.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico parte 2
Programación Dinámica

7 de febrero de 2025

Nombre	Padrón
Maximo Giovanettoni	110303
Franco Bossi	111122
Luca Saluzzi	108088
Agustin Conti	107800
Gabriel Peralta	101767

1. Introducción

En este trabajo se aborda la resolución de un juego de mesa de 2 jugadores en donde, dado un arreglo de monedas, cada jugador extrae una moneda de uno de los extremos. El que consigue el valor máximo acumulado es el ganador. Para buscar cuál es la jugada óptima se utiliza la técnica de Programación Dinámica.

En el siguiente informe se detalla el algoritmo planteado, la ecuación de recurrencia, y una demostración de que la ecuación permite obtener el máximo valor posible. Por último, se hace un análisis de la complejidad temporal, y mediciones que permitan corroborar la complejidad teórica.

1.1. Análisis de la consigna

En un principio se definirá lo que es la Programación Dinámica. Es un tipo de algoritmo el cual resuelve un problema complejo dividiéndolo en subproblemas más pequeños. Y, a diferencia de otros algoritmos, este almacena las soluciones anteriores en vez de volver a calcularlas. Además, usa un estilo de ecuación de recurrencia para poder calcular las soluciones (haciendo el código fácil de escribir pero difícil de pensarlo en sí).

Ahora, en esta parte, Mateo elige sus propias monedas y siempre elige la que mejor le conviene. Esto hace más complicado que Sofía gane. Entonces, una estrategia para que Sofía saque la máxima ganancia posible es una técnica de memoization el cual se define con la siguiente ecuación de recurrencia:

Considerando la función S como la representación del valor óptimo que obtendrá Mateo luego de que Sophia tome la moneda izquierda.

$$S(i,j) = \begin{cases} OPT(i+2,j) & \text{si } monedas[i+1] \geq monedas[j] \\ OPT(i+1,j-1) & \text{si } monedas[i+1] < monedas[j] \end{cases}$$

Y de manera análoga cuando Sophia elige la moneda de la derecha:

$$S(i,j) = \begin{cases} OPT(i+1,j-1) & \text{si } monedas[i] \geq monedas[j-1] \\ OPT(i,j-2) & \text{si } monedas[i] < monedas[j-1] \end{cases}$$

Obteniendo la ecuación de recurrencia de la siguiente forma:

$$OPT(i,j) = \begin{cases} \text{máx}(monedas[i] + S(i+1,j), monedas[j] + S(i,j-1)) & \text{si } i < j \\ monedas[i] & \text{si } i = j \\ 0 & \text{si } i < j \end{cases}$$

Resolución del problema

2. Descripcion del codigo y su implementación

La idea básica es que dos jugadores están tomando monedas de una secuencia de monedas. Cada jugador quiere maximizar la cantidad de monedas que puede tomar. Dado que cada jugador puede elegir la primera o la última moneda de la secuencia en su turno, el problema es inherentemente recursivo.

2.1. Construcción de la tabla de programación dinámica

```
OPT = [[0] * n for _ in range(n)]
decisiones = [[None] * n for _ in range(n)]
```

- OPT[i][j] almacena la mejor ganancia posible que Sophia puede obtener en el intervalo de monedas [i, j].
- decisiones[i][j] almacena la decisión óptima tomada ('izq' o 'der').

2.2. Llenado de la tabla

```
for k in range(2, n + 1):
    for izq in range(n - k + 1):
        der = izq + k - 1
```

- Se itera sobre subproblemas de diferentes tamaños (k).
- izq y der definen los límites del subarreglo de monedas en cada iteración.

2.3. Decisiones de Sophia

```
if monedas[izq + 1] >= monedas[der]:
    tomar_izq = monedas[izq] + OPT[izq + 2][der] if izq + 2 <= der else monedas[izq
    ]
else:
    tomar_izq = monedas[izq] + OPT[izq + 1][der - 1] if izq + 1 <= der - 1 else
    monedas[izq]
```

- Si Sophia elige la moneda de la izquierda, Mateo intenta reducir su ganancia en el siguiente turno.
- Se actualiza OPT[i][j] con la mejor opción posible.

