.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Que parezca programación dinámica

5 de mayo de 2025

Barletta Brenda
112184

Docampo Torrico Daniel Rodolfo
112395

Rivas Sofia Belen
112216

1. Consideraciones

Por medio del presente informe, se dará presentación al análisis del problema "Que parezca programación dinámica", cuyo objetivo es desarrollar una solución basada en programación dinámica para determinar si una cadena de caracteres desencriptada, sin espacios, puede corresponder a una secuencia válida de palabras en español.

1.1. Contexto del problema

La historia de fondo nos sitúa como miembros ascendidos de una organización mafiosa ficticia que ha detectado un posible soplón en sus filas. Este sujeto estaría filtrando información en mensajes encriptados sin espacios entre palabras, lo que dificulta validar si los mensajes tienen sentido. Aunque la semántica del mensaje no se evalúa, sí se requiere determinar si es una secuencia **potencialmente válida** de palabras. Para esto, se cuenta con un listado de palabras válidas (diccionario), y se desea verificar si es posible segmentar la cadena recibida de forma tal que todas las subcadenas sean palabras contenidas en dicho listado.

1.2. Consigna

Se nos solicita:

- Plantear y analizar una ecuación de recurrencia adecuada al problema.
- Proponer un algoritmo eficiente utilizando Programación Dinámica que determine si una cadena dada puede segmentarse en palabras válidas.
- Implementar el algoritmo y justificar su complejidad teórica.
- Estudiar cómo diferentes factores (como longitud de la cadena, tamaño del diccionario, etc.) afectan los tiempos de ejecución.
- Validar empíricamente la complejidad teórica mediante mediciones, gráficos y técnica de cuadrados mínimos.
- Presentar ejemplos de ejecución para corroborar la validez del algoritmo.

1.3. Objetivos del informe

Este trabajo tiene como objetivo principal analizar el problema de segmentación de cadenas en el contexto de verificación lingüística, mediante técnicas de programación dinámica. Se busca:

- Demostrar que es posible aplicar programación dinámica a problemas de verificación léxica.
- Formalizar una solución que determine si una cadena puede ser segmentada completamente en palabras del idioma.
- Analizar tanto teórica como empíricamente la eficiencia del algoritmo propuesto.
- Documentar los pasos, decisiones y experimentos realizados durante el desarrollo.

2. Análisis del problema

El problema planteado consiste en determinar si una cadena de caracteres sin espacios puede segmentarse completamente en palabras válidas de un idioma dado un diccionario. Esta situación se encuadra en un conjunto de problemas clásicos de programación dinámica, donde se busca tomar decisiones óptimas basadas en subproblemas previamente resueltos.

2.1. Enfoque inicial y entendimiento del problema

Desde un primer análisis, se interpretó que el problema consiste en validar si existe una segmentación de la cadena original tal que todas las subcadenas obtenidas estén presentes en un listado de palabras válidas. La condición es que no deben quedar letras sin asignar, ni permitir fragmentaciones incorrectas. Es decir, se busca una partición total de la cadena, utilizando únicamente combinaciones de palabras conocidas.

Este tipo de problema presenta una estructura característica de muchos problemas de programación dinámica: se puede abordar dividiendo el problema global en subproblemas más pequeños que comparten estructura similar y que pueden resolverse de forma acumulativa. En particular, la idea es verificar si la subcadena comprendida entre las posiciones 0 e i puede segmentarse en palabras válidas, reutilizando la información obtenida para subcadenas anteriores.

Además, se identificó rápidamente que este problema no busca optimizar una medida (como beneficio, costo o cantidad de cortes), sino simplemente determinar la existencia de una solución válida. Por ello, se opta por una estructura de decisión booleana que indique si una determinada posición en la cadena puede alcanzarse mediante una segmentación válida.

Durante esta etapa inicial también se tuvo en cuenta el contexto específico del idioma español, que facilita el análisis al no presentar una ambigüedad elevada en la segmentación (a diferencia de otros idiomas como el alemán). Esto permitió acotar el problema a una única solución válida, sin necesidad de explorar todas las segmentaciones posibles.

Finalmente, se planteó que, si bien el foco está en validar la cadena, también puede ser deseable reconstruir la segmentación encontrada. Por eso, desde el inicio se pensó en almacenar no solo la información booleana de alcanzabilidad, sino también los índices que permitirían retroceder en la cadena y reconstruir el mensaje completo.

2.2. Propuestas consideradas

El análisis se orientó desde el inicio a explorar distintas formas de aplicar la programación dinámica, comparando variantes estructurales y de implementación que permitieran construir una solución eficiente y extensible.

Una de las primeras propuestas consideradas fue implementar un enfoque **top-down con memoización**, dado que es, en muchos casos, la aproximación más intuitiva para quienes abordan por primera vez un problema de Programación Dinámica. En este enfoque se define una función recursiva que, dada una posición en la cadena, determina si es posible segmentar el sufijo restante. La función intenta cortar la cadena en todos los puntos posibles que generen una palabra válida, y guarda en memoria (memo) el resultado de cada posición para evitar recomputaciones redundantes.

