Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica

05 de Mayo de 2025

Facundo Barrasso
111942

Mateo Nowenstein
112063

Juana Rehl
112185

1. Introducción

Un algoritmo de Programación Dinámica permite resolver problemas de optimización dividiendo el problema original en subproblemas más pequeños, almacenando los resultados parciales para evitar cálculos redundantes. Esta técnica es especialmente útil cuando un problema presenta subestructura óptima y superposición de subproblemas.

El objetivo de este trabajo es profundizar en el estudio de la programación dinámica a través de un problema concreto. Se busca plantear la ecuación de recurrencia correspondiente, desarrollar el algoritmo que la implemente, analizar su correctitud y justificar su complejidad teórica y empírica.

1.1. Consigna

1. Hacer un análisis del problema, plantear la ecuación de recurrencia correspondiente y proponer un algoritmo por programación dinámica que obtenga la solución óptima al problema planteado: dado el listado de palabras y una cadena desencriptada, determinar si es un posible mensaje (es decir, si se lo puede separar en palabras del idioma), o no. Si es posible, determinar cómo sería el mensaje.

2. Demostrar que la ecuación de recurrencia planteada en el punto anterior en efecto nos asegura encontrar una solución, si es que la hay (y si no la hay, lo detecta).

3. Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Considerar analizar por separado cada uno de los algoritmos que se implementen (programación dinámica y reconstrucción), y luego llegar a una conclusión final.

4. Analizar si (y cómo) afecta a los tiempos del algoritmo planteado la variabilidad de los valores (mensajes, palabras del idioma, largos de las palabras, etc.).

5. Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado. Adicionalmente, el curso proveerá algunos casos particulares para validar.

6. Hacer mediciones de tiempos para corroborar la complejidad teórica indicada. Agregar los casos de prueba necesarios para dicha corroboración (generando sus propios conjuntos de datos). Esta corroboración empírica debe realizarse confeccionando los gráficos correspondientes y utilizando la técnica de cuadrados mínimos.

7. Agregar cualquier conclusión que les parezca relevante.

2. Análisis del problema

En este trabajo práctico se nos encomienda validar si una cadena sin espacios, resultado de un mensaje encriptado por un posible soplón, puede segmentarse completamente en palabras del idioma español. El mensaje se analiza usando un diccionario confiable, provisto por los lingüistas de Amarilla Pérez. No buscamos interpretar el contenido, sino confirmar si el mensaje es potencialmente válido. Si puede dividirse en palabras reales, se encienden las alarmas; si no, se descarta.

El objetivo es claro: dado un string s y un conjunto de palabras válidas D, determinar si s puede segmentarse completamente con palabras de D, utilizando programación dinámica. Si es así, el mensaje se acepta; si no, queda marcado como "no es mensaje". Aprovechamos subresultados: si un prefijo de la cadena ya puede separarse, podemos reutilizarlo para verificar segmentos más largos sin recalcular desde cero.

2.1 Análisis de la ecuación de recurrencia

Sea s una cadena de longitud n y D un conjunto de palabras válidas (diccionario). Definimos un arreglo booleano dp de tamaño $n+1$, donde $dp[i]$ indica si el prefijo $s[0:i]$ puede segmentarse en palabras que pertenecen a D.

La recurrencia queda definida de la siguiente manera:

$$dp[i] = \begin{cases} \text{True} & \text{si } \exists j \in [0, i) \text{ tal que } dp[j] = \text{True y } s[j:i] \in D \\ \text{False} & \text{en caso contrario} \end{cases}$$

Caso base:

$$dp[0] = \text{True}$$

Esto representa que la cadena vacía es segmentable (condición inicial para poder construir soluciones más grandes).

Significado

Para cada posición i en la cadena, evaluamos todos los posibles cortes j anteriores. Si existe al menos un índice j tal que:

- el prefijo $s[0:j]$ es segmentable ($dp[j] = \text{True}$), y
- la subcadena $s[j:i]$ pertenece al diccionario ($s[j:i] \in D$),

entonces el prefijo $s[0:i]$ también es segmentable ($dp[i] = \text{True}$).

