Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Algoritmo Programación Dinámica

8 de mayo de 2025

Nicolás Martín Guerrero 112514

1. Introducción del Trabajo

Luego de haber ayudado al Gringo y a Amarilla Pérez a encontrar a la rata que les estaba robando dinero ganado a fuerza de sudor y sangre (no de ellos), hemos sido ascendidos.

Amarilla detectó que hay un soplón en la organización, que parece que se contacta con alguien de la policía. En general eso no sería un problema, ya que la mafia trabaja en un distrito donde media policía está arreglada. El problema es que no parecería ser el caso. El soplón parece estarse contactando con mensajes encriptados.

La organización no está conformada por novatos; no es la primera vez que algo así sucede. Ya han desencriptado mensajes de este estilo, y han charlado amablemente con su emisor. El problema es que en este caso parece ser más complicado. El soplón de esta oportunidad parece encriptar todas las palabras juntas, sin espacios. Eso complica más la desencriptación y validar que el mensaje tenga sentido.

No interesa saber quién es el soplón (de momento), sino más bien qué información se está filtrando. El área de desencriptación se encargará de intentar dar posibles resultados, y nosotros debemos validar si, en principio, es un posible mensaje. Es decir, si nos dan una cadena desencriptada que diga .^{es}tanocheenelmuellealassiete", este sería un posible mensaje, el cual correspondería a .^{es}ta noche en el muelle a las siete", mientras que .^{es}tamikheestado"no lo es, ya que no podemos separar de forma de generar todas palabras del idioma español (en cambio, si fuera .^{es}tamiheestado"podría ser .^{es}ta mi he estado). No es nuestra labor analizar si el texto tiene potencialmente sentido o no, de eso se encargará otro área.

El Gringo nos sugirió que usemos programación dinámica para esto. Y, en general, cuando lo sugiere, es porque es necesario. Eso, y que no seguir con su sugerencia puede implicar desayunar con los peces.

A considerar: Dado que estamos trabajando en español, no es necesario considerar el caso que pueda haber más de una opción para la separación (por ejemplo, esto podría pasar con .^{es}tamosquea", que puede separarse en .^{es}tamos que a.^{o} bien .^{es}ta mosquea"), ya que son pocos casos, muy forzados y poco probables. Si el trabajo fuera en un idioma germánico (como el alemán) esto podría ser un poco más problemático.

1.1. Consigna del Problema

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 n de palabras, y una cadena desencriptada, determinar si es un posible mensaje (es decir, 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 con algunos casos particulares para que puedan usar inicialmente 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 sets de datos).

Esta corroboración empírica debe realizarse confeccionando gráficos correspondientes, y utilizando la técnica de cuadrados mínimos. Para esto, proveemos una explicación detallada, en conjunto de ejemplos.

2. Análisis del Problema

Para explicar mejor este problema, primero debemos definir qué es un algoritmo **Programación dinámica**. Esta técnica de diseño de algoritmos se utiliza para resolver problemas que pueden descomponerse en subproblemas más pequeños, aprovechando que la solución óptima del problema original puede obtenerse a partir de las soluciones óptimas de sus subproblemas. Y para evitar recomputar los mismos subproblemas varias veces, la programación dinámica almacena los resultados intermedios, lo que permite reducir drásticamente la complejidad temporal en muchos casos.

En nuestro caso, el problema consiste en determinar cuándo termina una palabra en un mensaje descifrado sin espacios. Debido a la incertidumbre del tamaño del mensaje y dado que los mensajes pueden tener palabras sin sentido, el mejor enfoque es verificar palabra por palabra. Como los mensajes tienen muchas palabras que pueden ser tratadas como subproblemas, la solución del mensaje final es la sumatoria de las soluciones de cada palabra.

3. Elección del Algoritmo

Cómo el objetivo es segmentar una cadena(sin espacios) en palabras válidas de un diccionario, colocando espacios donde correspondan. El objetivo es encontrar una segmentación válida completa, si existe, no la más corta ni la más larga, solo una correcta. Viendo a la cadena(de nombre str) como el problema principal y cada letra de esta como una posible palabra que sería un subproblema, podemos abordar este problema con un enfoque de programación dinámica.

Consideramos que en cada posición i de la cadena existe un subproblema que consiste en determinar si es posible segmentar el sufijo desde dicha posición usando únicamente palabras del diccionario. Por lo tanto, se plantea una relación de recurrencia que se resuelve de forma ascendente, almacenando en una tabla las soluciones a estos subproblemas.