2.4. Reconstrucción del camino óptimo

```
while left <= right:
    if decisiones[left][right] == 'izq':
        camino.append(('Sofia agarra la primera: ', monedas[left]))
        left += 1
    else:
        camino.append(('Sofia agarra la ultima', monedas[right]))
        right -= 1
    turnos += 1
```

- Se usa la matriz decisiones para reconstruir la secuencia de elecciones de Sophia y Mateo.

2.5. Demostración inductiva

Usaremos inducción para demostrar que la solución dada por la ecuación de recurrencia es correcta y maximiza el valor acumulado.

2.6. Base inductiva:

Para un solo elemento, OPT[i][i] = monedas[i]. Esto es correcto, ya que si solo queda una moneda, el jugador necesariamente la tomará, y ese será su valor máximo.

2.7. Paso inductivo:

Supongamos que la recurrencia es correcta para cualquier subsecuencia de longitud menor o igual a k-1. Ahora demostraremos que también es válida para una subsecuencia de longitud k.

El jugador tiene dos opciones, tomar la moneda izquierda o derecha. Si toma la izquierda, el valor que obtiene es el valor de esa moneda más lo que puede maximizar en las monedas restantes, sabiendo que el siguiente jugador tomará de forma óptima. El mismo razonamiento aplica si toma la moneda derecha.

El hecho de que tomemos el mínimo de las opciones del jugador siguiente asegura que estamos considerando que ambos jugadores juegan de forma óptima, lo que maximiza el valor acumulado para el primer jugador.

Dado que la recurrencia es válida para subsecuencias más pequeñas y seguimos esta lógica inductiva para subsecuencias más grandes, hemos demostrado por inducción que la recurrencia lleva al valor máximo acumulado.

3. Análisis de la solución

3.1. Análisis de la complejidad

En nuestra implementación, utilizamos un algoritmo basado en programación dinámica que almacena los valores óptimos de cada subproblema en una matriz de dimensiones nxn , donde n representa la cantidad de elementos en el conjunto de entrada. Esto nos permite calcular la solución de manera eficiente, evitando recomputaciones innecesarias.

La ecuación de recurrencia nos permitió definir una solución en $O(n^2)$, lo que es aceptable para valores razonables de . No estamos ante un algoritmo pseudo-polinomial, ya que el tiempo de ejecución depende exclusivamente de la cantidad de elementos y no de los valores individuales de estos.

Por otra parte, la reconstrucción de la solución se realiza en $O(n)$, ya que el algoritmo recorre la matriz de soluciones de manera secuencial, siguiendo los valores óptimos previamente calculados.

En consecuencia, la complejidad total del algoritmo, incluyendo cálculo y reconstrucción, sigue siendo $O(n^2)$.

3.2. Análisis de la complejidad de la reconstrucción

La reconstrucción de la solución se realiza recorriendo la matriz de resultados obtenida mediante programación dinámica.En cada paso, se identifica qué decisión se tomó en la matriz de estados y se almacena en una estructura auxiliar.

Dado que este proceso involucra un único recorrido de la matriz de nxn, la complejidad temporal de la reconstrucción es $O(n)$.

Como resultado, el tiempo de ejecución total del algoritmo, incluyendo cálculo y reconstrucción, sigue siendo $O(n^2)$, lo cual es eficiente para el problema planteado.

3.3. Análisis de la influencia de la variabilidad de los argumentos

El tamaño del conjunto de entrada es el principal factor que afecta el tiempo de ejecución. La variabilidad de los valores individuales de los elementos no influye en la complejidad del algoritmo, ya que este depende exclusivamente de la cantidad de elementos y no de su magnitud.

Restricciones como la presencia de valores negativos, repetidos o nulos pueden afectar la validez de la solución, pero no impactan en su eficiencia computacional.

En resumen, independientemente de los valores específicos en los elementos, la estrategia de programación dinámica garantiza que la solución óptima sea alcanzada en $O(n^2)$.

3.4. Optimalidad del algoritmo

Para demostrar que el algoritmo implementado garantiza la solución óptima al problema, es necesario deducir la ecuación de recurrencia utilizada y confirmar que el procedimiento seguido por el algoritmo respeta dicha ecuación.

Sea $M \in (\mathbb{Z}^+)^n$, con $n > 0$, un arreglo de monedas dispuestas en una secuencia sin un orden en particular. Nuestro objetivo es garantizar que Sophia obtenga la máxima suma posible, asumiendo que Mateo toma siempre la moneda más grande disponible en los extremos en su turno.

