Teoría de Algoritmos
(TB024) Curso Buchwald - Genender

Trabajo Práctico 2
Que parezca programación dinámica

5 de mayo de 2025

Conde Cardó,	Colombo Farre,	Willson,
Lucas Ariel	Iván Joel	Marina
112201	111671	103532

1. Introducción

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 "estanocheenelmuellealassiete", este sería un posible mensaje, el cual correspondería a "esta noche en el muelle a las siete", mientras que "estamikheestado"no lo es, ya que no podemos separar de forma de generar todas palabras del idioma español (en cambio, si fuera "estamiheestado"podría ser "esta 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 "estamosquea", que puede separarse en "estamos que a.º bien "esta 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

- 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.
- 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).
- 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.
- 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.).
- 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.
- 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.

2. Análisis del problema

Luego de leer la consigna reiteradas veces y realizar un análisis de la misma, comprendimos que el problema consiste en determinar si, dada una cadena de n caracteres, esta puede ser separada en una secuencia de palabras válidas (pertenecientes a un diccionario dado) o, en caso contrario, concluir que la misma no se trata de un mensaje válido. Asimismo, cuando la cadena efectivamente se trata de un mensaje, debemos determinar cuál es este.

Seguidamente, identificamos dos puntos clave para nuestra futura solución:

- Una misma palabra del diccionario puede ser utilizada múltiples veces.
- Cuando termina una palabra debe comenzar otra (en el carácter posterior al que dio por finalizada la anterior) o terminar el mensaje. Es otras palabras, no pueden quedar caracteres "sueltos" ni puede haber superposición entre palabras.

2.1. Ejemplo

Para describir mejor nuestro análisis del problema, consideramos pertinente mostrar un ejemplo sobre su comportamiento.

Si recibimos una cadena desencriptada que dice

universidaddebuenosaires

y un diccionario (set de palabras) cuyo contenido es

aire, aires, buen, buenos, colegio, de, es, manzana, no, nos, pelota, si, universidad, ver

Nuestra solución debería determinar, **si** es posible segmentar la cadena en múltiples palabras válidas (sin importar si tiene sentido o no la frase), e informar que el posible mensaje es

universidad de buenos aires

Teniendo en cuenta esto, concluimos que ya estábamos en condiciones de desarrollar un algoritmo que lo resuelva.

3. Soluciones iniciales

3.1. Backtracking

Dado que ya habíamos identificado qué debíamos resolver, nuestra primera propuesta consistió en implementar un algoritmo sencillo que, utilizando la técnica de Backtracking, nos permitió obtener rápidamente una solución que pasa todas las pruebas proporcionadas por la cátedra.

A continuación, se muestra el código correspondiente:

```
    def validar_mensaje(cadena, palabras):
        mensaje = []
        if _validar_mensaje_bt(cadena, 0, palabras, mensaje):
            return " ".join(mensaje)
        return MSG_NO_ES


    def _validar_mensaje_bt(cadena, inicio, palabras, mensaje):
        if inicio == len(cadena):
            return True

        for fin in range(inicio + 1, len(cadena) + 1):
            palabra = cadena[inicio:fin]
            if palabra in palabras:
                mensaje.append(palabra)
                if _validar_mensaje_bt(cadena, fin, palabras, mensaje):
                    return True
                mensaje.pop()
        return False

```

No obstante, este enfoque no resulta para nada eficiente y su complejidad es, en el peor de los casos, de orden exponencial. Esto se debe a que, explícitamente, se evalúan todas las segmentaciones posibles hasta hallar una que resuelva el problema.

Sin embargo, nos resultó interesante la sencillez que presenta al reconstruir el mensaje (simplemente se unen las palabras usadas, que son previamente guardadas en una lista durante el proceso recursivo) y nos sirvió como puntapié inicial para desarrollar un algoritmo mucho más eficiente y robusto, utilizando la técnica indicada por la consigna: *Programación Dinámica*.

3.2. Programación Dinámica

Como vimos en clase, la mejor forma de encarar un problema que debe ser resuelto por programación dinámica es pensando en qué casos base pueden ocurrir. Estos representan una situación en que la solución es inmediata, no requieren ningún tipo de evaluación ni procesamiento extra.

En nuestro análisis identificamos una situación con estas características,

> **¿Qué sucede si la cadena es de largo cero?**
>
> Cuando la cadena tiene largo cero, la solución debe ser una cadena vacía, pues un mensaje vacío sigue siendo un mensaje.
>
> Como se detallará más adelante (durante el análisis y demostración de la ecuación de recurrencia), este caso nos servirá de "piso" para armar soluciones a problemas más grandes.

Posteriormente, dado que el anterior era el único caso base que identificamos, nos fueron surgiendo nuevas incógnitas sobre cómo deberíamos resolver problemas más grandes. Algunas de ellas fueron...

¿Qué sucede si la cadena es de largo uno?

Cuando la cadena está compuesta por un único carácter, se presentan dos situaciones posibles:

1. ese carácter forma una palabra, como "y" en el idioma español, o
2. ese carácter no forma ninguna palabra por lo que no representa un mensaje válido.

¿Qué sucede si la cadena es de largo dos?

Aquí se empiezan a complejizar las opciones:

1. se forma un mensaje si los caracteres juntos representan una palabra,
2. si los caracteres separados forman palabras (por ejemplo "y" y "o", en el diccionario español) también se constituye un mensaje (caso de dos cadenas de largo uno),
3. si **por lo menos uno** de los dos caracteres no es una palabra válida no se forma un mensaje válido.

De esta manera, llegamos a un caso general que describiremos a continuación.

3.2.1. Subproblemas

Como se pudo observar, la solución a un problema grande puede ser construida en función de problemas más pequeños. Queda claro entonces que es fundamental a la hora de lograr una solución que: *luego de encontrar una palabra válida, se debe encontrar un mensaje válido para la subcadena restante.*

Esto se puede apreciar en el ejemplo dado durante el análisis del problema. Luego de encontrar la palabra *universidad*, el algoritmo debería identificar si existe una solución para la parte restante, *debuenosaires*.

Esto es exactamente lo que realiza nuestra solución por backtracking pero ésta debe una y otra vez explorar todos los casos posibles e ir verificando su validez (¡incluso en ramas para las cuales ya se había encontrado una solución!). Por ejemplo, en la subcadena mencionada se encontrarían las palabras *de* y *buen*, pero la nueva subcadena *osaires* no nos permitiría llegar a una solución válida. Por lo tanto, se volvería para atrás y se buscarían nuevas soluciones.

Si bien nuestro algoritmo por programación dinámica realizará un proceso similar, éste aplicará una técnica de optimización primordial en cualquier algoritmo con estas características: la **memoización**.

3.2.2. Memoización

Esta técnica consiste en almacenar resultados para evitar repetir análisis al evaluar un subproblema en una misma instancia que ya fue previamente evaluada. Un claro ejemplo del funcionamiento de esta técnica es el siguiente:

Dado el diccionario,

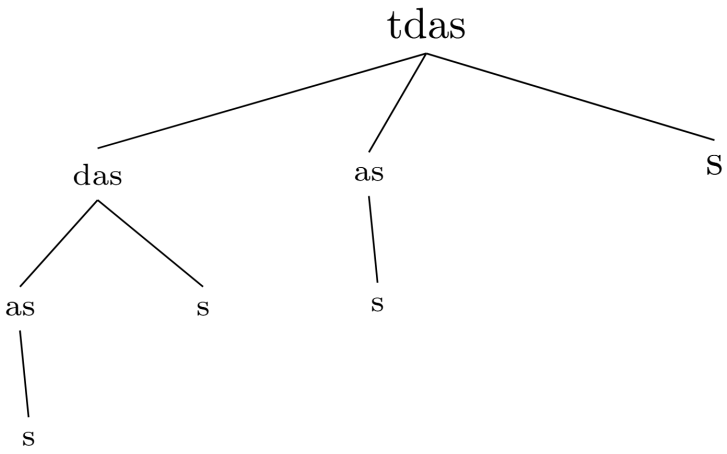
$$t,\ d,\ a,\ td,\ da,\ tda$$

y la cadena,

$$tdas$$

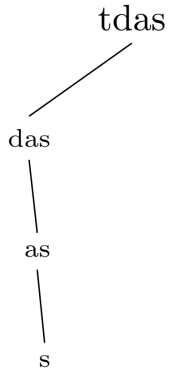
A simple vista se puede observar que la cadena no se trata de un mensaje válido, dado que la letra *s* no forma parte de ninguna de las palabras del diccionario y la cadena como palabra única tampoco.

