REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

«Explorando la Distancia entre Cadenas, una Operación a la Vez»

Damian Alexander Rojas Robles

17 de noviembre de 2024

16:45

Resumen

Este informe presenta dos enfoques para resolver el problema de transformación de cadenas: fuerza bruta y programación dinámica. La implementación en fuerza bruta utiliza recursión para explorar todas las combinaciones posibles de operaciones (sustitución, inserción, eliminación y transposición), pero su complejidad exponencial la hace impráctica para cadenas largas. En contraste, la programación dinámica optimiza el proceso utilizando una matriz para almacenar costos intermedios, logrando una mayor eficiencia.

La evaluación experimental se realizó en un equipo con especificaciones específicas, utilizando pruebas de tiempo y validación de costos. Los resultados muestran que la programación dinámica es considerablemente más rápida y eficiente en términos de memoria, especialmente para cadenas más largas.

Índice

1. Introducción

El diseño y análisis de algoritmos es una de las áreas más importantes en Ciencias de la Computación. Nos permite comprender cómo funcionan las soluciones a problemas reales y encontrar formas de hacerlas más rápidas y eficientes. Este campo ayuda a mejorar el rendimiento de los programas y que asegura que puedan manejar grandes cantidades de datos o funcionar bien incluso cuando los recursos son limitados. Un ejemplo interesante es el estudio de problemas relacionados con la transformación de cadenas, como calcular el costo más bajo para convertir una cadena en otra usando ciertas operaciones. Este tipo de problemas tiene aplicaciones prácticas en cosas como la edición de texto, lo que lo hace especialmente valioso.

A lo largo del tiempo, se han creado diferentes formas de abordar estos problemas. Al principio, se usaban métodos sencillos, como los algoritmos de fuerza bruta, que aunque fáciles de implementar, se vuelven poco eficientes cuando las entradas son muy grandes. La programación dinámica ofreció una mejora al almacenar y reutilizar resultados intermedios, lo que hace que los algoritmos sean mucho más rápidos y consuman menos recursos.

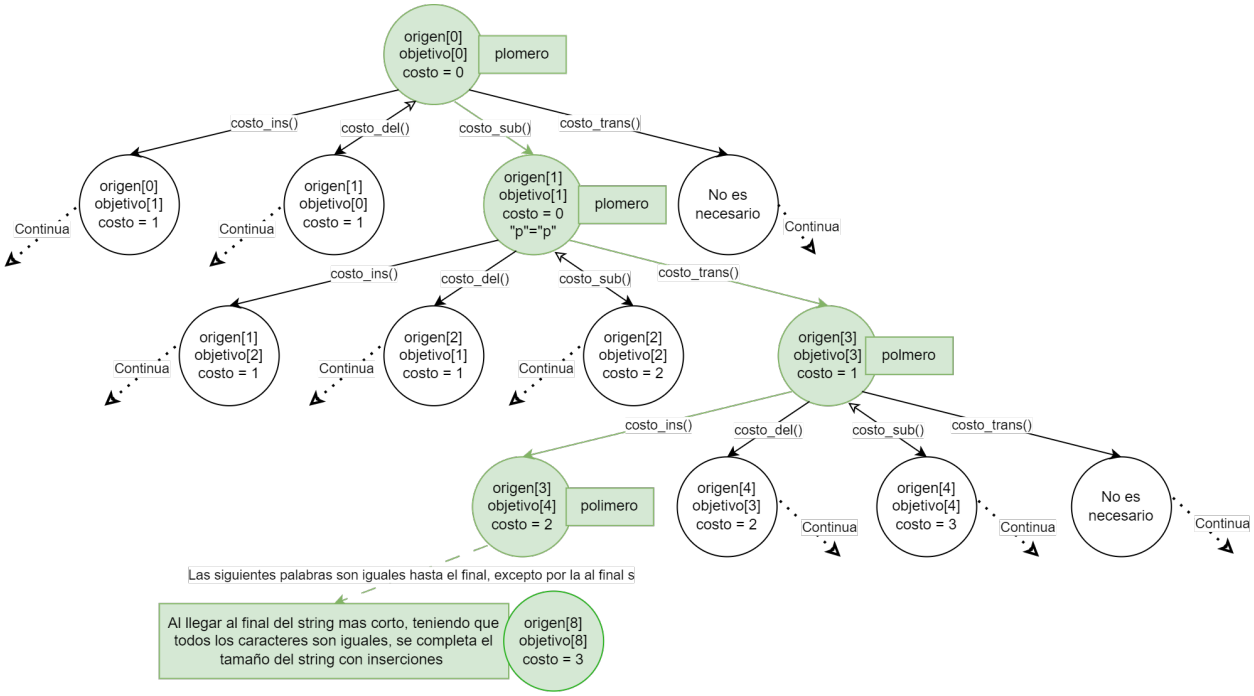
Este informe tiene como objetivo analizar y comparar dos algoritmos para la transformación de cadenas: el de fuerza bruta y el de programación dinámica. La idea principal es que, aunque la programación dinámica es generalmente más eficiente para cadenas largas, el enfoque de fuerza bruta podría ser más competitivo en casos específicos, como cuando las entradas son pequeñas o las cadenas tienen mucha similitud entre sí. Para poner a prueba esta idea, se implementarán ambos métodos, se medirán sus tiempos de ejecución en diferentes situaciones y se evaluará qué tan precisos son al calcular los costos mínimos de transformación.