Definimos $OPT(i, j)$ como la máxima cantidad de dinero que Sophia puede obtener si el subarreglo considerado es $M[i], M[i+1], ..., M[j]$.

El algoritmo sigue una estrategia de programación dinámica basada en la siguiente ecuación de recurrencia:

$$OPT(i, j) = \text{máx}(M[i] + S(p(i+1, j)), M[j] + S(p(i, j-1)))$$

donde $S(p(i+1,j))$ y $S(p(i,j-1)))$ representan la mejor opción para Sophia después de que Mateo haya realizado su movimiento óptimo.

3.5. Demostración de que la ecuación coduce al óptimo global

Para probar que encuentra efectivamente la solución óptima, analizamos las dos decisiones posibles para Sophia:

1. Tomar la moneda de la izquierda ($M[i]$)

- Si Sophia elige $M[i]$, Mateo tomará la moneda que maximice su ganancia entre $M[i+1]$ y $M[j]$.
- La ganancia restante para Sophia después de que Mateo tome su mejor opción es la subsolución

$$OPT(p(i+1,j))$$

.

2. Tomar la moneda de la derecha ($M[j]$)

- Si Sophia elige $M[j]$, Mateo tomará la moneda más grande entre $M[j-1]$ y $M[i]$.
- La ganancia restante para Sophia después de que Mateo juegue de manera óptima es

$$OPT(p(i,j-1))$$

.

Por lo tanto, la ecuación de recurrencia nos dice que la mejor opción para Sophia es:

$$OPT(i,j) = \text{máx}(M[i] + OPT(p(i+1,j)), M[j] + OPT(p(i,j-1)))$$

Dado que el algoritmo implementado sigue exactamente esta ecuación de recurrencia al llenar la tabla OPT, podemos concluir que se obtiene la solución óptima.

3.6. Justificación por reducción al absurdo

Supongamos que la solución óptima no sigue la ecuación de recurrencia establecida. Esto implicaría que:

$$OPT(i,j) \neq \text{máx}(M[i] + OPT(p(i+1,j)), M[j] + OPT(p(i,j-1)))$$

Es decir, la solución elegida por el algoritmo no es el máximo entre estas dos opciones. Como el conjunto de elecciones posibles tiene exactamente dos elementos, esto significa que:

$$OPT(i,j) = \text{mín}(M[i] + OPT(p(i+1,j)), M[j] + OPT(p(i,j-1)))$$

Pero esto contradice la estrategia óptima del juego, ya que Sophia siempre busca maximizar su ganancia. En consecuencia, la suposición inicial es incorrecta y se concluye que:

$$OPT(i,j) = \text{máx}(M[i] + OPT(p(i+1,j)), M[j] + OPT(p(i,j-1)))$$

lo cual es exactamente la ecuación utilizada en la implementación.

4. Mediciones

Se realizaron mediciones en base a crear arreglos de diferentes largos, yendo de 100 en 100 elementos, donde los elementos en cada caso fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`).

