Universidad Tecnológica Metropolitana

Facultad de Ingeniería

Análisis de Complejidad de Algoritmos de Particionamiento de Palíndromos

Trabajo 2 - Análisis de Algoritmos

Autor:
Diego Cabezas

Julio 2024

Abstract

Este informe presenta un análisis detallado de dos algoritmos para el particionamiento de palíndromos, evaluando su complejidad utilizando recurrencias y otros métodos. Se incluyen resultados visuales y una conclusión basada en el análisis teórico y empírico.

0.1 Introducción

Este informe compara cómo dos algoritmos abordan el particionamiento de palíndromos mediante recurrencia y programación dinámica, analizando su complejidad. Se incluyen gráficos y tablas para comparar los resultados de ambos algoritmos con diferentes longitudes de cadena y demostrar su impacto en el rendimiento.

Esta complejidad de un algoritmo la forma que se expresa en términos de la notación Big O, que describe el comportamiento del tiempo de ejecución o el uso de espacio en función del tamaño de la entrada. Por ejemplo, los algoritmos con complejidad $O(n^2)$ tienen un crecimiento cuadrático, mientras que los algoritmos con complejidad $O(n^3)$ tienen un crecimiento cúbico, lo que significa que para grandes valores de n, el tiempo de ejecución de los algoritmos cúbicos será significativamente mayor que el de los algoritmos cuadráticos, y esto lo podremos comprobar con el tiempo de resolución de cada cadena de caracteres con base en los dos algoritmos en un gráfico.

Se resolverán recurrencias para analizar el tiempo de ejecución de los algoritmos, aplicando programación dinámica cuando sea necesario. Además, se mostrarán resultados empíricos que evidencien las diferencias de rendimiento entre los algoritmos estudiados.

0.2 Análisis de Complejidad

0.2.1 Algoritmo 1: Optimización de subproblemas superpuestos en $O(n^3)$

Descripción

Como se menciona en el trabajo, el algoritmo utiliza dos arreglos, $C[i][j]$ y $P[i][j]$, para almacenar los resultados computados. $C[i][j]$ almacena el número mínimo de cortes necesarios para la subcadena $cadena[i..j]$, mientras que $P[i][j]$ almacena si la subcadena $cadena[i..j]$ es un palíndromo o no. El algoritmo comienza con subcadenas más pequeñas y gradualmente se construye hasta la cadena completa.

Recurrencia

Por lo que la recurrencia para este algoritmo se puede expresar de la siguiente manera:

$$C[i][j] = \min_{i \leq k < j} (C[i][k] + C[k+1][j] + 1)$$

Esta recurrencia se utiliza donde se busca minimizar algún costo o maximizar alguna ganancia, dividiendo el problema en subproblemas más pequeños.

Programación Dinámica

La programación dinámica se utiliza aquí para evitar la recomputación de subproblemas superpuestos. Al almacenar los resultados de subproblemas en los arreglos C y P, el algoritmo puede construir la solución para la cadena completa de manera eficiente.

Análisis de Complejidad

La complejidad temporal de este algoritmo es $O(n^3)$ debido a los tres bucles anidados necesarios para llenar las matrices C y P.

0.2.2 Algoritmo 1: Implementación del código $O(n^3)$

Código en Python

La función a crear para este algoritmo tiene como objetivo encontrar el número mínimo de cortes necesarios para dividir una cadena dada **s** en subcadenas, de modo que cada subcadena sea un palíndromo. También devuelve las particiones resultantes de la cadena original.

```python
#Algoritmo (O(n^3)): Algoritmo A
def palindrome_cuts_A(s):
    n = len(s)
    C = [[0 for _ in range(n)] for _ in range(n)]
    P = [[False for _ in range(n)] for _ in range(n)]

    for i in range(n):
        P[i][i] = True

    for L in range(2, n + 1):
        for i in range(n - L + 1):
            j = i + L - 1
            if L == 2:
                P[i][j] = (s[i] == s[j])
            else:
                P[i][j] = (s[i] == s[j]) and P[i + 1][j - 1]

    for i in range(n):
        if P[0][i]:
            C[0][i] = 0
        else:
            C[0][i] = float('inf')
            for j in range(i):
                if P[j + 1][i] and 1 + C[0][j] < C[0][i]:
                    C[0][i] = 1 + C[0][j]

    partitions = []
    start = 0
    while start < n:
        for end in range(n - 1, start - 1, -1):
            if P[start][end]:
                partitions.append(s[start:end + 1])
                start = end + 1
                break

    return C[0][n - 1], '-'.join(partitions)
```