De forma matematica:

$$OPT(i) = \begin{cases} \text{espacio} & \text{si } i{=}str \\ \text{el primer } str[i:j]\text{+”(espacio)-}OPT(j) & \text{tal que } str[i:j] \text{ pertenece al diccionario y } OPT(j) \text{ != none} \end{cases}$$

Esta ecuación refleja cómo se construye la solución desde el final hacia el inicio de la cadena: se busca el primer segmento válido desde la posición i que permita formar una segmentación completa, y se guarda en una tabla para evitar reevaluar los mismos sufijos.

El algoritmo itera hacia atrás desde la última posición de la cadena, asegurándose de que todos los subproblemas necesarios estén disponibles al momento de tomar decisiones. Si se logra llenar la posición $dp[0]$, se ha encontrado una segmentación completa de la cadena.

Este planteo asegura que, si existe una forma de segmentar la cadena utilizando únicamente palabras del diccionario, el algoritmo la encontrará. Además, como los resultados intermedios se almacenan en una tabla de forma iterativa, se evita el uso de la pila de llamadas y se logra una solución robusta y eficiente.

Cómo lo mencionamos previamente, nuestra estrategia se fundamenta en **encontrar palabras dentro del mensaje empezando por su ultima letra** para luego comparar si el segmento desde esa letra en adelante forma una palabra. Esto se ve principalmente en este fragmento del codigo:

```
for i in range(n - 1, -1, -1):
        for j in range(i + 1, n + 1):
            palabra = s[i:j]
            if palabra in diccionario and opt[j] is not None:
                opt[i] = palabra if opt[j] == "" else palabra + " " + opt[j]
                break
    return opt[0]
```

3.1. Demostración de Ecuación de Recurrencia

- D es el diccionario.
- es la cadena vacía.
- $\mathcal{O}_{\sqrt{\sqcup}}(i)$ retorna la segmentación óptima desde la posición i hasta el final.

Queremos probar mediante iduccion que: "Para toda posición $\mathcal{O}_{\sqrt{\sqcup}}(i)$retorna la segmentación válida si existe, o None en caso contrario."

Consideramos el caso base ($i = n$): $\mathcal{O}_{\sqrt{\sqcup}}(n)=$ entonces, es correcto (no hay más caracteres para segmentar).

- **Pasos inductivos**($i < n$)

Se asume que $\mathcal{O}_{\sqrt{\sqcup}}(i)$ es óptimo para todo $j > i$(hipótesis inductiva).Luego, Para $i < n$, si $str[i : j] \in D$ y $OPT(j)$ es válido, entonces $str[i : j] + "" + OPT(j)$ también lo es. El primer j válido asegura completitud.Entonces, la solución para $str[i :]$ depende solo de soluciones válidas para $str[j :]$ con $j > i$.

4. Complejidad del Algoritmo

4.1. Explicacion de la Complejidad

El algoritmo que implementamos(*segmentar texto*) utiliza un enfoque de programación dinámica con tabulacion, lo cual permite optimizar la resolución del problema evitando la resolución repetida de subproblemas ya resueltos.

```
    def segmentar_texto(s, diccionario):
    n = len(s)
    opt = [None] * (n + 1)
    opt[n] = ""
    for i in range(n - 1, -1, -1):
        for j in range(i + 1, n + 1):
            palabra = s[i:j]
            if palabra in diccionario and opt[j] is not None:
                opt[i] = palabra if opt[j] == "" else palabra + " " + opt[j]
                break
    return opt[0]

```

En este enfoque, se define una tabla `opt` de tamaño $n+1$, donde $opt[i]$ contiene la segmentación válida del sufijo de la cadena desde la posición i en adelante, si existe. El valor `opt[n]` se inicializa como cadena vacía, ya que representa el caso base (segmentar una cadena vacía es válido).

El ciclo externo itera desde el final hacia el inicio de la cadena, y el ciclo interno evalúa todas las posibles subcadenas $s[i:j]$. Si se encuentra una palabra válida que permite continuar con una segmentación completa desde j, se almacena la solución parcial en $opt[i]$.

```
for i in range(n - 1, -1, -1):
        for j in range(i + 1, n + 1):
            palabra = s[i:j]
            if palabra in diccionario and opt[j] is not None:
                opt[i] = palabra if opt[j] == "" else palabra + " " + opt[j]
                break
```

- El ciclo externo itera n veces (una por cada posición).
- En el peor caso, el ciclo interno también recorre hasta n posiciones por iteración externa.