Si bien este diseño resulta natural y directo de programar, especialmente cuando se visualiza el problema como una sucesión de elecciones recursivas, presenta ciertas limitaciones prácticas. En particular, puede implicar una mayor complejidad de implementación si se desea extender el algoritmo para permitir la reconstrucción del mensaje segmentado, ya que no se almacenan de forma explícita los puntos de corte. Además, en ciertos entornos o lenguajes, el uso excesivo de recursión puede generar problemas de profundidad de pila, especialmente para cadenas largas.

Estas observaciones motivaron el paso hacia un enfoque iterativo más controlado, utilizando una estrategia *bottom-up*, donde se construye iterativamente una tabla (por ejemplo, un arreglo booleano de longitud $n + 1$, siendo n la longitud de la cadena) en la que la posición i indica si

la subcadena `cadena[0:i]` puede segmentarse en palabras válidas. Este enfoque permite mayor control sobre el flujo de construcción y facilita la reconstrucción de la solución si se desea mostrar explícitamente el mensaje segmentado.

Se evaluaron también dos variantes para optimizar el tiempo de ejecución:

- **Limitar la longitud máxima de palabras consideradas**: dado que en español las palabras no suelen ser extremadamente largas, se definió una cota para evitar explorar prefijos que seguramente no estén en el diccionario.
- **Utilizar estructuras eficientes de búsqueda**: como conjuntos (‘set‘) para verificar la existencia de palabras válidas en tiempo constante.

Se analizaron también estructuras adicionales para permitir la reconstrucción del mensaje: por ejemplo, almacenar en cada posición de la tabla el índice anterior desde el cual se realizó un corte válido. Esto facilita reconstruir la secuencia de palabras al finalizar el procesamiento.

2.3. Contraejemplos a las propuestas anteriores

En el proceso de análisis se identificaron ciertos casos que permitieron descartar o ajustar variantes dentro del enfoque de programación dinámica.

Por ejemplo, en la versión **top-down con memoización**, se observó que, si bien se evitaban recálculos, la reconstrucción del mensaje segmentado resultaba poco natural y requería estructuras adicionales para rastrear los cortes, lo cual complicaba innecesariamente la implementación. Además, en ciertos lenguajes como Python, la recursividad excesiva puede ser limitada por el stack, lo que motivó un enfoque iterativo.

En cuanto a la implementación **bottom-up**, se identificaron contraejemplos donde no limitar la longitud máxima de palabras generaba exploraciones innecesarias. Por ejemplo, en una cadena como `'estamosenelmuelle'`, si no se establece un límite, se generan prefijos como `'estamosenelmu'`, que claramente no son palabras, y cuya verificación resulta costosa si el diccionario es grande. Este tipo de casos motivó la incorporación de una longitud máxima basada en la palabra más larga del diccionario.

También se consideró un caso como `'salirnosnoesunaopcion'`, en el que múltiples prefijos válidos (`'salir'`, `"salirnos"`, `'salirnosno'`) podrían generar múltiples caminos en la tabla. Esto obligó a definir una lógica clara: si en una posición ya se encontró una segmentación válida, se continúa el recorrido sin necesidad de buscar todas las posibles alternativas, ya que no se exige exhaustividad en la segmentación.

Estos ejemplos ayudaron a consolidar una implementación dinámica eficiente, clara y ajustada a las necesidades del problema.

2.4. Naturaleza del problema

Dada la estructura del problema y su análisis inicial, se concluye que este posee características fundamentales que lo vuelven especialmente adecuado para ser abordado mediante programación dinámica, tal como se requiere en la consigna.

El objetivo es determinar si una cadena de caracteres sin espacios puede ser completamente segmentada en fragmentos que correspondan a palabras válidas incluidas en un diccionario. Para lograrlo, se construye una estructura de decisión (en este caso, un arreglo booleano) donde cada posición $c[i]$ indica si la subcadena $s[0:i]$ puede ser segmentada correctamente.

Esta formulación se basa en el principio de subproblemas superpuestos: si sabemos que una subcadena anterior puede segmentarse correctamente, basta con verificar si el segmento restante forma una palabra válida para extender la solución. Este tipo de enfoque acumulativo es característico de muchos problemas clásicos de programación dinámica.

El problema presenta similitudes estructurales con otros problemas canónicos de este paradigma:

- **Corte de soga**: se busca particionar una unidad (longitud, número) en fragmentos válidos bajo ciertas condiciones. Aquí, el fragmento válido es una palabra del diccionario.
- **Cambio de monedas**: donde se intenta construir una suma exacta combinando elementos discretos. En nuestro caso, se intenta construir la cadena completa a partir de segmentos válidos.
- **Subset Sum**: donde se requiere encontrar una combinación de elementos que sumen un valor exacto. En esta analogía, se busca una combinación de posiciones de corte que cubra la cadena sin dejar caracteres fuera.

Además, el problema se resuelve eficientemente porque no busca maximizar ni minimizar una función objetivo, sino simplemente verificar la existencia de una partición válida. Esto permite el uso de una estructura booleana que reduce la complejidad conceptual y algorítmica del abordaje.