Sin embargo, nos interesa ver cómo se comportaría nuestro algoritmo por backtracking al intentar hallar una solución. Para esto, graficamos el siguiente árbol, donde cada nodo representa un llamado recursivo que se realiza con la subcadena indicada:



Se evidencia que, al no haber memoización, se ejecutan reiteradas veces llamados recursivos que intentan resolver subcadenas para las cuáles ya se había determinado que no hay solución.

Aplicando memoización, se pasarían a ejecutar los siguientes llamados:



Esto se debe a que, al haber determinado y memorizado que las subcadenas *as* y *s* no poseen solución, no se vuelve a analizar una posible solución para éstas. Así, logramos determinar que la cadena no representa un mensaje de una manera mucho más rápida y, cuando sí lo es, podemos llegar a éste sin repetir ramas ya evaluadas.

De esta forma, ya estamos en condiciones de presentar nuestra primera solución utilizando nociones de programación dinámica.

3.2.3. Primera solución aplicando PD

Nuestra primera propuesta, teniendo en cuenta los conceptos de programación dinámica, consiste en una modificación al algoritmo por backtracking. Aplicando la técnica de memoización anteriormente explicada, esta solución evita recálculos innecesarios y logra resolver el problema en tiempo polinomial.

Seguidamente, se muestra el código correspondiente:

```
    def validar_mensaje(cadena, palabras):
        mensaje = []
        if _validar_mensaje_dp(cadena, 0, palabras, mensaje, set()):
            return " ".join(mensaje)
        return MSG_NO_ES


    def _validar_mensaje_dp(cadena, inicio, palabras, mensaje, memoization):
        if inicio == len(cadena):
            return True

        if inicio in memoization:
            return False

        for fin in range(inicio + 1, len(cadena) + 1):
            palabra = cadena[inicio:fin]
            if palabra in palabras:
                mensaje.append(palabra)
                if _validar_mensaje_dp(cadena, fin, palabras, mensaje, memoization)
    :
                    return True
                mensaje.pop()

        memoization.add(inicio)
        return False

```

Este algoritmo sigue una estrategia recursiva **"top-down"**, cuya estructura es bastante difícil de entender. Esto complica el cálculo de complejidad (tanto temporal, como espacial) y vuelve más engorrosa la optimización de memoria que, en soluciones **"bottom-up"**, son más intuitivas y simples.

Es por esto que decidimos centrarnos en encontrar la ecuación de recurrencia para luego, en base a ella, desarrollar un algoritmo **"bottom-up"** más sencillo y eficaz.

4. Solución definitiva - Ecuación de recurrencia

Para formular la ecuación de recurrencia recopilamos todas las conclusiones que habíamos sacado hasta el momento. En primer lugar, se trataría de una solución **"bottom-up"**, los subproblemas se irían componiendo desde el caso base e irían analizando qué pasa al agregar cada letra. A su vez, en cada paso tendríamos en cuenta las soluciones encontradas para **subproblemas anteriores**.

Por la naturaleza del problema, nos resultó más sencillo plantear primero visualmente la solución y, en base a eso, elaborar la ecuación de recurrencia.

Pensamos a la solución como una matriz de soluciones parciales, *pd*, donde los encabezados de las filas y las columnas fueran las letras individuales que componen la cadena en el orden que aparecen. Dado que cada eje tiene a todas las letras de la cadena, el tamaño de la matriz es n^2, con n igual a la longitud de la cadena. Cada celda entonces representa si hay solución para la subcadena que va del índice i al j o si anteriormente se encontró solución hasta el índice j.

Si bien el problema original pide devolver el potencial mensaje que tiene la cadena, optamos por abstraernos de ese resultado y abordar inicialmente la versión booleana del mismo: determinar si existe o no una segmentación válida. La idea es que sepamos en cada paso si se puede obtener una solución y la formulación del mensaje dejarla para la función de reconstrucción de la que se hablará más adelante.

Por ejemplo, para una cadena como *siyo* y el diccionario $\{si, y, yo, o\}$ tendríamos una matriz

	s	i	y	o
s				
i				
y				
o				

Cuadro 1: Matriz de soluciones n^2

Iremos llenando la matriz por fila, analizando si cada vez que se agrega una letra nueva se encuentra una solución para la subcadena que se forma.

Entonces, para la primera fila donde $i = 1$:

- para $j = 1$, como $cadena[1:1]$ = "*s*" no pertenece al diccionario, será `Falso`
- para $j = 2$, como $cadena[1:2]$ = "*si*" pertenece al diccionario, será `Verdadero`
- para $j = 3$, como $cadena[1:3]$ = "*siy*" no pertenece al diccionario, será `Falso`
- para $j = 4$, como $cadena[1:4]$ = "*siyo*" no pertenece al diccionario, será `Falso`

	s	i	y	o
s	F	V	F	F
i				
y				
o				

Cuadro 2: Matriz de soluciones n^2

Para la segunda fila, donde $i = 2$, empezaremos a analizar desde $j = 2$ porque la cadena debe analizarse en el orden original, no tendría sentido evaluar $pd[2][1]$ para este problema.

- para $j = 2$, $cadena[2:2]$ = "*i*" no pertenece al diccionario.

  Pero sabemos que $cadena[1:2]$ = "*si*" sí pertenece porque lo averiguamos en el paso anterior. Entonces, dado que nuestro problema no es de optimización sino que buscamos encontrar

alguna solución para la subcadena, podemos evaluar para cada i, j si **hasta ese j** ya se encontró alguna solución anteriormente **o** si la nueva subcadena $cadena[2:2]$ pertenece por sí sola al diccionario en caso de que **hasta el caracter anterior** a $i = 2$ también se haya encontrado solución.

Entonces, si bien $cadena[2:2] =$ "i" no pertenece, ya tenemos una solución hasta ese j en $pd[1][2]$ y por lo tanto marcaremos como `Verdadero` esa celda.

- para $j = 3$, como $cadena[2:3] =$ "iy" no pertenece al diccionario ni tampoco hay una solución para el mismo j en $i - 1$, es decir, $pd[1][3] =$ `Falso`, marcaremos esa celda como `Falso`
- para $j = 4$, como $cadena[2:4] =$ "iyo" no pertenece al diccionario y $pd[2][3] =$ `Falso`, marcaremos esa celda como `Falso`

	s	i	y	o
s	F	V	F	F
i		V	F	F
y				
o				

Cuadro 3: Matriz de soluciones n^2

Para la tercera fila, donde $i = 3$, empezaremos a analizar desde $j = 3$.

- para $j = 3$, tenemos $cadena[3:3] =$ "y" que pertenece al diccionario pero por cómo estamos planteando las soluciones, tenemos que pedir que también sea `Verdadero` la posición $pd[2][2]$.

  Esto se debe a que, como mencionamos con anterioridad, en cada posición al poner V estamos asegurando que toda la subcadena anterior desde $i = 0$ es segmentable. Entonces, sabiendo que $cadena[3:3] =$ "y", tengo que asegurarme de que desde $i - 1$ hacia atrás también haya solución, con corroborar la $pd[i-1][i-1]$ es suficiente. En este caso, ambas condiciones se cumplen y por lo tanto $pd[3][3] =$ `Verdadero`
- para $j = 4$, ocurre algo análogo al punto anterior. $cadena[3:4] =$ "yo" pertenece al diccionario **y** $pd[i-1][i-1] = pd[3][3] =$ `Verdadero` por lo que podemos asegurar que $pd[3][4] =$ `Verdadero`.

	s	i	y	o
s	F	V	F	F
i		V	F	F
y			V	V
o				

Cuadro 4: Matriz de soluciones n^2

Para la cuarta y última fila, donde $i = 4$, evaluaremos $j = 4$.

- Además de ser $cadena[4:4] =$ "o" una palabra perteneciente al diccionario **y** $pd[3][3] =$ `Verdadero`, también sucede que $pd[3][4]$ era `Verdadero`, por lo que por cualquiera de las dos opciones podría marcarse la casilla como `Verdadero`.

Para poder decir si la cadena en su totalidad es segmentable en palabras del diccionario, basta con mirar el valor de $pd[n][n]$, con n igual a la longitud de la cadena. Si tomamos el ejemplo recién presentado, podemos decir que la cadena era completamente segmentable por ser $pd[4][4] =$ `Verdadero`.

	s	i	y	o
s	F	V	F	F
i		V	F	F
y			V	V
o				V

Cuadro 5: Matriz de soluciones n^2

Concluido el llenado de la matriz, estuvimos en condiciones de formular la ecuación de recurrencia.