Justificación de la ecuación de recurrencia

Para justificar la validez de la ecuación de recurrencia planteada:

$$dp[i] = \begin{cases} \text{True} & \text{si } \exists j \in [0, i) \text{ tal que } dp[j] = \text{True y } s[j:i] \in D \\ \text{False} & \text{en caso contrario} \end{cases}$$

utilizamos **inducción completa** sobre el tamaño del prefijo considerado (índice i en la cadena de entrada).

Base inductiva

Para $i = 0$, se tiene el caso base $dp[0] = \text{True}$, ya que la cadena vacía se considera segmentable (es decir, no requiere cortes). Esto sirve como punto de partida para construir soluciones más grandes.

Hipótesis inductiva

Suponemos que para todo $k < i$, los valores $dp[k]$ han sido correctamente calculados, es decir, cada uno refleja si el prefijo $s[0 : k]$ puede ser segmentado en palabras del diccionario D.

Paso inductivo

Demostraremos que, bajo esta hipótesis, el valor $dp[i]$ también será correctamente calculado según la ecuación de recurrencia.

La definición de $dp[i]$ indica que existe un índice $j < i$ tal que:

- $dp[j] = \text{True}$ (es decir, el prefijo $s[0 : j]$ puede segmentarse), y
- la subcadena $s[j : i] \in D$ (es una palabra válida del diccionario).

Si se cumple esta condición, entonces el prefijo $s[0 : i]$ puede segmentarse como la concatenación de dos partes válidas:

1. una segmentación válida del prefijo $s[0 : j]$, y
2. una palabra del diccionario que corresponde a la subcadena $s[j : i]$.

Por lo tanto, $dp[i] = \text{True}$.

En caso contrario, si no existe ningún j tal que $dp[j] = \text{True}$ y $s[j : i] \in D$, entonces no hay forma válida de segmentar $s[0 : i]$, y por lo tanto $dp[i] = \text{False}$.

Conclusión

Por el principio de **inducción completa**, concluimos que la ecuación de recurrencia es correcta para todo $i \in [0, n]$, y el algoritmo determina correctamente si una cadena puede segmentarse en palabras del diccionario.

3. Explicación del algoritmo

El algoritmo planteado está diseñado de la forma típica de un algoritmo de Programación Dinámica, esto es, recorrer el arreglo original de forma iterativa de abajo para arriba (es decir, de forma *bottom-up*), permitiendo "memorizar" las soluciones anteriores y encontrar la solución del problema actual mediante ellas. En otras palabras, como los problemas se van haciendo cada vez más grandes, lo mejor es empezar desde el subproblema más chico ya que es más fácil considerar todos los posibles casos, permitiendo que cuando llegue al problema más grande, hallar la solución del mismo sea mucho más fácil ya que contaríamos con las soluciones anteriores.

Código y explicación

Primero se leerán las palabras disponibles:

```python
def cargar_diccionario(ruta_archivo):
    palabras = set()
    with open(ruta_archivo, 'r', encoding='utf-8') as archivo:
        for linea in archivo:
            palabra = linea.strip()
            if palabra:
                palabras.add(palabra.lower())  # por las dudas paso a minuscula
    return palabras
```

Luego, por cada línea de la entrada estándar (puede ser desviada a un archivo) se analizará si esa línea efectivamente puede ser un posible mensaje:

```python
def analizar_texto(texto, diccionario):
    mensajes = []
    for linea in texto:
        linea = linea.strip()
        mensaje = es_mensaje_valido(linea, diccionario)
        mensajes.append(mensaje)
    return mensajes
```

Una vez ya nos guardamos las palabras posibles y la línea a desencriptar, procedemos al análisis del mensaje en sí, que es la parte en donde se utiliza la ecuación de recurrencia para hallar el posible mensaje:

```python
def es_mensaje_valido(cadena, diccionario):
    n = len(cadena)
    dp = [False] * (n + 1)
    dp[0] = True
    prev = [-1] * (n + 1)
    max_len = max(len(palabra) for palabra in diccionario)

    for i in range(1, n + 1):
        for j in range(max(0, i - max_len), i):
            if dp[j] and cadena[j:i] in diccionario:
                dp[i] = True
                prev[i] = j
                break

    return reconstruccion_palabras(prev, cadena) if dp[-1] else None
```