Figure 1: Algoritmo A

Función "palindrome cuts a"

La función "palindrome cuts a" encuentra el número mínimo de cortes necesarios para dividir la cadena "s" en subcadenas palindrómicas y devuelve también las particiones resultantes. Primero, se inicializan dos matrices: "P", que indica si una subcadena es un palíndromo, y "C", que guarda el número mínimo de cortes necesarios. Para subcadenas de longitud 1, se establece que son palíndromos. Luego, se verifica para subcadenas de mayor longitud si son palíndromos y se actualizan las matrices en consecuencia. A continuación, se calcula el número mínimo de cortes para cada subcadena. Finalmente, se generan las particiones palindrómicas basadas en la matriz "P" y se devuelve el número mínimo de cortes junto con las particiones formadas. La complejidad del algoritmo es $O(n^3)$ debido a los bucles anidados que iteran sobre las subcadenas y verifican si son palíndromos.

Por ejemplo, si la cadena de entrada es "abac", el resultado será ("aba-c"), indicando que se necesita un corte para dividir la cadena en subcadenas palindrómicas "aba" y "c".

0.2.3 Algoritmo 2: Programación dinámica en $O(n^2)$

Descripción

El algoritmo B, según el documento, resuelve el problema encontrando el sufijo que comienza en j y termina en el índice i ($1 \leq j \leq i \leq n-1$), que son palíndromos. Se realiza un corte aquí que requiere 1 + el mínimo de cortes del subtexto restante $[0, j-1]$. Para todos estos sufijos palíndromos que comienzan en j y terminan en i, se minimiza en $minCutDp[i]$. Finalmente, $minCutDp[n-1]$ será el número mínimo de cortes necesarios para el particionamiento palíndromo de la cadena dada.

Recurrencia

La recurrencia para este algoritmo se puede expresar de la siguiente manera:

$$minCutDp[i] = \min_{j \leq i}(1 + minCutDp[j-1])$$

Esta recurrencia, lo que quiere decir es que aquí, minCutDp[i] representa el número mínimo de cortes necesarios para particionar la subcadena que termina en la posición i en subcadenas palíndromas. La recurrencia calcula este valor considerando todos los posibles índices j desde 0 hasta i, y actualiza minCutDp[i] como el mínimo entre su valor actual y 1 + minCutDp[$j-1$], donde j es una posición de corte y minCutDp[$j-1$] es el número mínimo de cortes necesarios para la subcadena anterior a j. Esta estrategia normalmente se combina con una verificación adicional para determinar si una subcadena es un palíndromo, actualizando así minCutDp[i] solo cuando la subcadena entre j e i (inclusive) es un palíndromo.

Programación Dinámica

Aquí nuevamente la programación dinámica se utiliza aquí para evitar la recomputación de subproblemas. Al almacenar los resultados de subproblemas en el arreglo $minCutDp$, el algoritmo puede construir la solución de manera eficiente.