2. Diseño y Análisis de Algoritmos

2.1. Fuerza Bruta

> *"Indeed, brute force is a perfectly good technique in many cases; the real question is, can we use brute force in such a way that we avoid the worst-case behavior?"*
>
> — Knuth, 1998 [2]

Esta solución busca el menor costo para transformar una cadena de caracteres 'origen' en otra cadena 'objetivo'. Cada operación tiene un costo asignado: la sustitución cuesta 2 si los caracteres son diferentes, la inserción y eliminación cuestan 1, y la transposición cuesta 1. La función 'transformar' es recursiva y evalúa todas las posibles transformaciones aplicables en cada posición de las cadenas. Como tal, el programa no transforma el string, sino que calcula directamente el costo de estas operaciones.
El programa ejecuta cada una de las posibilidades para cada caracter del string, ejecutando todas las opciones y eligiendo la menor de estas. En el siguiente ejemplo se revisa un camino optimo en donde vemos el costo de transformar 'plomero' a 'polimeros'



El programa revisa todos los casos posible en el string recursivamente, llegando al ultimo caracter del string, donde retorna en caso de que haya una diferencia en el tamaño del string. cuando llega al nivel mas abajo, comienza a evaluar, retornando el costo de cada transformacion.

Algoritmo 1: Algoritmo de transformación por fuerza bruta entre dos cadenas para encontrar el costo mínimo de conversión. (i y j indices de origen y objetivo, respectivamente)

```
1  Procedure TRANSFORMAR(origen, objetivo, i, j, costo_acumulado)
2      if i = longitud de origen y j = longitud de objetivo then
3          return costo_acumulado
4      else if i = longitud de origen then
5          return costo_acumulado + (longitud de objetivo - j)
6      else if j = longitud de objetivo then
7          return costo_acumulado + (longitud de origen - i)
8      costo_min ← ∞
9      costo_sust ← costo_acumulado + COSTO_SUB(origen[i], objetivo[j])
10     costo_min ← mín(costo_min, TRANSFORMAR(origen, objetivo, i+1, j+1, costo_sust))
11     costo_inser ← costo_acumulado + COSTO_INS(objetivo[j])
12     costo_min ← mín(costo_min, TRANSFORMAR(origen, objetivo, i, j+1, costo_inser))
13     costo_elim ← costo_acumulado + COSTO_DEL(origen[i])
14     costo_min ← mín(costo_min, TRANSFORMAR(origen, objetivo, i+1, j, costo_elim))
15     if i + 1 < longitud de origen y j + 1 < longitud de objetivo y origen[i] = objetivo[j+1] y origen[i+1] = objetivo[j] then
16         costo_transp ← costo_acumulado + COSTO_TRANS(origen[i], origen[i+1])
17         costo_min ← mín(costo_min, TRANSFORMAR(origen, objetivo, i+2, j+2, costo_transp))
18     return costo_min
19 Procedure COSTO_SUB(a, b)
20     return if a = b then
21         0
22     else
23         2
24 Procedure COSTO_INS(b)
25     return 1
26 Procedure COSTO_DEL(a)
27     return 1
28 Procedure COSTO_TRANS(a, b)
29     return if a = b then
30         0
31     else
32         1
```

El programa tiene un orden temporal de $O(4^{\text{mín}(m,n)})$, ya que por cada carácter realiza 4 llamadas recursivas, repitiendo el mismo proceso. Al llegar al final del string más corto, devuelve la distancia de diferencia con el otro string. Espacialmente, al pasar los strings por referencia, el orden espacial es $O(m+n)$.

