencia

- Caso base: Si $i = 0$ o $j = 0$, entonces $LCS(i, j) = 0$.
- Si $s[i-1] = t[j-1]$, entonces $LCS(i, j) = 1 + LCS(i-1, j-1)$.
- Si $s[i-1] \neq t[j-1]$, entonces $LCS(i, j) = \text{máx}(LCS(i-1, j), LCS(i, j-1))$.

#### 2.2.3. Identificación de subproblemas

Los subproblemas están definidos por los pares de índices $(i, j)$ que representan los prefijos $s[0..i-1]$ y $t[0..j-1]$ de las cadenas de entrada. Se resuelven y almacenan en la tabla DP para evitar repeticiones.

#### 2.2.4. Estructura de datos y orden de cálculo

Se utiliza una matriz bidimensional dp de tamaño $(n+1) \times (m+1)$ donde $dp[i][j]$ almacena la longitud de la LCS entre los prefijos $s[0..i-1]$ y $t[0..j-1]$. El cálculo se realiza en orden ascendente (de menor a mayor), llenando fila por fila desde $dp[0][0]$ hasta $dp[n][m]$.

#### 2.2.5. Algoritmo utilizando programación dinámica

**Algoritmo 2:** Algoritmo con programación dinámica para diferencias de secuencias

**Input:** Secuencias $s$, $t$

**Output:** Lista de diferencias entre $s$ y $t$

```
Crear matriz dp de dimensiones (n + 1) × (m + 1) inicializada en 0;
for i ← 1 to n do
    for j ← 1 to m do
        if s[i − 1] = t[j − 1] then
            dp[i][j] ← dp[i − 1][j − 1] + 1;
        end
        else
            dp[i][j] ← máx(dp[i − 1][j], dp[i][j − 1]);
        end
    end
end
Reconstruir LCS desde matriz dp;
Usar LCS para segmentar las diferencias;
return lista de diferencias
```

**Complejidad**

- Tiempo: $O(n \cdot m)$, al llenar la tabla de DP y reconstruir la LCS.
- Espacio: $O(n \cdot m)$ por la matriz DP.

**Resumen Comparativo**

| **Característica** | **Fuerza Bruta** | **Programación Dinámica** |
|---|---|---|
| Estrategia | Enumeración completa | Subproblemas + memoización |
| Complejidad temporal | $O(2^n \cdot m)$ | $O(n \cdot m)$ |
| Complejidad espacial | $O(n+m)$ | $O(n \cdot m)$ |
| Ventaja | Simple para $n$ pequeño | Escalable para entradas grandes |
| Desventaja | Inviable para $n > 20$ | Requiere más memoria |

Cuadro 1: Comparación general entre los algoritmos diseñados

## 3. Implementaciones

A continuación se encontrara con el link al repositorio que contiene todos los archivos y programas que se utilizaron para la medición y elavoración de este informe, en la carpeta code para cada algoritmo encontrara la carpeta data, la cual contiene los archivos de inputs, outputs, las metricas y los gráficos, mientras que en la carpeta de scripts encontrara lo relacionado a la creación de datos de entrada y la generación de gráficos.

https://github.com/xReNatS/Algoritmos-y-complejidad-2025/tree/master/INF221-2025-1-TAREA-2-3

# 4. Experimentos

Para la ejecución de las pruebas se utilizó con computador de escritorio con las siguientes características: procesador Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz 3.00 GHz, 16 GB de memoria DDR4, almacenamiento SSD NVMe y sistema operativo Windows 11 pro. Todas las implementaciones se compilaron con g++ 14.2.0, haciendo uso de la consola CMD de Windows. Para la generación de gráficos se utilizó Python 3.10 con numpy, pandas, matplotlib y la libreria seaborn para el diseño.

