.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Algoritmo Programación Dinámica

5 de mayo de 2025

Nicolás Martín Guerrero 112514

1. Introducción del Trabajo

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

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

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

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

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

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

1.1. Consigna del Problema

1. Hacer un análisis del problema, plantear la ecuación de recurrencia correspondiente y proponer un algoritmo por programación dinámica que obtenga la solución óptima al problema planteado: Dado el listado n de palabras, y una cadena desencriptada, determinar si es un posible mensaje (es decir, se lo puede separar en palabras del idioma), o no. Si es posible, determinar cómo sería el mensaje.

2. Demostrar que la ecuación de recurrencia planteada en el punto anterior en efecto nos asegura encontrar una solución, si es que la hay (y si no la hay, lo detecta).

3. Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Considerar analizar por separado cada uno de los algoritmos que se implementen (programación dinámica y reconstrucción), y luego llegar a una conclusión final.

4. Analizar si (y cómo) afecta a los tiempos del algoritmo planteado la variabilidad de los valores (mensajes, palabras del idioma, largos de las palabras, etc.).

5. Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado. Adicionalmente, el curso proveerá con algunos casos particulares para que puedan usar inicialmente para validar.

6. Hacer mediciones de tiempos para corroborar la complejidad teórica indicada. Agregar los casos de prueba necesarios para dicha corroboración (generando sus propios sets de datos).

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

2. Análisis del Problema

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

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

3. Elección del Algoritmo

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

Consideramos que en cada posición i de la cadena existe un subproblema que consiste en determinar si es posible segmentar el inicio de una palabra a partir de dicha posición. Por lo tanto, se plantea una ecuación de recurrencia que intenta formar una palabra válida $str[i:j]$ (que pertenezca al diccionario) y continuar recursivamente con el resto de la cadena,a partir de la posición j.

De forma matematica:

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

Esta ecuación se utiliza porque modela con precisión la estrategia necesaria para resolver el problema: desde cada posición se prueban todos los posibles segmentos iniciales que formen palabras válidas, y por cada uno de ellos se intenta segmentar el resto de la cadena. Si en algún punto se alcanza exitosamente el final de la cadena, significa que se ha encontrado una segmentación completa válida.

Este planteo asegura que, si existe una forma de segmentar la cadena utilizando únicamente palabras del diccionario, el algoritmo la encontrará. Además, como se almacenan los resultados intermedios de cada posición i mediante memoización, se evita recalcular soluciones ya halladas, logrando así una implementación eficiente.

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

```
for j in range(i + 1, len(s) + 1):
    palabra = s[i:j]
    if palabra in diccionario:
        resto = segmentar_texto(s, diccionario, memo, j)
        if resto is not None:
            memo[i] = palabra if resto == "" else palabra + " " + resto
            return memo[i]
```

4. Complejidad del Algoritmo

4.1. Explicacion de la Complejidad

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

```
    def segmentar_texto(s, diccionario, memo, i):
    if i in memo:
        return memo[i]
    if i == len(s):
        return ""

    for j in range(i + 1, len(s) + 1):
        palabra = s[i:j]
        if palabra in diccionario:
            resto = segmentar_texto(s, diccionario, memo, j)
            if resto is not None:
                memo[i] = palabra if resto == "" else palabra + " " + resto
                return memo[i]

    memo[i] = None
    return None

```

La función principal de segmentación implementa una estrategia de programación dinámica con memoización para evitar resolver múltiples veces los mismos subproblemas. La función recursiva $segmentar_texto$ recibe como entrada la cadena, el diccionario, una memoria (*memo*) y la posición actual i desde donde se intenta segmentar.

El bucle principal examina todos los posibles cortes de la subcadena que comienza en i, evaluando si forman una palabra válida. Este comportamiento se ve reflejado en el siguiente fragmento:

```
for j in range(i + 1, len(s) + 1):
    palabra = s[i:j]
    if palabra in diccionario:
        resto = segmentar_texto(s, diccionario, memo, j)
```

Este bucle interno puede ejecutarse hasta n veces en cada llamada (donde n es la longitud de la cadena), por lo que tiene una complejidad $O(n)$ por llamada.

Gracias al uso de memoización:

```
if i in memo:
    return memo[i]
```

cada subproblema para una posición i se resuelve solo una vez, y su resultado se guarda. Como hay a lo sumo n posiciones diferentes en la cadena, la complejidad total del algoritmo de segmentación es:

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

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

```
memo[i] = palabra if resto == "" else palabra + " " + resto
return memo[i]
```

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

Dado que el algoritmo recorre todas las posiciones posibles i de la cadena, y en cada una de ellas evalúa todas las extensiones posibles j, se concluye que su complejidad temporal es $\mathcal{O}\left(n^2\right)$ y , siendo eficiente para cadenas de longitud moderada.

4.2. Prueba de la Complejidad con Casos Pseudo-Aleatorios

4.3. Variabilidad de la Complejidad

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

1. **Tamaño del mensaje**: A mayor longitud del mensaje, mayor es la cantidad de subproblemas posibles, y por tanto, el número de llamadas recursivas. Esto afecta directamente la cantidad de veces que se invoca la función recursiva antes de que el resultado esté memoizado.
2. **Tamaño del diccionario**: Aunque el diccionario sea un *set* para obtener búsquedas en tiempo constante, su tamaño puede influir indirectamente: a mayor cantidad de palabras, mayor probabilidad de que un segmento $s[i : j]$ coincida con alguna palabra, lo que puede aumentar la profundidad de la recursión, osea, una mayor cantidad de llamadas recusivas.
3. **Longitud promedio de las palabras del diccionario**: Si las palabras del diccionario son muy cortas, habrá muchas combinaciones posibles desde cada posición i, lo que puede aumentar la cantidad de ramificaciones en el árbol de llamadas recursivas antes de encontrar una segmentación válida (o descartar la posibilidad).

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

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