REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

«Encontrando Diferencias entre dos Secuencias»

David Kripper

24 de mayo de 2025

18:20

Resumen

Este trabajo aborda el problema de detectar diferencias entre dos secuencias de texto mediante la métrica de la subsecuencia común más larga (LCS). Se comparan dos paradigmas: fuerza bruta, que examina todas las subsecuencias posibles y garantiza la corrección a costa de un crecimiento exponencial, y programación dinámica, que almacena resultados intermedios y reduce la complejidad. Las implementaciones en C++ emplean la biblioteca `chrono` *para medir tiempos y scripts en Python para generar casos de prueba y gráficar resultados. En un MacBook Air M2, la fuerza bruta pasa de microsegundos a más de 40 minutos cuando* $n > 17$*, mientras que la programación dinámica mantiene tiempos del orden de* 10^{-5} *segundos en todo el rango. La evidencia valida la teoría: la fuerza bruta solo es viable en instancias sumamente pequeñas, mientras que la programación dinámica ofrece una solución práctica y escalable para aplicaciones de procesamiento de lenguaje natural, bioinformática y corrección ortográfica.*

Índice

1. Introducción

El presente trabajo se situa en el campo del Análisis y Diseño de Algoritmos en Ciencias de la Computación, un área fundamental para resolver problemas computacionales de manera eficaz y eficiente. Dentro de este campo, los problemas de edición y comparación de cadenas de texto han recibido gran atención, especialmente en aplicaciones que requieren medir similitudes entre secuencias, como el procesamiento de lenguaje natural, la bioinformática [1] y la corrección ortográfica. Entre las distintas métricas para medir similitud o diferencia entre secuencias, el análisis de la subsecuencia común más larga (LCS, por sus siglas en inglés) destaca como una herramienta fundamental.

Una de las estrategias más utilizadas para comparar cadenas es el uso de la subsecuencia común más larga, la cual permite identificar los segmentos que se comparten entre dos secuencias. A partir de la LCS, es posible deducir los fragmentos que difieren entre ambas, lo que permite no solo cuantificar su diferencia sino también obtener una representación visual de estas mismas.

El propósito de este informe es estudiar y comparar dos enfoques algorítmicos para resolver este problema: el método de fuerza bruta, basado en una búsqueda exhaustiva de todas las subsecuencias comunes, y el método de programación dinámica, que optimiza la búsqueda al ir recordando lo que ya ha calculado, en lugar de volver a hacerlo desde cero. Ambos algoritmos serán evaluados en cuanto a su eficiencia y efectividad para detectar las diferencias entre dos cadenas de texto, analizando cómo escalan en tiempo frente a entradas de diferentes tamaños.

Este estudio permite introducir conceptos avanzados como el uso de programación dinámica para optimizar procesos combinatorios, y explorar las ventajas y limitaciones de los enfoques de fuerza bruta frente a técnicas algorítmicas más eficientes. El informe tiene como objetivo proporcionar una visión tanto práctica como teórica de cada paradigma, ofreciendo una base sólida para el análisis de algoritmos en contextos donde la comparación entre secuencias es una operación fundamental.

2. Diseño y Análisis de Algoritmos

En este trabajo se analizan dos enfoques fundamentales para resolver el problema de detección de diferencias entre secuencias de texto a partir de la subsecuencia común más larga: el enfoque de fuerza bruta y el de programación dinámica. Ambos algoritmos abordan el mismo objetivo pero lo hacen desde perspectivas distintas, cada una con sus ventajas y limitaciones en cuanto a eficiencia, escalabilidad y complejidad de implementación.

El enfoque de fuerza bruta se basa en la exploración exhaustiva de todas las posibles subsecuencias comunes entre dos cadenas, utilizando técnicas de backtracking. Esta estrategia permite encontrar la subsecuencia común más larga probando cada combinación posible, lo que asegura una solución correcta, aunque a un alto costo computacional. Este método es útil para comprender el comportamiento del problema en instancias pequeñas, ya que considera todos los caminos posibles. Sin embargo, su complejidad crece de forma exponencial con el tamaño de las cadenas, lo que limita seriamente su aplicabilidad a casos reales de gran tamaño.

