TEORÍA DE ALGORITMOS

(75.29) CURSO BUCHWALD - GENENDER

Trabajo Práctico 2
Que parezca programación dinámica

5 de mayo de 2025

Julen Leonel Gaumard
111379

Noelia Salvatierra
100116

Franco Macke
105974

1. Análisis del problema

Tenemos:

- Una cadena encriptada S sin separaciones
- Un diccionario de palabras válidas en español

Debemos determinar:

- Si es posible separar S en una secuencia de palabras de D. En caso afirmativo, cuál sería esa separación.
- Si podemos determinar que los primeros i caracteres forman una secuencia válida de palabras, entonces solo necesitamos verificar si el resto de la cadena también puede separarse en palabras válidas.
- Para cada posición en la cadena, podemos probar todas las posibles palabras que podrían comenzar desde allí y verificar si el resto de la cadena puede separarse recursivamente.

Ecuación de recurrencia

Sea `oracion[0..n-1]` la cadena a segmentar. Definimos `existencia_parcial` tal que `existencia_parcial[i]` es verdadero si la subcadena de longitud i de la cadena (es decir, `oracion[0..i-1]`) puede segmentarse completamente en palabras del diccionario.

Entonces, definimos la ecuación de recurrencia tal que:

$$\text{existencia\_parcial}[i] = \bigvee_{j=0}^{i-1} (\text{existencia\_parcial}[j] \wedge \text{oracion}[j:i] \in \text{diccionario})$$

Esto significa que `existencia_parcial[i]` va a ser verdadero si existe al menos un índice $j < i$ tal que:

- el prefijo `oracion[0..j-1]` puede segmentarse (es decir, `existencia_parcial[j] = True`)
- la subcadena `oracion[j..i-1]` pertenece al diccionario

El caso base es:

$$\texttt{existencia\_parcial}[0] = \texttt{True}$$

ya que una cadena vacía la consideramos una palabra válida.

2. Demostración

La ecuación de recurrencia obtenida es:

$$\text{existencia\_parcial}[i] = \bigvee_{j=0}^{i-1} (\text{existencia\_parcial}[j] \wedge \text{oracion}[j:i] \in \text{diccionario})$$

Donde:

existencia_parcial$[i]$ es una tabla booleana que va a ser una $P(n)$ proposición, con $i \in [0, n]$ tal que:

$$\text{existencia\_parcial}[i] = \text{True} \iff \text{la subcadena oracion}[0:i] \text{ puede segmentarse completamente en palabras del diccio}$$

Demostraremos que esta ecuación llega a la solución óptima, por el principio de inducción matemático y usando el método directo:

Paso base: $i = 0$

La subcadena vacía oracion$[0:0]$ se puede segmentar (no contiene caracteres), por lo tanto:

$$\text{existencia\_parcial}[0] = \text{True}$$

Hipótesis verdadera: Supongamos que para todo $k < i$, existencia_parcial$[k] = \text{True}$ si y solo si el prefijo oracion$[0:k]$ PUEDE SEGMENTARSE completamente en palabras del diccionario.

Paso inductivo: Demostraremos que bajo esta hipótesis, existencia_parcial$[i]$ también se establece correctamente.

Según la ecuación de recurrencia:

$$\text{existencia\_parcial}[i] = \bigvee_{j=0}^{i-1} (\text{existencia\_parcial}[j] \wedge \text{oracion}[j:i] \in D)$$

Esto significa que:

- Existe un índice $j < i$ tal que existencia_parcial$[j] = \text{True}$ (es decir, oracion$[0:j]$ puede segmentarse), y
- La subcadena oracion$[j:i]$ pertenece al diccionario.

Entonces, se puede construir la segmentación de oracion$[0:i]$ como la concatenación de dos partes válidas:

$$\text{oracion}[0:j] + \text{oracion}[j:i]$$

y por lo tanto, existencia_parcial$[i] = \text{True}$.

Por otro lado, si no existe ningún $j < i$ que cumpla esa condición, entonces no existe una forma de segmentar oracion$[0:i]$, y correctamente:

$$\text{existencia\_parcial}[i] = \text{False}$$

Conclusión: Por el principio de inducción matemática, la proposición $P(n)$ (tabla existencia_parcial está correctamente construida para todo $i \in [0, n]$).