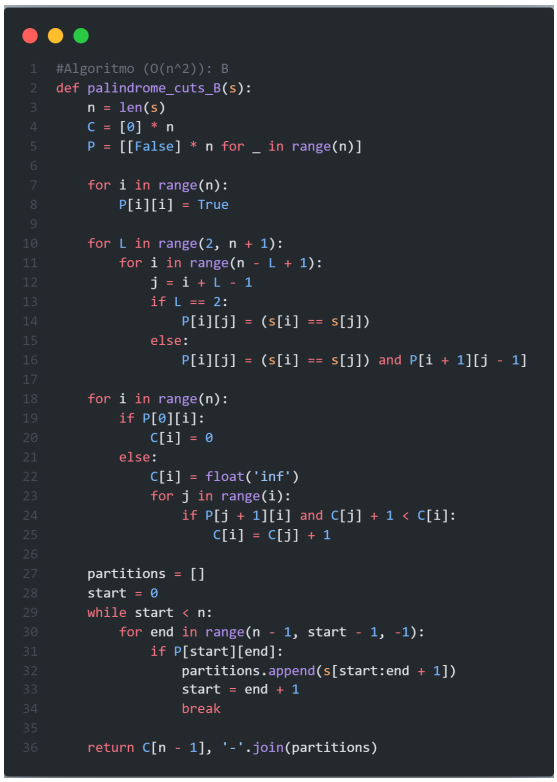
Análisis de Complejidad

La complejidad temporal de este algoritmo es $O(n^2)$ debido a los dos bucles anidados necesarios para llenar el arreglo $minCutDp$.

0.2.4 Algoritmo 2: Implementación del código $O(n^2)$

Código en Python

Para hacer el código con base en la recurrencia, este debe calcular el mínimo número de cortes necesarios para particionar una cadena en subcadenas palíndromas. Entonces, como idea inicial debemos inicializar la matriz P para identificar subcadenas palíndromas. Utilizamos la matriz P para calcular la matriz C de cortes mínimos, iteramos sobre posibles puntos de corte para actualizar C basado en la recurrencia y finalmente, construimos la partición de subcadenas palíndromas para obtener el resultado. Con esto en cuenta, tenemos el siguiente código:



```python
#Algoritmo (O(n^2)): B
def palindrome_cuts_B(s):
    n = len(s)
    C = [0] * n
    P = [[False] * n for _ in range(n)]

    for i in range(n):
        P[i][i] = True

    for L in range(2, n + 1):
        for i in range(n - L + 1):
            j = i + L - 1
            if L == 2:
                P[i][j] = (s[i] == s[j])
            else:
                P[i][j] = (s[i] == s[j]) and P[i + 1][j - 1]

    for i in range(n):
        if P[0][i]:
            C[i] = 0
        else:
            C[i] = float('inf')
            for j in range(i):
                if P[j + 1][i] and C[j] + 1 < C[i]:
                    C[i] = C[j] + 1

    partitions = []
    start = 0
    while start < n:
        for end in range(n - 1, start - 1, -1):
            if P[start][end]:
                partitions.append(s[start:end + 1])
                start = end + 1
                break

    return C[n - 1], '-'.join(partitions)
```

Figure 2: Algoritmo B

Función "palindrome cuts b"

El código "palindrome cuts b" toma una cadena s y calcula el número mínimo de cortes necesarios para dividirla en subcadenas palíndromas. Utiliza dos matrices: C, que almacena el número mínimo de cortes necesarios para cada posición, y P, que es una matriz booleana que indica si una subcadena es un palíndromo. Primero, inicializa P para subcadenas de longitud 1 y luego actualiza P para subcadenas más largas, comprobando si los extremos coinciden y si la subcadena interior es un palíndromo. Luego, calcula C actualizando el número mínimo de cortes para cada posición. Si una subcadena desde el inicio hasta i es un palíndromo, no se necesitan cortes (C[i] = 0), de lo contrario, se considera cada posible punto de corte j para minimizar C[i]. Finalmente, construye la lista de subcadenas palíndromas a partir de los cortes calculados y devuelve tanto el número mínimo de cortes como la cadena de subcadenas palíndromas separadas por guiones, tal cual como lo resuelve el algoritmo anterior.

0.3 Resultados

A partir de la solución que entrega el programa, ingresando la opción 1. Este genera un .txt con la cadena, que luego la carga para solucionarla con los algoritmos y así poder obtener los tiempos de resolución de los algoritmos con las cadenas de caracteres que van del [100-1000], además que guarda los tiempos en un .txt y de esta manera podemos hacer una tabla para comparar los resultados. Puesto que en el menú principal la opción 2. genera un gráfico con los tiempos obtenidos en la opción 1, así podemos comparar el rendimiento de dos algoritmos de complejidad $O(n^3)$ y $O(n^2)$ en función del tamaño de la cadena de entrada.