## 4.1. Dataset (casos de prueba)

Los casos de prueba para evaluar el comportamiento algoritmo de programación con fuerza bruta, se generaron mediante un script en Python (input_generator.py). Se crearon archivos de entrada que contienen pares de secuencias aleatorias de longitud acotada, con un número de casos (`k`) que aumenta progresivamente.

La idea consistió en variar el valor de `k` comenzando desde 10, y aumentándolo de forma escalonada: en tramos de 10 hasta 100, luego en pasos más amplios (150, 2250, etc.) hasta alcanzar un máximo de 10 millones. Para cada archivo se generaron hasta 100.000 pares de secuencias.

Cada secuencia fue construida con caracteres aleatorios del alfabeto inglés en mayúsculas (`A-Z`), con una longitud $n$ y $m$ elegidas aleatoriamente entre 1 y un límite superior de 20. Esta acotación se hizo debido al alto tiempo que le tomaba al algoritmo de fuerza bruta en realizar la tarea (como se vera más adelante) y para que el tiempo de ejecución para la creación de los inputs no fuera demasiado extensa. Sin embargo se realizó otro generador de inputs (que sigue la misma lógica) para testear de mejor manera al algoritmo de programación dinámica pues tuvo mejores resultados en los tiempos (como se vera más adelante tambien) con un limite superior de 100 para la longitud de $n$ y $m$.

Esta generación progresiva de casos permite observar el comportamiento del algoritmo frente a distintos volúmenes de entrada, manteniendo constante la complejidad por caso pero variando la carga total del archivo.

## 4.2. Resultados

En esta sección se presentan los resultados experimentales obtenidos al medir el tiempo de ejecución y el uso de memoria de las implementaciones de los algoritmos de fuerza bruta y programación dinámica para la detección de diferencias entre dos secuencias. Los datos fueron generados utilizando scripts en Python que ejecutan los programas sobre archivos de entrada generados previamente con distintos tamaños de entrada $k$ (número de casos). Todos los archivos generados se encuentran disponibles en la carpeta `data`.

## 4.3. Metodología de experimentación

Se generaron múltiples archivos de entrada con un número de casos creciente, desde $k = 10$ hasta $k = 10^7$, usando secuencias aleatorias de caracteres alfabéticos en mayúscula. Las longitudes de las

secuencias se limitaron a 20 caracteres para la fuerza bruta y a 100 caracteres para la programación dinámica, para asegurar una ejecución razonable en ambos algoritmos.

Para la generación de los distintos inputs es necesario ejecutar para ambos algoritmos el programa `input_generator.py` que se encuentra en la carpeta `scripts` de cada algoritmo, despues solo se requiere ejecutar el programa principal de cada algoritmo haciendo uso del comando `make run`.

La ejecución de cada algoritmo fue cronometrada utilizando la biblioteca `time` y el consumo de memoria fue medido con `psutil`. Posteriormente, los datos se almacenaron en archivos CSV, y las visualizaciones fueron realizadas con `matplotlib`, generando gráficos de tiempo y memoria en función de $k$, utilizando una escala logarítmica para una mejor visualización.

### 4.4. Análisis de resultados

La Figura 1 muestra el tiempo de ejecución del algoritmo de fuerza bruta. Se observa un crecimiento exponencial, pero para efectos prácticos para valores $k$ mayores a 1,000 se puso como valor por defecto un tiempo de 60 segundos, esto puesto que para valores de $k$ mayores a 1,000, el tiempo de ejecución se volvia demasiado extenso, lo cual confirma su ineficiencia frente a grandes volúmenes de datos.