2.5. Condiciones para que haya solución

Las condiciones necesarias para considerar que una cadena puede segmentarse correctamente son las siguientes:

- La cadena debe cubrirse completamente mediante fragmentos consecutivos que pertenezcan al diccionario.
- Las particiones deben ser contiguas y no solapadas, es decir, deben reconstruir la cadena original sin omisiones ni repeticiones.
- El algoritmo debe encontrar, a lo largo del recorrido, una secuencia de cortes que respete la condición de validez del diccionario y que comience desde el índice cero, alcanzando exactamente el final de la cadena.
- Si en la posición final $c[n]$ del arreglo de decisiones se encuentra un valor `True`, entonces existe al menos una forma de segmentar correctamente la cadena. Caso contrario, no es un mensaje válido.

3. Ecuación de recurrencia

El problema consiste en determinar si una cadena de caracteres s, sin espacios y de longitud n, puede segmentarse completamente en palabras válidas que pertenezcan a un diccionario D. Para modelar esta situación, se define una función booleana $c[i]$ tal que:

- $c[i]$ = `True` si la subcadena $s[0 : i]$ (los primeros i caracteres de s) puede segmentarse completamente en palabras del diccionario D,
- $c[i]$ = `False` en caso contrario.

La ecuación de recurrencia que se utiliza para resolver el problema es la siguiente:

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

Justificación:

- El caso base $c[0]$ = `True` representa que la cadena vacía puede segmentarse trivialmente.
- Para $i > 0$, se busca algún índice $j < i$ tal que la subcadena anterior $s[0 : j]$ pueda segmentarse (es decir, $c[j]$ = `True`) y que el segmento actual $s[j : i]$ sea una palabra válida (pertenezca al diccionario D). Si tal j existe, se concluye que $s[0 : i]$ también es segmentable.

Ejemplo ilustrativo

Sea la cadena:

$$s = \texttt{'ellamarco'}$$

Y el diccionario:

$$D = \{\texttt{'el'}, \texttt{'ella'}, \texttt{'mar'}, \texttt{'marco'}, \texttt{'arco'}\}$$

Una segmentación válida es:

$$\texttt{'ella'}, \quad \texttt{'marco'}$$

A continuación se presenta cómo se construye el arreglo c (de tamaño $n + 1 = 10$) paso a paso:

Construcción del arreglo `c`:

Índice i	0	1	2	3	4	5	6	7	8	9
Carácter $s[i]$		e	l	l	a	m	a	r	c	o
Valor $c[i]$	T	F	T	F	T	F	F	T	F	T

Paso a paso:

- $c[0]$ = `True`: caso base.
- $c[2]$ = `True`: $s[0 : 2]$ = `'el'` $\in D$ y $c[0]$ = `True`.
- $c[4]$ = `True`: $s[0 : 4]$ = `'ella'` $\in D$ y $c[0]$ = `True`.

- $c[7] =$ `True`: $s[4:7] =$ `'mar'` $\in D$ y $c[4] =$ `True`.
- $c[9] =$ `True`: $s[4:9] =$ `'marco'` $\in D$ y $c[4] =$ `True`.

Interpretación final:

El valor $c[9] =$ `True` indica que la cadena completa puede segmentarse en palabras del diccionario. La ecuación de recurrencia permite construir la solución de forma progresiva, evaluando en cada paso si existe una segmentación válida hasta ese punto.

3.1. Demostración de la correctitud de la ecuación

La correctitud de la ecuación de recurrencia propuesta puede demostrarse mediante el método de **inducción matemática** sobre la longitud del prefijo considerado.

Hipótesis: Sea s una cadena de longitud n y D un diccionario de palabras válidas. Definimos $c[i]$ como una variable booleana que indica si el prefijo $s[0:i]$ puede segmentarse completamente en palabras de D. Afirmamos que la ecuación:

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

es correcta, es decir: $c[i] =$ `True` si y solo si $s[0:i]$ puede segmentarse completamente en palabras del diccionario.

Demostración por inducción:

Caso base: $i = 0$ La cadena vacía puede considerarse segmentable trivialmente, ya que no contiene caracteres y no requiere cortes. Por definición, $c[0] =$ `True`, lo cual coincide con la ecuación dada.

Paso inductivo: Supongamos que la afirmación es verdadera para todas las posiciones menores que i, es decir, que para todo $j < i$, $c[j] =$ `True` si y solo si $s[0:j]$ puede segmentarse completamente en palabras válidas.

Queremos demostrar que $c[i] =$ `True` si y solo si existe algún $j < i$ tal que:

- $s[0:j]$ es segmentable (es decir, $c[j] =$ `True`), y
- $s[j:i] \in D$ (es una palabra válida).

($\Rightarrow$) Supongamos que $c[i] =$ `True`. Entonces, por definición, el prefijo $s[0:i]$ puede segmentarse completamente. Esto implica que existe una última palabra válida que termina en la posición i, y que todo lo anterior también puede segmentarse. Es decir, existe algún $j < i$ tal que $s[j:i] \in D$ y $s[0:j]$ es segmentable, por lo tanto $c[j] =$ `True`. Esto valida la condición de la ecuación.