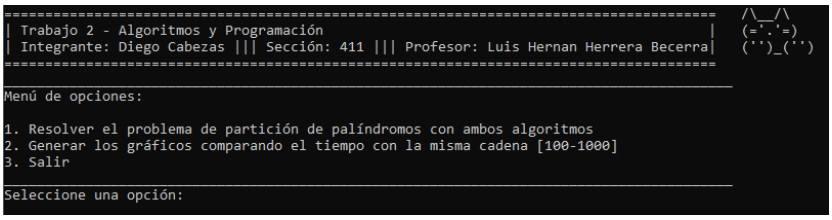


Figure 3: Imagen del menú con la opción 1

Tabla con los resultados

Tiempos ordenados	**Tiempo de ejecución (segundos)**	
de 100 a 1000	**O(n^3)**	**O(n^2)**
100	0.00100112	0.00100112
200	0.00400376	0.00300312
300	0.00900865	0.00600505
400	0.01801610	0.01100993
500	0.05004644	0.01901722
600	0.04003620	0.02702427
700	0.05505061	0.03703356
800	0.07206607	0.05004549
900	0.09108329	0.06305766
1000	0.11510444	0.07907200

Table 1: Comparación de tiempos de ejecución de algoritmos O(n^3) y O(n^2).

Con esta tabla permite entender manera clara cómo cambia el tiempo de ejecución a medida que aumenta el tamaño de los datos para estos dos tipos de algoritmos, proporcionando una visualización directa de su eficiencia relativa bajo diferentes escenarios de entrada.

Esto teniendo en cuenta, las características y componentes, que forman el computador en donde se ejecuta y se resuelve el programa, ya que puede variar los segundos, pero el aumento siempre va a ser significativo al haber un mayor número de datos.

Gráfico

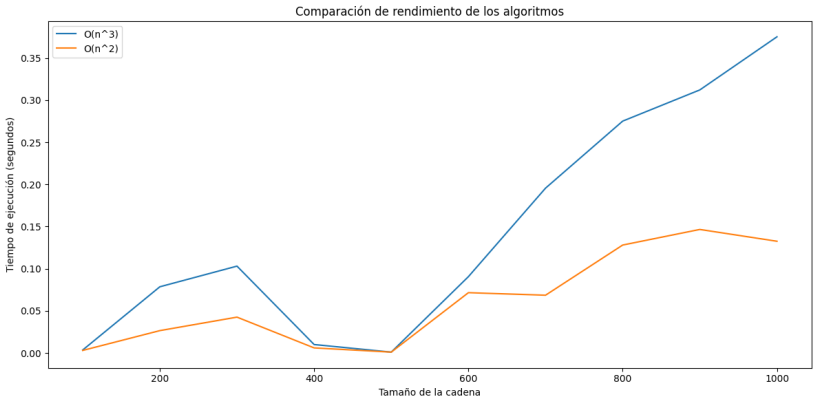


Figure 4: Gráfico con Ambos algoritmos - Tiempo v/s Tamaño

- Línea Azul ($O(n^3)$): Representa el tiempo de ejecución del algoritmo con complejidad $O(n^3)$. Este algoritmo muestra un crecimiento más pronunciado en el tiempo de ejecución a medida que aumenta el tamaño de la cadena.
- Línea Naranja ($O(n^2)$): Representa el tiempo de ejecución del algoritmo con complejidad $O(n^2)$. Este algoritmo es más eficiente y su tiempo de ejecución crece más lentamente en comparación con el algoritmo de $O(n^3)$.

Por lo descrito lo que podemos entender con el gráfico de forma más clara, la línea azul muestra un crecimiento más rápido y variable. Esto es típico de algoritmos con complejidad $O(n^3)$, donde el tiempo de ejecución es proporcional al cubo del tamaño de la entrada. Se observa que, a partir de tamaños de cadena más grandes (alrededor de 600), el tiempo de ejecución aumenta significativamente. La línea naranja, en contraste, muestra un crecimiento más uniforme y menos pronunciado. Esto es consistente con la complejidad $O(n^2)$, donde el tiempo de ejecución es proporcional al cuadrado del tamaño de la entrada.