3. Complejidad teorica

Primero, vamos a analizar el código analíticamente:

```python
def procesar_texto(oraciones, diccionario):
    """
    Procesa una lista de oraciones y un diccionario, y devuelve una lista de oraciones segmentadas.
    :param oraciones: Lista de oraciones a procesar.
    :param diccionario: Diccionario utilizado para la segmentaci n.
    """
    resultado = []

    for oracion in oraciones:
        oracion_segmentada = segmentar_oracion(oracion, diccionario)

        if len("".join(oracion_segmentada)) < len(oracion):
            resultado.append("No es un mensaje")
        else:
            resultado.append(" ".join(oracion_segmentada))

    return resultado


def segmentar_oracion(oracion, diccionario):
    n = len(oracion)
    conjunto_diccionario = set(diccionario)
    max_long_palabra = max(len(palabra) for palabra in diccionario)

    existencia_parcial = [False] * (n + 1)
    existencia_parcial[0] = True
    path = [None] * (n + 1)

    for i in range(1, n + 1):
        for j in range(i):
            if existencia_parcial[j] and oracion[j:i] in conjunto_diccionario:
                existencia_parcial[i] = True
                path[i] = (j, oracion[j:i])
                break

    if not existencia_parcial[n]:
        return []

    return reconstruir_segmentacion(path, n)


def reconstruir_segmentacion(path, n):
    """
    Reconstruye la segmentaci n de una oraci n a partir del camino dado.
    :param path: Lista de tuplas que representan el camino de segmentaci n.
    :param n: Longitud de la oraci n original.
    :return: Lista de palabras segmentadas.
    """
    resultado = []
    idx = n
    while idx > 0:
        j, palabra = path[idx]
        resultado.append(palabra)
        idx = j

    return resultado[::-1]
```

Listing 1: Algoritmo principal

Analizando línea por línea:

- `resultado = []`: asignación Complejidad: $\mathcal{O}(1)$.
- Bucle principal for:

- Se ejecuta m veces (una por cada oracion), siendo m el tamaño de oraciones.
- En cada iteración:
  - `segmentar_oracion(...)`: (se justifica con detalle más adelante). $\mathcal{O}(k + (n^3))$ con k, el tamaño del diccionario.
  - `condicional if`: $\mathcal{O}(1)$.(los condicionales se ejecutan a tiempo constante)
  - `append(...)`: tiempo constante. $\mathcal{O}(1)$.
  - `condicional else`: $\mathcal{O}(1)$.
  - `append(...)`: $\mathcal{O}(1)$.
- Total por iteración: $\mathcal{O}(k + (n^3))$.

- Complejidad total del bucle: $\mathcal{O}(m * (k + (n^3))$.
- return resultado: $\mathcal{O}(1)$.

Complejidad total del algoritmo:

$$\mathcal{O}(1) + \mathcal{O}(m * (k + (n^3))) + \mathcal{O}(1) = \boxed{\mathcal{O}(m * (k + (n^3)))}$$

Entonces, analíticamente podemos decir que el algoritmo es $O(m * (k + (n^3)))$ con m = len(oraciones), n = len(oracion) y k = len(diccionario).

Observamos que si el diccionario de palabras supera a n, la complejidad se puede aproximar a: $O(mxk)$.

Si en cambio, m es mucho mas grande que el diccionario de palabras, la complejidad se puede aproximar a: $O(m)$, donde m es la cantidad de oraciones a segmentar.

Segmentar_oracion(...):

- `len(oracion)`: Complejidad: $\mathcal{O}(n)$.
- `set(diccionario)`: Complejidad: $\mathcal{O}(k)$ con k el tamaño del diccionario.
- `existencia_parcial = [False] * (n + 1)`: Complejidad: $\mathcal{O}(n)$.
- `existencia_parcial[0] = True`: Complejidad: $\mathcal{O}(1)$.
- `path = [None] * (n + 1)`: Complejidad: $\mathcal{O}(n)$.
- Bucle principal for:
  - Se ejecuta n veces (una por i caracter), siendo n el tamaño de oración.
  - En cada iteración:
    - `bucle secundario for`:
      - Se ejecuta i veces (una por j caracter) con L ¡n.
      - En cada iteración:
      - `condicional if con slice oracion[j:i]`: Los slices no son de tiempo constante.Complejidad $\mathcal{O}(i)$.
      - `resto de asignaciones`: Complejidad $\mathcal{O}(1)$.
      - `break`: Complejidad $\mathcal{O}(1)$.
      - Total por iteración: $\mathcal{O}(i)$.
    - Complejidad total del bucle secundario: $\mathcal{O}(i^2)$.
  - Total por iteración: $\mathcal{O}(i^2)$.
- Complejidad total del bucle principal: $\mathcal{O}((i^2)xn)$. En el peor caso, i = n, entonces podemos aproximar la complejidad a $\mathcal{O}((n^3)$

- `return reconstruir_segmentacion(...)`: función que reconstruye el mensaje çon sentido", dada la solucion por programacion dinamica y tiene asignaciones, llamadas a append(...), que son de tiempo constante, y un bucle while que en el peor caso, se recorre hasta que idx menor o igual a 0, con idx = n, entonces la complejidad es: $\mathcal{O}(n)$.

