.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Que parezca programación dinámica

6 de mayo de 2025

M. B. García Lapegna
111848

M. U. Sáenz Valiente
111960

T. Goncalves Rei
111405

Índice

1. Consideraciones

Este informe se distribuirá de la siguiente manera:

- Sección 2: se enunciara el problema a resolver así como también las condiciones en las que debe ser resuelto.
- Sección 3: se planteara una ecuación de recurrencia que permite resolver el problema algoritmicamente utilizando programación dinamica. Luego se explicaran los detalles sobre la implementación.
- Sección 4: en esta sección demostraremos que la ecuación de recurrencia aplicada permite determinar siempre de forma optima si el texto es valido o no dado un conjunto de palabras.
- Sección 5: se analizara la complejidad algorítmica de la solución. Para ello, se presentaran un conjunto de pruebas y mediciones realizadas al algoritmo.
- Por ultimo, se explayaran las conclusiones del trabajo presentado.

2. Introducción

En este informe se presentará un problema, y se propondrán y compararan diferentes formas de solucionarlo utilizando algoritmos de *Programación Dinamica*. En su desarrollo, se verán las ventajas y desventajas de cada solución. Dada la solución final, se demostrará que la ecuación de recurrencia es optima, y se explicara la complejidad del algoritmo haciendo uso de gráficos comparativos.

2.1. Enunciado

Para resolver este problema contamos con la siguiente información:

"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."

Con esta información, durante el informe buscaremos desglosar el problema hasta llegar a una ecuación de recurrencia que nos permita resolverlo de forma algoritmica, usando *Programación Dinamica*.

Se puede ver que el problema no solicita hacer una optimización, sino que solo se busca determinar si existe alguna posible solución. Por lo tanto, nuestros subproblemas (y tambien nuestro problema general) seran de decisión booleana. Explicaremos esto en detalle mas adelante.

2.2. Condiciones

Para resolver este problema, se tendran en cuenta las siguientes condiciones:

- El problema debe resolverse utilizando *Programacion Dinamica*
- 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

3. Resolución

En esta sección se presentara el algoritmo propuesto como solución.

3.1. Planteo del problema a resolver

Se nos otorga una *cadena desencriptada* y un listado de n palabras en el idioma español, nuestra tarea es determinar si la cadena es un posible mensaje. Decimos que una cadena es un mensaje si podemos separar los caracteres de manera tal que todas las palabras formadas pertenezcan al listado que nos dan.

3.2. Una primera idea

Teniendo en cuenta que para resolver el problema se pretende agrupar el texto en palabras validas, una primera idea podria ser pensar en los subproblemas como la cantidad de palabras que se pueden formar hasta el caracter i inclusive, e intentar maximizar esta cantidad. Esto puede ser tentador ya que funciona en algunos casos, pero no es optimo ya que no contempla necesariamente que la totalidad del texto sea valido. A continuación se presenta un contrajemplo.

- Palabras (P): 'uñass', 'ssol'
- Texto a comprobar: 'uñassol'

Si se sigue con la idea del subproblema antes propuesto, una posible ecuación de recurrencia que resuelva el problema sería seria la siguiente:

$$OPT(i) = max_{\forall k<i/[c_k,\ldots,c_i]\in P}(1 + OPT(k))$$

Si se aplica esta ecuación hasta resolver todos los subproblemas, la información que se guardaría en cada optimo seria insuficiente para poder reconstruir la secuencia de palabras que forman el texto. Incluso tampoco sería posible determinar si el texto puede ser valido si se insertaran espacios. Con esto, podemos ver que esta idea es incorrecta, y no permite solucionar el problema.

3.3. Planteo del algoritmo utilizando programación dinámica

Para resolver el problema anteriormente planteado, utilizaremos un arreglo al que llamaremos **M**, este nos servirá para aplicar la técnica de *memoizacion*, la cual se basa en ir almacenando los resultados de subproblemas ya resueltos para luego reutilizarlos para resolver un problema general, de esta manera no se recalculan cosas que previamente fueron calculadas.

Ya que nuestro enigma es saber si cierta cadena de caracteres es o no un posible mensaje, nuestro arreglo **M** entonces contendrá esta información. Es decir, sus valores variaran entre *True* y *False* haciendo alusión a si la cadena es o no es un mensaje.

Haremos que la solución a nuestro problema original (determinar si la cadena completa es o no un mensaje) se construya a partir de soluciones a subproblemas de este, es decir, mediante la composición de estos problemas mas pequeños (los cuales calcularemos previamente) podamos encontrar una solución a nuestro problema general. De esta manera, mediante *programacion dinamica* encontraremos la solución al problema general.

Nuestro **subproblema** sera determinar si hasta cierto caracter de la cadena se forma o no un mensaje. Esto lo haremos diciendo que el *indice+1* de nuestro arreglo **M** representara el *indice* de la cadena. Por lo tanto (y contando el caso base, el cual sera explicado posteriormente) el largo de nuestro arreglo solución sera igual al *largo de la cadena + 1*. De esta manera y con lo mencionado anteriormente, cada posición i del arreglo **M** representara si hasta el *indice* $i-1$ de la cadena, se forma o no un mensaje (True en caso de ser un mensaje y False en caso contrario).

De esta manera en el arreglo **M** tendremos *"memoizadas"* la solución de los subproblemas hasta un caracter determinado, teniendo asi en la ultima posición la solución general para la cadena, siendo este nuestro problema original.

El **caso base** de nuestro algoritmo sera que una cadena vacía **es un mensaje**.