Por su parte, el enfoque de programación dinámica resuelve el mismo problema de manera mucho más eficiente. Se basa en dividir el problema en subproblemas más pequeños y almacenar los resultados intermedios. Esto evita el cálculo repetido y mejora considerablemente el tiempo de ejecución. Este enfoque permite escalar a instancias de mayor tamaño, manteniendo su complejidad, lo que lo hace particularmente adecuado para contextos prácticos como la comparación de documentos o secuencias biológicas.

A continuación, se presenta un análisis detallado de cada algoritmo. Este análisis permitirá comparar ambos métodos y evidenciar por qué la programación dinámica resulta más adecuada para instancias grandes, sin dejar de valorar lo que ofrece la implementación por fuerza bruta.

2.1. Fuerza Bruta

La solución de fuerza bruta desarrollada en esta tarea [3] [5] busca identificar las diferencias entre dos secuencias de texto a partir de la subsecuencia común más larga (LCS, por sus siglas en inglés). Para ello, el algoritmo explora exhaustivamente todas las posibles subsecuencias comunes entre dos cadenas, con el fin de encontrar aquella de mayor longitud. Esta subsecuencia se utiliza posteriormente para determinar los fragmentos que difieren entre ambas cadenas. Debido a la naturaleza exhaustiva de este enfoque, el algoritmo resulta poco eficiente cuando se trabaja con cadenas largas.

La solución se implementa mediante una función recursiva que explora todas las combinaciones posibles de caracteres compartidos entre las dos cadenas. El algoritmo comienza comparando los primeros caracteres de ambas cadenas. Si coinciden, se agregan a la subsecuencia común y se continúa con los siguientes índices. En caso contrario, se realizan dos llamadas recursivas: una avanzando en la primera

cadena y otra avanzando en la segunda, evaluando así todas las posibilidades. Una vez obtenida la LCS, se recorre nuevamente cada cadena para identificar los bloques que no pertenecen a dicha subsecuencia, generando así los pares de substrings que representan las diferencias entre ambas.

Este enfoque garantiza una solución correcta, ya que evalúa todas las opciones posibles para obtener la LCS. Sin embargo, su principal desventaja es el alto costo computacional. La complejidad temporal del algoritmo es exponencial, ya que en el peor de los casos el número de llamadas recursivas crece de manera combinatoria. Para dos cadenas de longitudes `n` y `m`, la complejidad temporal se puede expresar de la siguiente manera:

$$O(2^{\text{máx}(n,m)})$$

Esto se debe a que cada posición puede generar múltiples caminos posibles según las decisiones tomadas. Como consecuencia, el enfoque de fuerza bruta se vuelve impractico e infructifero para entradas largas, siendo útil únicamente para analizar instancias pequeñas.

En cuanto al análisis espacial, el principal consumo de memoria proviene de la pila de llamadas recursivas. Cada vez que el algoritmo compara caracteres, realiza una nueva llamada recursiva que se mantiene en la pila hasta retornar. Esto implica un uso de memoria proporcional a la profundidad máxima de la recursión, que en el peor caso es:

$$O(\text{máx}(n, m))$$

Esta complejidad espacial es aceptable en términos de uso de memoria, aunque no compensa la ineficiencia temporal en instancias grandes.

Algoritmo 1: Algoritmo de Fuerza Bruta para Diferencias entre Secuencias

```
1  Procedure LCSBT(s, i, t, j, actual, mejor)
2      if i = largo(s) or j = largo(t) then
3          if largo(actual) > largo(mejor) then
4              mejor ← actual
5          return
6      if s[i] = t[j] then
7          agregar s[i] a actual
8          LCSBT(s, i+1, t, j+1, actual, mejor)
9          quitar último carácter de actual
10     LCSBT(s, i+1, t, j, actual, mejor)
11     LCSBT(s, i, t, j+1, actual, mejor)
12 Procedure LCSFuerzaBruta(s, t)
13     actual ← cadena vacía
14     mejor ← cadena vacía
15     LCSBT(s, 0, t, 0, actual, mejor)
16     return mejor
17 Procedure SequenceDifference(s, t)
18     lcs ← LCSFuerzaBruta(s, t)
19     diferencias ← lista vacía
20     i, j, k ← 0
21     while i < largo(s) or j < largo(t) do
22         if k < largo(lcs) and i < largo(s) and j < largo(t) and s[i] = t[j] and s[i] = lcs[k] then
23             i++, j++, k++
24         else
25             bloqueS, bloqueT ← cadenas vacías
26             while i < largo(s) and (k ≥ largo(lcs) or s[i] ≠ lcs[k]) do
27                 agregar s[i] a bloqueS
28                 i++
29             while j < largo(t) and (k ≥ largo(lcs) or t[j] ≠ lcs[k]) do
30                 agregar t[j] a bloqueT
31                 j++
32             if bloqueS ≠ vacío or bloqueT ≠ vacío then
33                 agregar (bloqueS, bloqueT) a diferencias
34     return diferencias
```