Complejidad total segmentar oracion(...):

$$\mathcal{O}(n) + \mathcal{O}(k) + \mathcal{O}(k + (n^3)) + \mathcal{O}(n) = \boxed{\mathcal{O}(k + (n^3))}$$

.

4. Ejemplos de ejecución

Oraciones generadas: `zunaoirá`

Resultados del procesamiento:

`Evaluando subcadena: z`
Es válido hasta la posición 0?: True
Existe en el diccionario?: False

`Evaluando subcadena: zu`
Es válido hasta la posición 0?: True
Existe en el diccionario?: False

`Evaluando subcadena: u`
Es válido hasta la posición 1?: False
Existe en el diccionario?: False

`Evaluando subcadena: zun`
Es válido hasta la posición 0?: True
Existe en el diccionario?: False

`Evaluando subcadena: un`
Es válido hasta la posición 1?: False
Existe en el diccionario?: False

`Evaluando subcadena: n`
Es válido hasta la posición 2?: False
Existe en el diccionario?: False

`Evaluando subcadena: zuna`
Es válido hasta la posición 0?: True
Existe en el diccionario?: True

`Evaluando subcadena: zunao`
Es válido hasta la posición 0?: True
Existe en el diccionario?: False

`Evaluando subcadena: unao`
Es válido hasta la posición 1?: False
Existe en el diccionario?: False

`Evaluando subcadena: nao`
Es válido hasta la posición 2?: False
Existe en el diccionario?: False

`Evaluando subcadena: ao`
Es válido hasta la posición 3?: False

Existe en el diccionario?: False

`Evaluando subcadena: o`
Es válido hasta la posición 4?: True
Existe en el diccionario?: False

`Evaluando subcadena: zunaoi`
Es válido hasta la posición 0?: True
Existe en el diccionario?: False

`Evaluando subcadena: unaoi`
Es válido hasta la posición 1?: False
Existe en el diccionario?: False

`Evaluando subcadena: naoi`
Es válido hasta la posición 2?: False
Existe en el diccionario?: False

`Evaluando subcadena: aoi`
Es válido hasta la posición 3?: False
Existe en el diccionario?: False

`Evaluando subcadena: oi`
Es válido hasta la posición 4?: True
Existe en el diccionario?: False

`Evaluando subcadena: i`
Es válido hasta la posición 5?: False
Existe en el diccionario?: False

`Evaluando subcadena: zunaoir`
Es válido hasta la posición 0?: True
Existe en el diccionario?: False

`Evaluando subcadena: unaoir`
Es válido hasta la posición 1?: False
Existe en el diccionario?: False

`Evaluando subcadena: naoir`
Es válido hasta la posición 2?: False
Existe en el diccionario?: False

`Evaluando subcadena: aoir`
Es válido hasta la posición 3?: False
Existe en el diccionario?: False

`Evaluando subcadena: oir`
Es válido hasta la posición 4?: True
Existe en el diccionario?: False

`Evaluando subcadena: ir`
Es válido hasta la posición 5?: False
Existe en el diccionario?: False

`Evaluando subcadena: r`
Es válido hasta la posición 6?: False
Existe en el diccionario?: False

`Evaluando subcadena: zunaoirá`
Es válido hasta la posición 0?: True
Existe en el diccionario?: False

`Evaluando subcadena: unaoirá`
Es válido hasta la posición 1?: False
Existe en el diccionario?: False

`Evaluando subcadena: naoirá`
Es válido hasta la posición 2?: False
Existe en el diccionario?: False

`Evaluando subcadena: aoirá`
Es válido hasta la posición 3?: False
Existe en el diccionario?: False

`Evaluando subcadena: oirá`
Es válido hasta la posición 4?: True
Existe en el diccionario?: True

Oración: `zunaoirá` **Resultado:** `zuna oirá`

5. Análisis de la complejidad por cuadrados minimos

Vamos a corroborar la complejidad calculada anteriormente de manera empírica. Para ello, seteamos la cantidad de palabras y el tamaño de las oraciones a descifrar relativamente grandes. Los casos generados se hacen utilizando palabras del diccionario generadas aleatoriamente de esta manera:

```python
def generate_random_dictionary(size, min_length=3, max_length=MAX_WORD_LENGTH):
    """Genera un diccionario aleatorio de palabras."""
    dictionary = []
    for _ in range(size):
        word_length = random.randint(min_length, max_length)
        word = ''.join(random.choice(string.ascii_lowercase) for _ in range(
    word_length))
        dictionary.append(word)
    return dictionary

def generate_valid_text(dictionary, length):
    """Genera un texto aleatorio usando palabras del diccionario."""
    return ''.join(random.choice(dictionary) for _ in range(length))
```

Listing 2: Generación de diccionario y texto válido aleatorios

Primero, queremos comprobar que efectivamente, la complejidad de `segmentar_oracion()` es $\mathcal{O}(n^3 + k)$.