El programa utiliza recursión en los métodos que alteran el string. La sustitución implica eliminar e insertar un elemento, lo cual tiene un costo acorde. Sin embargo, la transposición introduce una recursión adicional, ya que, aunque pueda implicar una eliminación e inserción en posiciones cercanas, su costo justifica evaluarla.

2.2. Programación Dinámica

Dynamic programming is not about filling in tables. It's about smart recursion!

Erickson, 2019 [1]

2.2.1. Descripción de la solución recursiva

Las funciones de coste funcionan de la misma manera que en el caso anterior, pero con la variación de que la función `transformar` utiliza programación dinámica para evaluar todas las posibles transformaciones en cada posición de las cadenas, almacenando los costos acumulados de cada subproblema en una matriz para evitar recalcular los mismos valores.

2.2.2. Relación de recurrencia

Este código utiliza programación dinámica para calcular el costo mínimo de transformar una cadena `s1` en otra `s2`. Para ello, utiliza una matriz `dp` donde `dp[i][j]` representa el costo mínimo para transformar los primeros i caracteres de `s1` en los primeros j caracteres de `s2`. Se definen costos para cuatro operaciones: sustitución, inserción, eliminación y transposición. La matriz se llena iterativamente, evaluando el costo mínimo entre estas operaciones. Al final, `dp[m][n]` contiene el costo mínimo de la transformación, que se imprime.

2.2.3. Identificación de subproblemas

El algoritmo identifica los subproblemas a resolver mediante la partición de las cadenas en segmentos más pequeños. Cada celda de la matriz `dp` corresponde a un subproblema que calcula el costo de transformar un prefijo de la cadena `s1` en un prefijo de la cadena `s2`. Estos subproblemas se resuelven de manera acumulativa, asegurando que los valores previamente calculados sean reutilizados en lugar de recalcularse.

2.2.4. Estructura de datos y orden de cálculo

El programa utiliza una matriz bidimensional `dp` de tamaño $(m+1) \times (n+1)$, donde m y n son las longitudes de las cadenas `s1` y `s2`, respectivamente. El cálculo se realiza llenando esta matriz celda por celda, de izquierda a derecha y de arriba hacia abajo. Cada celda `dp[i][j]` se calcula considerando los costos de las operaciones posibles (sustitución, inserción, eliminación y transposición).

2.2.5. Algoritmo utilizando programación dinámica

- Inicializa la matriz `dp` con dimensiones $(m+1) \times (n+1)$.
- Llena la primera fila y la primera columna con valores base, correspondientes a los costos acumulados para transformar cadenas vacías.

- Itera sobre los índices i y j para calcular el costo mínimo en cada celda `dp[i][j]`, evaluando las operaciones de transformación.
- Al finalizar, el valor en `dp[m][n]` representa el costo mínimo de transformar `s1` en `s2`.

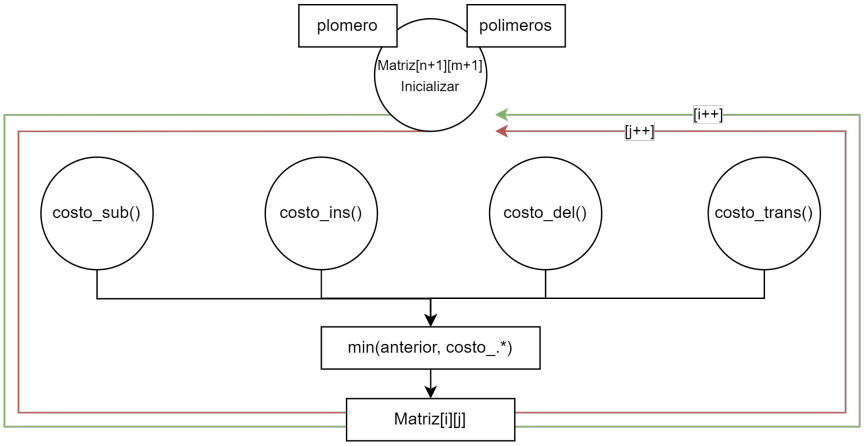


Figura 1: Visualización del proceso de programación dinámica.