Nos resultó natural plantear una disyunción para cada par i, j. Ésta sería verdadera de cumplirse alguna de las opciones hasta j en cada caso.

Como fue expuesto en el ejemplo práctico, para cada subcadena analizamos si $cadena[i:j]$ era una palabra en su totalidad **y** $pd[i-1][i-1] =$ `Verdadero` **o** si para alguna subcadena anteriormente analizada ya se encontró solución **hasta** j, es decir, si $pd[i-1][j] =$ `Verdadero`.

Hay un caso adicional que no fue mencionado en su momento con el fin de presentar la complejidad de la ecuación de forma incremental. Para la primera fila de la matriz, también aplica el razonamiento $pd[i-1][j]$ debe ser `Verdadero` **o** $cadena[i:j] \in$ `diccionario` **y** $pd[i-1][i-1]$, pero en este caso el equivalente sería considerar $cadena[i:j] \in$ `diccionario` **y** $i = 1$. Es decir, podríamos tomar que su antecedente es la cadena vacía que siempre consideramos `Verdadero`. De esta manera, quedan cubiertas todas las alternativas posibles para $i <= j$:

$$pd[i][j] = \begin{cases} V & \text{si} \quad pd[i-1][j] \\ V & \text{si} \quad i = 1 \land cadena[i:j] \in \texttt{diccionario} \\ V & \text{si} \quad pd[i-1][i-1] \land cadena[i:j] \in \text{diccionario} \\ F & \text{cualquier otro caso} \end{cases}$$

La ecuación se puede reescribir de forma tal que abarque todos los casos posibles. Aquí, $pd(j) = V$ significa que hasta el índice j la cadena es segmentable

$$pd(j) = \bigvee_{0 \leq i \leq j \leq n} \Big(pd(i-1) \land \text{cadena}[i:j] \in \text{diccionario} \Big)$$

Para empezar, tenemos la disyunción que expone que alguno de los casos de la conjunción que tiene dentro debe ser verdadera para que haya solución hasta el índice j de la cadena. Los valores que se pueden adoptar de i y j son $0 \leq i \leq j \leq n$, donde 0 sería el caso base que es siempre `Verdadero` y n la longitud de la cadena. Aquí consideramos que los índices de la cadena van de 1 a n, mientras que los de pd empiezan en 0.

Luego, tenemos la conjunción. En la forma reescrita de la ecuación de recurrencia queda más en claro que no importa desde dónde venía "arrastrando" el valor `Verdadero` cuando observábamos $pd[i-1][j]$, ni cuáles son las segmentaciones posibles hasta ese punto. Lo importante es que, si $pd(j) =$ `Verdadero`, hasta j para **por lo menos un** i entre 0 y j, $cadena[i:j]$ es una palabra del diccionario y todo lo anterior representado por $pd(i-1)$ es segmentable. Esto incluye al caso base y será `Falso` cuando no sea posible segmentar como demostraremos a continuación.

4.1. Demostración

Para demostrar que la ecuación de recurrencia planteada encuentra siempre un mensaje en caso de ser posible o detecta cuando no lo sea, se seguirá el método inductivo.

Como fue detallado en el punto anterior, la ecuación de recurrencia es

$$pd(j) = \bigvee_{0 \leq i \leq j \leq n} \Big(pd(i-1) \wedge \text{cadena}[i:j] \in \text{diccionario} \Big)$$

Entonces, esta ecuación plantea que $pd(j)$ será `Verdadero` si

- $pd(i-1) =$ `Verdadero`, se encontró una solución hasta alguna posición $i-1$ en la cadena y
- la subcadena $cadena[i:j]$ pertenece al diccionario

Si alguno de esos dos términos es `Falso` para todos los i posibles, entonces $pd(j)$ también lo será.

Nuestro objetivo en esta demostración es probar por inducción que

$$\forall h / n_0 \leq h \leq n : v[P(h) \rightarrow P(h+1)] = \text{V}$$

siendo $n_0 = 0$, n el largo de la cadena y $P(h)$ la proposición "la subcadena $cadena[0:h]$ es segmentable si y sólo si $pd(h) =$ `Verdadero`".

Para empezar es necesario plantear un caso base, $P(n_0)$. En nuestro problema será el de una cadena vacía cuyo resultado para $pd(0)$ es `Verdadero`.

Tomaremos a $P(h)$ como nuestra hipótesis inductiva y trataremos de probar que la implicancia es verdadera. Gracias al principio de inducción, podremos afirmar que la tesis también era verdadera $\forall h \geq 0$ y que por lo tanto nuestra ecuación de recurrencia describe el comportamiento del algoritmo en cualquier caso.

Para probar la veracidad de la implicancia seguiremos el método directo. Asumiremos que la hipótesis es verdadera, es decir, que

$$pd(h) = \text{True} \Leftrightarrow \text{cadena[0:h] para } h < n$$

y buscaremos llegar a que vale para $h+1$ también.

En nuestro caso, dado que tenemos un `si y sólo si`, tendremos que probar la implicancia en ambos sentidos.

cadena[0:h+1] segmentable → pd(h+1)=`Verdadero`

Nuestro antecedente es $cadena[0:h+1]$ segmentable. Esto quiere decir que existe algún $i-1$ tal que:

- la subcadena $cadena[0:i-1]$ es segmentable
- la subcadena $cadena[i:h+1]$ pertenece al diccionario

Por hipótesis inductiva, tener una subcadena segmentable hasta el subíndice $i-1$ implica $pd(i-1) =$ Verdadero. Además, como $cadena[i:h+1]$ pertenece al diccionario, se cumplen las dos condiciones de la conjunción de la ecuación de recurrencia, por lo tanto se cumple la disyunción y por ende se puede afirmar que $pd(h+1) =$ Verdadero.

De esta forma queda mostrado que la implicancia $cadena[0:h+1]$ segmentable → $pd(h+1) =$ `Verdadero` es verdadera.

pd(h+1)=`Verdadero` → cadena[0:h+1] segmentable

En este caso, el antecedente que asumimos verdadero es $pd(h+1) =$ `Verdadero`. Según la ecuación de recurrencia, esto quiere decir que existe un i tal que:

- $pd(i-1) =$ `Verdadero` y
- la subcadena $cadena[i : h+1]$ pertenece al diccionario

Por hipótesis inductiva, $pd(i-1) =$ `Verdadero` a su vez significa que $cadena[0 : i-1]$ es segmentable. Entonces, $cadena[0 : h+1]$ es completamente segmentable ya que $cadena[0 : i-1]$ lo es y $cadena[i : h+1]$ pertenece al diccionario.

De esta manera, queda demostrado que la implicancia $pd(h+1) =$ `Verdadero` $\rightarrow$ $cadena[0, h+1]$ segmentable también es verdadera.

Habiendo demostrado la doble implicancia, se puede afirmar que se cumple la proposición para $h+1$. Con esto podemos entonces asegurar que

$$pd(j) = \texttt{Verdadero} \Leftrightarrow cadena[i : j] \text{ segmentable } \forall i, j / 0 \leq i \leq j \leq \text{len(cadena)}$$

siendo $pd(j)$ la ecuación de recurrencia que indica que hasta j la cadena es segmentable.

En particular, $pd(n) =$ `Verdadero` $\Leftrightarrow$ la cadena completa es segmentable, n igual a la longitud de la misma.

Si la cadena no puede ser segmentada completamente, entonces no existe una secuencia de palabras del diccionario que cubra toda la cadena, y por la dirección $\Leftarrow$ de la implicancia, se sigue que $pd(n) =$ `Falso`.

En la ecuación de recurrencia esto se vería como que no se cumple la conjunción para ningún i, es decir, que $pd(i-1) =$ `Falso` o $cadena[i : j] \notin diccionario$ para todos los $0 \leq i \leq j \leq n$.

Así queda mostrado que el algoritmo detecta correctamente tanto los casos en los que la subcadena $cadena[i : j]$ es segmentable como aquellos en los que no lo es, incluido el caso en el que $i = 0$, $j = n$ cuyo valor determinará si la cadena completa es o no un mensaje válido.