2.2. Programación Dinámica

El enfoque de programación dinámica garantiza una solución correcta, donde cada subproblema se resuelve exactamente una vez y sus resultados se almacenan para reutilizarlos cuando sea necesario. De esta forma se evita la explosión de llamadas recursivas típica de la fuerza bruta. La complejidad temporal del algoritmo por programación dinámica corresponde a:

$$O(n \times m)$$

pues cada celda se computa en tiempo constante. En cuanto a memoria, la tabla ocupa el mismo orden:

$$O(n \times m)$$

2.2.1. Descripción de la solución recursiva

Pensemos en dos cadenas, S (largo n) y T (largo m). Para cualquier par de índices (i, j) llamo $L(i, j)$ a la cantidad de letras que comparte la subsecuencia común más larga entre los primeros i caracteres de S y los primeros j de T. Si alguno de los dos prefijos está vacío, no hay nada que comparar: $L(0, j) = L(i, 0) = 0$.

El resto lo decidimos de la siguiente manera:

- **Coinciden los últimos caracteres**. Cuando S_i y T_j son iguales, simplemente "heredo" la mejor solución de los prefijos que los preceden y le sumo uno:

$$L(i, j) = 1 + L(i-1,\ j-1).$$

- **No coinciden**. Aquí pruebo dos caminos:

  1. Ignoro el último carácter de S y miro $L(i-1, j)$.
  2. Ignoro el último de T y miro $L(i, j-1)$.

  Me quedo con el que deje una subsecuencia más larga:

$$L(i, j) = \text{máx}\big(L(i-1,\ j),\ L(i,\ j-1)\big).$$

Una vez llenada toda la tabla, el valor de la esquina inferior derecha, $L(n, m)$, me dice cuántas letras tiene la LCS completa. Para saber cuáles son, camino desde esa esquina hacia arriba-izquierda siguiendo los pasos que no cambian el valor, hasta volver al origen $(0, 0)$.

2.2.2. Relación de recurrencia

La tabla L se llena utilizando la siguiente lógica: Si las letras finales de los prefijos coinciden, me cuelgo de la mejor solución que ya tenía arriba y a la izquierda y le sumo uno. Si no, comparo lo que obtengo al saltarme el último carácter de S con lo que obtengo al saltarme el de T, y me quedo con el mayor de los dos.

2.2.3. Identificación de subproblemas

La gracia de la PD está en partir el problema grande en problemas más chiquititos.

Cada casilla (i, j) de la matriz pregunta: "¿cuántos caracteres tiene la LCS entre $S[1..i]$ y $T[1..j]$?". De ese modo aparecen $(n+1)(m+1)$ subproblemas. Para resolver uno cualquiera sólo necesito, como máximo, los tres vecinos inmediatos:

$$(i-1,\ j), \quad (i,\ j-1), \quad (i-1,\ j-1).$$

2.2.4. Estructura de datos y orden de cálculo

- `matrizDP`: tabla $(n+1) \times (m+1)$ con los valores $L(i, j)$.
- `Llenado`: bottom-up, cada casilla usa los tres vecinos ya calculados.