		p	l	o	m	e	r	o	s
	0	1	2	3	4	5	6	7	8
p	1	0	1	2	3	4	5	6	7
o	2	1	2	1	2	3	4	5	6
l	3	2	1	2	3	4	5	6	7
i	4	3	2	3	4	5	6	7	8
m	5	4	3	4	5	6	7	8	9
e	6	5	4	5	6	7	8	9	10
r	7	6	5	6	7	8	9	10	11
o	8	7	6	7	8	9	10	11	12
s	9	8	7	8	9	10	11	12	13

Figura 2: Matriz resultante después de las iteraciones.

Después de completar las iteraciones necesarias, la matriz completa quedaría tal que así. Los valores corresponden al mínimo de la función de transformación, donde `dp[m][n]` es el valor óptimo.

Algoritmo 2: Algoritmo basado en programación dinámica para calcular el costo mínimo de transformar una cadena $s1$ en otra cadena $s2$ mediante tabulacion.

```
1  Procedure TRANSFORMAR(s1, s2)
2      m ← longitud de s1
3      n ← longitud de s2
4      Crear matriz dp de tamaño (m+1) × (n+1) con todos los valores igual a ∞
5      dp[0][0] ← 0
6      for i ← 1 to m do
7          dp[i][0] ← dp[i−1][0] + COSTO_DEL(s1[i-1])
8      for j ← 1 to n do
9          dp[0][j] ← dp[0][j−1] + COSTO_INS(s2[j-1])
10     for i ← 1 to m do
11         for j ← 1 to n do
12             costo_sust ← dp[i−1][j−1] + COSTO_SUB(s1[i-1], s2[j-1])
13             dp[i][j] ← mín(dp[i][j], costo_sust)
14             costo_inser ← dp[i][j−1] + COSTO_INS(s2[j-1])
15             dp[i][j] ← mín(dp[i][j], costo_inser)
16             costo_elim ← dp[i−1][j] + COSTO_DEL(s1[i-1])
17             dp[i][j] ← mín(dp[i][j], costo_elim)
18             if i > 1 y j > 1 y s1[i−1] = s2[j−2] y s1[i−2] = s2[j−1] then
19                 costo_transp ← dp[i−2][j−2] + COSTO_TRANS(s1[i-2], s1[i-1])
20                 dp[i][j] ← mín(dp[i][j], costo_transp)
21     return dp[m][n]
22 Procedure COSTO_SUB(a, b)
23     return if a = b then
24         0
25     else
26         2
27 Procedure COSTO_INS(b)
28     return 1
29 Procedure COSTO_DEL(a)
30     return 1
31 Procedure COSTO_TRANS(a, b)
32     return if a = b then
33         0
34     else
35         1
```

El programa tiene un orden temporal de $O(m \times n)$, donde m y n son las longitudes de las cadenas. Esto se debe a que se llena una matriz de tamaño $(m+1) \times (n+1)$, realizando un número constante de operaciones por celda para calcular los costos de las transformaciones posibles.

El algoritmo tiene un orden espacial de $O(m \times n)$, ya que utiliza una matriz de tamaño $(m+1) \times (n+1)$ para almacenar los resultados intermedios.

3. Implementaciones

La implementación presentada resuelve el problema de transformación de cadenas utilizando dos enfoques distintos: fuerza bruta y programación dinámica. Además, se incluye un script en Bash para automatizar la ejecución de los casos de prueba almacenados en una estructura de carpetas.

El archivo 'BruteForce.cpp' implementa un enfoque de fuerza bruta mediante recursión, explorando todas las combinaciones posibles de operaciones (sustitución, inserción, eliminación y transposición) para transformar una cadena en otra. Determina el costo mínimo al evaluar cada opción y selecciona la que genera el menor costo acumulado.