Hacemos el análisis de cantidad de palabras del diccionario k constante.

Constantes de longitud de palabras

$$\texttt{MAX\_WORD\_LENGTH} = 400 \quad \text{(Longitud máxima de las palabras)}$$

$$\texttt{NUMBER\_OF\_WORDS\_IN\_ONE\_SENTENCE} = 100 \quad \text{(Cantidad de palabras por oracion)}$$

Cantidad de oraciones por iteración

$$\texttt{sizes} = [50,\ 100,\ 125,\ 150,\ 170,\ 200,\ 250,\ 300,\ 400]$$

$$\texttt{NUMBER\_OF\_SENTENCES} = 1 \quad \text{(Cantidad de oraciones por caso, m en nuestro análisis)}$$

Gráfico de complejidad temporal

A continuación se muestra el gráfico de complejidad temporal empírica usando las variables anteriormente mencionadas, acompañado de la curva que mejor ajusta la tendencia observada.

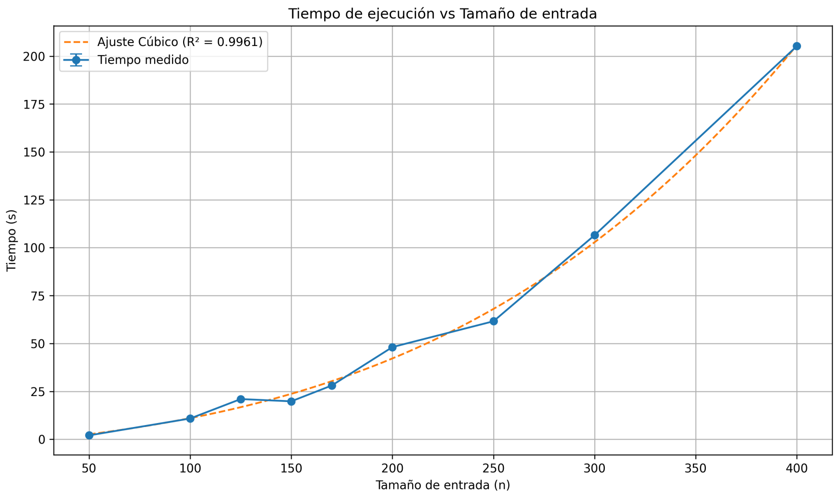

Figura 1: Ajuste de complejidad temporal con curva cúbica

Ajuste por mínimos cuadrados

La curva ajustada corresponde a la siguiente expresión:

$$y = 2{,}11 \times 10^{-6} \cdot x^3 + 2{,}12 \times 10^{-4} \cdot x^2 + 1{,}00 \cdot x - 3{,}23$$

Efectivamente, observamos que la complejidad de `segmentar_oracion()` resulta $\mathcal{O}(n^3)$.

Subiendo m (`NUMBER_OF_SENTENCES`) a un valor proporcional a n, aumentaría la complejidad a:

$$T(n, m) = m \cdot O(n^3)$$

$$\text{Si } m = O(n), \text{ entonces:}$$

$$T(n, m) = O(n) \cdot O(n^3) = O(n^4)$$

6. Variabilidad de los valores

Complejidad con diccionario

Ahora, mantenemos n constante y variamos la cantidad de palabras del diccionario.

```
DICTIONARY_SIZES = [1000, 2000, 5000, 7000, 10000, 15000, 20000]
```