($\Leftarrow$) Supongamos que existe $j < i$ tal que $c[j] =$ `True` y $s[j:i] \in D$. Entonces, $s[0:j]$ puede segmentarse, y la subcadena restante $s[j:i]$ es una palabra válida. La concatenación de ambas constituye una segmentación completa del prefijo $s[0:i]$, por lo tanto $c[i] =$ `True`.

Conclusión: Por el principio de inducción, se cumple que para todo $i \in [0, n]$, el valor de $c[i]$ indica correctamente si el prefijo $s[0:i]$ puede segmentarse completamente en palabras del diccionario. Esto garantiza la **correctitud** de la ecuación de recurrencia utilizada en el algoritmo.

4. Algoritmos propuestos

En esta sección se presentan los algoritmos desarrollados para abordar el problema de segmentación de una cadena en palabras válidas mediante programación dinámica. La implementación se divide en dos partes: por un lado, el algoritmo de decisión, que determina si la segmentación es posible; y por otro, el algoritmo de reconstrucción, que permite recuperar la secuencia de palabras si la segmentación es válida.

4.1. Algoritmo de programación dinámica propuesto

El algoritmo de decisión utiliza un enfoque *bottom-up* para construir una tabla de decisión que permite evaluar si cada prefijo de la cadena puede segmentarse en palabras válidas pertenecientes a un diccionario predefinido.

Notación utilizada:

- s: cadena de entrada a analizar, sin espacios.
- n: longitud de la cadena s, es decir, $n = |s|$.
- D: conjunto de palabras válidas (el diccionario), donde cada palabra es una cadena de caracteres.
- $c[i]$: variable booleana que indica si la subcadena $s[0 : i]$ (es decir, los primeros i caracteres) puede segmentarse completamente en palabras que pertenecen a D. Corresponde directamente con la función $c[n]$ definida en la ecuación de recurrencia.
- `indices`$[i]$: posición desde la cual se generó una palabra válida que termina en el índice i, utilizada para la reconstrucción del mensaje.
- `longitud_maxima`: longitud de la palabra más larga en el diccionario. Se utiliza para limitar la cantidad de cortes posibles y optimizar el rendimiento.

Se define un arreglo booleano c de tamaño $n + 1$, donde cada posición $c[i]$ representa si la subcadena $s[0 : i]$ puede segmentarse completamente en palabras del diccionario. El caso base es $c[0] =$ `True`, ya que la cadena vacía se considera segmentable por definición.

Para cada posición i, el algoritmo analiza si existe algún índice j en el rango $[\text{máx}(0, i - L), i)$ (donde L es la longitud máxima de palabra en el diccionario) tal que $c[j] =$ `True` y la subcadena $s[j : i] \in D$. Si se cumple esta condición, entonces se marca $c[i] =$ `True` y se registra en `indices[i]` el punto de corte válido correspondiente.

Cabe destacar que este algoritmo no realiza directamente la reconstrucción del mensaje segmentado. En su lugar, construye dos estructuras auxiliares: un arreglo booleano que indica la segmentabilidad de cada prefijo de la cadena y un arreglo de índices que permite, si fuera necesario, reconstruir la secuencia original de palabras. Esta separación entre verificación y reconstrucción favorece la claridad y modularidad del diseño.

```python
def algoritmo(s, D):
    n = len(s)
    c = [False] * (n + 1)
    indices = [-1] * (n + 1)

    longitud_maxima = 0
    for palabra in D:
        if len(palabra) > longitud_maxima:
            longitud_maxima = len(palabra)

    c[0] = True

    for i in range(1, n + 1):
        for j in range(max(0, i - longitud_maxima), i):
            if c[j]:
                palabra = s[j:i]
                if palabra in D:
                    c[i] = True
                    indices[i] = j
                    break

    return c, indices
```

4.1.1. Complejidad del algoritmo de programación dinámica

En esta subsección se procederá a analizar el costo computacional del algoritmo propuesto para determinar la segmentabilidad de una cadena. Se consideran tanto la complejidad temporal como la espacial, teniendo en cuenta las operaciones principales involucradas en la construcción de la tabla de decisiones.

Complejidad temporal:

El algoritmo recorre la cadena de entrada s de longitud n, evaluando en cada posición i (de 1 a n) si es posible segmentar el prefijo $s[0:i]$ mediante palabras del diccionario D.

Para cada i, se itera sobre posibles cortes desde $j = \max(0, i - L)$ hasta $j = i - 1$, donde L representa la longitud máxima de palabra en el diccionario. Esto implica como máximo L iteraciones por cada posición i.

En cada iteración del bucle interno, se realiza un corte de la subcadena $s[j:i]$. En `Python`, esta operación de *slicing* no es constante: implica la creación de una nueva cadena, cuyo costo es proporcional a su tamaño, es decir:

$$\text{Costo de } s[j:i] = O(i - j) \leq O(L)$$

Dado que esto ocurre hasta L veces por cada i, el costo del bucle interno en el peor caso es $O(L^2)$. Como este proceso se repite para cada una de las n posiciones, la complejidad temporal total del algoritmo resulta:

$$\boxed{O(n \cdot L^2)}$$

Cabe mencionar que si se utilizara un lenguaje donde el *slicing* es constante (como en implementaciones que permiten vistas sobre arreglos, sin copias), la complejidad podría reducirse a $O(n \cdot L)$.