El archivo 'Dinamic.cpp' implementa un enfoque basado en programación dinámica, que es considerablemente más eficiente que la fuerza bruta. Este método utiliza una matriz para almacenar los costos intermedios de transformar subcadenas, eliminando cálculos redundantes. La matriz se llena iterativamente, evaluando todas las operaciones posibles y seleccionando la que minimiza el costo acumulado. Este enfoque tiene una complejidad temporal $O(m \times n)$, donde m y n son las longitudes de las cadenas.

El enfoque de fuerza bruta, aunque sencillo, tiene una complejidad exponencial y un uso constante de memoria, lo que lo hace impráctico para cadenas largas. En cambio, la programación dinámica es significativamente más eficiente, con complejidad temporal $O(m \times n)$ y uso de memoria proporcional al tamaño de las cadenas, siendo más adecuada para aplicaciones reales. No obstante, su implementación es más compleja debido a la necesidad de gestionar y llenar una matriz.

El script 'test.sh' automatiza la ejecución masiva de pruebas iterando sobre subcarpetas del directorio 'testcases/', donde cada archivo 'test.txt' contiene dos cadenas de prueba. Ejecuta el programa 'main' con estos datos como entrada estándar y muestra los resultados, permitiendo comparar resultados o medir tiempos eficientemente.

4. Experimentos

> "*Non-reproducible single occurrences are of no significance to science.*"
>
> —Popper,2005 [3]

Para la experimentación y evaluación del desempeño del algoritmo propuesto, se utilizó un equipo con las siguientes especificaciones técnicas:

- **Sistema Operativo:** Windows 10 Home Single Language, 64 bits (versión 10.0, compilación 19045).
- **Procesador:** Intel® Core™ i5-10300H CPU @ 2.50GHz (8 núcleos).
- **Memoria RAM:** 16 GB (16384 MB).
- **Disco Duro:** Unidad NVMe SSD de 512 GB.
- **Tarjeta Gráfica:** NVIDIA GeForce GTX 1650.

El programa utiliza las librerias estandar de c++, el uso de la libreria <string>, para manejo de los string de entrada. Especificamente en el programa de fuerza bruta, se utiliza la libreria <limits>, ya que se utiliza para conseguir el valor maximo perimitido por un int. En la solucion por programacion dinamica se utiliza la lirberia <vector>para lacreacion y manejo de la matriz para la resolucion del problema. Ademas de que se uso la libreria <chrono>para el calculo temporal de las funciones a la hora de resolver los distintos casos. Se utilizó un entorno WSL para la ejecucion de los programas.

4.1. Dataset (casos de prueba)

Pruebas realizadas

Para la experimentación actual, implementamos dos tipos principales de pruebas:

1. **Prueba temporal:** En esta prueba, nos enfocamos en medir el tiempo que cada método (fuerza bruta y programación dinámica) tarda en resolver una serie de ejercicios generados mediante programas en Python, independientemente del resultado de la función. Los casos evaluados incluyen:
   - **Cadenas de distintos tamaños**
   - **Cadenas completamente iguales**
   - **Una cadena vacía y otra no vacía**
   - **Cadenas de igual tamaño pero con letras completamente distintas**
   - **Entradas que solo pueden resolverse mediante transposición**

   Este enfoque permitió comparar directamente los tiempos de ejecución entre ambos métodos en diferentes escenarios.

2. **Prueba de validación de costos:** En esta prueba utilizamos un dataset limitado con entradas seleccionadas y cuales resultados esperados fueron calculados manualmente. Posteriormente, comparamos estos valores con los resultados obtenidos por el programa, asegurándonos de que los costos fueran correctos en todos los casos.

Estas pruebas nos permitieron analizar la eficiencia en términos de rendimiento.

Ejemplos de entradas para pruebas

A continuación, se presentan ejemplos de entradas utilizadas para evaluar el rendimiento y la precisión del algoritmo, clasificados según las características de las cadenas:

1. **Cadenas de distintos tamaños:**
   - Entrada:

     ```
     "st"
     "abcdef"
     ```

   - Explicación: Una cadena es significativamente más grande que la otra, lo que genera múltiples inserciones.
2. **Cadenas completamente iguales:**
   - Entrada:

     ```
     "algoritmo"
     "algoritmo"
     ```

   - Explicación: Ambas cadenas son idénticas, lo que debería resultar en un costo de transformación igual a cero.
3. **Una cadena vacía y otra no vacía:**
   - Entrada:

     ```
     ""
     "transformar"
     ```

   - Explicación: Una de las cadenas está vacía, lo que implica que todas las operaciones serán inserciones.
4. **Cadenas de igual tamaño pero con letras completamente distintas:**
   - Entrada:

```
"abcde"
"vwxyz"
```

- Explicación: Ambas cadenas tienen el mismo tamaño, pero no tienen caracteres en común, por lo que el resultado implicará únicamente sustituciones.