Por lo tanto, la complejidad total es:

$$\mathcal{O}(n^2)$$

A diferencia de otros algoritmos dinámicos que almacenan decisiones por separado y luego requieren una fase explícita de reconstrucción, en este caso la segmentación se va construyendo directamente en *opt*:

Esto permite evitar una segunda fase de reconstrucción y simplifica el diseño, aunque el proceso de concatenación de strings podría tener un impacto menor en eficiencia, mitigado por el uso de tabulación.

4.2. Prueba de la Complejidad con Casos Pseudo-Aleatorios

Con el objetivo de comprobar la complejidad teórica del algoritmo de segmentación que implementamos, realizamos una serie de pruebas midiendo el *tiempodeejecución* frente a la *longitudtotaldeltexto* (en caracteres). Esperamos obtener una relación cuadrática entre ambos, que cumpla la forma $\mathcal{O}(n^2)$.

Para calcular el error por **cuadrados mínimos** a la función cuadrática se utilizó el siguiente fragmento de código con la librería de python *scipy*:

```
    f_teorica = lambda x, c1, c2: c1 * x**2 + c2
    c, _ = scipy.optimize.curve_fit(f_teorica, x, y)
    y_fit = f_teorica(x, *c)

    error_total = np.sum((y - y_fit) ** 2)
    print(f"Error cuadr tico total: {error_total:.6e}")
```

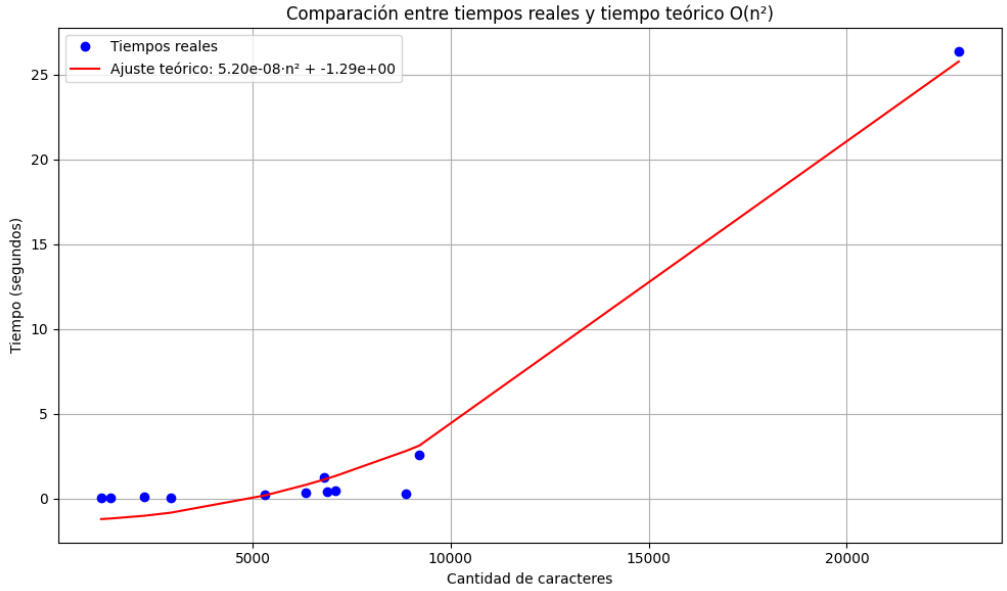

Figura 1: Gráfico de Complejidad con Casos dados por la catedra.
Error cuadrático total: 1.358995e+01

El primer caso fue realizado tomando los resultados dados por los datos proporcionados por la cátedra. En el gráfico se observa que los tiempos de ejecución se alinean con una curva cuadrática ajustada por cuadrados mínimos. El error cuadrático total fue de 13.58, lo cual sugiere una fuerte relación entre el modelo teórico y los datos medidos.