Complejidad espacial:

Se utilizan dos arreglos auxiliares de tamaño $n + 1$:

- Un arreglo booleano $c[0 \ldots n]$ para almacenar si cada prefijo puede segmentarse correctamente.

- Un arreglo `indices`$[0 \dots n]$ para registrar desde qué posición se generó la última palabra válida en cada caso.

Ambas estructuras requieren espacio lineal en función de n, y no se emplean estructuras adicionales proporcionales a n^2 ni copias innecesarias.

$$\boxed{O(n)}$$

Conclusión: La complejidad del algoritmo de programación dinámica propuesto es:

Tiempo: $O(n \cdot L^2)$ **Espacio:** $O(n)$

4.2. Algoritmo de reconstrucción propuesto

Notación utilizada:

- s: cadena original que se intentó segmentar.
- `indices`: arreglo auxiliar generado por el algoritmo de programación dinámica, donde `indices`[i] indica desde qué posición se cortó la palabra que termina en i.
- i: posición final desde la que se inicia la reconstrucción. En general, coincide con $n = \text{len}(s)$, es decir, el final de la cadena.

En caso de que la segmentación de la cadena sea posible, se implementa un algoritmo adicional que permite reconstruir la secuencia original de palabras a partir del arreglo de índices generado en la etapa de decisión. Esta reconstrucción es necesaria únicamente si se desea visualizar el mensaje separado en palabras legibles.

El procedimiento comienza desde la última posición $i = n$, que representa el final de la cadena, y retrocede utilizando los valores almacenados en el arreglo `indices`. Cada entrada `indices[i]` indica la posición desde la cual se generó una palabra válida que termina en i. En cada iteración se toma el segmento comprendido entre el punto de corte y el final actual, y se lo almacena en una lista. Al finalizar el recorrido hacia atrás, se invierte el orden de las palabras para obtener la secuencia original.

Este algoritmo tiene complejidad lineal $O(n)$, ya que recorre la cadena hacia atrás sin repetir pasos, y permite reconstruir cualquier segmentación válida identificada por el algoritmo de decisión.

```python
def reconstruccion(s, indices, i):
    palabras = []

    if indices[i] == -1:
        return []

    while i > 0:
        j = indices[i]
        palabras.append(s[j:i])
        i = j

    palabras.reverse()
    return palabras
```

4.2.1. Complejidad del algoritmo de reconstrucción

En esta sección se analiza el costo computacional del algoritmo de reconstrucción, cuya finalidad es obtener la secuencia de palabras válidas a partir del arreglo de índices generado previamente

por la programación dinámica. El análisis considera el caso en que existe una segmentación válida (es decir, `indices`[n] $\neq -1$).

Complejidad temporal:

El algoritmo recorre la cadena de forma regresiva desde la posición final $i = n$, utilizando el arreglo `indices` para saltar hacia atrás en cada palabra segmentada. En cada iteración se realiza:

- Una consulta a `indices`[i], de costo $O(1)$.
- Un corte de subcadena $s[j : i]$, que implica copiar $i - j$ caracteres.

Como cada carácter de la cadena original aparece en una única subcadena reconstruida, el costo acumulado de todos los *slicing* es $O(n)$. Además, el número de iteraciones está acotado por la cantidad de palabras en la segmentación, que en el peor caso puede ser n (una palabra por carácter).

Por último, la operación `palabras.reverse()` también se realiza en tiempo lineal. Por lo tanto, la complejidad temporal total del algoritmo es:

$$\boxed{O(n)}$$

Complejidad espacial:

El algoritmo utiliza una lista auxiliar para almacenar las palabras reconstruidas. La longitud total de las cadenas almacenadas es igual a la longitud de la cadena original s, ya que cada carácter pertenece a una única palabra.

$$\boxed{O(n)}$$

Conclusión: El algoritmo de reconstrucción es eficiente, y posee complejidad lineal en tiempo y espacio:

Tiempo: $O(n)$ **Espacio:** $O(n)$

5. Análisis de variabilidad de valores

En esta sección se analiza cómo distintos factores vinculados a la entrada afectan al comportamiento práctico del algoritmo de programación dinámica propuesto. Si bien su complejidad teórica está acotada por $O(n \cdot L^2)$, donde n es la longitud del mensaje a analizar y L la longitud máxima de palabra en el diccionario, el tiempo real de ejecución puede variar significativamente según la naturaleza del input.

1. Longitud del mensaje (n)

La longitud del mensaje es el factor dominante del algoritmo. Dado que se recorren los n caracteres de izquierda a derecha, el tiempo de ejecución escala linealmente con respecto a n. Además, en cada posición i se intenta realizar cortes hacia atrás, hasta una distancia máxima de L, lo que implica $O(L^2)$ operaciones por posición debido al slicing.