4.2. Implementación del algoritmo

Pusimos en funcionamiento la ecuación de recurrencia desarrollada mediante la implementación del siguiente algoritmo que, a diferencia del presentado anteriormente, sigue una estructura **bottom-up**:

```
    def validar_mensaje(cadena, palabras):
        if len(cadena) == 0:
            return ""

        matriz = [[False for _ in range(len(cadena))] for _ in range(len(cadena))]

        for i in range(len(cadena)):
            for j in range(i, len(cadena)):
                if matriz[i - 1][j]:
                    matriz[i][j] = True

                palabra = cadena[i : j + 1]
                if palabra in palabras and (i == 0 or matriz[i - 1][i - 1]):
                    matriz[i][j] = True

        if matriz[-1][-1] == False:
            return MSG_NO_ES

        return reconstruir_mensaje(matriz, cadena)

```

Tal como se aprecia, la función comienza creando una matriz de tamaño n^2, siendo n el largo de la cadena a analizar. En cada celda se guardan estados booleanos, inicializados en `False`, que indican si se encontró una solución en el subproblema dado. En esta implementación bottom-up, esa es la manera de aplicar la memoización.

Luego, dado que las cadenas son arreglos de caracteres, iteramos anidadamente las n posiciones del arreglo.

Allí es donde aplicamos la lógica de nuestra ecuación de recurrencia y evaluamos:

- si ya se encontró una solución para el subproblema finalizado en la misma posición, pero arrancando la subcadena (formada por los i, $j+1$ en iteración) en una anterior **o**
- si la subcadena actual forma una palabra del diccionario y el subproblema restante hacia atrás, es decir, en la posición $i-1$ para ambas entradas de la matriz representa una solución válida, seteamos la celda actual de la matriz con el valor `True`.
- En cualquier otro caso se determina que en el subproblema dado no se encuentra una solución y, por ende, se mantiene el valor **False** seteado por defecto al comienzo.

Finalmente, nuestro algoritmo realiza una última tarea. Si la última posición de la matriz se mantuvo en `False`, se determina que la cadena no es un mensaje válido. Si en cambio la última posición es `True`, se determina que se encontró un mensaje válido y se lo reconstruye retornando el valor dado por la siguiente función:

```
    def reconstruir_mensaje(matriz, cadena):
        mensaje = []
        i, j = len(matriz) - 1, len(matriz[0]) - 1

        while i >= 0:
            if i > 0 and matriz[i][j] == matriz[i - 1][j]:
                i -= 1
                continue
            palabra = cadena[i : j + 1]
            mensaje.append(palabra)
            i, j = i - 1, i - 1

        return " ".join(mensaje[::-1])

```

Esta recorre la matriz desde la última posición, siguiendo el camino dado por los estados válidos y almacenando las palabras formadas en un arreglo que, al final de la función, es invertido y retornado en forma de mensaje, con las palabras separadas por espacios.

4.3. Optimizaciones

Una vez consolidada la estructura base del programa, formulamos algunas optimizaciones para nuestro código.

Estas nos permitieron reducir los tiempos de ejecución en gran medida y ocupar menos memoria, principalmente al ejecutarlo con cadenas muy largas.

```
1     def validar_mensaje(cadena, palabras):
2         if len(cadena) == 0:
3             return ""
4
5         validacion = [None] * len(cadena)
6         largo_min, largo_max = buscar_largos_extremos(palabras)
7
8         for ini in range(len(cadena)):
9             for i in range(ini + largo_min - 1, min(ini + largo_max, len(cadena))):
10                fin = i + 1
11
12                if validacion[i] is None:
13                    if cadena[ini:fin] in palabras and (ini == 0 or validacion[ini
    - 1] is not None):
14                        validacion[i] = ini
15
16            if validacion[-1] is not None:
17                break
18
19        if validacion[-1] is None:
20            return MSG_NO_ES
21
22        return reconstruir_mensaje(validacion, cadena)
23
```

Esta nueva implementación:

- **Busca el largo mínimo y máximo de las palabras del diccionario**: como será explicado a detalle en el próximo apartado, esta optimización le suma un término lineal respecto de otra variable a la complejidad del algoritmo base. Sin embargo, al realizar la búsqueda de palabras únicamente desde el posible inicio de la misma hasta el intervalo entre el largo mínimo y el máximo, resulta de una optimización extremadamente eficiente, especialmente en cadenas largas o cuando el diccionario de palabras presentado no posee un largo muy grande.

  Un punto muy importante es que el largo de las palabras suele ser muy reducido. Por ejemplo, en el idioma español la palabra más corta posee una sola letra, mientras que la más larga 23. De esta forma, logramos que el ciclo *for* interno y la formación de subcadenas tengan una complejidad prácticamente despreciable respecto al potencial tamaño de la cadena entera. Esto quedará más claro cuando estudiemos en detalle de complejidad.

- **Utiliza un arreglo en lugar de matriz**: Inicialmente almacenábamos los resultados en una matriz de tamaño n^2. Con esta optimización logramos reducir el espacio necesario reemplazándola por un arreglo de tamaño n, en donde la **i-ésima posición** representa el **índice de finalización** de la palabra formada y **su contenido** es el **índice de comienzo** de la palabra en cuestión. De esta forma no solo logramos una gran reducción en la complejidad espacial, que pasa de ser $O(n^2)$ a ser $O(n)$, sino que también simplificamos considerablemente la lógica de la función de reconstrucción. A continuación de esta sección, se apreciará que dicha función ahora simplemente recorre el arreglo desde la última posición, formando y almacenando las palabras, y saltando a la posición anterior a la almacenada.

- **Finaliza tempranamente**: se aplica en las líneas 16 y 17, es un corte que se da cuando el algoritmo ya encontró una solución para la última palabra (es decir, la que termina en la última posición) y finaliza su ejecución. Evitando así, evaluaciones innecesarias.

Los nuevos algoritmos de reconstrucción y de búsqueda de largos de palabras, luego de aplicar estas "mejoras"son:

```
    def reconstruir_mensaje(validacion, cadena):
        i = len(validacion) - 1
        palabras = []

        while i >= 0:
            palabras.append(cadena[validacion[i] : i + 1])
            i = validacion[i] - 1

        return " ".join(palabras[::-1])

    def buscar_largos_extremos(palabras):
        min, max = float('inf'), 0

        for palabra in palabras:
            if len(palabra) < min:
                min = len(palabra)
            if len(palabra) > max:
                max = len(palabra)

        return min, max

```

Adicionalmente, incorporamos un último avance que, si bien no mejora directamente la rapidez de nuestro algoritmo, permite al usuario del mismo setear manualmente los valores de las variables *largo_min* y *largo_max*.

De esta forma, si se lo quiere ejecutar reiteradas veces con un mismo diccionario, se puede evitar la complejidad extra de buscar para cada cadena estos largos, haciéndolo una sola vez.

Presentamos entonces el código definitivo de nuestro algoritmo principal:

```
    def validar_mensaje(cadena, palabras, largo_min = None, largo_max = None):
        if len(cadena) == 0:
            return ""

        validacion = [None] * len(cadena)

        if largo_min is None or largo_max is None:
            largo_min, largo_max = buscar_largos_extremos(palabras)

        for ini in range(len(cadena)):
            for i in range(ini + largo_min - 1, min(ini + largo_max, len(cadena))):
                fin = i + 1

                if validacion[i] is None:
                    if cadena[ini:fin] in palabras and (ini == 0 or validacion[ini
    - 1] is not None):
                        validacion[i] = ini

            if validacion[-1] is not None:
                break

        if validacion[-1] is None:
            return MSG_NO_ES

        return reconstruir_mensaje(validacion, cadena)

```

Aquí, si se especifican manualmente valores para *largo_min* y *largo_max* antes de invocar a la función, se omite el recálculo de estos al ejecutar la misma.

Aclaración: Teniendo en cuenta que la consigna expresa de forma clara que el problema debe ser resuelto para una única cadena, decidimos, a la hora de hacer mediciones con múltiples cadenas y un mismo diccionario, no utilizar esta optimización.

4.4. Análisis de la complejidad temporal

Inicialmente nuestro algoritmo, como ya mencionamos, crea un arreglo de largo n donde se almacenan las soluciones. Dado que n representa el largo de la cadena, esto tiene una complejidad de $\mathcal{O}(n)$.

Luego, se realiza la búsqueda del largo mínimo y máximo de las palabras del diccionario. Este algoritmo consta de un ciclo `for` que recorre todas las palabras que, si describimos su cantidad como k, resulta en una complejidad de $\mathcal{O}(k)$.

Posteriormente se recorren los n índices de la cadena. Estos son utilizados como posibles inicios de palabras que podrían terminar en el índice igual al valor en iteración del ciclo que tiene anidado. Allí, se recorre todo el intervalo de largos que podría tener una palabra. Puesto que `largo_min` y `largo_max` podrían valer uno y un valor mayor a n respectivamente, determinamos que este ciclo realiza, en el peor de los casos, n iteraciones.