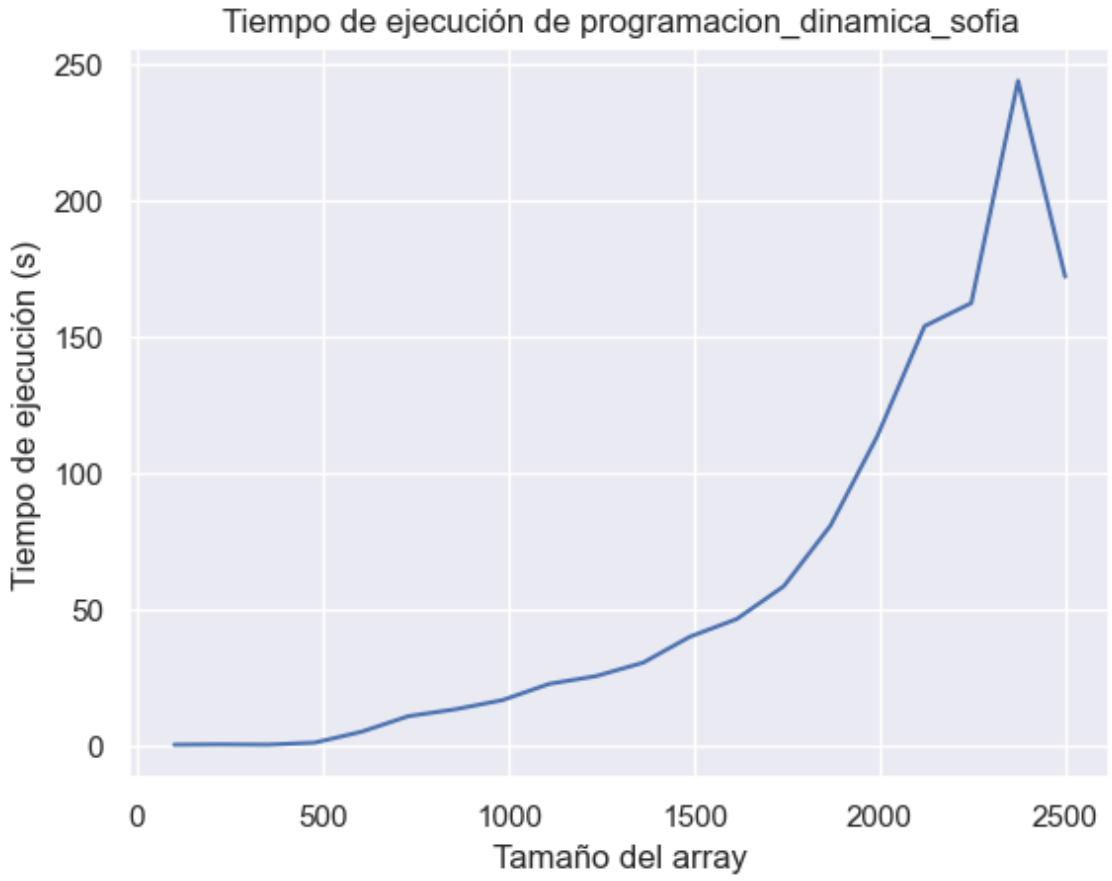

Figura 1: Tiempo de Ejecución

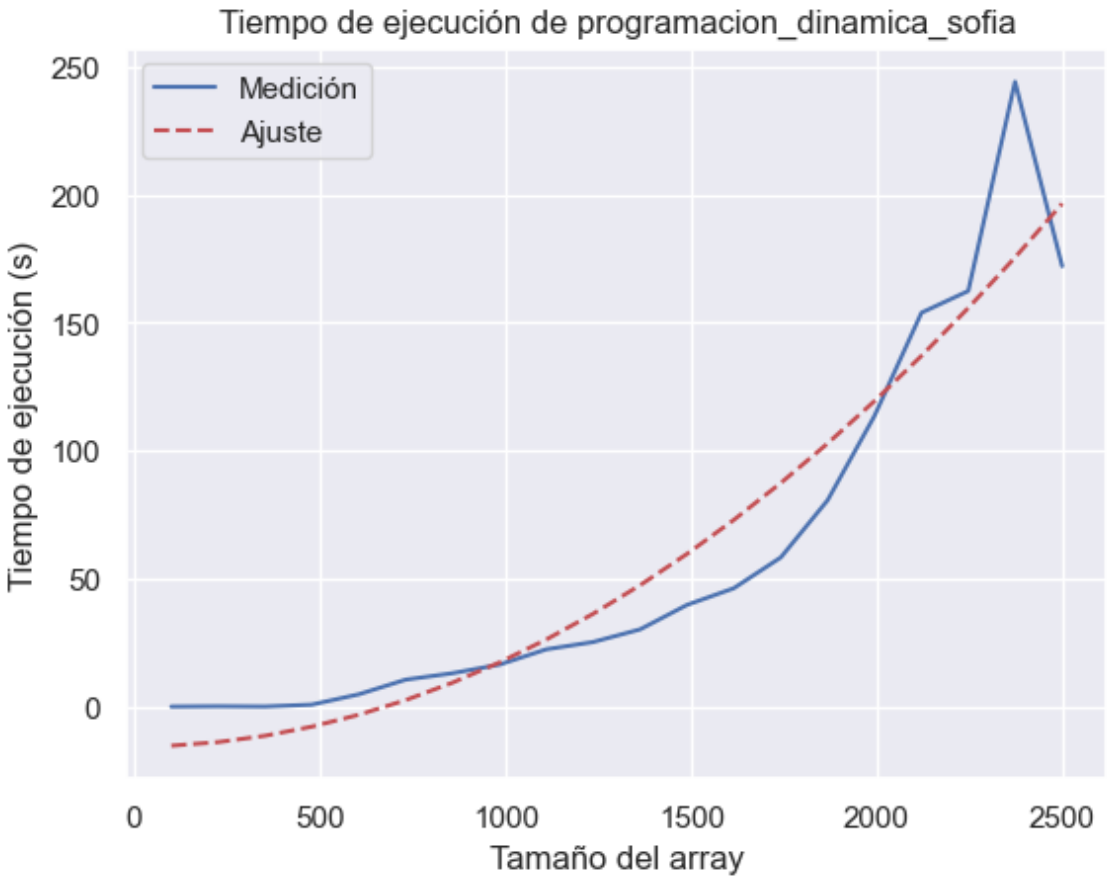

Figura 2: Tiempo de