Mensajes largos y segmentables provocarán muchas actualizaciones en la tabla de decisión, pero posiblemente terminen antes gracias al `break`. En cambio, mensajes largos no segmentables activarán el peor caso: se exploran todas las divisiones posibles en cada i sin encontrar ninguna válida.

2. Longitud máxima de palabra (L)

El parámetro L determina la cantidad de iteraciones posibles del bucle interno para cada i, y también impacta el costo del slicing $s[j : i]$, que en Python es lineal en $i - j$.

- Si el diccionario contiene palabras de gran longitud (por ejemplo, técnicas o compuestas), el número de cortes considerados será menor, pero el costo de crear cada subcadena será más alto.
- Si, por el contrario, predominan palabras cortas, se realizan más cortes por posición i, pero con slicing menos costoso.

En términos prácticos, un L muy alto puede degradar el rendimiento empírico significativamente, a pesar de que la complejidad teórica lo considera fijo.

3. Cantidad de palabras válidas en el diccionario ($|D|$)

La cardinalidad del diccionario no afecta la complejidad asintótica siempre que se utilice una estructura eficiente (como un `set` o `dict` en Python) con acceso en $O(1)$. Sin embargo, tiene un efecto indirecto sobre el comportamiento práctico:

- Un diccionario más extenso incrementa la probabilidad de que una subcadena $s[j : i]$ sea válida, haciendo que el algoritmo finalice más rápidamente en muchos casos (gracias al `break`).
- Un diccionario pobre en cobertura semántica puede aumentar los tiempos, ya que el algoritmo deberá explorar más posibilidades sin éxito.

4. Distribución de palabras en el diccionario

No solo importa la cantidad de palabras, sino cómo están distribuidas en términos de longitud:

- Diccionarios con predominancia de palabras cortas (`'a'`, `'de'`, `'lo'`, etc.) generan más combinaciones posibles y pueden ralentizar el análisis si el mensaje no es segmentable.

- Diccionarios con palabras muy específicas (largas, técnicas) tienden a reducir el espacio de búsqueda, pero pueden fallar en segmentar textos realistas.

Además, si hay muchas palabras solapadas (por ejemplo, `'mar'`, `'marco'`, `'marcar'`), el algoritmo podría encontrar múltiples caminos válidos, aunque en esta implementación solo se conserva uno.

5. Estructura léxica del mensaje

El mensaje a analizar puede tener distintas características que afecten el rendimiento:

- **Segmentable con facilidad:** cuando las palabras aparecen en el mensaje de forma clara y sin ambigüedad, el algoritmo encuentra rápidamente los cortes y utiliza pocas iteraciones.
- **No segmentable:** el peor caso ocurre cuando ningún corte válido se encuentra y se debe evaluar todo el rango de cortes posibles en cada i.
- **Ambigüedad estructural:** cuando hay múltiples cortes posibles (por ejemplo, en `'estamosquea'`), el algoritmo se detendrá en el primero que encuentre, sin explorar alternativas. Si ese camino no conduce a una solución global, no se descubrirá.

Este tipo de comportamiento puede impactar de manera desigual según el mensaje, a pesar de que todos tengan la misma longitud n.

6. Eficiencia del acceso al diccionario

El tiempo de verificación de pertenencia $s[j : i] \in D$ se asume constante $O(1)$, siempre que D esté implementado como un conjunto hash. Si se utilizara una lista o estructura secuencial, el acceso pasaría a ser $O(|D|)$, lo cual alteraría por completo la complejidad y haría que el algoritmo ya no sea eficiente para diccionarios grandes.

Conclusión

Si bien el algoritmo presenta una complejidad teórica bien definida, su comportamiento empírico puede variar significativamente según las siguientes variables:

- La longitud total del mensaje n, que marca el ritmo principal de ejecución.
- La longitud máxima de palabra L, que impacta en la profundidad y costo de cada iteración.
- La implementación y tamaño del diccionario D, que define la eficiencia de validación de palabras.
- La estructura interna del mensaje y su grado de segmentabilidad.

6. Ejemplos de ejecución

En esta sección se presentan ejemplos concretos que ilustran el funcionamiento del algoritmo propuesto. Se incluyen casos de segmentación exitosa, mensajes imposibles de segmentar, y escenarios donde existen palabras solapadas en el diccionario. Para cada caso se detallan la cadena original, el diccionario, la longitud de entrada y el resultado final.

Ejemplo 1: Segmentación exitosa y directa

Entrada:

- Cadena: `'ellamarco'` (longitud $n = 9$)
- Diccionario: {`'el'`, `'ella'`, `'mar'`, `'marco'`, `'arco'`}
- Longitud máxima de palabra: $L = 5$

Segmentación encontrada:

`['ella', 'marco']`

Interpretación: El algoritmo detecta correctamente una partición completa del mensaje en palabras válidas, eligiendo ``‘ella’'' seguida de `"marco"`, sin dejar caracteres sin cubrir.

Ejemplo 2: Robustez ante cadenas no segmentables

Entrada:

- Cadena: `'estamikheestado'` (longitud $n = 16$)
- Diccionario: {`'esta'`, `'mi'`, `'he'`, `'estado'`}
- Longitud máxima de palabra: $L = 7$

Segmentación encontrada:

`[]` (no segmentable)

Interpretación: A pesar de que contiene palabras del diccionario, el mensaje no puede cubrirse completamente sin dejar fragmentos sin sentido. El algoritmo retorna correctamente una reconstrucción vacía.