Se debe recordar que el costo de formar una palabra con `slices` es $\mathcal{O}(l)$, siendo l el largo de la cadena. En este caso, como determinamos que cada palabra podría tener un largo máximo de n caracteres, a la complejidad anterior debemos adicionarle este $\mathcal{O}(n)$ extra. Entonces, si cada bucle recorre n posiciones y adentro realiza una operación $\mathcal{O}(n)$, su complejidad total es $\mathcal{O}(n^3)$.

Asimismo, se realizan operaciones $\mathcal{O}(l)$ a lo largo de todo el algoritmo. Como se realiza una cantidad constante de éstas, su peso es insignificante respecto al de los aspectos recién evaluados.

Por lo tanto, llegamos a la conclusión de que la complejidad temporal resultante de nuestro algoritmo es $\mathcal{O}(k + n^3)$ o, si se prefiere mayor especificidad, $\mathcal{O}(max(k, n^3))$. Está dada por los tres aspectos más importantes del problema: la búsqueda de largos, la pareja de ciclos anidados e, internamente, la formación de subcadenas.

Antes de retornar la solución se realiza la reconstrucción del mensaje que, como recorre el arreglo formando las palabras y, finalmente, juntando todo, tiene una complejidad de $\mathcal{O}(n)$. Esta llamada resulta insignificante respecto a la que determinamos como complejidad resultante.

A esta altura ya estamos en condiciones de mostrar por qué nuestra última optimización resulta muy eficiente, aunque dependa exclusivamente de cómo se utilice el algoritmo.

Supongamos que se quiere analizar m mensajes, con $m > 1$, utilizando nuestra solución y un mismo diccionario de palabras. Sin implementar la última optimización, esto tendría una complejidad de $\mathcal{O}(m * (k + n^3))$, suponiendo también que todas las cadenas tienen el mismo largo. En cambio, incorporando la mejora, esta complejidad resultaría $\mathcal{O}(k + m * n^3))$. Al sustraer k de nuestro algoritmo, logramos que la complejidad de la operación pase a ser lineal en función de esta variable, ya que el procesamiento de palabras se realiza una única vez, y ahora m solo multiplica al $\mathcal{O}(n^3)$ que proviene de nuestra función. Por lo tanto, proporciona una mejora sustancial en los tiempos de ejecución.

5. Mediciones y análisis de variabilidad

Para corroborar que la complejidad hallada analíticamente sea correcta, realizamos mediciones de tiempos de ejecución para problemas de distintos tamaños de cadenas y diccionarios con una diversa cantidad de palabras.

Fue necesario realizar un ajuste con una curva cúbica y analizar el error asociado a dicho ajuste para corroborar la complejidad teórica planteada. Seguimos la técnica de cuadrados mínimos cuyo objetivo es hallar una función que minimice la suma de los cuadrados de las diferencias entre los valores medidos y los calculados por la función que ajusta.

Dados n puntos $p = (x_i, y_i)$, el error cuadrático queda definido como $ec_i = (y_i - f(x_i))^2$. Lo que se busca entonces es dar con la $f(x)$ que minimice la sumatoria de dichos errores, esto se conoce como el error cuadrático total.

Para hallar la función más adecuada usamos `curve_fit` de scipy que calcula los parámetros óptimos que minimizan el error cuadrático al ajustar con una función propuesta. Basándonos en el análisis teórico anteriormente mencionado, propusimos $f(x) = c_1 \cdot x^3 + c_2$.

Inicialmente, decidimos probar el algoritmo principal, sin la reconstrucción, y con casos de uso "comunes". Estos se dan cuando las cadenas están formadas por palabras de un idioma corriente, en este caso el español, en el que la cantidad de letras de cada palabra tiene aproximadamente una esperanza de 5 caracteres y una varianza de 3. De esta forma lograríamos tener una representación más exacta de la complejidad temporal que nuestro algoritmo presenta en la mayoría de los casos.

Para esto, usamos el archivo `supergigante.txt` provisto por la cátedra. Este diccionario contiene 1.000 palabras compuestas por entre 3 y 18 caracteres y que cumplen con las características mencionadas en el párrafo anterior. Luego, tomando palabras de allí, generamos cadenas de prueba, asegurándonos de que todas formen mensajes válidos. Las cadenas generadas tienen, ascendentemente, entre 5 y 10.000 caracteres.

Finalmente, seteamos nuestra herramienta medidora de tiempos de ejecución para que realice el promedio de 4 ejecuciones de cada ejemplo. Así, conseguimos minimizar el impacto de cualquier perturbación ajena a nuestro algoritmo.

Al medir las ejecuciones, obtuvimos como resultado esta curva de *tiempo de ejecución (s) vs. tamaño de la cadena (n)*:

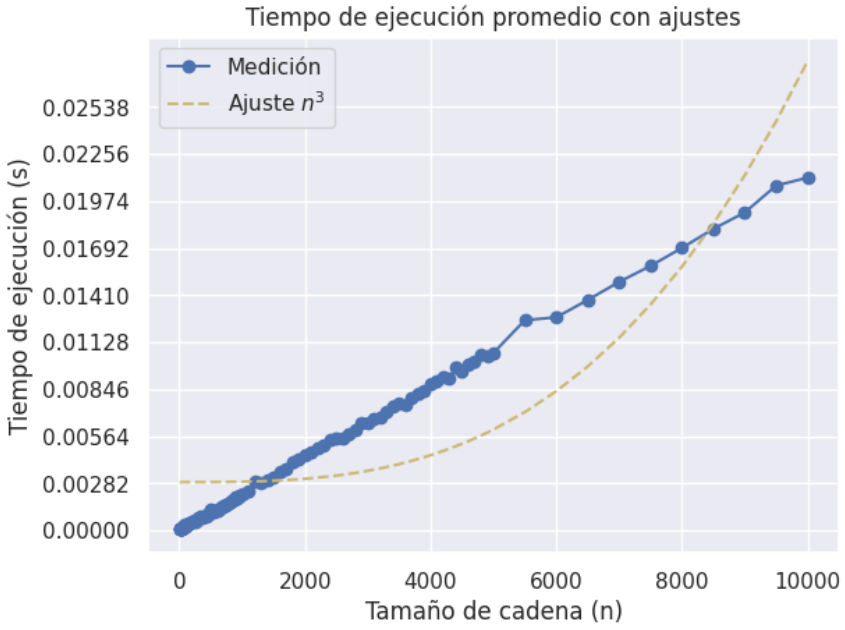


Figura 1: Tiempo de ejecución vs tamaño de la cadena. Algoritmo optimizado, diccionario español de 1.000 palabras, set de cadenas con 10.000 caracteres máximo

Como el tamaño del diccionario, k, es 1.000, de antemano supimos que no influiría en el crecimiento temporal de nuestras pruebas ya que en la mayoría de los casos es considerablemente inferior al largo de la cadena al cubo, n^3. Sin embargo, los tiempos no se aproximan en absoluto al ajuste cúbico al que, en teoría, deberían acercarse.

Para intentar comprender por qué se estaba dando ese fenómeno, decidimos exponer nuestra primera solución bottom-up, sin ninguna optimización, a estas mismas pruebas. Tras ejecutarlas obtuvimos el siguiente resultado:

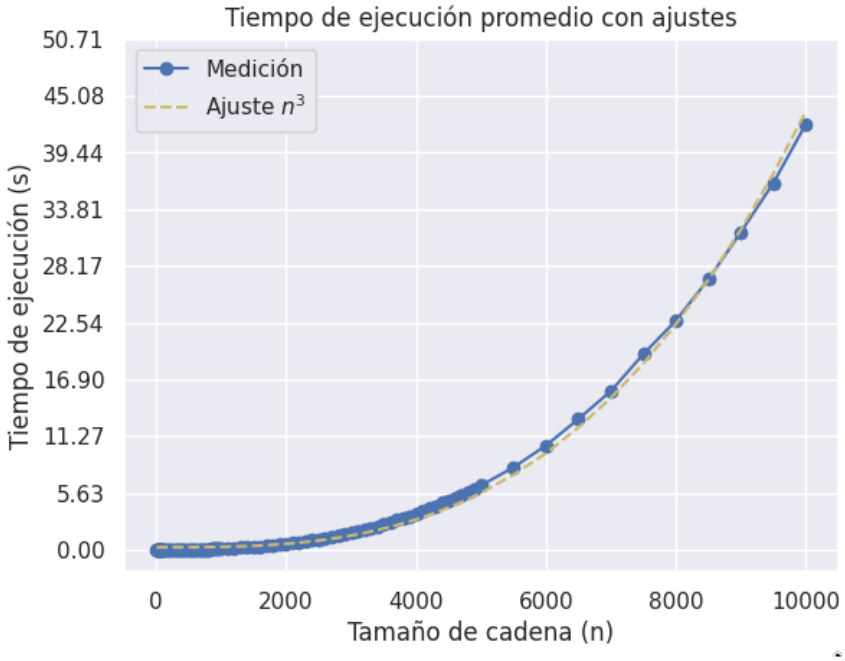


Figura 2: Tiempo de ejecución vs tamaño de la cadena. Algoritmo no-optimizado, diccionario español de 1.000 palabras, set de cadenas con 10.000 caracteres máximo

Se puede observar que, en este caso, la curva sí se aproxima de forma casi perfecta al ajuste cúbico. Además, el error cuadrático total dio 16.122502480948366, que para la cantidad de puntos utilizados y teniendo en cuenta los tiempos de ejecución, se puede decir que es un muy buen ajuste. Asimismo, es notable que los tiempos de ejecución resultaron considerablemente superiores en comparación con los obtenidos para la prueba anterior como se puede observar en el Cuadro 6.