Primero se inicializa el arreglo de soluciones para el largo de la cadena $+1$, ya que cada subproblema tendrá un largo diferente, siendo $i \in [0, n]$ y $j \in [\text{máx}(0, i - \text{max\_len}), i]$, con $j \leq i$ (se inician todas en `False`, siendo `False` el indicador de que no hay ninguna cadena que termine en la posición i). Este `max_len` guarda la longitud de la palabra del diccionario más largo, permitiendo que para cada subproblema solo se consideren los intervalos de hasta `max_len` de largo, evitando muchas iteraciones innecesarias. Luego, sabemos que el caso base, es decir, ninguna palabra, siempre estará en el diccionario, por lo que ponemos `True` en la posición 0. Finalmente, pasaremos al análisis de todas las palabras: si en la posición j (posición de inicio del subproblema) hay un fin de palabra (posición i o de fin de la palabra anterior) y además la palabra en el intervalo $[j, i]$ se encuentra en el diccionario, entonces marcamos el final de la palabra como `True` y repetimos esto pero con otro i (agrandamos el subproblema).

Si al haber recorrido toda la cadena llegamos a que no se pudo reconstruir toda la cadena usando palabras del diccionario, entonces este no es un posible mensaje, así que devolvemos `None`.

Si obtuvimos un posible mensaje de la cadena a partir de palabras existentes, será necesario reconstruir el mensaje encriptado. Esto se hará a la inversa, es decir, desde el final de la cadena hasta el principio, viendo siempre que el anterior sea una palabra válida:

```python
def reconstruccion_palabras(prev, cadena):
    palabras = []
```

```
    i = len(cadena)
    while i > 0:
        j = prev[i]
        palabras.append(cadena[j:i])
        i = j
    palabras.reverse()
    return palabras
```

Sea `palabras` el arreglo en el que guardaremos las palabras e i el índice que indicará hasta qué subproblema se está analizando, así como también la condición del ciclo. El algoritmo es el siguiente: mientras que $i > 0$ (estamos dentro de un subproblema), diremos que el inicio de la palabra será la posición i del arreglo de previos, que se guarda en cada posición la posición de inicio de la palabra. Luego, agregamos la palabra al arreglo de `palabras` y disminuimos i. Finalmente, invertimos el arreglo y lo devolvemos.

3.1. Análisis de complejidad

Para el análisis de complejidad sólo tendremos en cuenta el algoritmo que busca un posible mensaje y no todo el código que se encarga de guardar las palabras en un diccionario y de leer las líneas de mensajes encriptados, ya que multiplicar esta complejidad por la cantidad de mensajes de cada archivo sería una diferencia despreciable y sumar la complejidad de cargar las palabras a nuestro diccionario seria $\mathcal{O}(n)$ y no cambiaria la complejidad total.

```
n = len(cadena)
dp = [False] * (n + 1)
dp[0] = True
prev = [-1] * (n + 1)
```

Iniciar el arreglo de soluciones y de previos: $\mathcal{O}(n)$, siendo n el largo de la cadena/mensaje encriptado.

```
max_len = max(len(palabra) for palabra in diccionario)
```

Conseguir la longitud de palabra máxima: $\mathcal{O}(m)$, siendo m la cantidad de palabras del diccionario.

```
for i in range(1, n + 1):
    for j in range(max(0, i - max_len), i):
        if dp[j] and cadena[j:i] in diccionario:
            dp[i] = True
            prev[i] = j
            break
```

En esta parte del código se utilizan dos ciclos `for` anidados. El primero recorre los índices de la cadena desde 1 hasta n, por lo que tiene complejidad $\mathcal{O}(n)$. El segundo bucle itera desde $\max(0, i - \text{max\_len})$ hasta $i - 1$, es decir, como máximo k veces, siendo $k = n - \text{max\_len}$.