Luego creamos nuevos mensajes que usan palabras de diccionarios dados por la catedra, como *corto.txt* y *grande.txt*, utilizamos el siguiente script para generar esta información mas aleatoria:

```
    import random

# Leer el diccionario
with open("Ejemplos\gigante.txt", "r") as file:
    palabras = [line.strip() for line in file]

# Generar mensajes aleatorios
num_mensajes = 2000  # Cantidad de mensajes a generar
longitud_min = 10    # M nimo de palabras por mensaje
longitud_max = 15   # M ximo de palabras por mensaje

mensajes = []
for _ in range(num_mensajes):
    num_palabras = random.randint(longitud_min, longitud_max)
```

```
    mensaje = "".join(random.choices(palabras, k=num_palabras))
    mensajes.append(mensaje)

# Guardar los mensajes en un archivo
with open("Ejemplos_random\mensajes_2000_gigante_largo.txt", "w") as file:
    for mensaje in mensajes:
        file.write(mensaje + "\n")
```

Generamos dos tipos de archivos de entrada adicionales:

1. En el primer conjunto de pruebas, mantuvimos **fijo un diccionario** (de tamaño mediano) y aumentamos progresivamente la cantidad de mensajes. Esto nos permitió observar cómo escala el tiempo de ejecución en función de la cantidad total de texto procesado, manteniendo constante la variedad de palabras.

2. En el segundo conjunto, mantuvo **fija la cantidad de mensajes** (100 mensajes por archivo), pero se modificó el tamaño del diccionario, utilizando versiones progresivamente más grandes. De esta forma, evaluamos cómo impacta la variabilidad de palabras y la ambigüedad potencial en el rendimiento del algoritmo.

y se obtuvieron los siguientes graficos:

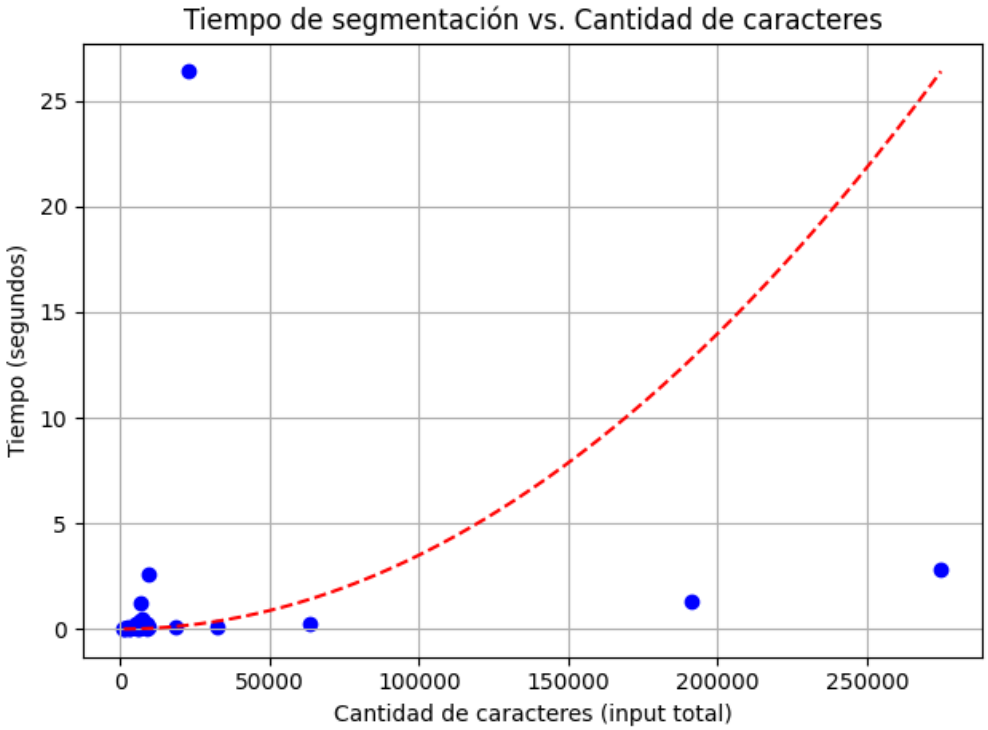

Figura 2: Gráfico de Complejidad con todos los ejemplos.