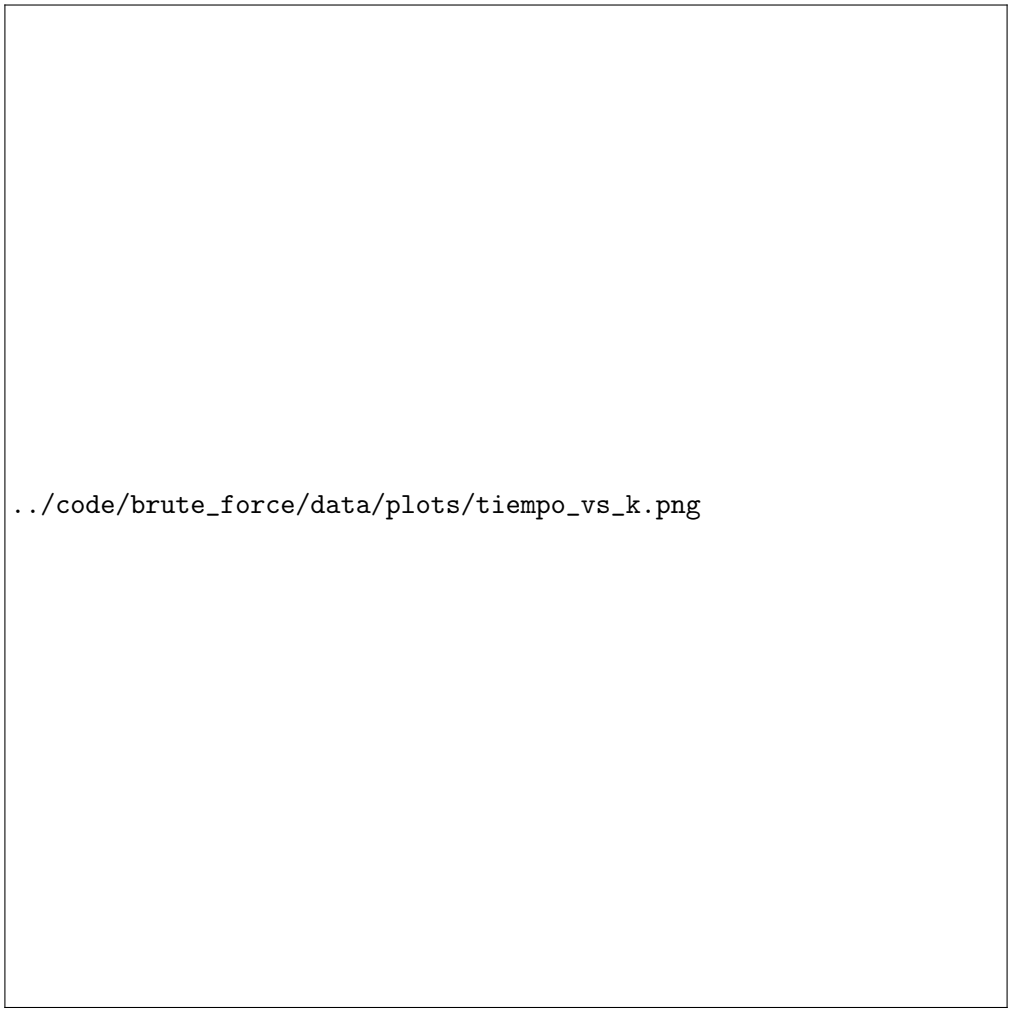


Figura 1: Tiempo de ejecución vs número de casos $k$ para algoritmo de fuerza bruta.

En la Figura 2, que representa el uso de memoria del algoritmo de fuerza bruta, se observan valores anómalos extremadamente altos (del orden de $10^{19}$ MB), debido a que se fijo tambien el valor de la memoria con $k$ mayor a 1,000. Sin embargo, en general, el consumo de memoria no escala significativamente con $k$, dado que cada subproblema es procesado de forma independiente y no se almacenan estructuras auxiliares significativas.