Luego, siendo C la cadena dada a analizar y P el conjunto de palabras que nos otorgan, por cada carácter (cada subproblema) operaremos con la siguiente ***ecuación de recurrencia***:

$$M[i] = \ OR_{\forall k<i} \ (C[k+1:i] \in P \ AND \ M[k])$$

Lo que esta ecuación nos dice es que, para determinar si la cadena " C " es o no un mensaje hasta cierto carácter c_i, se recorren todos los caracteres de la cadena hasta el carácter $i-1$ (es decir, k ira tomando los valores desde 0 hasta $i-1$) y por cada una de estas iteraciones se evalúa si la union de los caracteres que van desde k hasta i inclusive forman un texto perteneciente al conjunto P.

De no ser así se sigue con la próxima iteración, ya que al tener un *AND*, que una de las expresiones sea falsa implica que la expresión general lo sea. En caso contrario, si la unión de estos caracteres formaran una palabra perteneciente al conjunto P, para que efectivamente sea un mensaje hasta el caracter i los caracteres previos a este caracter k(inclusive) también deberían formar un mensaje independientemente a este ultimo recientemente hallado. En caso contrario nuevamente la condición final también seria falsa. Esta información la tendremos *memoizada* en el arreglo M, ya que es la solución previamente calculada para el carácter k. De esta manera, en caso de que en alguna de las iteraciones(del indice i) se cumplan ambas condiciones entonces la cadena sera un mensaje hasta el carácter i, por lo que la información que se guardaria en nustro arreglo de M seria *True*.

3.4. Implementación

A continuación se mostrara el algoritmo de lo anteriormente presentado

```
1 def cadena_es_mensaje(palabras,mensaje):
2
3     if len(mensaje)==0 : return True , None
4     if len(palabras)==0 : return False , None
5
6     #Paso lista de palabras a set
```

```
    palabras = set(palabras)

    #Creacion del arreglo que se utilizara para la "memoizacion"

    M = [False] * (len(mensaje)+1)

    #Caso base -> cadena vacia
    M[0] = True

    for indice in range(1,len(mensaje)+1): # Recorre caracter por caracter la
    cadena
        es_palabra = False

        for indice_anterior in range(indice): # Recorre desde el inicio de la
    cadena hasta indice - 1

            palabra = []

            for i in range(indice_anterior, indice):
                palabra.append(mensaje[i])
            palabra = "".join(palabra)

            if palabra in palabras: #si la palabra pertenece al conjunto
                anterior_es_mensaje= M[indice_anterior]
                if anterior_es_mensaje: # si los caracteres previos tambien forman
    un mensaje
                    es_palabra = True
                    break
        M[indice] = es_palabra

    if M[len(mensaje)] :
        return True , reconstruccion_mensaje(mensaje,M)

    return False , None
```

Como se puede observar, viendo la ultima posición del arreglo M se puede saber si la cadena es o no un mensaje. En base a esto, se retorna False(si no lo es) o True junto con la reconstrucción del mensaje que forma (detallado en la **Sección 3.4**).

Como se lo menciono previamente en el primer ciclo *for* se recorre carácter por carácter a la cadena, ademas internamente se iteran todos los caracteres previos a este. Dentro de esta iteración se toma la unión previamente descrita y se corrobora si esta efectivamente forma parte del conjunto de palabras (lista que nos pasan por parámetro (*1*)), de ser así (recordemos que el arreglo **M** guarda información sobre si es o no un mensaje hasta el carácter), ademas de ser una palabra perteneciente al conjunto, los caracteres previos a esta palabra también deben formar un mensaje. Si se cumple esto obtendríamos un *True*, y como planteamos en nuestra ecuación de recurrencia, si en una iteración logramos obtener un *True*, significa que también hay un mensaje valido hasta ese carácter, por lo que, nos guardamos esa información en el arreglo **M**. En caso de que no se cumpla lo antes descripto se guardaria un *False* para ese indice.

(*1*) Se paso el listado de palabras a un *set de datos* para que los checkeos de pertenencia al conjunto sean una operacion constante, se desarrollara mas sobre esto en la **Seccion 5.5.1** .

3.5. Posible optimización

Si bien el planteo de la sección anterior era optimo (se demostrara mas adelante) y funcionaba en la mayoria de casos, hay escenarios en los que su rendimiento empeora drasticamente. Por ejemplo, si un texto cuenta con una gran cantidad de caracteres, por cada uno de ellos deberan recorrerse todos los demas. Luego, cada vez que se recorre un caracter anterior, se debera formar la posible palabra para comprobar si esta existe. Es decir, para un caracter n, se deberan recorrer sus $n-1$ caracteres anteriores y con ellos se formaran $n-1$ palabras distintas.

Una posible solución a este problema es la siguiente: sabemos, por ejemplo, que la palabra mas larga del español es .[el]ectroencefalografista", con 23 letras. Esto nos da pie a estar seguros (al menos

para textos en español) de que luego de recorrer 23 caracteres anteriores sin encontrar una palabra posible, no habra otra palabra posible por encontrar, aunque sigamos recorriendo. Si generalizamos esta idea, podemos decir que si r es el largo maximo de palabra para un set de palabras posibles, entonces no hara falta recorrer mas de r letras en cada paso para buscar una posible palabra. Esto supone una gran ventaja a la versión anterior, ya que la cantidad de iteraciónes por cada letra no dependera del largo del texto posible, como si pasaba antes. En este nuevo escenario, para un caracter n, se deberan recorrer solo sus r caracteres anteriores (de haberlos) y formar con ellos r palabras distintas. Ademas, por ser r el largo de una palabra, podra considerarse despreciable para textos muy extensos. En la **sección 5** se explayara sobre la dimensión de estas mejoras en diferentes escenarios.