Ejemplo 3: Manejo de palabras solapadas

Entrada:

- Cadena: `'ellamar'` (longitud $n = 7$)
- Diccionario: {`'el'`, `'ella'`, `'mar'`}
- Longitud máxima de palabra: $L = 4$

Segmentación encontrada:

`['ella', 'mar']`

Interpretación: El algoritmo resuelve correctamente la ambigüedad: aunque `'el'` es un prefijo válido, al seleccionar `'ella'` logra una segmentación completa, lo cual valida su eficiencia en escenarios con cortes alternativos.

Ejemplo 4: Segmentación de mensaje extenso

Entrada:

- Cadena: `'estamosenelmuellealasdiez'` (longitud $n = 27$)
- Diccionario: {`'estamos'`, `'en'`, `'el'`, `'muelle'`, `'a'`, `'las'`, `'diez'`}
- Longitud máxima de palabra: $L = 7$

Segmentación encontrada:

`['estamos', 'en', 'el', 'muelle', 'a', 'las', 'diez']`

Interpretación: El algoritmo demuestra su escalabilidad para mensajes largos, construyendo una segmentación completa y precisa que respeta el orden y las palabras del diccionario.

Observaciones finales

- En todos los casos segmentables, se obtiene una reconstrucción ordenada y exhaustiva de la cadena original.
- En entradas no segmentables, el algoritmo responde con exactitud, sin forzar cortes artificiales.
- La combinación de análisis iterativo y reconstrucción permite interpretar con claridad el recorrido del algoritmo y verificar su resultado paso a paso.

7. Mediciones

En esta sección se realizarán mediciones empíricas con el fin de contrastar la complejidad teórica del algoritmo implementado con su comportamiento real. Según lo analizado, la complejidad del algoritmo depende principalmente de dos factores:

- n: longitud de la cadena de entrada.
- L: longitud de la palabra más larga del diccionario.

La complejidad teórica del algoritmo es $\mathcal{O}(n \cdot L^2)$. Por lo tanto, se espera que, al mantener L constante, el crecimiento sea lineal respecto a n, y que al mantener n constante, el crecimiento sea cuadrático respecto a L.

Para validar esta hipótesis, se realizaron dos tipos de experimentos:

- Medición de tiempo de ejecución en función de n, con $L = 7$ fijo.
- Medición en función de L, con $n = 10000$ fijo.

Los datos fueron generados sintéticamente: se construyeron diccionarios artificiales con palabras de longitud fija, y las cadenas de entrada se crearon concatenando palabras válidas del diccionario. Las mediciones se realizaron promediando 10 ejecuciones por tamaño para reducir la variabilidad.

7.1. Análisis con n variable ($L = 7$ fijo)

En esta primera instancia se midió el tiempo de ejecución para cadenas de longitudes crecientes, manteniendo constante la longitud de las palabras del diccionario en $L = 7$. Los tamaños de entrada utilizados fueron: **5000, 9000, 13000, 17000, 21000, 25000**.

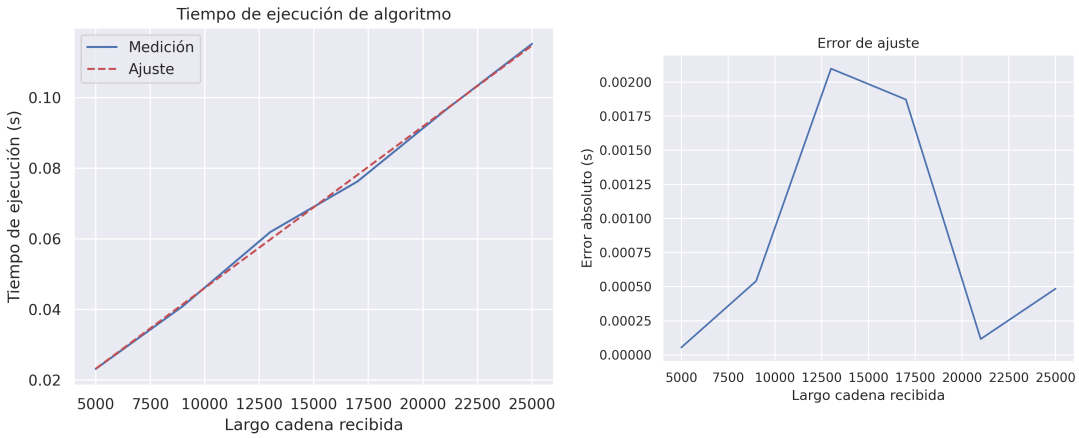

Figura 1: Ajuste lineal para n variable. Error cuadrático total = **0.0246**

Como puede observarse, el gráfico presenta una relación lineal clara entre el tamaño de la cadena y el tiempo de ejecución. El ajuste por mínimos cuadrados se aproxima con una función de la forma $f(n) = c_1 \cdot n + c_2$, lo cual confirma empíricamente la dependencia lineal respecto de n cuando L es constante.