Dentro del bucle interno se crea un `slice` de la forma `cadena[j:i]`, operación cuya complejidad es de $\mathcal{O}(k)$ en el peor caso. Además, se verifica si dicha subcadena pertenece al diccionario, lo cual tiene costo $\mathcal{O}(1)$ al estar implementado como un `set`. Por lo tanto, el costo del ciclo interno es $\mathcal{O}(k)$ y como este se repite n veces, la complejidad total del algoritmo es $\mathcal{O}(n \cdot k)$.

¿Por qué no es $\mathcal{O}(n \cdot k^2)$? La confusión podría surgir al considerar el costo del slice cadena[j:i]. Aunque el costo del slice depende de la longitud del segmento (i - j), el rango del bucle interno está limitado por `max_len`. Esto significa que el costo del slice nunca excede $\mathcal{O}(k)$, y no se multiplica por k nuevamente. Si el slice no estuviera limitado por (`max_len`) y se recorriera toda la cadena en cada iteración, entonces la complejidad podría ser $\mathcal{O}(n \cdot k^2)$. Sin embargo, en este caso, el rango del bucle interno está explícitamente limitado por (`max_len`), lo que reduce la complejidad a $\mathcal{O}(n \cdot k)$.

```python
def reconstruccion_palabras(prev, cadena):
    palabras = []
    i = len(cadena)
    while i > 0:
        j = prev[i]
        palabras.append(cadena[j:i])
        i = j
    palabras.reverse()
    return palabras
```

Al contar con un arreglo de previos completado anteriormente, la complejidad baja enormemente con respecto a utilizar un ciclo `for` dentro del `while`. Primero, en cada iteración del bucle `while i >0` se crea un `slice` con complejidad $\mathcal{O}(k)$ en el peor caso. Al sumar todas las longitudes de las cadenas, sabremos que nunca será mayor a n, por lo que el costo total de todos los `slices` será de $\mathcal{O}(n)$, siendo n el largo total de la cadena original. Además, utilizar `palabras.reverse()` tiene complejidad $\mathcal{O}(n)$ también, dando como complejidad final $\mathcal{O}(n)$. Al ejecutar la reconstrucción solo una vez, esta complejidad se suma al $\mathcal{O}(n \cdot k)$ anterior y termina siendo un termino despreciable.

Luego, llegamos a que la complejidad final del algoritmo de programacion dinámica es de $\mathcal{O}(n \cdot k)$. Para $n =$ `len(cadena)` y $k = n - \text{max\_len}$ (max_len = longitud de la palabra mas larga del diccionario). Por simplicidad, podemos asumir que k siempre va a ser menor a n, entonces la complejidad siempre estara acotada por una funcion cuadratica $\mathcal{O}(n^2)$

4. Variabilidad de los Datos

El objetivo de esta sección es analizar cómo distintas características del conjunto de datos afectan el comportamiento del algoritmo, tanto en términos de complejidad teórica como de rendimiento observado en la práctica. Para ello, se identificaron y evaluaron distintos factores que pueden influir sobre el tiempo de ejecución, considerando cómo varía este al modificar uno a la vez, manteniendo los demás constantes. A continuación, se detallan los efectos de cada variable individual sobre el desempeño del sistema.

1. Tamaño del mensaje (cantidad de caracteres)

A medida que crece la longitud del mensaje, también lo hace la cantidad de subproblemas a resolver, ya que el algoritmo de programación dinámica debe considerar más posiciones dentro del mensaje. Esto implica un crecimiento lineal del tiempo de ejecución con respecto al largo del mensaje.

Impacto: Directamente proporcional. Mensajes más largos aumentan significativamente el tiempo de procesamiento.

2. Cantidad de mensajes

Aunque este factor no modifica la complejidad del algoritmo por mensaje, sí afecta el tiempo total de ejecución cuando se procesan múltiples mensajes de manera consecutiva, ya que cada uno se evalúa de forma independiente.

Impacto: No afecta la complejidad individual, pero incrementa el tiempo total cuando se procesan grandes volúmenes de datos.