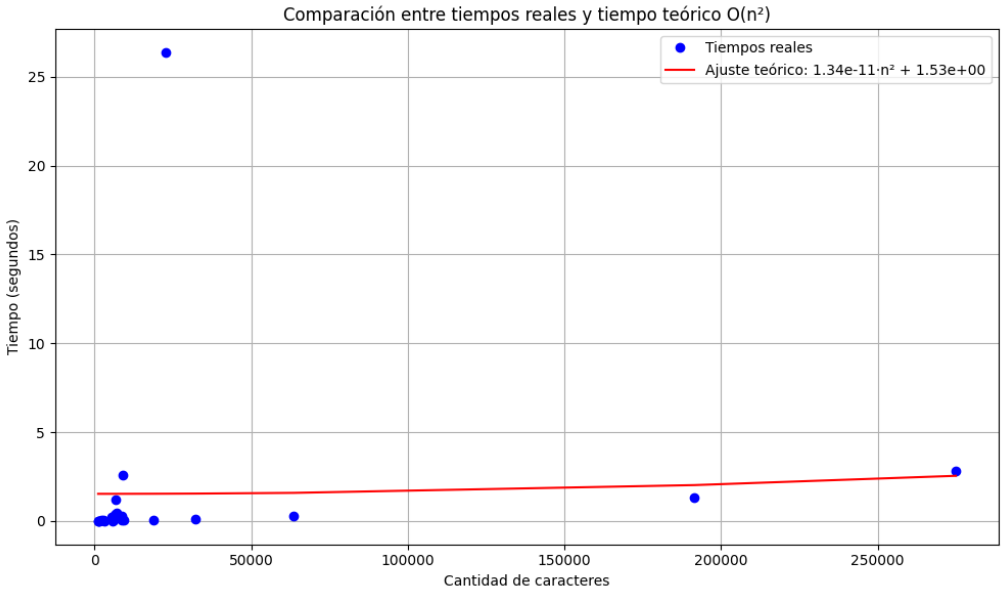

Figura 3: Gráfico de Complejidad con Casos Pseudo-Aleatorios.
error cuadratico total: 6.549898e+02 con mis ejemplos

Como se puede observar en ambos gráficos, los nuevos datos generados aumentan en tamaño(cantidad de caracteres) pero el tiempo de ejecución no incrementa como se espera. En la figura 2 se puede ver como estos nuevos datos hacen que la complejidad teórica no se vea reflejada en la practica, aumentando mucho el error cuadrático.

Este desvío puede explicarse por la estructura de los datos: en muchos casos, el algoritmo encuentra rápidamente una segmentación válida sin necesidad de evaluar todas las combinaciones posibles. Esto hace que, en la práctica, la cantidad de operaciones ejecutadas sea menor a la esperada en el peor caso.

Finalmente, realizamos una tercera serie de pruebas **manteniendo constante tanto el diccionario como la cantidad de mensajes** (1000), pero aumentando la cantidad de palabras por mensaje, es decir, la longitud total de cada uno. Esta variante nos permitió observar cómo escala el algoritmo cuando se incrementa directamente el volumen de caracteres procesados, sin variar la complejidad de palabras ni el número de decisiones a tomar en paralelo.

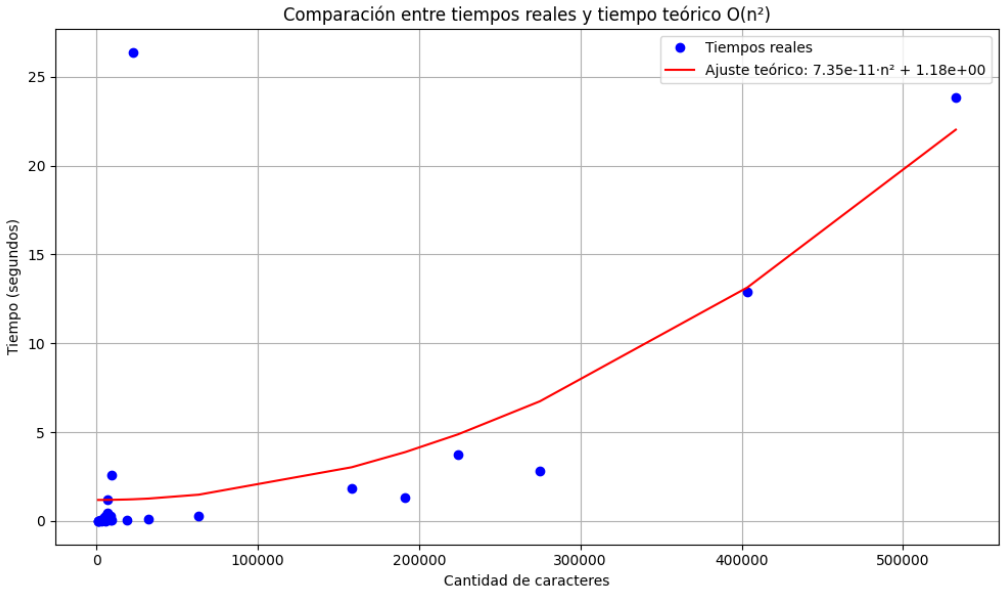

Figura 4: Gráfico de Complejidad con mensajes largos.
error cuadratico total: 6.843579e+02 con mis ejemplos