2.2.5. Algoritmo utilizando programación dinámica [3] [2]

Algoritmo 2: Diferencias entre dos secuencias usando PD

```
1  Procedure LCSDP(S, T)
2      n ← |S|,
3      m ← |T|
4      crear matriz L[0..n][0..m] inicializada en 0
5      for i ← 1 to n do
6          for j ← 1 to m do
7              if S_i = T_j then
8                  L[i][j] ← 1 + L[i − 1][j − 1]
9              else
10                 L[i][j] ← máx(L[i − 1][j], L[i][j − 1])
11     i ← n,
12     j ← m,
13     lcs ← ε
14     while i > 0 and j > 0 do
15         if S_i = T_j then
16             agregar_al_principio S_i a lcs; i−−; j−−
17         else if L[i − 1][j] ≥ L[i][j − 1] then
18             i−−
19         else
20             j−−
21     return lcs
22 Procedure SequenceDifference(S, T)
23     lcs ← LCSDP(S, T)
24     diferencias ← []
25     i, j, k ← 0
26     while i < |S| or j < |T| do
27         if k < |lcs| and i < |S| and j < |T| and S_i = T_j = lcs_k then
28             i++, j++, k++
29         else
30             bloqueS, bloqueT ← ε
31             while i < |S| and (k ≥ |lcs| or S_i ≠ lcs_k) do
32                 append S_i a bloqueS; i++
33             while j < |T| and (k ≥ |lcs| or T_j ≠ lcs_k) do
34                 append T_j a bloqueT; j++
35             if bloqueS ≠ ε or bloqueT ≠ ε then
36                 append (bloqueS, bloqueT) a diferencias
37     return diferencias
```

3. Implementaciones

La implementación de esta tarea fue desarrollada en C++ y organizada de manera separada para facilitar la modularización entre los distintos algoritmos y el manejo de archivos de entrada/salida. A continuación, se describe la estructura general del proyecto:

- `code`: Carpeta principal que contiene todo el código fuente, subdividido por paradigma.
  - `brute_force`: Contiene la implementación del algoritmo de fuerza bruta. Incluye a:
    - `brute_force.cpp`: Archivo principal que lee los datos de entrada, ejecuta el algoritmo y guarda los resultados.
    - `algorithm/sequence_difference.cpp y algorithm/sequence_difference.h`: Implementan la lógica del cálculo de diferencias a partir de la LCS utilizando fuerza bruta.
    - `makefile`
  - `dynamic_programming`: Contiene la implementación del algoritmo basado en programación dinámica. Tiene una estructura similar a `brute_force`, pero con una lógica optimizada.
- `data`: Carpeta donde se organizan los datos de prueba y resultados:
  - `inputs.txt`: Archivo con los casos de prueba utilizados
  - `outputs.txt`: Resultados generados por los algoritmos.
  - `measurements.txt`: Contiene las mediciones de tiempo de ejecución y tamaño de entrada para cada ejecución de ambos algoritmos.
  - `plots`: Carpeta donde se guardan las gráficas generadas con Python.
- `scripts`: Contiene scripts auxiliares en Python que automatizan tareas como:
  - `input_generator.py`: Genera automáticamente casos de prueba en el formato que se pide en la tarea.
  - `plot_generator.py`: Lee las mediciones y genera gráficos del rendimiento de los algoritmos.

https://github.com/dakrima/INF221-2025-1-TAREA-2-3

4. Experimentos

Para realizar los casos de prueba de los algoritmos implementados se utilizó Hardware, entorno de software y condiciones de entrada especificas.

Descripción del Hardware.

Se utilizó un MacBook Air Apple M2 (2022) con la especificaciones siguientes [4]:

- **Procesador:** Chip M2 Apple con 4 núcleos de rendimiento @3.49 GHz y 4 núcleos de eficiencia @2.42 GHz. GPU de 8 núcleos. Neural Engine de 16 núcleos.
- **Memoria RAM:** 8 GB de memoria unificada LPDDR5.
- **Almacenamiento SSD:** 256 GB de almacenamiento SSD.

Entorno de Software.

- **Sistema Operativo:** macOS Sequoia 15.0.1.
- **Compilador C++:** Se utiliza el compilador g++ 16.0.0, con el estándar de C++17: -std=c++17

Condiciones de entrada.

Durante los experimentos se utilizaron condiciones de entrada controladas y reproducibles, permitiendo comparar de manera consistente el comportamiento de los algoritmos evaluados bajo los mismos parámetros.

- **Cadenas de texto:** Para evaluar ambos algoritmos, se generaron pares de cadenas de texto de manera aleatoria, con longitudes variables desde 2 hasta 20 caracteres. Estas cadenas fueron construidas a partir de letras mayúsculas del alfabeto inglés, utilizando un script generador programado en Python. Cada par de cadenas fue procesado siguiendo el mismo formato especificado en el enunciado de la tarea: para cada caso, se almacena la longitud de la cadena, seguida de la cadena misma en una línea separada.

4.1. Dataset (casos de prueba)

Para realizar los experimentos, se generaron casos de prueba compuestos por pares de cadenas de texto, con el principal objetivo de evaluar el rendimiento de los algoritmos de fuerza bruta y programación dinámica. Los strings fueron generados de forma aleatoria utilizando caracteres en mayúsculas del abecedario inglés.