Listing 3: Tamaños del diccionario utilizados

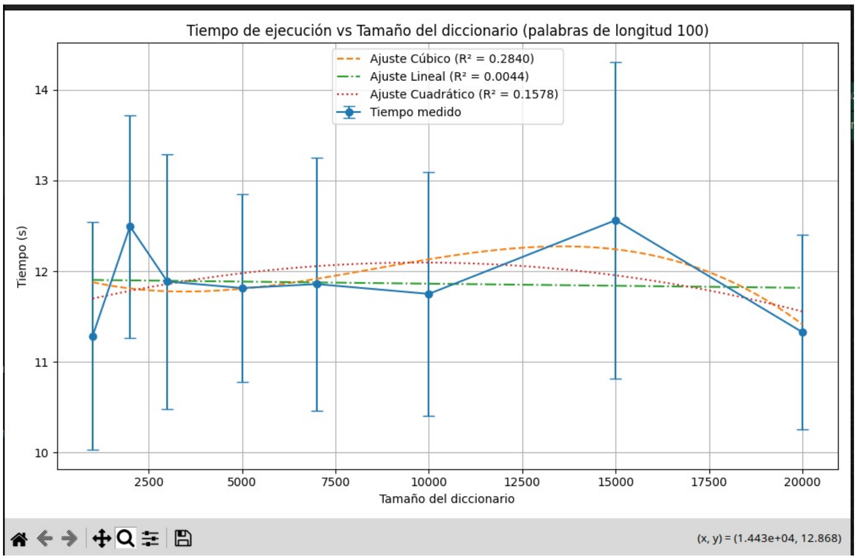

Figura 2: Mantengo n constante y aumento k

Podemos ver que tarda prácticamente lo mismo con un diccionario de lontitud 1.000 y uno de 20.000. Podemos ver que cuando n es grande, k termina siendo absorbido en la complejidad.

Variante n chico y k muy grande

Para mostrar la dependencia con la cantidad de palabras del diccionario, proponemos el caso donde n es chico y k muy grande. Esto resulta en el gráfico siguiente, siendo la mejor aproximación algo lineal.

```
HUGE_DICTIONARY_SIZES = [num for num in range(10_000, 1_000_000, 50_000)]
```

Listing 4: Generación de tamaños grandes de diccionario

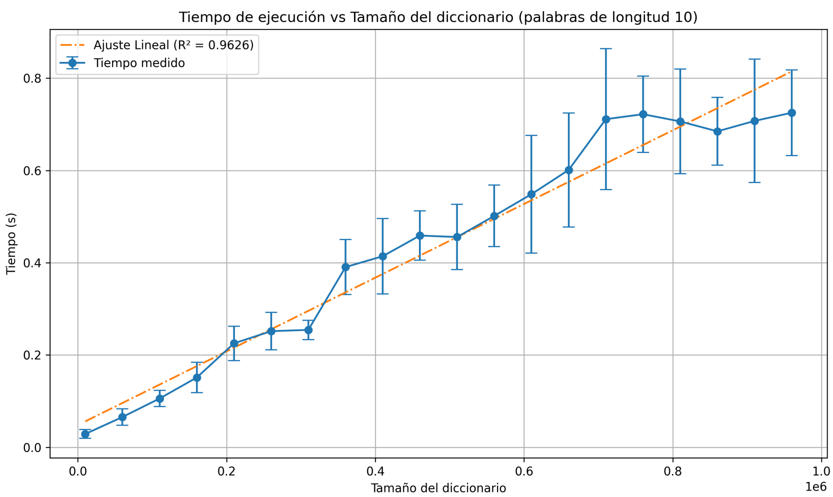

Figura 3: Mantengo n chica y constante aumentando k

Claramente, k no puede ser obviada, ya que si es mucho más grande que n, afecta a la complejidad temporal.

Variaciones de k

Cuando $k \propto n^3$:

$$O(k + n^3) = O(c \cdot n^3 + n^3) = O(n^3 + n^3) = O(2n^3) = O(n^3)$$

$$\therefore\ O(k + n^3) = O(n^3)$$

Cuando $k \ll n^3$:

$$O(k + n^3) = O(n^3)$$

El término k es despreciable frente a n^3

Cuando $k \gg n^3$:

$$O(k + n^3) = O(k)$$

El término n^3 es despreciable frente a k

7. Anexo

Demostración de si y solo si de `existencia_parcial` mediante inducción matemática

Vamos a demostrar, que para todo $i \in [0, n]$ se cumple:

`existencia_parcial[i]` = `True` si y sólo si `oracion[0:i]` puede segmentarse completamente en palabras del diccionario.

Paso base: $i = 0$

La subcadena `oracion[0:0]` corresponde a la cadena vacía. Por convención, la cadena vacía se considera segmentada correctamente. Además, el algoritmo establece explícitamente:

$$\texttt{existencia\_parcial[0]} = \texttt{True}$$

Por lo tanto, se cumple la propiedad en el paso base:

$$\texttt{existencia\_parcial[0]} = \texttt{True} \iff \texttt{oracion[0:0]} \text{ es segmentable.}$$

Hipótesis inductiva: VAMOS A SUPONER QUE EL ALGORITMO FUNCIONA DE MANERA CORRECTA PARA ESTOS CASOS!