Con estos nuevos cambios, la ecuación de recurrencia tendra una leve modificación, que impactara sobre todo en los costes de tiempo (no asi en el funcionamiento del algoritmo como tal).

$$M[i] = OR_{\forall k \in [i-r:i]} \ (C[k+1:i] \in P \ AND \ M[k]),$$
$$* \text{ Si } i - r < 0, \text{ entonces } k \in [0, i]$$

A continuación se presentara el algoritmo con esta nueva mejora final. Se obviaran las explicaciónes ya que los cambios respecto a la version anterior son leves, y refiere a lo explicado antes en esta sección.

```
def cadena_es_mensaje(lista_palabras,mensaje):
    """
    palabras : listado de palabras posibles
    mensaje  : cadena que se evalua
    """

    if len(mensaje)==0 : return True , None
    if len(lista_palabras)==0 : return False , None

    #Se pasa lista de palabras a set y se guarda la longitud de la mas larga
    palabras = set()
    longitud_max=0
    for p in lista_palabras:
        palabras.add(p)
        longitud_p =len(p)
        if longitud_p > longitud_max:
            longitud_max = longitud_p

    #Creo el arreglo que se utilizara para la "memoizacion"
    M = [False] * (len(mensaje)+1)

    #Caso base -> cadena vacia
    M[0] = True

    #Recorremos cadena
    for indice in range(1,len(mensaje)+1):
        es_palabra = False
        indice_anterior_desde = max(0, indice-longitud_max)

        for indice_anterior in range(indice_anterior_desde, indice):
            palabra = []

            for i in range(indice_anterior, indice):
                palabra.append(mensaje[i])
            palabra = "".join(palabra)

            if palabra in palabras:
                anterior_es_mensaje= M[indice_anterior]
                if anterior_es_mensaje:
                    es_palabra = True
                    break
        M[indice] = es_palabra

    if M[len(mensaje)] :
        return True , reconstruccion_mensaje(mensaje,M,palabras)
```

```
46
47      return False , None
```

3.6. ¿Por que es Programación Dinamica?

Para explicar porque nuestra ecuación de recurrencia permite resolver el problema usando programación dinamica, primero referiremos a una definición certera:

'*...In a way, we can thus view dynamic programming as operating dangerously close to the edge of brute-force search: although it's systematically working through the exponentially large set of possible solutions to the problem, it does this without ever examining them all explicitly*' - Algorithm Design, p.251

Para buscar una solución, entonces, podriamos pensar primero en cuales decisiones deben tomarse en cada paso, y cuales son las opciones para cada decisión. Por las caracteristicas del problema que estamos intentando resolver, se puede ver que la decisión sera de tipo booleana. En este caso, la decisión sera determinar si existe un texto valido hasta el caracter i. Para esta decisión, las opciones estaran dadas por la ecuación de recurrencia: para cada caracter anterior se tiene la opción de empezar una palabra nueva a partir de ese caracter $k < i$. Es decir, para el caracter i, existen $i - 1$ opciones que refieren a cada caracter donde puede empezar la ultima palabra del texto.

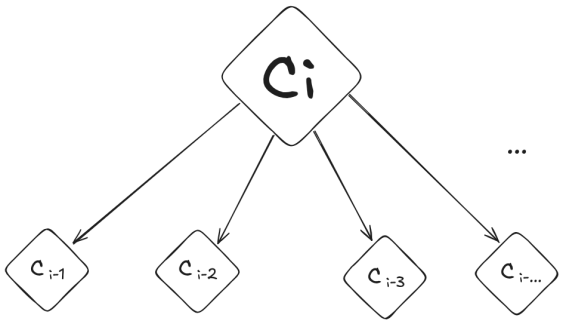


Figura 1: Árbol de decisiones de cada subproblema i

Si se pretendiera recorrer cada una de las soluciones explicitamente, la complejidad de nuestro algoritmo creceria rapidamente hasta llegar a una complejidad exponencial. La principal causa de que la complejidad crezca es la alta cantidad de calculos repetidos que se hacen para resolver un subproblema (los subproblemas anteriores ya hicieron los calculos necesarios para los subproblemas anteriores a ellos) Esto puede optimizarse usando *memoización*: si cada subproblema se resuelve en base a subproblemas anteriores, entonces podemos *memoizarlos* para tenerlos disponibles en la resolución de subproblemas proximos. Es por eso que en nuestro algoritmo cada resultado se guarda en un arreglo **M**. Usando esta tecnica, ver el resultado de cualquier subproblema anterior tiene complejidad constante.

Como se menciono antes, nuestra ecuación de recurrencia es la siguiente:

$$M[i] = \ OR_{\forall k \in [i-r:i]} \ (C[k+1:i] \in P \ AND \ M[k]),$$
$$\text{* Si } i - r < 0, \text{ entonces } k \in [0, i]$$

Se puede ver con esto que cada subproblema se resuelve gradualmente utilizando soluciones a subproblemas anteriores, que estaran *memoizados* en el arreglo **M**. Por lo tanto, nuestro algoritmo es de *Programación Dinamica*