Los tamaños de entrada utilizados van desde los 2 hasta los 20 caracteres por cadena, lo que fue suficiente para evidenciar diferencias de rendimiento entre ambos algoritmos sin comprometer demasiado el tiempo de ejecución del enfoque de fuerza bruta, cuya complejidad impide trabajar con longitudes

demasiado largas. A partir de estos rangos, se generaron 4 pares de cadenas para cada tamaño, con el fin de obtener una medición más robusta del comportamiento del tiempo de ejecución.

Cada caso fue almacenado en el archivo inputs.txt, siguiendo el formato solicitado en el enunciado: un número inicial K que indica la cantidad de casos, seguido por K pares de líneas que contienen la longitud y el contenido de cada cadena. Los resultados correspondientes fueron escritos en el archivo outputs.txt, y las mediciones de tiempo y de longitud fueron registradas en measurements.txt.

Cabe destacar que los casos de prueba fueron generados automáticamente mediante un script en Python, lo que garantiza la reproducibilidad de los datos y la coherencia en el análisis. No se incorporaron casos externos al formato definido por el enunciado. Esta generación de datos permite un análisis representativo del desempeño de los algoritmos.

4.2. Resultados

Para medir los tiempos de ejecución de los algoritmos desarrollados, se utilizó la biblioteca `<chrono>` de C++, la cual permite capturar con alta precisión el intervalo de tiempo transcurrido entre el inicio y término de cada ejecución. Estas mediciones se realizaron automáticamente dentro del mismo programa, con el objetivo de garantizar consistencia y evitar interferencias externas.

Los casos de prueba fueron generados automáticamente mediante un script en Python, el cual permite crear pares de cadenas de texto aleatorias con longitudes controladas. Estos casos se almacenan en archivos de texto ubicados en la carpeta `brute_force_input` y `dynamic_programming_input`, respectivamente, y son leídos por los programas en C++ al momento de ejecutar cada algoritmo.

Durante la ejecución, los programas calculan la cantidad de substrings distintos entre las cadenas, y además registran el tiempo total de ejecución junto con la longitud promedio de los pares procesados. Posteriormente, se generaron gráficos a partir de estas mediciones utilizando scripts en Python, los cuales producen las visualizaciones correspondientes en la carpeta `plots` respectiva para cada parádigma.

Este flujo garantiza la reproducibilidad de los experimentos, permitiendo observar que los resultados obtenidos reflejan fielmente el comportamiento real de los algoritmos. De este modo, fue posible comparar de forma objetiva el rendimiento de ambos enfoques en función del tamaño de entrada.

Algoritmo de fuerza bruta

Para evaluar el rendimiento del algoritmo se generaron cuatro pares de cadenas idénticas en longitud para cada tamaño, desde 2 hasta 20 caracteres. Ese mismo caso se repitió sucesivamente para los diferentes tamaños. Los resultados de dichas mediciones se presentan en la tabla que sigue.

n	Tiempo (s)
2	$5{,}708 \times 10^{-6}$
3	$6{,}167 \times 10^{-6}$
4	$1{,}8833 \times 10^{-5}$
5	$1{,}8500 \times 10^{-5}$
6	$4{,}5750 \times 10^{-5}$
7	$1{,}131\,67 \times 10^{-4}$
8	$4{,}617\,50 \times 10^{-4}$
9	$1{,}331\,46 \times 10^{-3}$
10	$5{,}099\,33 \times 10^{-3}$
11	$1{,}822\,43 \times 10^{-2}$
12	$5{,}982\,46 \times 10^{-2}$
13	$1{,}765\,83 \times 10^{-1}$
14	$6{,}323\,00 \times 10^{-1}$
15	$2{,}272\,75$
16	$1{,}102\,35 \times 10^{1}$
17	$3{,}712\,02 \times 10^{1}$
18	$1{,}612\,97 \times 10^{2}$
19	$5{,}588\,69 \times 10^{2}$
20	$2{,}544\,17 \times 10^{3}$

Cuadro 1: Tiempos de ejecución para distintos largos n de cadenas de texto.

El gráfico generado se presenta en la imagen que sigue.

Figura 1: Tiempos de ejecución para el algoritmo de fuerza bruta.