Supongamos que para un cierto $i \geq 0$, y para todo $k \leq i$, se cumple:

$$\texttt{existencia\_parcial[k]} = \texttt{True} \iff \texttt{oracion[0:k]} \text{ puede segmentarse completamente.}$$

Paso inductivo:

Queremos probar que para $i + 1$:

$$\texttt{existencia\_parcial[i+1]} = \texttt{True} \iff \texttt{oracion[0:i+1]} \text{ puede segmentarse completamente.}$$

($\Rightarrow$) Supongamos que `existencia_parcial[i+1]` = `True`.
Por definición del algoritmo, esto implica que existe al menos un índice $j \in [\text{máx}(0, i+1-L), i]$, donde L es la longitud máxima de las palabras del diccionario, tal que:

- `existencia_parcial[j]` = `True`, y
- `oracion[j:i+1]` $\in$ `diccionario`

Por la hipótesis inductiva, `oracion[0:j]` puede segmentarse. Dado que `oracion[j:i+1]` es una palabra válida, entonces:

$$\texttt{oracion[0:i+1]} = \texttt{oracion[0:j]} + \texttt{oracion[j:i+1]}$$

también puede segmentarse completamente.

($\Leftarrow$) Supongamos que `oracion[0:i+1]` puede segmentarse completamente.
Entonces existe una secuencia de índices $0 = j_0 < j_1 < \ldots < j_k = i + 1$ tal que cada subcadena `oracion[`$j_{m-1} : j_m$`]` pertenece al diccionario.

Sea $j = j_{k-1}$. Entonces:

- `oracion[0:j]` puede segmentarse (por hipótesis inductiva $\Rightarrow$ `existencia_parcial[j]` = `True`)

- oracion[j:i+1] $\in$ diccionario

Por lo tanto, el algoritmo encontrará esta combinación válida y establecerá:

existencia_parcial[i+1] = True

Por el principio de inducción matemática, se concluye que para todo $i \in [0, n]$:

existencia_parcial[i] = True $\iff$ oracion[0:i] puede segmentarse completamente en palabras del diccionario.

7.1. Corrección Complejidad teorica por la palabras mas grande del diccionario

```
def procesar_texto(oraciones, diccionario):
    """
    Procesa una lista de oraciones y un diccionario, y devuelve una lista de
    oraciones segmentadas.
    :param oraciones: Lista de oraciones a procesar.
    :param diccionario: Diccionario utilizado para la segmentaci n.
    """
    resultado = []

    for oracion in oraciones:
        oracion_segmentada = segmentar_oracion(oracion, diccionario)

        if len("".join(oracion_segmentada)) < len(oracion):
            resultado.append("No es un mensaje")
        else:
            resultado.append(" ".join(oracion_segmentada))

    return resultado


def segmentar_oracion(oracion, diccionario):
    n = len(oracion)
    conjunto_diccionario = set(diccionario)
    max_long_palabra = max(len(palabra) for palabra in diccionario)

    existencia_parcial = [False] * (n + 1)
    existencia_parcial[0] = True
    path = [None] * (n + 1)

    for i in range(1, n + 1):
        for j in range(max(0, i - max_long_palabra), i):
            if existencia_parcial[j] and oracion[j:i] in conjunto_diccionario:
                existencia_parcial[i] = True
                path[i] = (j, oracion[j:i])
                break

    if not existencia_parcial[n]:
        return []

    return reconstruir_segmentacion(path, n)


def reconstruir_segmentacion(path, n):
    """
    Reconstruye la segmentaci n de una oraci n a partir del camino dado.
    :param path: Lista de tuplas que representan el camino de segmentaci n.
    :param n: Longitud de la oraci n original.
    :return: Lista de palabras segmentadas.
    """
    resultado = []
    idx = n
    while idx > 0:
```

```
        j, palabra = path[idx]
        resultado.append(palabra)
        idx = j

    return resultado[::-1]
```

Listing 5: Algoritmo principal

Analizando línea por línea:

- `resultado = []`: asignación Complejidad: $\mathcal{O}(1)$.
- Bucle principal for:
  - Se ejecuta m veces (una por cada oracion), siendo m el tamaño de oraciones.
  - En cada iteración:
    - `segmentar_oracion(...)`: (se justifica con detalle más adelante). $\mathcal{O}(k + (L^2) \cdot n)$ con k, el tamaño del diccionario y L el tamaño mas grande de una palabra del diccionario, siendo L MUCHO MENOR que n.
    - `condicional if`: $\mathcal{O}(1)$.(los condicionales se ejecutan a tiempo constante)
    - `append(...)`: tiempo constante. $\mathcal{O}(1)$.
    - `condicional else`: $\mathcal{O}(1)$.
    - `append(...)`: $\mathcal{O}(1)$.
  - Total por iteración: $\mathcal{O}(k + (L^2) \cdot n)$.
- Complejidad total del bucle: $\mathcal{O}(m \cdot (k + (L^2) \cdot n))$.
- return resultado: $\mathcal{O}(1)$.