3. Tamaño del diccionario (cantidad de palabras)

Un diccionario más grande implica una mayor cantidad de palabras candidatas para verificar en cada posición del mensaje. No obstante, si se utilizan estructuras eficientes como conjuntos (`set`)

o *tries*, como en nuestro caso, la verificación de subcadenas puede realizarse en tiempo constante o proporcional al largo de la subcadena.

Impacto: Moderado. Aumenta la cantidad de comparaciones posibles, pero el uso de estructuras eficientes mantiene el rendimiento aceptable.

4. Longitud de las palabras en el diccionario

La longitud promedio de las palabras también influye en el tiempo de ejecución. Cuando el diccionario contiene principalmente palabras cortas, el número de formas válidas de segmentar un mensaje aumenta considerablemente. Esto se debe a que las palabras cortas permiten muchas más combinaciones posibles de corte dentro del mensaje, lo que incrementa el espacio de búsqueda del algoritmo. Como resultado, se generan más estados intermedios en la tabla de programación dinámica, elevando el tiempo de procesamiento.

Por el contrario, si las palabras del diccionario son más largas, hay menos combinaciones posibles por posición del mensaje. Esto restringe las formas en que se puede segmentar el mensaje, reduciendo la ramificación del algoritmo y mejorando el rendimiento general.

Impacto teórico: Las palabras cortas tienden a empeorar el rendimiento al aumentar la cantidad de combinaciones posibles, mientras que las largas mejoran la eficiencia al restringir el espacio de búsqueda.

Entre todos los factores analizados, el que mayor impacto tiene sobre la eficiencia del algoritmo es la longitud del mensaje, seguido por la longitud promedio de las palabras en el diccionario. La cantidad de mensajes y el tamaño del diccionario afectan más al rendimiento general del sistema que al tiempo de procesamiento por mensaje individual.

4.1. Análisis de la variabilidad en los tiempos de ejecución según el tipo de entrada

Para comprender cómo distintas características de los datos afectan el rendimiento del algoritmo, se realizaron pruebas modificando los valores de cada variable de forma controlada. Utilizamos una función generalizada que permite fijar ciertas variables y alterar únicamente aquellas que se desean estudiar, manteniendo constantes las demás. Esto permitió aislar el impacto individual de cada parámetro sobre el tiempo de ejecución.

En el siguiente gráfico se observa cómo varía el tiempo promedio de ejecución en función de diferentes tipos de test. Las pruebas fueron diseñadas para modificar una única característica del conjunto de datos por vez. Se destaca que los mayores tiempos de ejecución corresponden a mensajes más largos y a grandes volúmenes de mensajes, mientras que los tests con palabras cortas no mostraron un aumento significativo en el tiempo, a diferencia de lo que se predice en teoría.