Los datos obtenidos para el algoritmo de fuerza bruta revelan con mucha claridad el carácter exponencial de su tiempo de ejecución. Si se observan los puntos correspondientes a tamaños de entrada entre $n = 2$ y $n = 10$, los tiempos permanecen en el orden de los microsegundos, de modo que la curva luce prácticamente plana. Una vez que uno supera el largo de $n = 10$, la pendiente de la curva comienza a inclinarse. Para $n = 10 - 15$ se empieza a percibir un incremento en tiempo de ejecución. A partir de $n = 16$, el comportamiento del algoritmo y su tiempo de ejecución se multiplica exponencialmente alcanzando más de 2.500 segundos. Esta aceleración casi vertical en el gráfico refleja la complejidad teórica del algoritmo: al explorar todas las subsecuencias posibles de cada cadena, el trabajo efectivo se duplica con cada carácter adicional.

El resultado práctico es una escalabilidad nula: el algoritmo solo resulta utilizable para entradas muy pequeñas. A partir de $n = 15$ o más, se vuelve poco manejable. Esta complejidad teorica tan pobre justifica el paso a enfoques más sofisticados, como lo es la programación dinámica, que reduce el problema a complejidad cuadrática permitiendo procesar cadenas de longitudes mucho mayores.

La tabla y el gráfico no solo corroboran la validez del algoritmo de fuerza bruta, sino que también ponen de manifiesto su inviabilidad práctica y la necesidad de técnicas de optimización para cualquier aplicación que involucre secuencias de tamaño moderado.

Algoritmo de programación dinámica

Para evaluar el rendimiento del algoritmo se generaron cuatro pares de cadenas idénticas en longitud para cada tamaño, desde 2 hasta 20 caracteres. Ese mismo caso se repitió sucesivamente para los diferentes tamaños. Los resultados de dichas mediciones se presentan en la tabla que sigue.

n	Tiempo (s)
2	8,2500 $\times 10^{-6}$
3	1,6917 $\times 10^{-5}$
4	1,0084 $\times 10^{-5}$
5	1,0750 $\times 10^{-5}$
6	1,4709 $\times 10^{-5}$
7	1,4875 $\times 10^{-5}$
8	1,5125 $\times 10^{-5}$
9	1,6750 $\times 10^{-5}$
10	1,8750 $\times 10^{-5}$
11	1,2042 $\times 10^{-5}$
12	2,5667 $\times 10^{-5}$
13	2,1333 $\times 10^{-5}$
14	2,3875 $\times 10^{-5}$
15	2,4416 $\times 10^{-5}$
16	2,5209 $\times 10^{-5}$
17	3,0000 $\times 10^{-5}$
18	2,6917 $\times 10^{-5}$
19	3,6292 $\times 10^{-5}$
20	1,8500 $\times 10^{-5}$

Cuadro 2: Tiempos de ejecución para programación dinámica (tamaño de entrada n)

El gráfico generado se presenta en la imagen que sigue.

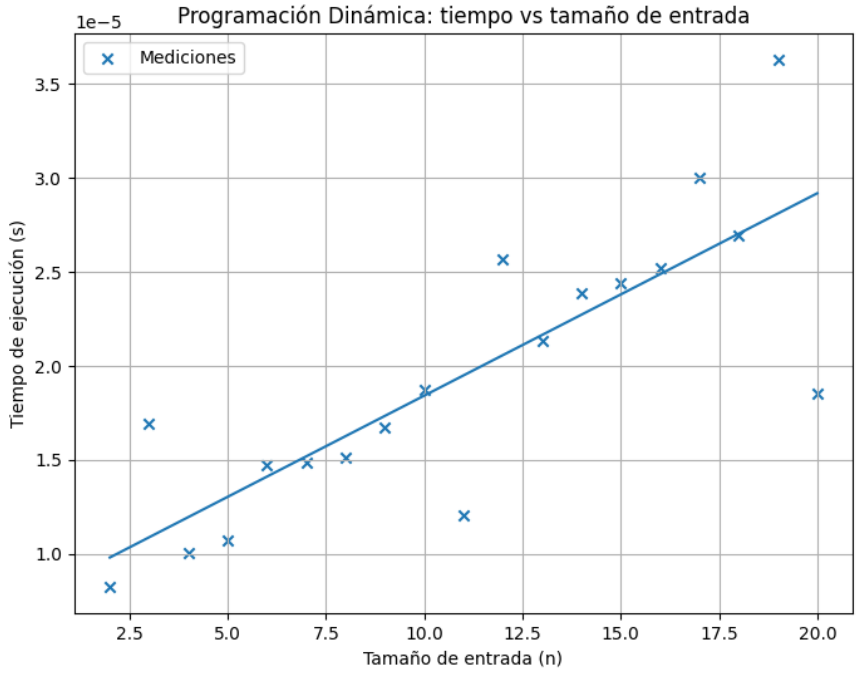

Figura 2: Tiempos de ejecución para el algoritmo de programación dinámica.