Complejidad total del algoritmo:

$$\mathcal{O}(1) + \mathcal{O}(m \cdot (k + (L^2) \cdot n)) + \mathcal{O}(1) = \boxed{\mathcal{O}(m \cdot (k + (L^2) \cdot n))}$$

Entonces, analíticamente podemos decir que el algoritmo es $O(mx(k + (L^2)xn))$ con m = len(oraciones), n = len(oracion) y k = len(diccionario). Podemos notar que como L ES MUCHO MAS CHICO que n, el algoritmo es mucho màs ràpido, auque a simple vista no se vea en la complejidad.

Observamos que si el diccionario de palabras supera a n, la complejidad se puede aproximar a: $O(mxk)$.

Si en cambio, m es mucho mas grande que el diccionario de palabras, la complejidad se puede aproximar a: $O(m)$, donde m es la cantidad de oraciones a segmentar.

Segmentar_oracion(...):

- `len(oracion)`: Complejidad: $\mathcal{O}(n)$.
- `set(diccionario)`: Complejidad: $\mathcal{O}(k)$ con k el tamaño del diccionario.
- `max long palabra`: Complejidad: $\mathcal{O}(L)$ con L la palabra mas grande del diccionario.
- `existencia_parcial = [False] * (n + 1)`: Complejidad: $\mathcal{O}(n)$.
- `existencia_parcial[0] = True`: Complejidad: $\mathcal{O}(1)$.
- `path = [None] * (n + 1)`: Complejidad: $\mathcal{O}(n)$.

- Bucle principal for:
  - Se ejecuta n veces (una por i caracter), siendo n el tamaño de oración.
  - En cada iteración:
    - `bucle secundario for`:
      - Se ejecuta L veces (una por j caracter) con L menor que n la palabra mas grande del diccionario.
      - En cada iteración:
      - `condicional if con slice oracion[j:i]`: Los slices no son de tiempo constante.Complejidad $\mathcal{O}(L)$.
      - `resto de asignaciones`: Complejidad $\mathcal{O}(1)$.
      - `break`: Complejidad $\mathcal{O}(1)$.
      - Total por iteración: $\mathcal{O}(L)$.
    - Complejidad total del bucle secundario: $\mathcal{O}(L^2)$.
  - Total por iteración: $\mathcal{O}(L^2)$.
- Complejidad total del bucle principal: $\mathcal{O}((L^2) \cdot n)$.
- `return reconstruir_segmentacion(...)`: función que reconstruye el mensaje çon sentido", dada la solucion por programacion dinamica y tiene asignaciones, llamadas a append(...), que son de tiempo constante, y un bucle while que en el peor caso, se recorre hasta que idx menor o igual a 0, con idx = n, entonces la complejidad es: $\mathcal{O}(n)$.

Complejidad total segmentar oracion(...):

$$\mathcal{O}(n) + \mathcal{O}(k) + \mathcal{O}(L) + \mathcal{O}((L^2) \cdot n) + \mathcal{O}(n) = \boxed{\mathcal{O}(k + (L^2) \cdot n)}$$

.

Análisis de la complejidad por cuadrados mínimos

Caso L y k chico, m constante y n variable

En este caso, podemos ver que si $L \ll n$, la complejidad depende de n y m. Mantuvimos m constante y n variable. Si m también aumentara, sería algo cuadrático. Podemos considerar que es el caso más común, ya que L es a lo sumo tan grande como la palabra más larga del diccionario, lo cual comparado con la longitud de una oración, termina siendo despreciable.