3.7. Reconstrucción

Como se lo menciono previamente, en caso de que efectivamente la cadena en general sea un mensaje, debemos reconstruirlo, es decir, si hay un mensaje debemos decir cual es.

```
def reconstruccion_mensaje(mensaje,M,conjunto_palabras):
    palabras = []
    indice = len(M)-1

    palabraActual = []
    while indice != 0:
        palabraActual.append(mensaje[indice-1])
        indice = hallar_inicio(M,indice,palabraActual,mensaje)

        palabraActual.reverse()
        palabra = "".join(palabraActual)

        if palabra in conjunto_palabras:
            palabras.append(palabra)
            palabraActual = []
        else:
            palabraActual.reverse()

    palabras.reverse()
    mensaje = " ".join(palabras)
    return mensaje

def hallar_inicio(M,finActual,palabraActual,mensaje):
    for indice in range(finActual-1,-1,-1):
        if M[indice] == True :
            return indice
        palabraActual.append(mensaje[indice-1])
```

La idea del algoritmo es muy sencilla, recorre el arreglo **M** de atrás hacia adelante, guardándose los caracteres de la cadena (utilizando y actualizando el indice a medida que va iterando sobre el arreglo M), asi hasta encontrar un *True* en el arreglo **M**, cuando lo halla significa que hasta ese indice había un mensaje valido, por lo que se analiza si la palabra formada con los caracteres hasta el momento es una palabra perteneciente al conjunto, de ser así se la agrega al listado de palabras. En caso contrario, se sigue iterando y sumando caracteres a los previos hasta formar una palabra perteneciente al conjunto. Finalmente se unen en un mensaje todas las palabras encontradas.

4. Demostración

En esta seccion demostraremos que nuestra ecuación de recurrencia determina correctamente si un texto es valido dado un conjunto de palabras determinado. Es decir, puede obtenerse ese mismo texto uniendo palabras $p_i \in P$. Demostraremos la idea inicial del algoritmo, ya que la segunda versión es solo una optimización y no supone un cambio en el funcionamiento ni en como se determina la solución.

4.1. Consideraciones

En adelante, consideraremos:

- $[c_1, c_2, ..., c_n]$ = secuencia (palabra o texto) formada por los caracteres $c_1, c_2, ..., c_n$.
- $[A - Z]$ = Conjunto de letras del abecedario español.
- $OPT(0) = True$. Los demas indices $i, i > 0$ corresponderan al caracter i del texto a comprobar (contando desde 1 a n).

4.2. Hipótesis

Suponemos que:

- $\exists$ n palabras p_i, con $p_i \in P = \{p_1, p_2, ..., p_n\}$, $p_i = [c_1 ... , c_k]$, $i \in [0, n]$.
- $\exists$ m caracteres que forman una cadena desencriptada / c_i es un caracter $\in C = [c_1, c_2, ..., c_m]$, $c_i \in [A - Z]$
- Para los fines de esta demostración reformularemos nuestra ecuación de recurrencia de la siguiente manera:

$$OPT(i) = \bigvee_{k=0}^{i-1} [c_{k+1}, ..., c_i] \in P \wedge OPT(k)$$

4.3. Tesis

Nuestra ecuación de recurrencia determina correctamente si existe un texto valido formado por todos los caracteres de un texto hasta i inclusive. Es decir,

$$OPT(i) = True \iff [c_1, c_2, ..., c_i] \text{ es un texto valido}$$

Para ello demostraremos por metodo inductivo:

- **Caso base:** $OPT(1) = True \iff [c1]$ es un texto valido **(sección 4.4.1)**
- **Paso inductivo:** Teniendo $OPT(1), OPT(2), ..., OPT(k)$:

$$OPT(k+1) = True \iff [c_1, c_2, ..., c_{k+1}] \text{ es un texto valido } \textbf{(sección 4.4.2)}$$

4.4. Demostración

4.4.1. Caso base

Se tiene c_1.

- Si $[c_1]$ es un texto valido:
  $\Rightarrow [c_1] \in P$ y ademas $OPT(0) = True$
  $\Rightarrow$ por la ecuación de recurrencia, $OPT(1) = (True \wedge True) = True$.
- Si $[c_1]$ no es un texto valido:
  $\Rightarrow [c_1] \notin P$ y ademas $OPT(0) = True$
  $\Rightarrow$ por la ecuación de recurrencia, $OPT(1) = (False \wedge True) =$ False

Queda demostrado entonces que $OPT(1)$ determina correctamente si existe una texto valido hasta el caracter 1 inclusive.

4.4.2. Paso inductivo

Tenemos $OPT(1), OPT(2), ..., OPT(K)$

- Si existe una secuencia valida hasta $k+1$ inclusive.
  $\Rightarrow [c_1, ..., c_k + 1]$ es una secuencia valida
  $\Rightarrow \exists j < k+1$ / $[c_1, ..., c_j]$ es una secuencia valida y ademas $[c_{j+1}, ..., c_{k+1}] = p_i \in P$
  $\Rightarrow [c_1, ..., c_j]$ es una secuencia valida $\Rightarrow OPT(j) = True$
  $\Rightarrow OPT(j) = True \wedge [c_{j+1}, ..., c_{k+1}]$ es una secuencia valida.
  $\Rightarrow$ por la ecuación de recurrencia, $OPT(K+1) = (True \wedge True) = True$
- Por otro lado, si no existe una secuencia valida hasta el indice $k+1$ inclusive.
  $\Rightarrow \nexists j$ / $[c_1, ..., c_j]$ sea una secuencia valida y $[c_{j+1}, ..., c_{k+1}] \in P$
  $\Rightarrow \nexists j$ / $OPT(j) = True \wedge [c_{j+1}, ..., c_{k+1}] \in P$
  $\Rightarrow$ por la ecuación de recurrencia, $OPT(K+1) = (False \wedge False/True) = False$

Demostramos así que dados los optimos hasta el indice k, se determinara correctamente el resultado para $k+1$.