Los datos obtenidos para el algoritmo de programación dinámica revelan con mucha claridad la eficiencia prevista para este enfoque. Todas las mediciones permanecen en el rango de los 10^{-5} segundos. En la práctica, el algoritmo es prácticamente instantáneo para cadenas de hasta veinte caracteres.

El gráfico muestra que los puntos se agrupan en torno a una recta ascendente, lo que sugiere un crecimiento lineal dentro de los casos evaluados en la ejecución. Teóricamente, el algoritmo tiene complejidad cuadrática, pero con tamaños tan pequeños la parte cuadrática todavía no se vuelve dominante y la porción de la curva que alcanzamos a ver puede aproximarse razonablemente por una línea.

Desde la perspectiva de la escalabilidad, la diferencia con la fuerza bruta es abismal. Aunque el algoritmo de programación dinámica necesita utilizar una matriz para su realización, el costo de llenarla es proporcional a nm y, para los tamaños estudiados, se logran procesar en tiempos muy bajos.

La tabla y el gráfico evidencian un algoritmo con rendimiento estable: su complejidad cuadrática no representa un obstáculo visible en este rango de entradas, y la ganancia frente a la solución de fuerza bruta justifica plenamente el uso de programación dinámica para calcular subsecuencias en aplicaciones reales.

5. Conclusiones

El informe confirma la premisa planteada en la introducción: al comparar cadenas de texto, elegir bien el paradigma algorítmico marca la diferencia entre una solución altamente impractica y una herramienta realmente utilizable. Los resultados muestran que el enfoque de fuerza bruta se vuelve impracticable apenas las secuencias superan un largo de 12 carácteres, al crecer su tiempo de ejecución de microsegundos a decenas de minutos en pocos incrementos de tamaño. Por el contrario, la programación dinámica mantiene tiempos estables, validando en la práctica su complejidad.

Estos hallazgos responden directamente a la premisa inicial: disponer de un método que detecte diferencias entre cadenas de forma eficiente y que escale con la longitud de entrada. Además, nos muestran la relevancia de reutilizar las subsoluciones para así lograr una mejoría ante complejidades exponenciales. En trabajos e investigaciones donde la comparación de secuencias sea una parte importante, por no decir indispensable, el paradigma dinámico resulta completamente recomendable.

En síntesis, el trabajo demuestra que la optimización basada en programación dinámica logra de manera robusta los objetivos de eficiencia y escalabilidad, mientras que el paradigma de fuerza bruta es útil unicamente para instancias sumamente pequeñas. Extender este estudio a técnicas que logren reducir aún más las complejidades representa una línea natural de trabajo futuro, pero no altera la conclusión central: la selección acertada del paradigma algorítmico es el factor decisivo para abordar con éxito el problema de la comparación de secuencias.

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 2ε (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

Referencias

[1] Bruno Contreras-Moreira. *Algoritmos en bioinformática estructural*. 2022.ª ed. Digital.CSIC, 2018. DOI: 10.20350/digitalcsic/8544. URL: https://doi.org/10.20350/digitalcsic/8544.

[2] Thomas H. Cormen et al. *Introduction to Algorithms. Capítulo "Dynamic Programming"*. 3.ª ed. MIT Press, 31 de jul. de 2009. URL: https://mitpress.mit.edu/9780262533058/introduction-to-algorithms/.

[3] GeeksforGeeks. *Longest Common Subsequence (LCS)*. 4 de mar. de 2025. URL: https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/.

[4] Apple Inc. *MacBook Air (M2, 2022) - Especificaciones técnicas*. 2024. URL: https://support.apple.com/es-cl/111867.

[5] Programiz. *Longest Common Subsequence*. 23 de mayo de 2025. URL: https://www.programiz.com/dsa/longest-common-subsequence.