Se observa que, si bien los puntos presentan cierta dispersión y el error cuadrático total fue de 684.36, las mediciones tienden a alinearse con una curva de crecimiento cuadrático. Esto respalda la hipótesis de que el algoritmo tiene una complejidad temporal teórica de $\mathcal{O}(n^2)$, aunque su comportamiento práctico se ve afectado por factores como:

1. La distribución y estructura de los mensajes.
2. La rapidez con la que se encuentra una segmentación válida desde cada posición.
3. La eficiencia del corte temprano en el ciclo interno.

En conjunto, el comportamiento empírico observado es compatible con una complejidad cuadrática, especialmente en entradas grandes, aunque en casos más estructurados o más chicos, el rendimiento tiende a ser mejor que el peor caso teórico.

4.3. Variabilidad de la Complejidad

Aunque la complejidad teórica del algoritmo es $\mathcal{O}\left(n^2\right)$, donde n es la longitud de la cadena a segmentar, los tiempos reales de ejecución pueden variar según ciertas características de los datos de entrada:

1. **Tamaño del mensaje**: A mayor longitud del mensaje, mayor es la cantidad de subproblemas posibles, y por tanto, el número de iteraciones. Esto afecta directamente la cantidad de veces que se utiliza la función de recurrencia antes de que se encuentre el resultado.

2. **Tamaño del diccionario**: Aunque el diccionario sea un *set* para obtener búsquedas en tiempo constante, su tamaño puede influir indirectamente: a mayor cantidad de palabras, mayor probabilidad de que un segmento $s[i:j]$ coincida con alguna palabra del diccionario implica que, desde cada posición i, habrá más candidatos válidos para iniciar una segmentación. Esto puede incrementar la cantidad de iteraciones necesarias dentro del ciclo interno, ya que el algoritmo debe evaluar múltiples subcadenas antes de encontrar una que permita completar la segmentación (o confirmar que no es posible).

3. **Longitud promedio de las palabras del diccionario**: Si las palabras del diccionario son muy cortas, el costo computacional se ve influido por la cantidad de combinaciones posibles desde cada posición: a mayor número de coincidencias con el diccionario, mayor será el esfuerzo requerido para explorar todas las alternativas válidas y almacenar la mejor solución en la tabla

Entonces, aunque la ***complejidad teórica*** está acotada por la longitud de la cadena, el **rendimiento práctico** del algoritmo depende significativamente de la estructura y variabilidad del `diccionario` y los mensajes a analizar.

Esta variabilidad también se reflejó en los gráficos obtenidos a partir de las mediciones prácticas. En particular, vemos que el **mayor impacto** en el tiempo de ejecución no proviene simplemente de la *cantidad total de caracteres*, sino de la **cantidad de palabras por mensaje**, es decir, del número de decisiones de segmentación que el algoritmo debe tomar. En los experimentos donde se mantuvo constante la cantidad de mensajes pero se incrementó la cantidad de palabras por mensaje, el tiempo creció de forma mucho mayor. Esto indica que, si bien el costo en teoría depende del `largo total de la cadena`, el **factor dominante en la práctica** es la cantidad de cortes válidos posibles en cada mensaje.

Sin embargo, la variabilidad **no modifica** la complejidad temporal en términos asintóticos. Gracias a la `tabla de programación dinámica`, cada subproblema (cada índice de inicio en la cadena) se resuelve como máximo una vez, y en cada caso se prueban hasta $\mathcal{O}(n)$ posibles extensiones de palabra. Esto establece una **cota superior** de $\mathcal{O}\left(n^2\right)$ que se mantiene *independientemente* del contenido específico del diccionario o del mensaje. Es decir, aunque ciertos casos particulares puedan resolverse más rápidamente en la práctica (por ejemplo, si se encuentra una solución válida muy temprano), el algoritmo siempre está preparado para enfrentar el **peor escenario**, en el que debe explorar **todos los posibles segmentos** desde cada posición. Por lo tanto, la **complejidad global** del algoritmo sigue siendo $\mathcal{O}\left(n^2\right)$.