Ante esta situación, empezamos a estudiar cómo la **variabilidad en los datos** podría estar generando tamaña diferencia entre las dos versiones.

n	Tiempos (s)	
	Con optimizaciones	Sin optimizaciones
5	7.433891296386719e-05	2.841949462890625e-05
10	6.427764892578126e-05	2.5463104248046874e-05
50	0.00016036033630371094	0.0003009319305419922
100	0.000289154052734375	0.0007805824279785156
250	0.00054595718383789	0.0007805824279785156
500	0.0012988567352294922	0.024972200393676758
750	0.001545858383178711	0.058911705017089845
1000	0.002163410186767578	0.11606497764587402
2500	0.00556182861328125	1.046910858154297
5000	0.01064891815185547	6.394043874740601
7500	0.015861892700195314	19.437801837921143
8500	0.018080472946166992	26.85020399093628
10000	0.021150922775268553	42.262178373336795

Cuadro 6: Comparación de tiempos de ejecución usando las dos versiones del algoritmo con el mismo diccionario y set de cadenas

Tras un largo análisis, nos hicimos una pregunta clave: **¿qué sucede si las palabras del diccionario son muy cortas respecto al largo total de la cadena?**.

Algunas de las cadenas de esta prueba tenían largos muy grandes respecto del tamaño de las palabras del diccionario que iban entre 3 y 18 caracteres. Al buscar `largo_min` y `largo_max` se acotaba mucho el rango del `for` anidado, llegando a hacer una cantidad insignificante de iteraciones para el largo original de la cadena. Además, el costo de generar subcadenas también resultaría extremadamente bajo en comparación ya que las mismas serían de un largo máximo igual a 18 y los `slices` en la práctica no demorarían tanto como calculamos con la complejidad teórica igual a $\mathcal{O}(n)$.

Con esta información, creamos otro set de pruebas, esta vez con cadenas cuyos largos alcanzaban los 200.000 caracteres. También le agregamos un ajuste lineal al gráfico, a `curve_fit` le pasamos entonces $f(x) = c_1 \cdot x + c_2$, donde el tiempo de ejecución es directamente proporcional al tamaño de la entrada n. Los resultados fueron los siguientes:

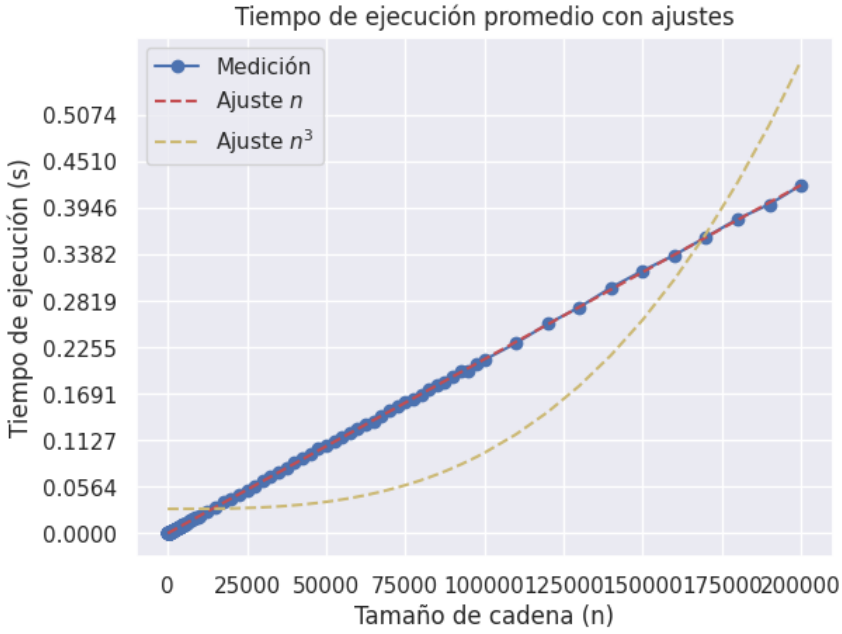


Figura 3: Tiempo de ejecución vs tamaño de la cadena. Algoritmo optimizado, diccionario español de 1.000 palabras, set de cadenas con 200.000 caracteres máximo

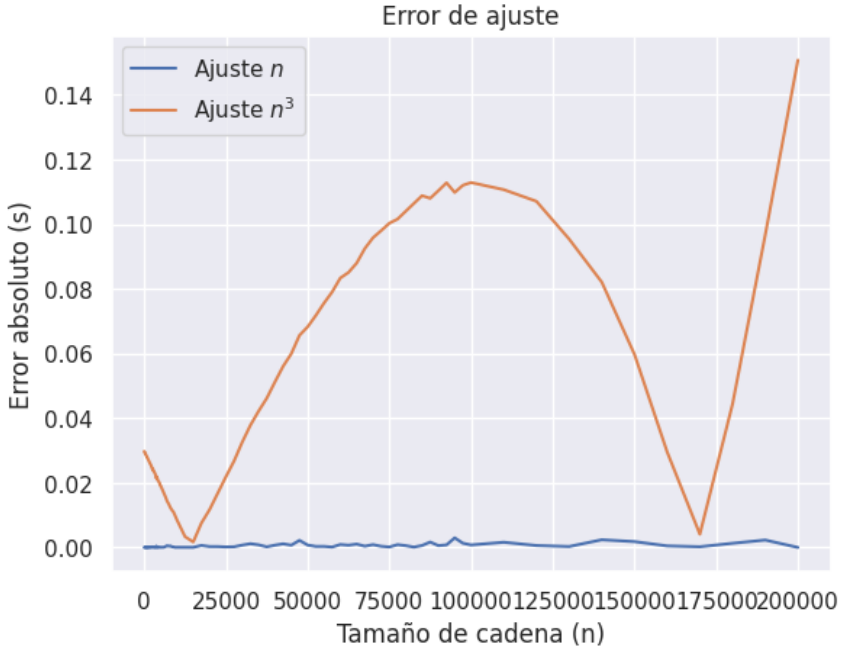


Figura 4: Error absoluto vs tamaño de entrada. Algoritmo optimizado, diccionario español de 1.000 palabras, set de cadenas con 200.000 caracteres máximo

Ahora, es claramente visible que nuestro algoritmo optimizado se comporta con una complejidad temporal lineal ante un diccionario y set de cadenas con estas características. Además, el error cuadrático total para este ajuste dio 5.341086470376983e-05, indicando incluso numéricamente que es un muy buen ajuste.

Por consiguiente, pudimos concluir que cuando las palabras del diccionario son cortas en comparación a n, los tiempos de ejecución dependen únicamente del máximo entre la cantidad de

palabras del diccionario (factor que será analizado más adelante), k, y la longitud de lineal la cadena, n, no la cuadrática. Entonces, podemos asumir que este mismo comportamiento ocurriría al desencriptar cadenas largas en cualquier idioma, como los que posiblemente envíe *el soplón*.

Seguros de que nuestro análisis de complejidad era correcto, probamos un escenario diferente. En este caso, creamos un diccionario pequeño de 500 palabras de entre 1 y 10.000 caracteres (caso poco común para los diccionarios reales). Estas palabras estaban formadas por caracteres aleatorios. Los tamaños de las cadenas nuevamente presentan una longitud ascendente hasta llegar a los 10.000 caracteres.