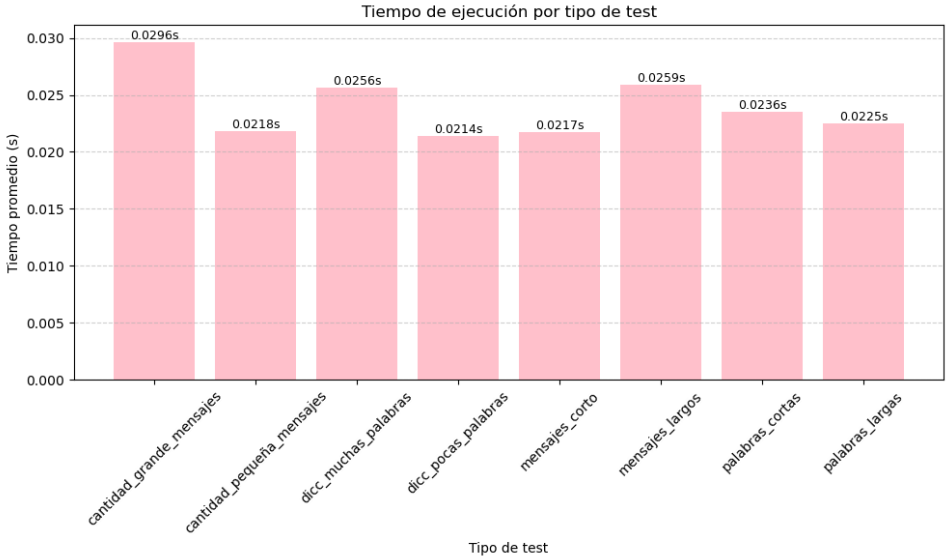

Figura 1: Tiempo promedio de ejecución según el tipo de entrada

5. Mediciones de tiempos para corroborar la complejidad teórica

5.1. Ajuste

Una vez comprobado que la complejidad es $\mathcal{O}(n^2)$, nos dispusimos a realizar los diferentes tests para verificar que, efectivamente, en la práctica se aproxima a una función cuadrática de la forma:

$$f(x) = 2{,}086 \cdot 10^{-13}x^2 + 2{,}476 \cdot 10^{-5}x + 4{,}503 \cdot 10^{-2}$$

Además, para realizar dichos tests creamos nuestros propios sets de datos, asegurándonos de que fueran lo suficientemente variados para obtener un análisis preciso. En ellos, los sets de datos varían de un rango de 10 hasta 100000 palabras por cada mensaje teniendo dos mensajes por archivo (uno válido y uno inválido), asegurándonos que el tiempo que tarda crece cuadráticamente en relación a la cantidad de palabras por cada mensaje.

Finalmente, para obtener la mayor precisión posible en cuanto al tiempo estimado de ejecución de cada set, corrimos cada test 20 veces con los mismos sets de datos, permitiendo la mayor equidad posible con respecto a la variabilidad de los valores entre sets y ejecuciones.

Para lograr esto, decidimos crear cada set de datos de manera aleatoria, es decir, eligiendo palabras para el diccionario a partir del diccionario proporcionado por la catedra y luego eligiendo palabras al azar de nuestra muestra para los mensajes. Luego, como se mencionó anteriormente, se ejecutaron estas pruebas en un archivo auxiliar llamado `pruebas_mediciones.py`, en el que se ejecutaba `tp2.py` junto con el nombre del archivo con el diccionario enviado como parámetro y cada archivo a descifrar como *stdin*.

En lo que respecta a la aproximación por cuadrados mínimos, esta la realizamos con ayuda de las bibliotecas de Python `Matplotlib` y `Numpy`, las cuales proveen una gran cantidad de herramientas para trabajar con matrices y gráficos. Esta vez elegimos una menor cantidad de datos comparado con el trabajo anterior porque aunque la complejidad es similar, el formato de trabajo y los procesos que deben ocurrir hacen que la ejecucion se alargue. Igualmente con los sets de datos generados se puede apreciar que el tiempo que tarda la ejecucion se puede aproximar con una funcion cuadrática.

En el gráfico generado, se puede observar:

- **Puntos rojos**: tiempos promedio de los distintos sets de datos junto con la cantidad de elementos del mismo.
- **Línea azul**: curva ajustada por mínimos cuadrados, que representa una función cuadrática.
- **Línea verde**: curva ajustada por mínimos cuadrados, que representa una función cúbica.
- **Comportamiento general**:
  - **ajuste cuadrático**: sigue de manera más consistente la tendencia de los puntos rojos en todo el rango de datos, desde tamaños pequeños hasta grandes.
  - **ajuste cúbico**: muestra mayores desviaciones en algunos intervalos, especialmente en los valores mas bajos, lo que indica que el modelo cúbico podría estar sobreajustando los datos.
- **Simplicidad del modelo**: La función cuadrática es más simple que la cúbica, lo que la hace más adecuada para describir una relación que ya se sabe teóricamente que es $\mathcal{O}(n^2)$. El modelo cúbico introduce un término adicional (x^3) que no aporta mejoras significativas al ajuste.