5. **Entradas que solo pueden resolverse mediante transposición:**

- Entrada:

```
"ab"
"ba"
```

- Explicación: Las cadenas son iguales en contenido pero tienen los caracteres intercambiados, lo que hace necesario el uso de la operación de transposición.

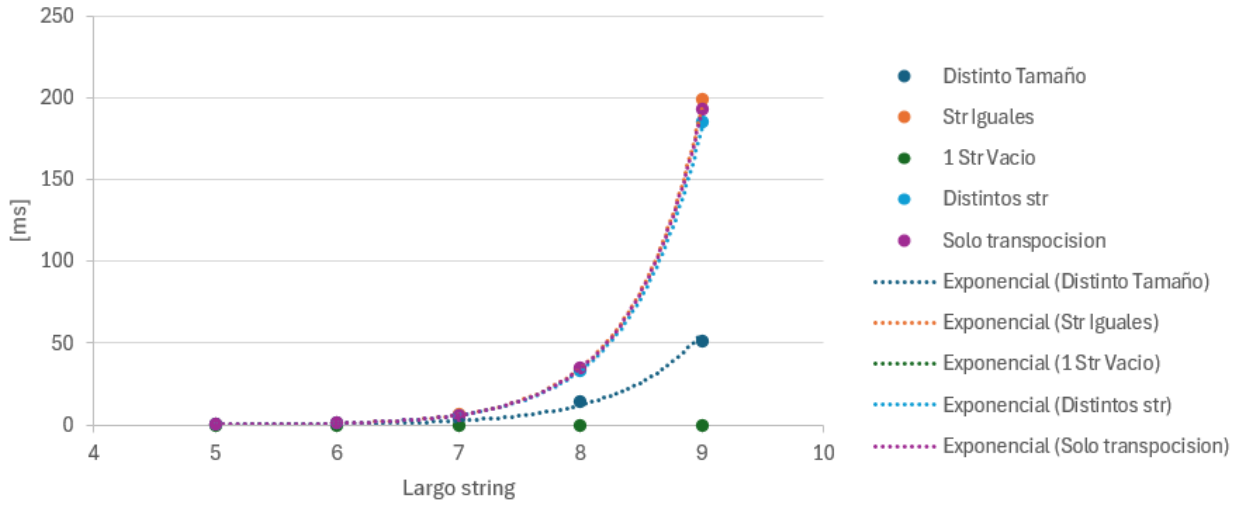
4.2. Resultados

En esta sección se presentan los resultados experimentales obtenidos para evaluar el rendimiento de los algoritmos presentados. Los resultados se resumen en las tablas a continuacion, armadas con los tiempos promedio en milisegundos [*ms*] para cada categoría y diferentes tamaños de entrada. Para cada tamaño, se realizaron tres pruebas independientes (*test1*, *test2*, y *test3*) cuyos valores se promediaron para destacar el comportamiento general del algoritmo. Los tiempos destacados en **negrita** representan los valores promedio obtenidos.

Fuerza Bruta

La tabla muestra los resultados obtenidos al probar el algoritmo de fuerza bruta con diferentes tipos de casos. Este método, aunque simple en su concepto, examina todas las posibles formas de transformar una cadena en otra y elige la que tiene el menor costo. Sin embargo, debido a que utiliza recursión y tiene una complejidad exponencial, el tiempo que tarda en ejecutarse puede variar mucho dependiendo de la longitud y las características de las cadenas analizadas.