$\therefore$ Concluimos que nuestra ecuación de recurrencia determina correctamente si un texto es valido dado un conjunto de palabras con el que puede estar formado.

5. Mediciones

Se realizaron mediciones para así poder determinar el tiempo real de ejecución del algoritmo con distintos tamaños de entrada. Con esto, mas adelante se analizó su complejidad temporal.

Estas mediciones ayudan a visualizar como se comporta el algoritmo.

5.1. Complejidad

Para poder analizar la complejidad del algoritmo, desglosaremos el código en distintas secciones.

Definimos :

- m a la cantidad de palabras.
- n a la cantidad de caracteres de la cadena.

5.1.1. Complejidad construcción arreglo de memoizacion

Como recibimos las palabras en una *lista*, si lo dejáramos así, ver si la unión de ciertos caracteres en una iteración es efectivamente una de las palabras de esta lista, nos obligaría a en el peor de los casos tener que recorrer la lista por completo. Esto seria $O(m)$ por cada unión de caracteres que probemos. Por lo tanto se opto por hacer lo siguiente:

```
1 palabras = set(palabras)
```

Al pasar el listado de palabras a un *set* logramos reducir todas estas operaciones lineales (que se harían por cada carácter y por cada uno con los caracteres previos a este) a que sean operaciones constantes.

Convertir la lista en un set de datos nos costara $O(m)$, pero luego cuando dentro de las iteraciones se quiera corroborar si la unión de ciertos caracteres pertenece no al conjunto de palabras nos costara $O(1)$. En síntesis, esta optimización nos permitirá abaratar operaciones que nos podrían

llegar a costar $O(m)$ cada una. Utilizando un set, el $O(m)$ se convierte en una operacion constante. Es decir, la unica vez que "se recorrera"la lista sera para pasarla a un set al principio (pero sera la primera y única vez) que nos costara $O(m)$.

Luego construir nuestro arreglo de *"memoizacion"* **M**,

```
M = [False] * (len(mensaje)+1)
```

nos costara $O(n+1)$ lo que es igual a $O(n)$.

Ahora bien, ir construyendo nuestra solución sobre el arreglo **M** sera ir caracter por caracter de la cadena, y por cada caracter recorrer sus r caracteres anteriores, siendo r el largo de la palabra mas larga. Por cada digito que retrocedemos al recorrer los r anteriores, se debe crear una nueva palabra con esos digitos. Hacer esto ultimo tiene complejidad $O(r^2)$ en total.

Finalmente, el proceso de recorrer los r anteriores y formar la palabra se hara por cada uno de los n caracteres. Por ello, la construcción de la solución tendra una complejidad de $O(n \times r^2)$.

En las siguientes secciones compararemos las complejidades de la primera versión del algoritmo y su versión optimizada, donde podrá verse que la mejora es notória.

5.1.2. Complejidad del armado del arreglo M en la versión inicial

- Pasar la lista de palabras a un set de palabras: $O(m)$
- Construir el arreglo de *memoizacion*: $O(n)$
- Recorrer los caracteres de la cadena y recorrer caracteres anterioreres al actual : $O(n \times (n-1))$
- Formar nueva palabra para cada caracter anterior, desde ese caracter hasta el actual: $O(n)$ (peor caso)

Lo cual nos lleva a que la complejidad del armado de nuestro arreglo solución **M** sea :

$$\Rightarrow O(m) + O(n) + O(n \times (n-1)) \times O(n)$$
$$\Rightarrow O(m) + O(n) + O(n^3)$$
$$\Rightarrow O(m + n + n^3)$$

En este caso n^3 funciona como cota superior para n, por lo que:

$$\Rightarrow O(m + n^3)$$

Por lo tanto, la complejidad total del armado del arreglo **M** es : $O(m + n^3)$

5.1.3. Complejidad del armado del arreglo M con la optimización

Como vimos antes, saber el largo de la palabra mas larga de nuestro set nos permitia agregar una gran optimización para el coste del algoritmo. En este caso, si definimos r como ese largo maximo, nuestra complejidad se transforma de la siguiente manera:

- Pasar la lista de palabras a un set de palabras y buscar el largo de la palabra mas larga: $O(m)$
- Construir el arreglo de *memoizacion*: $O(n)$
- Recorrer los caracteres de la cadena y recorrer sus r caracteres anteriores (como maximo): $O(n \times r)$
- Formar nueva palabra para cada caracter anterior, desde ese caracter hasta el actual: $O(r)$

Lo cual nos lleva a que la complejidad del armado de nuestro arreglo solución **M** sea :

$$\Rightarrow O(m) + O(n) + O(n \times r) \times O(r)$$
$$\Rightarrow O(m) + O(n) + O(n \times r^2)$$
$$\Rightarrow O(m + n + n \times r^2)$$

En este caso $n \times r^2$ funciona como cota superior para n, por lo que:

$$\Rightarrow O(m + n \times r^2)$$

Como dijimos antes, para el idioma español r sera un valor pequeño (podemos acotarlo superiormente por 30), y por lo tanto r^2 tambien lo sera. Por lo tanto, para textos muy extensos, podremos considerar que r es despreciable frente a n, y por lo tanto nuestra complejidad sera la siguiente:

- Para el caso general, $T(n) = O(m + n * r^2)$.
- Para el idioma español, $T(n) = O(m+n)$ por ser r despreciable. Aplica tambien para cualquier idioma y texto donde $n > r^2$.

5.2. Complejidad Reconstrucción

En el caso de que la cadena forme un mensaje se lo debe reconstruir . A continuación, se analizara la complejidad de reconstruirlo.