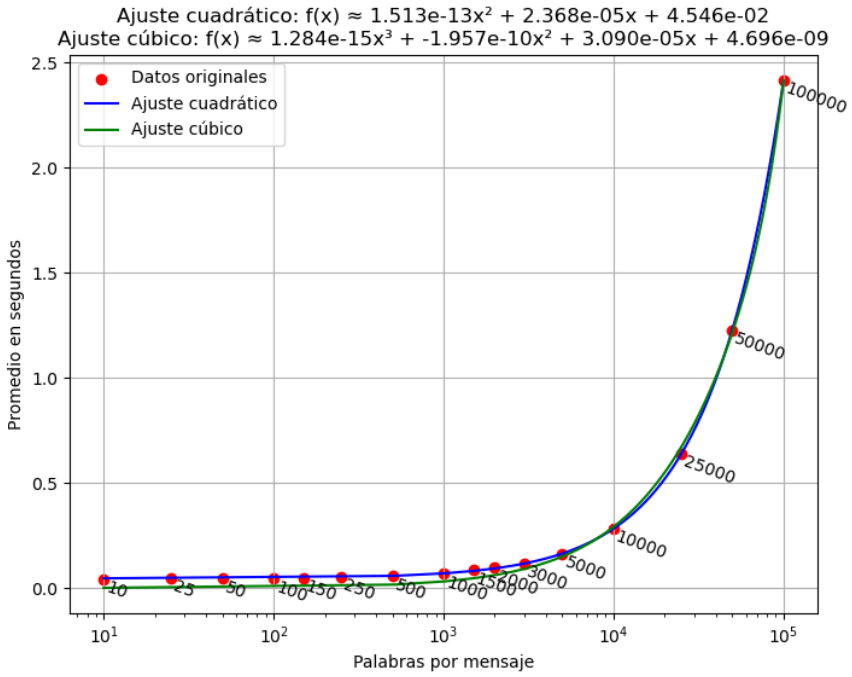

Figura 2: Curvas ajustadas por mínimos cuadrados vs. tiempos de ejecución

Comportamiento: Los resultados obtenidos no solo confirman la complejidad teórica del algoritmo, sino que también validan su comportamiento empírico. La curva cuadrática ajustada describe adecuadamente los datos, mientras que la curva cúbica, aunque más compleja, no mejora significativamente el ajuste y presenta errores mayores en algunos puntos. Esto refuerza la importancia de combinar análisis teóricos con experimentación para evaluar correctamente el rendimiento de un programa.

5.1.1. Error de ajuste

El error de ajuste mide la diferencia absoluta entre los valores reales observados y los valores predichos por los modelos ajustados. En este caso, se evaluaron dos modelos: uno cuadrático

y otro cúbico. Como se observa en el gráfico **"Error de ajuste cuadrático vs. cúbico"**, el modelo cuadrático presenta un error significativamente menor en comparación con el cúbico, lo que confirma que el ajuste cuadrático es más adecuado para describir los datos.