	Distinto Tam. [ms]	**Str Iguales [ms]**	**1 Str Vacío [ms]**	**Distintos Str [ms]**	**Solo Transposición [ms]**
5	**0.209766667**	**0.2022**	**0.001**	**0.2143**	**0.206066667**
test1	0.1994	0.1985	0.001	0.2086	0.1988
test2	0.2307	0.2036	0.0011	0.2351	0.1992
test3	0.1992	0.2045	0.001	0.1992	0.1995
6	**0.214666667**	**1.151733333**	**0.00166667**	**1.084166667**	**1.0974**
test1	0.2421	1.1352	0.001	1.0851	1.0847
test2	0.1992	1.1225	0.0009	1.0812	1.1227
test3	0.2081	1.1251	0.001	1.0814	1.1258
7	**4.302233333**	**6.211533333**	**0.000966667**	**6.0378**	**6.062966667**
test1	6.9041	6.5234	0.001	6.448	6.4338
test2	6.1986	6.077	0.001	6.4517	6.0974
test3	5.9828	6.0432	0.001	6.4493	6.0722
8	**14.01323333**	**34.0214**	**0.00133333**	**33.5678**	**35.26496667**
test1	5.0211	34.3423	0.0011	32.8275	33.8369
test2	1.6996	33.1743	0.001	32.9435	34.9697
test3	35.319	34.5429	0.0011	33.9435	34.9375
9	**51.67986667**	**198.698**	**0.00113333**	**184.9345667**	**192.8536667**
test1	77.7683	222.328	0.001	185.553	190.984
test2	74.2817	189.866	0.0011	185.829	182.521
test3	2.9984	183.9	0.001	183.422	205.002



Casos de prueba base

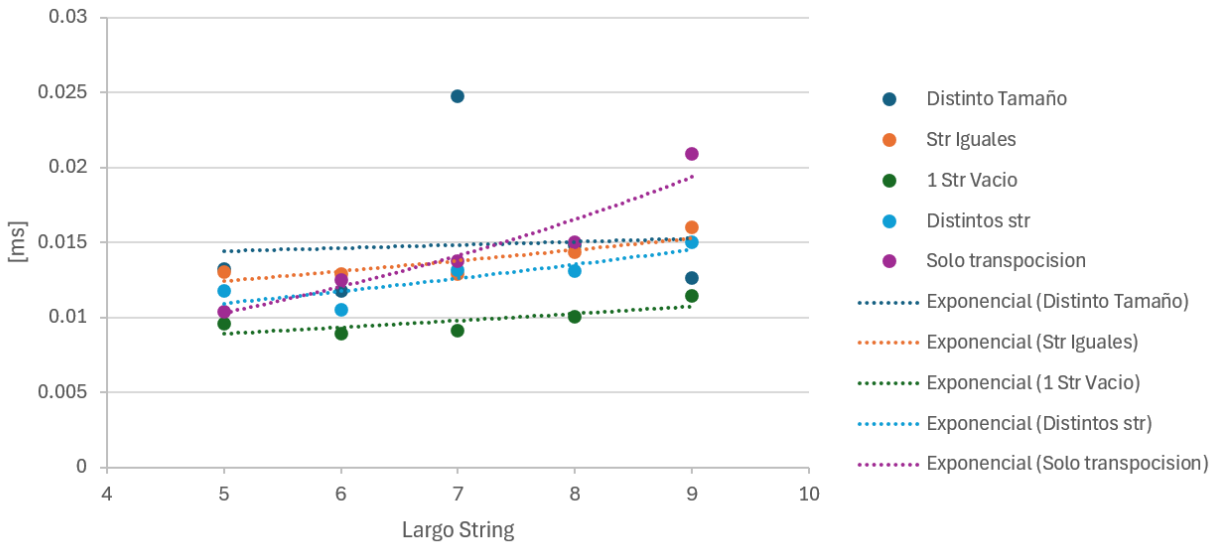
Se utilizaron varios casos de prueba básicos para evaluar el rendimiento del código de programación dinámica. Estos casos consisten en diferentes configuraciones de entrada, donde se varían características clave del algoritmo. Los resultados de las pruebas se presentan en la siguiente tabla.

Prueba	Cadenas	Resultado
1		**0**
2	`nada` `anda`	**1**
3	`componentes` `comerciales`	**10**
4	`datos` `estructuras`	**12**
5	`hola`	**4**

Programacion Dinamica

La tabla presenta los resultados experimentales obtenidos al aplicar el algoritmo basado en Programación Dinámica a distintos casos de prueba. Este enfoque optimiza el proceso de transformación al almacenar y reutilizar soluciones a subproblemas, lo que reduce significativamente la cantidad de cálculos redundantes en comparación con métodos más simples como el de fuerza bruta. Gracias a esta técnica, el algoritmo logra una complejidad polinómica, haciendo que su tiempo de ejecución sea mucho más eficiente, incluso para cadenas de mayor longitud o complejidad.