Para esto, debemos volver a recorrer el arreglo solución **M**, lo cual nuevamente tendrá una complejidad $O(n)$. Internamente dentro de esta iteración se busca volver a formar la palabra que previamente hizo que el mensaje como tal sea valido, para esto cada vez que en nuestro arreglo tengamos un indice con solución igual a *True* debemos corroborar que los caracteres hasta el momento sean una palabra perteneciente al conjunto. Para esto :

```
palabraActual.reverse()
palabra = "".join(palabraActual)
```

damos vuelta los caracteres obtenidos hasta el momento(recordar que recorremos desde el hacia el inicio), lo cual tiene una complejidad de $O(k)$ siendo k la cantidad de caracteres en la palabra actual. Si definimos r como el largo de la palabra mas larga, podemos decir que este procedimiento es $O(k)$. En adelante, acotaremos superiormente $O(k)$ por $O(r)$.

Luego de dar vuelta los caracteres, se las une utilizando un *join()* lo cual también cuesta $O(r)$, por lo que , en conjunto seria $O(r) + O(r) = O(r + r) = O(2r) = O(r)$.

Luego de esto, si la unión anterior es efectivamente una palabra, se la agrega a una *lista de palabras* lo cual es una operación constante. Pero en caso contrario:

```
palabraActual.reverse()
```

se vuelve a hacer una operación $O(r)$. Esta operación se hara como maximo por cada uno de los r caracteres anteriores que se recorreran, con lo cual tendra una complejidad de $O(r^2)$.

Entonces, hasta el momento en el peor de los casos realizamos r^2 operaciones por cada carácter de la cadena, lo cual nos cuesta $O(n \times r^2)$.

Finalmente, una vez que se recorrieron todos los caracteres tendremos una lista de palabras que forman el mensaje, pero por lo mencionado anteriormente esta lista también estará al revés, por lo que:

```
palabras.reverse()
mensaje = " ".join(palabras)
```

nuevamente debemos dar vuelta las palabras, que nos costara $O(n)$ (esto ya que justamente las palabras formadas son la cantidad de caracteres de la cadena original), luego unirlas para formar el mensaje nos costara lo mismo. De manera tal que, la complejidad de este fragmento de código es $O(n) + O(n) = O(n + n) = O(2n) = O(n)$.

5.2.1. Complejidad Total de la Reconstrucción

- Recorrer carácter por carácter la cadena y realizar las operaciones previamente descritas para saber si es o no una palabra perteneciente al conjunto: $O(n \times k) = (n \times r)$
- : Construir el mensaje con palabras halladas y armadas: $O(n)$

Por lo que la complejidad total de la reconstrucción es:

$$\Rightarrow O(n \times r^2) + O(n)$$

$O(n \times r^2)$ actua de cota superior para $O(n)$, por lo que la complejidad finalmente es : $O(n \times r^2)$

5.3. Complejidad del algoritmo en su totalidad

- Si la cadena **no es un mensaje**: en este caso lo único que se hace es construir el arreglo solución, por lo que la complejidad total sera $O(m + n \times r^2)$
- Si la cadena **es un mensaje**:
  - Construir el arreglo solución: $O(m + n * r^2)$
  - Reconstruir el mensaje : $O(n \times r^2)$

  Por lo que la complejidad **total** sera:

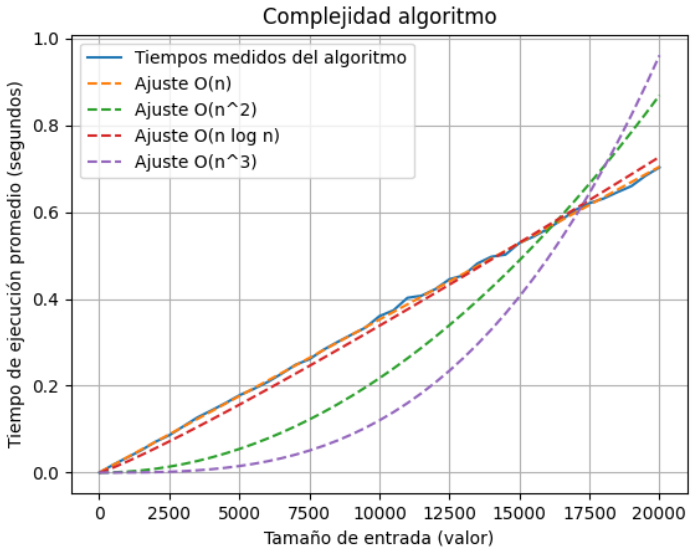
$$\Rightarrow O(m + n \times r^2) + O(n \times r^2)$$
$$\Rightarrow O(m + n \times r^2 + n \times r^2)$$
$$\Rightarrow O(m + 2(n \times r^2))$$

Por lo que la complejidad finalmente sera: $O(m + n \times r^2)$

$\therefore$ El algoritmo tiene complejidad teórica final $O(m + n \times r^2)$ en todos los casos.

5.4. Justificación de complejidad con gráficos

A continuación se realizaran gráficos comparativos entre el algoritmo planteado y las funciones $O(n)$, $O(n \times log(n))$, $O(n^2)$ y $O(n^3)$. Los datos utilizados para la gráfica son pruebas que se realizaron con distintos sets de datos.