Mientras tanto, para tamaños de cadena pequeños como 100 caracteres, la diferencia entre los dos algoritmos no es muy significativa, aunque el algoritmo $O(n^2)$ todavía es más rápido. A medida que aumenta el tamaño de la cadena, la diferencia de rendimiento se vuelve más notable. Para cadenas de 1000 caracteres, el algoritmo $O(n^3)$ es significativamente más lento que el algoritmo $O(n^2)$. Y cabe destacar que un momento o en algunas otras ejecuciones la línea azul muestra picos y valles, indicando que el tiempo de ejecución puede ser inconsistente para ciertos tamaños de cadena. Esta variabilidad podría deberse a la naturaleza del algoritmo y su sensibilidad a ciertas entradas.

Finalmente, con este análisis demuestra claramente cómo la complejidad algorítmica afecta el rendimiento. El algoritmo de $O(n^2)$ es mucho más eficiente que el de $O(n^3)$, especialmente para entradas grandes, lo que lo hace ideal para problemas con entradas grandes. Pero a largo plazo y con una cadena de caracteres muy grande, ambos algoritmos van a tardar mucho en resolver el problema, pero por la recurrencia de $O(n^2)$ lo hará de manera más rápida y eficaz.

0.4 Conclusión

Como conclusión se ha presentado un análisis comparativo detallado de dos algoritmos para el particionamiento de palíndromos, evaluando sus complejidades temporales y la eficiencia en función del tamaño de la cadena de entrada. A través del uso de recurrencias y programación dinámica, hemos demostrado cómo la complejidad algorítmica influye en el rendimiento y la escalabilidad de los algoritmos.

El Algoritmo A, con una complejidad temporal de $O(n^3)$, aunque efectivo, muestra un crecimiento significativo en el tiempo de ejecución a medida que aumenta el tamaño de la cadena. Este comportamiento es típico de algoritmos cúbicos, donde cada incremento en el tamaño de la entrada resulta en un incremento exponencial en el tiempo de ejecución.

Por otro lado, el Algoritmo B, con una complejidad temporal de $O(n^2)$, presenta una mejora notable en la eficiencia. La implementación de la programación dinámica en este algoritmo permite un manejo más eficiente de los subproblemas, resultando en un tiempo de ejecución considerablemente menor para cadenas grandes. Este rendimiento más consistente y menos variable lo convierte mejor que su algoritmo competidor.

Los resultados demostrados con el código hecho en Python, más apoyados por gráficos y tablas, respaldan lo dicho, mostrando de manera clara cómo el Algoritmo B supera al Algoritmo A en términos de tiempo de ejecución para cadenas de diferentes longitudes. A través de este análisis, se ha evidenciado la importancia de seleccionar recurrencias optimizadas y de menor complejidad para optimizar el rendimiento de futuros algoritmos.

Para finalizar, el análisis realizado no solo resalta la superioridad del Algoritmo B en términos de eficiencia y escalabilidad, sino que también proporciona una comprensión profunda de las técnicas utilizadas en la optimización de algo-

ritmos. Gracias a este estudio, sirve como una guía para futuro el desarrollo de futuros algoritmos donde la eficiencia es la clave del éxito.

0.5 Bibliografía

- BibliografíaCampbell, S. (2024, marzo 16). *Programa Palindrome en Python*. Guru99. https://www.guru99.com/es/palindrome-program-in-python.html
- Code, S. [@salmcode2894]. (2022, agosto 10). *Cómo tomar capturas de código profesionales en Visual Studio Code*. Youtube.
- *¿Cómo crear archivos de texto con Python?* (2021, mayo 11).
- Dux, C. (2024, febrero 1). *Guía: Notación Big O - Gráfico de complejidad de tiempo*. freecodecamp.org. https://www.freecodecamp.org/espanol/news/hoja-de-trucos-big-o/
- *Guía de desarrollo en LaTeX*. (s/f). Manualdelatex.com. Recuperado el 14 de julio de 2024, de https://manualdelatex.com/tutoriales
- *Palíndromo en Python - Parzibyte's blog*. (2021, mayo 19). Parzibyte's blog; parzibyte. https://parzibyte.me/blog/2021/05/19/palindromo-python/