	Distinto Tam.	Str Iguales	1 Str Vacío	Distintos Str	Solo Transposición
5	**0.0132**	**0.013033333**	**0.009566667**	**0.011733333**	**0.0104**
test1	0.0145	0.0132	0.0108	0.0144	0.0111
test2	0.0101	0.0113	0.005	0.0104	0.0111
test3	0.0131	0.0121	0.0114	0.0105	0.011
6	**0.011733333**	**0.012066667**	**0.008933333**	**0.0131**	**0.012466667**
test1	0.0121	0.0122	0.0101	0.0141	0.0114
test2	0.0102	0.0115	0.008	0.011	0.0115
test3	0.0131	0.0124	0.0099	0.0119	0.013
7	**0.024733333**	**0.012066667**	**0.009133333**	**0.013133333**	**0.013766667**
test1	0.0319	0.0131	0.011	0.0152	0.0143
test2	0.0112	0.0122	0.008	0.012	0.0115
test3	0.0106	0.0116	0.009	0.015	0.0132
8	**0.014833333**	**0.014333333**	**0.010333333**	**0.013066667**	**0.015033333**
test1	0.0112	0.0146	0.0101	0.0128	0.0143
test2	0.0123	0.014	0.0099	0.0125	0.0139
test3	0.0131	0.0143	0.0098	0.013	0.0147
9	**0.0132**	**0.0146**	**0.0114**	**0.015**	**0.020866667**
test1	0.013	0.0143	0.0108	0.0149	0.0238
test2	0.0129	0.0139	0.011	0.0126	0.0166
test3	0.0111	0.0168	0.0107	0.0162	0.0222



Casos de prueba base

Al igual que en el caso de la fuerza bruta, se evaluo el codigo frente a casos de prueba bases, en los que se variaban caracteristicas importantes del codigo. Los resultaos se prensentan en la siguiente tabla.

Prueba	Cadenas	Resultado
1		**0**
2	nada anda	**1**
3	componentes comerciales	**10**
4	datos estructuras	**12**
5	hola	**4**

5. Conclusiones

En conclusión, los resultados muestran claramente las diferencias en el rendimiento entre los enfoques de fuerza bruta y programación dinámica para la transformación de cadenas. La solución de fuerza bruta, aunque efectiva en pequeños conjuntos de datos, tiene una complejidad exponencial que la hace ineficiente para cadenas más largas. Por otro lado, la programación dinámica ofrece una solución mucho más eficiente, con una complejidad temporal de $O(m \times n)$, lo que la hace ideal para aplicaciones prácticas. Este análisis resalta la importancia de elegir el enfoque adecuado según el tamaño y las características de los datos, lo que mejora considerablemente el rendimiento en situaciones reales. Las pruebas experimentales confirmaron que la programación dinámica no solo reduce el tiempo de ejecución, sino que también optimiza el uso de memoria, lo que la hace más adecuada para trabajar con grandes volúmenes de datos. Aunque ambos métodos tienen su utilidad, la programación dinámica destaca como la opción preferida cuando se busca eficiencia y escalabilidad.

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 10 de noviembre de 2024.

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

Aquí puede agregar tablas, figuras u otro material que no se incluyó en el cuerpo principal del documento, ya que no constituyen elementos centrales de la tarea. Si desea agregar material adicional que apoye o complemente el análisis realizado, puede hacerlo en esta sección.

> Esta sección es solo para material adicional. El contenido aquí no será evaluado directamente, pero puede ser útil si incluye material que será referenciado en el cuerpo del documento. Por lo tanto, asegúrese de que cualquier elemento incluido esté correctamente referenciado y justificado en el informe principal.

Referencias

[1] Jeff Erickson. *Algorithms*. Jun. de 2019. ISBN: 978-1-792-64483-2.

[2] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201896850.

[3] K. Popper. *The Logic of Scientific Discovery*. Routledge Classics. Taylor & Francis, 2005. ISBN: 9781134470020. URL: https://books.google.cl/books?id=LWSBAgAAQBAJ.