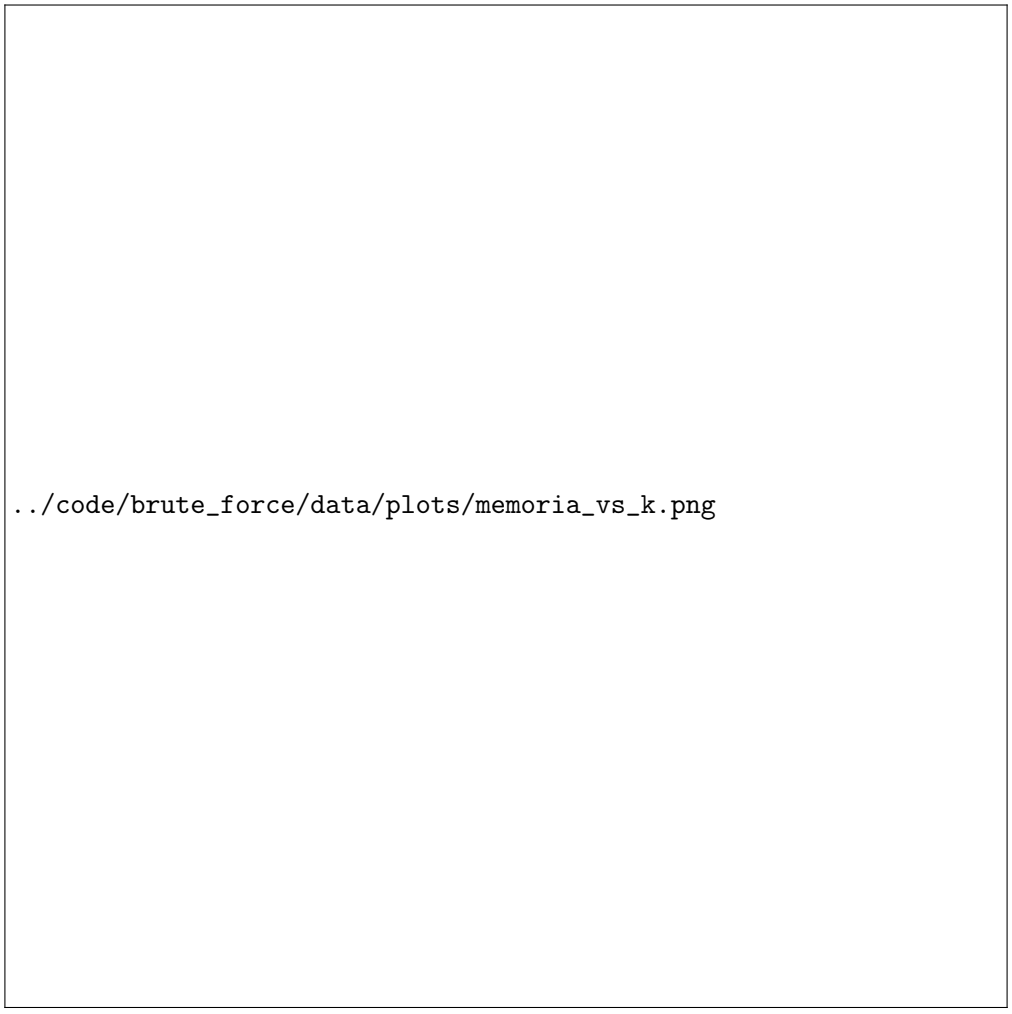


Figura 2: Uso de memoria vs número de casos $k$ para algoritmo de fuerza bruta.

Por otro lado, la Figura 3 muestra el tiempo de ejecución del algoritmo de programación dinámica. En este caso, el algoritmo muestra un crecimiento mucho más controlado del tiempo de ejecución, manteniéndose por debajo de los 8 segundos incluso para $k = 10^7$. Esto es consistente con la complejidad $O(nm)$ de este enfoque, que resulta mucho más eficiente al aumentar la cantidad de casos.


Figura 3: Tiempo de ejecución vs número de casos $k$ para algoritmo de programación dinámica.