Al llevar a cabo la medición temporal, obtuvimos estos resultados:

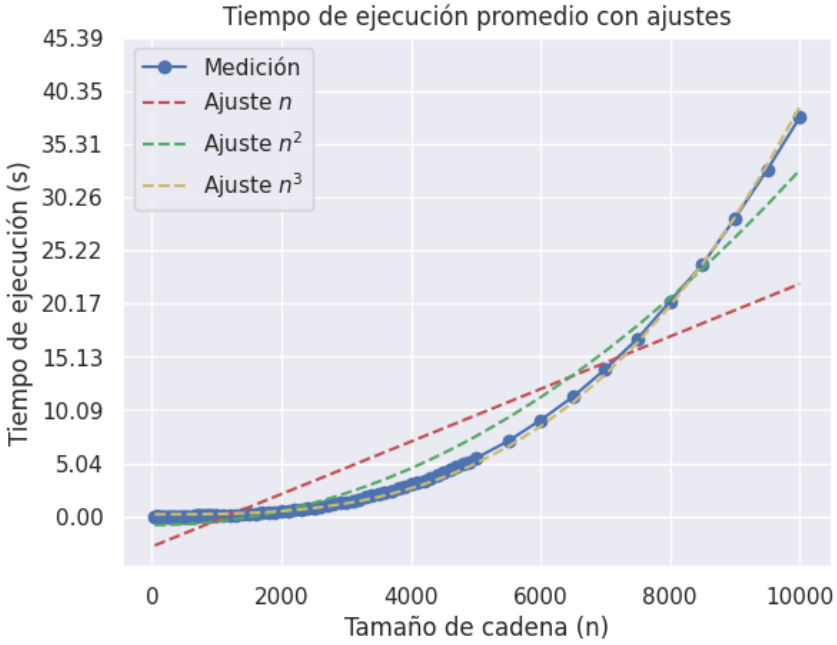


Figura 5: Tiempo de ejecución vs tamaño de la cadena. Algoritmo optimizado, diccionario aleatorio de 500 palabras, set de cadenas con 10.000 caracteres máximo

Aquí mostramos distintos ajustes (lineal, cuadrático y cúbico) junto a los datos medidos y, como esperábamos, se nota con claridad que el cúbico es el que mejor se aproxima. Su error cuadrático total fue 6.790984805668033, muy bueno para la cantidad de puntos utilizados y los valores temporales obtenidos.

Asimismo, al comparar los errores de cada ajuste comprobamos lo recién afirmado:

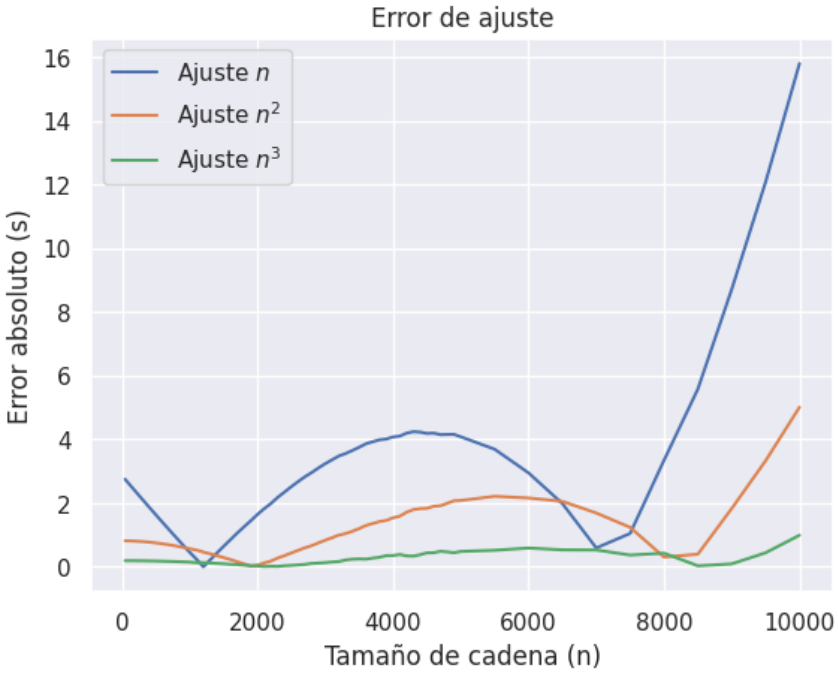


Figura 6:

Con este gráfico, estuvimos en condiciones de confirmar que, si la diferencia entre los largos mínimo y máximo de las palabras del diccionario resulta mayor o igual al largo de la cadena en evaluación, el algoritmo presenta un comportamiento cúbico. Esto se da porque ante este escenario, el rango del `for` anidado y los `slices` de la cadena no son para nada despreciables, más bien son totalmente comparables a n. Por lo tanto, la complejidad que habíamos obtenido teóricamente, cuando n^3 es mayor a k, es correcta.

Finalmente nos resta analizar una última situación referente a la variabilidad de los datos, **¿Qué sucede cuando k es mayor a n^3?**

Para probar este escenario creamos un diccionario con 750.000 palabras random, de entre 1 y 175 caracteres, ligeramente superior al largo máximo de la mayoría de los idiomas pero, aún así, de un largo pequeño. Creamos cadenas tenían entre 5 y 90 caracteres. Como $90^3 < 750{,}000$, nuestra sospecha era que todos los tiempos de ejecución debería ser prácticamente iguales.

Entonces, ejecutamos las mediciones y obtuvimos estos resultados:

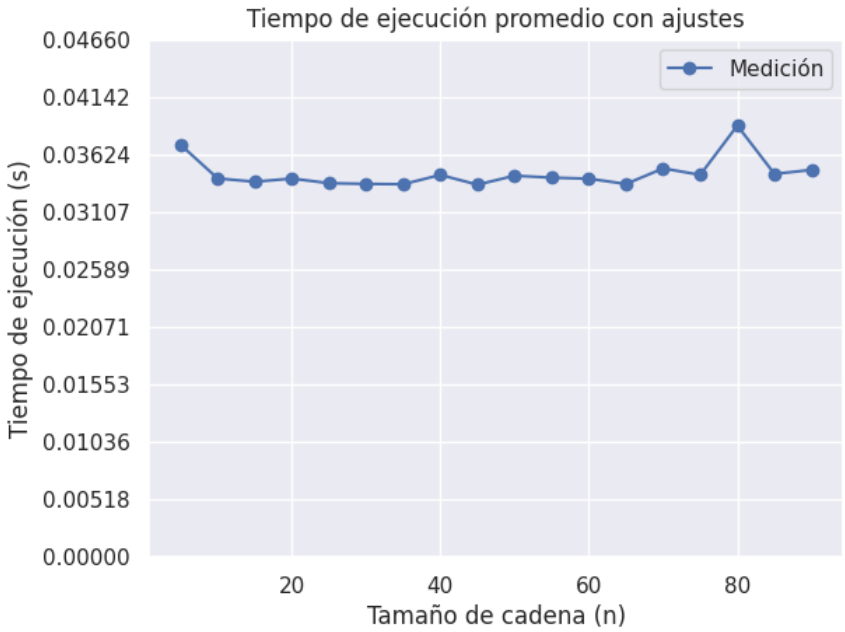


Figura 7: Tiempo de ejecución vs tamaño de la cadena. Algoritmo optimizado, diccionario aleatorio de 750.000 palabras, set de cadenas con 90 caracteres máximo

n	Tiempos (s)
5	0.037118959426879886
10	0.034082221984863284
20	0.03406920433044434
40	0.03439493179321289
60	0.03405852317810058
80	0.038832569122314455
90	0.03485007286071777

Cuadro 7: Algunos tiempos de ejecución. Caso donde $k > n^3$

Gracias a esto, pudimos confirmar nuestra sospecha: cuando $k > n^3$ los tiempos de ejecución dependen de la cantidad de palabras del diccionario y no del largo de la cadena, tardando prácticamente lo mismo para cualquier valor de n.

De esta manera pudimos aseverar que:

1. La complejidad analizada en el apartado anterior es correcta, $\mathcal{O}(\texttt{max}(k, n^3))$.
2. Podría incluirse una última optimización en la que se compare el largo del diccionario y el largo de la cadena al cubo para, en caso de que $k > n^3$, se omita la búsqueda de largos y simplemente opte por ejecutar el algoritmo en tiempo cúbico. Así, la complejidad en el peor caso sería siempre $\mathcal{O}(n^3)$.

Para finalizar con las mediciones, mostraremos el tiempo de ejecución para la reconstrucción del mensaje. En el análisis teórico de la complejidad para esta función, habíamos llegado a la conclusión de que era igual a $\mathcal{O}(n)$, n igual al tamaño de la cadena a reconstruir.