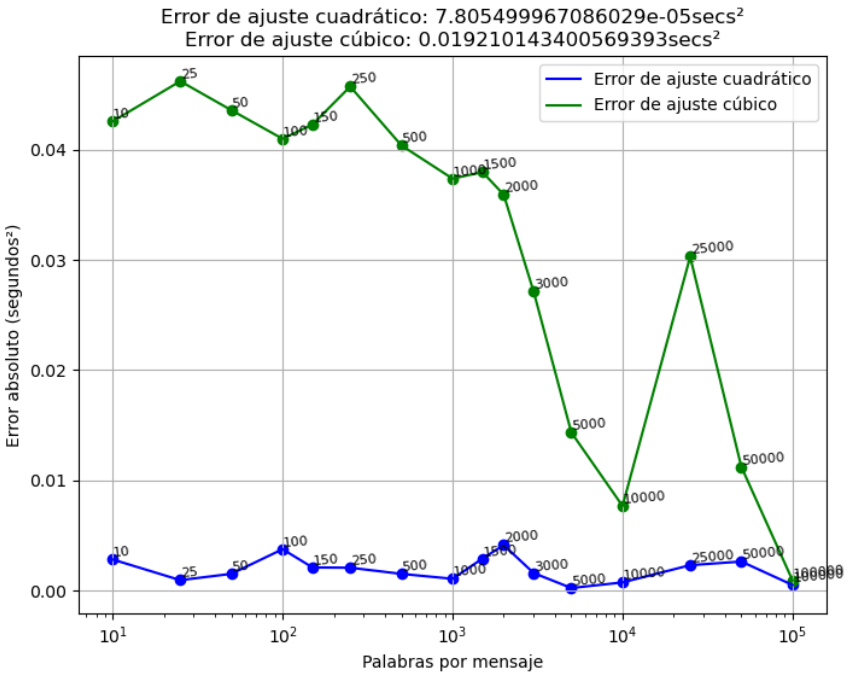

Figura 3: Error de ajuste cuadrático vs. cúbico

Interpretación del Gráfico: En el gráfico, el eje X muestra la cantidad de palabras en cada mensaje por cada prueba realizada (en escala logarítmica), mientras que el eje Y representa el error absoluto en segundos.

- **Puntos azules**: Representan los errores individuales para cada tamaño de prueba en el ajuste cuadrático.
- **Puntos verdes**: Representan los errores individuales para el ajuste cúbico.
- **Líneas conectadas**: Visualizan la tendencia general del error para ambos modelos.
- **Tendencia General**: El error de ajuste cuadrático es bajo en la mayoría de los casos, lo que indica que el modelo cuadrático describe adecuadamente los datos. Por otro lado, el ajuste cúbico presenta errores mucho más altos y una mayor variabilidad a lo largo de todas las muestras, lo que sugiere que no es el modelo más adecuado para estos datos.
- **Picos de Error**: En el ajuste cúbico, se observan picos de error más pronunciados, lo que podría deberse a que el modelo cúbico sobreajusta los datos o no captura correctamente la relación subyacente. En contraste, el ajuste cuadrático mantiene un error bajo y consistente, con picos menores que podrían atribuirse a variaciones aleatorias en los datos o en la ejecución de las pruebas.

Comparación del Error Cuadrático Total: El error cuadrático total, que es la suma del cuadrado de los errores individuales, aunque no es directamente interpretable en la misma escala que los datos originales, es útil para comparar ajustes o evaluar la magnitud relativa de los errores:

- **Error cuadratico total del ajuste cuadrático**: $7{,}805499967086029 \cdot 10^{-5}\,\text{secs}^2$.

- **Error cuadratico tota del ajuste cúbico**: 0,019210143400569393 secs^2.

Estos valores refuerzan que el modelo cuadrático es significativamente más preciso que el cúbico, ya que su error cuadrático total es mucho menor. Además, al tener un valor bajo, complementa a la afirmación de que el ajuste cuadrático es correcto.

Significado: El error de ajuste es una métrica clave para evaluar la calidad del modelo. En este caso, el modelo cuadrático es el más adecuado para describir la relación entre la cantidad de transferencias y el tiempo medido, como lo demuestra su bajo error cuadrático total y su comportamiento más consistente en el gráfico.

Por otro lado, el modelo cúbico, aunque más complejo, no mejora la precisión del ajuste y presenta errores más altos, lo que indica que no es el modelo correcto para estos datos. Esto refuerza la importancia de elegir un modelo que no solo se ajuste a los datos, sino que también sea interpretable y consistente con la teoría subyacente.

En conclusión, el modelo cuadrático describe correctamente los datos generales y confirma la complejidad teórica acotada por $\mathcal{O}(n^2)$ del algoritmo.

6. Conclusiones

En este Trabajo Práctico 2 hemos abordado el problema de validar mensajes cifrados sin espacios mediante un enfoque de Programación Dinámica. Partiendo de la definición formal de la recurrencia y su justificación por inducción, construimos iterativamente la tabladp y el vectorprev para reconocer y reconstruir cualquier segmentación de la cadena en palabras del diccionario. A través de experimentos controlados —variando longitud de mensajes y tamaño/composición del diccionario— comprobamos que el procedimiento identifica con precisión todos los casos “mensaje válido” y rechaza los “no mensajes”. De este modo, se cumple el objetivo de proporcionar una herramienta automática, robusta y de sencilla implementación para detectar posibles filtraciones de texto encriptado. Además, el trabajo demuestra la eficacia de combinar fundamento teórico y validación empírica para garantizar la fiabilidad de algoritmos de segmentación de texto.