La Figura 4 muestra el uso de memoria del algoritmo de programación dinámica. Al igual que con el enfoque de fuerza bruta, se presentan algunos picos anómalos en los valores medidos. Estos valores no son consistentes con el comportamiento real del algoritmo, que debería tener un uso de memoria lineal respecto al número de casos y cuadrático respecto a la longitud máxima de las secuencias.


Figura 4: Uso de memoria vs número de casos $k$ para algoritmo de programación dinámica.

## 5. Conclusiones

Los resultados obtenidos a lo largo del experimento permiten afirmar que el uso de programación dinámica representa una mejora significativa frente a la estrategia de fuerza bruta para resolver el problema de detección de diferencias entre dos secuencias. Esta mejora no solo se refleja en la reducción del tiempo de ejecución, sino también en la capacidad de escalar hacia entradas más grandes sin comprometer la viabilidad computacional del algoritmo.

El análisis comparativo entre ambos enfoques evidencia cómo la elección de una estrategia algorítmica adecuada puede determinar la eficiencia y aplicabilidad de una solución en contextos reales. La fuerza bruta, aunque conceptualmente clara, demuestra ser inviable para volúmenes altos de datos, mientras que la programación dinámica logra mantener un desempeño estable incluso en escenarios más exigentes.

De este modo, los resultados respaldan la necesidad de recurrir a enfoques más estructurados y optimizados para problemas combinatorios, especialmente cuando se espera escalar el uso del algoritmo. La información empírica recolectada no solo valida las complejidades teóricas discutidas, sino que también ilustra con claridad el impacto que tienen en la práctica.

Este trabajo, en definitiva, reafirma la importancia del análisis algorítmico desde una doble perspectiva: teórica y experimental, permitiendo una comprensión más integral del comportamiento de las soluciones frente a distintas condiciones de entrada.

## 6. Condiciones de entrega

- La tarea se realizará individualmente (esto es grupos de una persona), sin excepciones.
- La entrega debe realizarse vía `http://aula.usm.cl` en un tarball en el área designada al efecto, en el formato `tarea-2 y 3-rol.tar.gz` (`rol` con dígito verificador y sin guión).

  Dicho tarball debe contener las fuentes en LaTeX 2$\varepsilon$ (al menos `tarea-2 y 3.tex`) de la parte escrita de su entrega, además de un archivo `tarea-2 y 3.pdf`, correspondiente a la compilación de esas fuentes.
- Si se utiliza algún código, idea, o contenido extraído de otra fuente, este **debe** ser citado en el lugar exacto donde se utilice, en lugar de mencionarlo al final del informe.
- Asegúrese que todas sus entregas tengan sus datos completos: número de la tarea, ramo, semestre, nombre y rol. Puede incluirlas como comentarios en sus fuentes LaTeX (en TeX comentarios son desde % hasta el final de la línea) o en posibles programas. Anótese como autor de los textos.
- Si usa material adicional al discutido en clases, detállelo. Agregue información suficiente para ubicar ese material (en caso de no tratarse de discusiones con compañeros de curso u otras personas).
- No modifique `preamble.tex`, `tarea_main.tex`, `condiciones.tex`, estructura de directorios, nombres de archivos, configuración del documento, etc. Sólo agregue texto, imágenes, tablas, código, etc. En el códigos funte de su informe, no agregue paquetes, ni archivos .tex (a excepción de que agregue archivos en `/tikz`, donde puede agregar archivos .tex con las fuentes de gráficos en `TikZ`).
- La fecha límite de entrega es el día 6 de junio de 2025.

**NO SE ACEPTARÁN TAREAS FUERA DE PLAZO.**

- Nos reservamos el derecho de llamar a interrogación sobre algunas de las tareas entregadas. En tal caso, la nota de la tarea será la obtenida en la interrogación.

**NO PRESENTARSE A UN LLAMADO A INTERROGACIÓN SIN JUSTIFICACIÓN PREVIA SIGNIFICA AUTOMÁTICAMENTE NOTA 0.**

# A. Apéndice 1