Cabe aclarar que esta medición se hizo con 650 `RUNS_PER_SIZE` para poder lograr un resultado observable.

Salta a la vista que el ajuste es precisamente lineal como se esperaba.

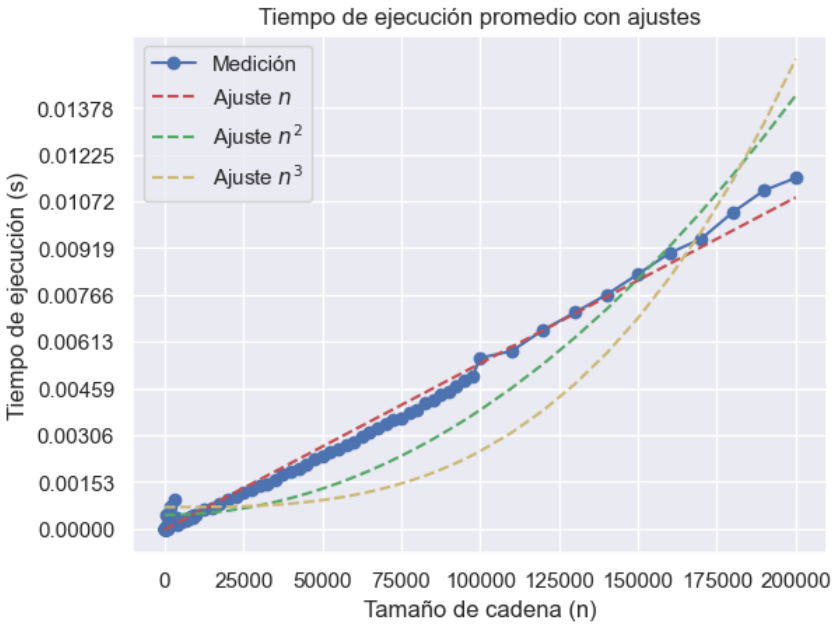


Figura 8: Tiempo de ejecución vs tamaño de la cadena. Algoritmo de reconstrucción, diccionario español de 1.000 palabras, set de cadenas con 200.000 caracteres máximo

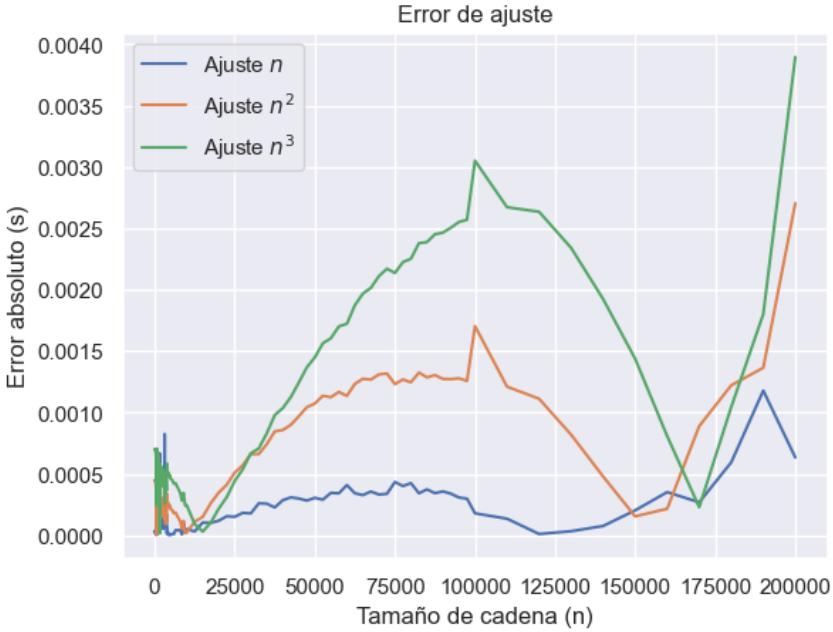


Figura 9: Error absoluto vs tamaño de entrada. Algoritmo de reconstrucción, diccionario español de 1.000 palabras, set de cadenas con 200.000 caracteres máximo

6. Conclusiones

Durante este trabajo hemos estudiado un problema que en principio podía resultar sencillo pero conforme fuimos descubriendo sus aristas, logramos identificar varios puntos de dificultad.

Para familiarizarnos con la solución esperada, analizamos algunos ejemplos prácticos y elaboramos una propuesta backtracking con la esperanza de sacar alguna conclusión antes de seguir.

Efectivamente, este *approach* nos sirvió para reconocer un comportamiento que queríamos evitar con programación dinámica: no volver a analizar subcadenas ya exploradas. Adoptamos entonces la noción de *memoización* y esbozamos un primer algoritmo con tintes de programación dinámica.

Con éste entendimos la necesidad de plantear primero una ecuación de recurrencia para luego desarrollar programa pero con un esquema *bottom-up*.

Una vez obtenida la ecuación y demostrada su efectividad para obtener el resultado correcto en todos los casos, implementamos el código esperado. No conformes con el desempeño temporal y espacial de nuestro programa, decidimos aplicar algunas optimizaciones.

Llegado el momento del análisis de la variabilidad y cómo afecta a los tiempos de ejecución, nos encontramos con varios aspectos a tener en cuenta.

En este problema hay varias variables que pueden influir y estudiarlas en simultáneo es bastante complejo. A pesar de eso, pudimos analizar cómo se podían combinar para sacar algunas conclusiones importantes:

- las optimizaciones realizadas fueron clave para mejorar los tiempos de ejecución ante ciertos escenarios
- en idiomas como el nuestro, el largo de las palabras puede resultar bastante chico y por lo tanto similar entre ellas. Este aspecto impacta considerablemente en problemas como el de este trabajo práctico
- ante un diccionario con palabras de largo con mucha variabilidad, el algoritmo optimizado se comporta como el peor caso posible
- la relación entre el tamaño del diccionario y el de las cadenas es sumamente relevante para el problema ya que su complejidad se describe a partir de ella
- el algoritmo de reconstrucción se ve afectado temporalmente únicamente por el largo de la cadena

Una vez más, consideramos que aplicar las técnicas de diseño vistas en clase en problemas de mayor complejidad nos ayudaron a lograr un entendimiento más profundo de los temas. Al ejercitar la búsqueda de la ecuación de recurrencia y la posterior demostración de su efectividad nos permitió elaborar un código preciso para el problema, seguros de lo que ya habíamos podido corroborar en los pasos anteriores. Pudimos ver con muchísima claridad cómo la variabilidad de los valores afectan a los tiempos de ejecución y lo importante que es observar todas las variables presentes.

7. Correcciones

7.1. Análisis de la complejidad temporal

Durante nuestro análisis previo, consideramos que el costo temporal de iterar el rango total que podría tener una palabra (dado por los valores obtenido anteriormente, denominados `largo_min` y `largo_max`) y formar subcadenas utilizando `slices` resulta ser $O(n^2)$ (sin contemplar la iteración externa que recorre todos los posibles inicios de la cadena). Sin embargo, pudimos notar que esta no era una cota correcta para el problema en cuestión.

Si bien es cierto que el largo máximo de una palabra a formar no puede superar a n, pues una palabra más extensa que la cadena nunca podría ser una solución válida, el valor l, correspondiente al largo máximo efectivamente posible, hallado con la utilización de la función `encontrar_largos_extremos`, resulta ser una cota sustancialmente mejor. Esto se puede observar claramente en gráficos como el de la página 18, donde un valor de l despreciable en comparación de n permite que el algoritmo se ejecute, en la práctica, con una dependencia casi lineal respecto del largo de la cadena.

Asimismo, y como ya mencionamos reiteradas veces, esta situación es recurrente al ejecutar el algoritmo con palabras pertenecientes a cualquier idioma moderno. Según **Guinness World Records**, la palabra más larga, entre todos los idiomas, tiene solo 195 letras y pertenece al sánscrito, hablado por muy pocas personas en la actualidad. De esta forma, nuestro algoritmo dejaría de depender cúbicamente de n para cualquier cadena mayor a este largo máximo y, considerando que los mensajes probablemente tengan largos ampliamente superiores, es evidente que nuestra conclusión inicial era errónea.

En consecuencia, la complejidad temporal mencionada al comienzo de esta sección presenta un costo de $O(l^2)$ e implica, por lo tanto, que la complejidad total resultante sea $O(k + nl^2)$ o, siendo más específicos, $O(max(k, nl^2))$.