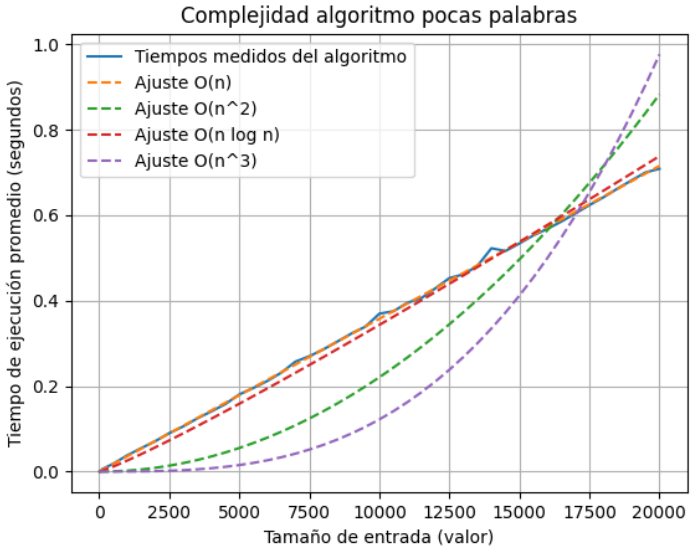


En la imagen podemos ver como nuestro algoritmo se ajusta mejor a la recta lineal. Recordemos que se recorre el largo de la cadena, y por cada uno de los caracteres de esta se itera el largo de la palabra mas larga, pero en el idioma español el largo de una palabra se hara despreciable en comparación con el largo de la cadena. Por esta razón las iteraciones quedan acotadas y el comportamiento del algoritmo tiende a ser lineal, es decir, $O(n)$ siendo n la cantidad de caracteres de la cadena. En otras palabras, la longitud de la palabra mas larga sera constante durante la iteración de los caracteres de la cadena, por lo que en la practica el algoritmo parece ser lineal. Como aumentamos la cantidad de entradas de mensaje pero dejamos fija la lista de palabras, osea una longitud constante, es esperable que los tiempos se ajusten como se muestra en el gráfico.

5.5. Análisis según escenario

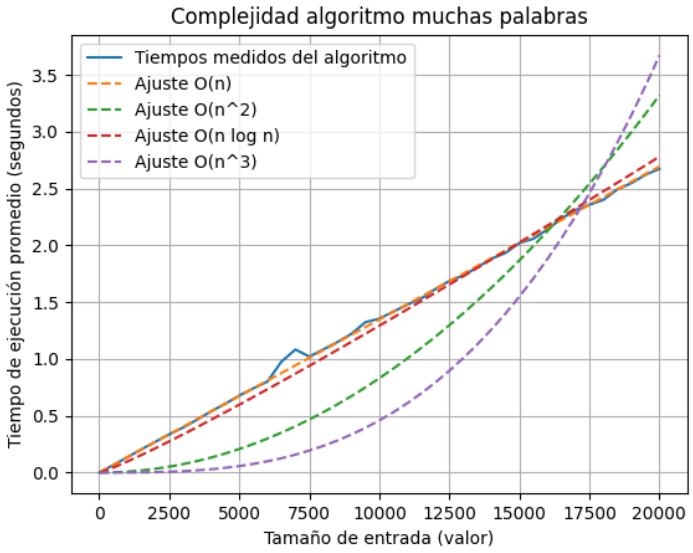
En este apartado, se detallara como se comporta el algoritmo a través de distintos set de pruebas, variando sus entradas.

5.5.1. Textos con menor cantidad de palabras



Se puede observar como en este escenario el algoritmo se comporta de igual manera que en el gráfico anterior pese a que se cambio el set de palabras.

5.5.2. Textos con mayor cantidad de palabras



Al igual que en el gráfico anterior, podemos notar que tampoco se ven variaciones significativas.

De esta manera podemos concluir que si el factor es la cantidad de palabras, el algoritmo se comporta de igual manera. Esto tiene sentido ya que al ponerlas en un set de datos independientemente de la cantidad de palabras siempre serán constantes los chequeos de pertenencia.

5.6. Error Cuadrático Medio

El ECM(error cuadrático medio) es una medida que permite acotar el error cometido al aproximar una serie de puntos mediante una función especifica. En general, el error cuadrático se define de la siguiente manera:

Dado un conjunto de puntos del tipo (x_i, y_i):

$$ECM = \sum_{i=0}^{k} r_i, r_i = (y_i - f(x_i))^2 \tag{1}$$

En nuestro algoritmo, vamos a calcular el error medio al ajustar los puntos resultantes de nuestras mediciones (véase gráfico de la sección 5.3) a funciones correspondientes a las complejidades $O(n \times log(n))$ y $O(n^2)$. Estas funciones tienen la siguiente forma:

Función $O(n^3) \implies f(n) = a * n^3, a \in \mathbb{R}^+$

Función $O(n^2) \implies f(n) = a * n^2, a \in \mathbb{R}^+$

Función $O(n \times log(n)) \implies f(n) = a * n * log(n), a \in \mathbb{R}^+$

Función $O(n) \implies f(n) = a * n, a \in \mathbb{R}^+$

Realizando el calculo del ECM con estas dos funciones, llegamos a los siguientes resultados:

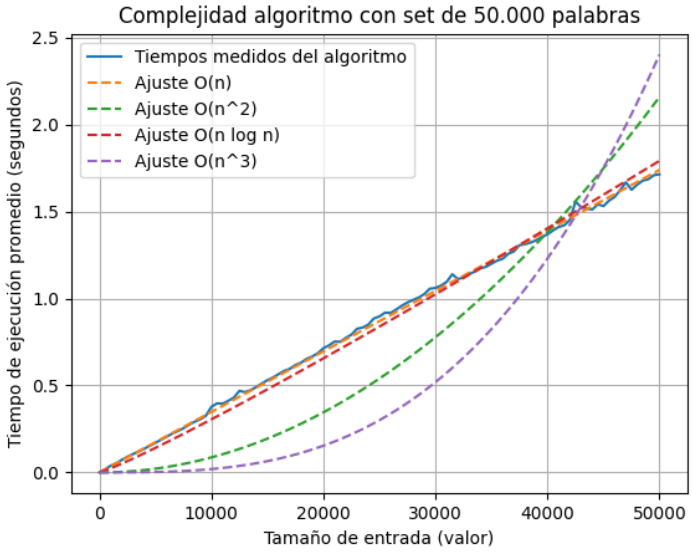
- Con función cubica: ECM con $O(n^3) = 2{,}432 \times 10^{-2}$
- Con función cuadrática: ECM con $O(n^2) = 9{,}604 \times 10^{-3}$
- Con función lineal-logarítmica: ECM con $O(n \times log(n)) = 2{,}274 \times 10^{-4}$
- Con función lineal: ECM con $O(n) = 1{,}621 \times 10^{-5}$

De esta manera se comprueba que la función que mejor ajusta al rendimiento de nuestro algoritmo es la función correspondiente a la complejidad $O(n)$.

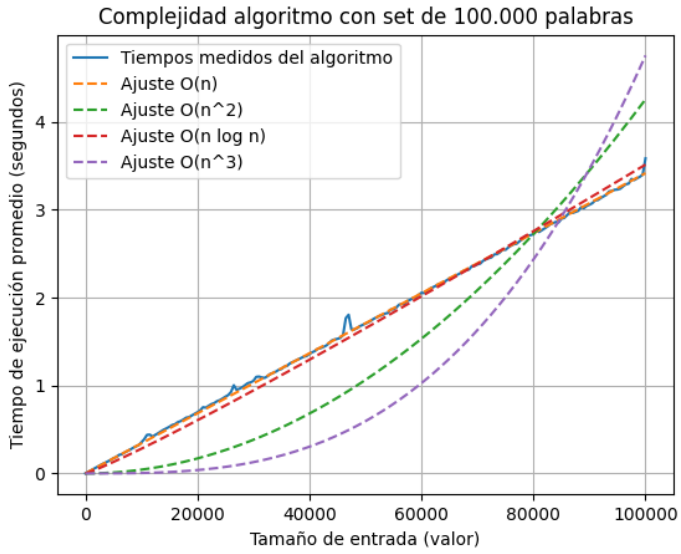
5.7. Complejidad algorítmica según volumen

En este apartado se mostraran los gráficos correspondientes a la ejecución del algoritmo con sets de mayor rango, para asi poder observar con mayor exactitud a que tiende su comportamiento.

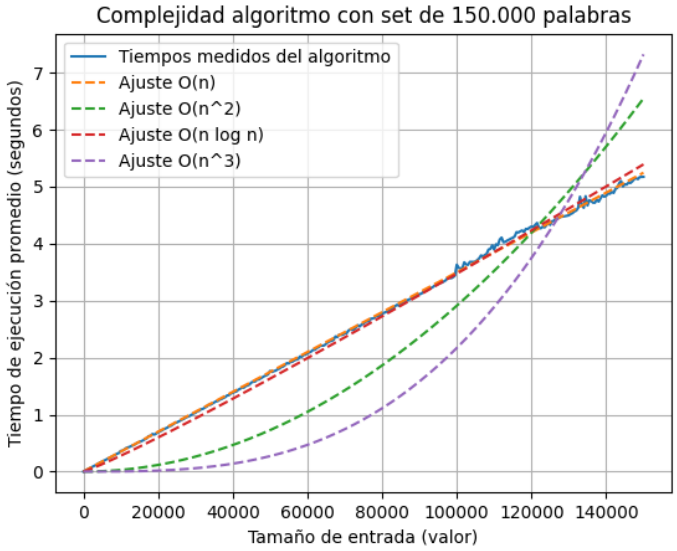
5.7.1. Comportamiento con set de 50.000 palabras



5.7.2. Comportamiento con set de 100.000 palabras



5.7.3. Comportamiento con set de 150.000 palabras



Como se puede observar en los distintos gráficos, comprobamos que la cantidad de entradas de palabras de los textos, no influye directamente sobre la complejidad del algoritmo, es decir, este se comporta de igual manera sin importar el largo de la cadena. El mismo mantiene un crecimiento del tiempo de ejecución controlado y cercano a la función lineal. Con esto podemos concluir que, mas alla de la complejidad teórica, el desempeño de nuestro algoritmo en lo practico resulta altamente eficiente para volúmenes grandes de texto.

6. Conclusión

En este informe pudimos abordar el uso de *Programación Dinamica* para resolver un problema determinado, y observamos como esta tecnica permite optimizar los tiempos de ejecución frente a otras tecnicas en las que se debe explorar un espacio de soluciones amplio para encontrar una solución valida.

Una vez planteada una ecuación de recurrencia, implementamos un algoritmo que permitio resolver el problema con una complejidad polinomica. Pudimos ver como el algoritmo podia deducirse a partir de la ecuación de recurrencia, lo cual supuso un gran punto a favor de la *Programación Dinamica* como tecnica de diseño de algoritmos.

Luego, demostramos por metodo inductivo que esta ecuación de recurrencia permitía encontrar la solución en cualquier caso, pudiendo determinar si el texto era valido o no correctamente.

Dada la versión final del algoritmo, se analizo tanto su comportamiento como su rendimiento en diferentes escenarios y volúmenes, haciendo uso de diferentes graficos, que fueron analizados en detalle.