7.2. Análisis con L variable ($n = 10000$ fijo)

En esta segunda medición se mantuvo constante el tamaño de la cadena de entrada ($n = 10000$), y se variaron las longitudes máximas de las palabras del diccionario L entre: **5, 10, 15, 20, 25, 30**.

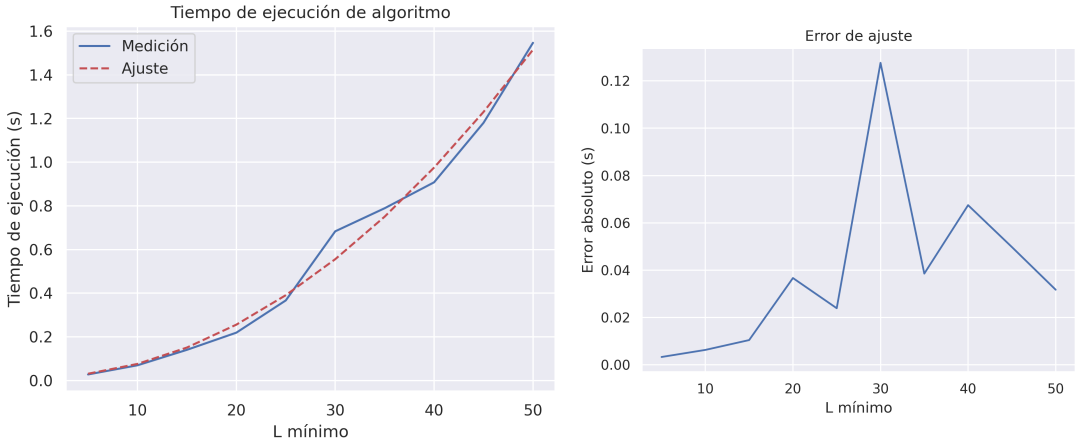

Figura 2: Ajuste cuadrático para L variable. Error cuadrático total = **0.02789681559222275**

En este caso, el comportamiento del tiempo de ejecución crece de forma cuadrática con la longitud máxima de las palabras. El ajuste con una parábola de la forma $f(L) = c_1 \cdot L^2 + c_2$ se adapta correctamente a los datos medidos, validando la hipótesis de que la complejidad es cuadrática en L para n fijo.

Conclusión

Los resultados obtenidos son coherentes con la complejidad teórica $\mathcal{O}(n \cdot L^2)$. Se observaron:

- Comportamiento lineal respecto de n, cuando L es constante.
- Comportamiento cuadrático respecto de L, cuando n es constante.

Las pequeñas desviaciones observadas en las mediciones pueden atribuirse a efectos del entorno de ejecución (recolección de basura, slicing de strings, caching), pero no contradicen la tendencia general.

En conclusión, las mediciones experimentales validan el análisis teórico de complejidad del algoritmo.

8. Conclusiones

A lo largo del presente informe se desarrolló una solución eficiente y bien fundamentada al problema de segmentación de cadenas sin espacios en palabras válidas, empleando herramientas de programación dinámica. El trabajo permitió abordar de manera estructurada un desafío que, si bien tiene un objetivo binario (es posible segmentar o no), requiere un análisis detallado del espacio de soluciones posibles y una gestión eficiente de recursos computacionales.

El algoritmo propuesto fue diseñado bajo un enfoque *bottom-up*, lo que facilitó un control más explícito del proceso de decisión y permitió estructurar la solución de forma iterativa. Este enfoque no solo favoreció la eficiencia, sino que también permitió introducir una lógica clara y verificable para la reconstrucción posterior del mensaje original.

Entre los principales puntos del trabajo, se destacan:

- La formulación precisa de una ecuación de recurrencia, que define con claridad las condiciones bajo las cuales una cadena puede considerarse segmentable.
- La implementación de un algoritmo con complejidad temporal $O(n \cdot L^2)$, adecuado para cadenas largas y diccionarios acotados, siendo n la longitud de la cadena y L la longitud máxima de palabra.
- La correcta separación de responsabilidades entre el algoritmo de decisión (verificación de segmentabilidad) y el algoritmo de reconstrucción (recuperación de palabras originales), promoviendo modularidad y claridad en el código.
- La validación del algoritmo con múltiples ejemplos de ejecución, demostrando que la solución no solo es teóricamente correcta, sino también práctica y aplicable a casos reales.
- Un análisis detallado de la complejidad computacional, que incluyó tanto la teoría como su comportamiento empírico frente a distintos tipos de entradas.

Además, se abordó con claridad cómo influyen en el rendimiento variables como la longitud del mensaje, el tamaño del diccionario, y la distribución de palabras dentro de éste. Estas consideraciones permitieron entender no sólo cómo funciona el algoritmo, sino también por qué funciona bien bajo ciertas condiciones y qué características del problema impactan directamente en su desempeño.

El conjunto de decisiones tomadas durante el desarrollo —desde la elección del enfoque iterativo hasta la implementación del corte anticipado con `break` en la búsqueda de particiones— se tradujo en una solución precisa, eficiente y robusta.