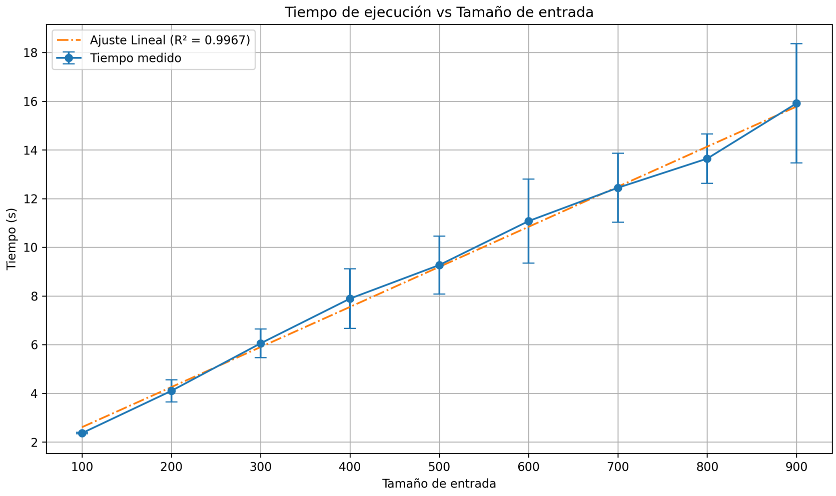

Figura 4: Mantengo m constante y aumento n

Podemos ver entonces que:

$$\mathcal{O}(m \cdot (k + (L^2) \cdot n)) = \boxed{\mathcal{O}(m \cdot n)}$$

Caso L y n chico, k grande y m constante

Para valores de $n \ll k$, podemos ver cómo la complejidad sigue resultando lineal, pero dependiendo de la cantidad de palabras del diccionario:

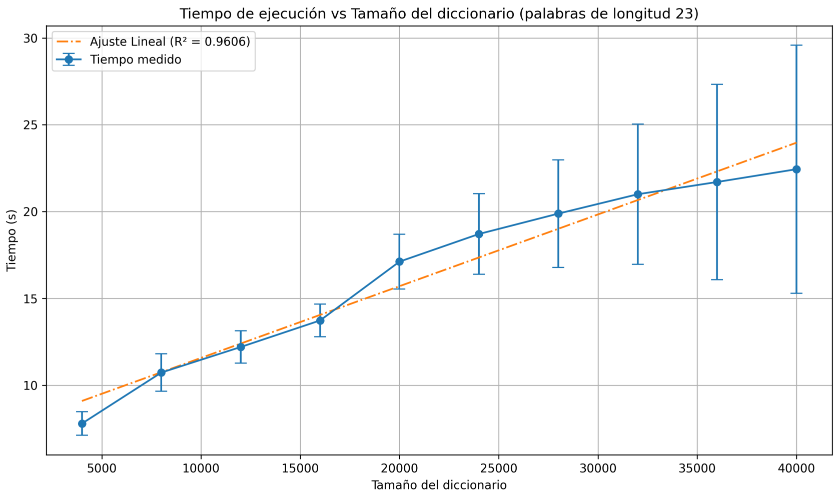

Figura 5: Mantengo n chico y aumento k

Podemos ver entonces que:

$$\mathcal{O}(m \cdot (k + (L^2) \cdot n)) = \boxed{\mathcal{O}(m \cdot k)}$$

Caso k y n chico, $L \gg$n

En este caso, L no es despreciable a comparación de n. En este caso, la complejidad pasa a ser cúbica, quedando el gráfico de esta forma:

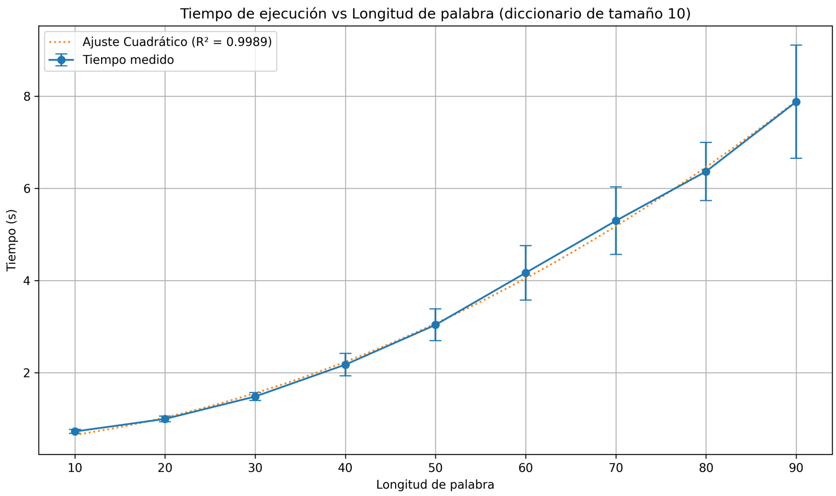

Figura 6: L grande, m constante

Y la complejidad queda:

$$\mathcal{O}(m \cdot (k + (L^2) \cdot n)) = \boxed{\mathcal{O}(m \cdot L^2)}$$