Ejecución con su error.

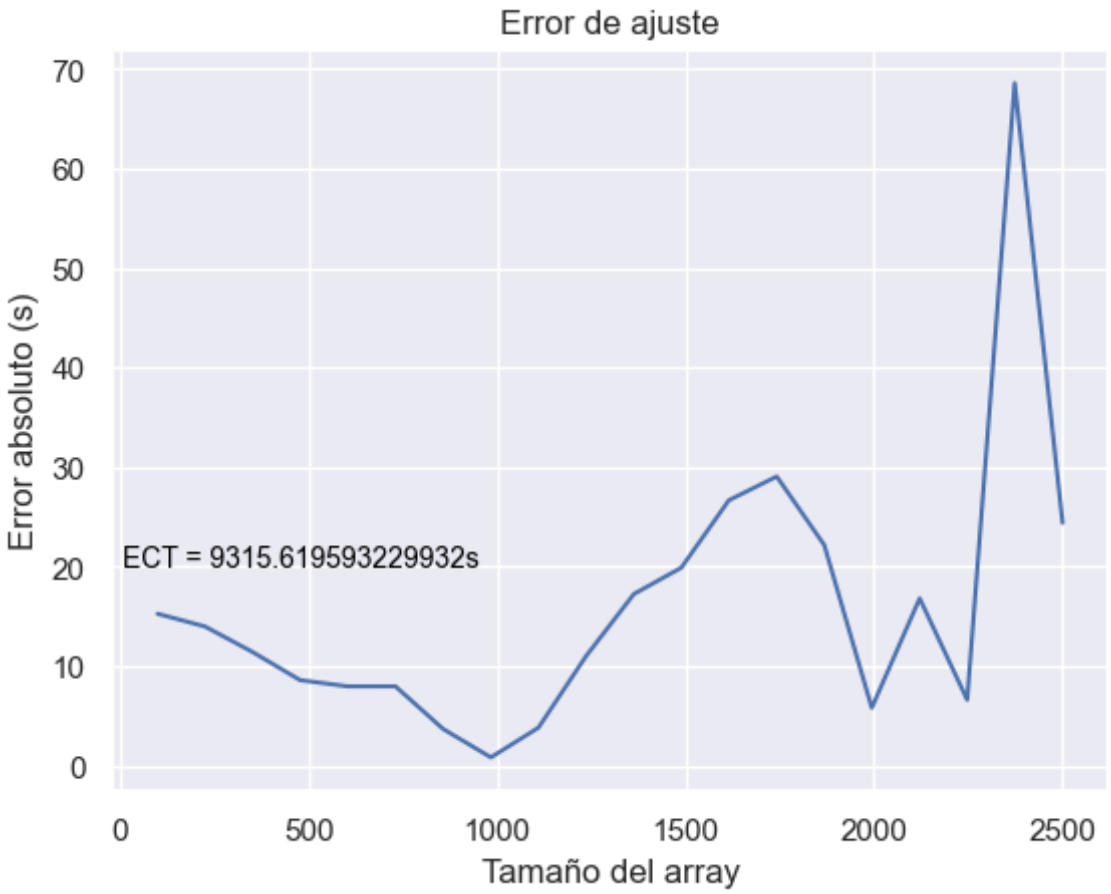

Figura 3: Error de ajuste

Como se puede apreciar, ambos algoritmos tienen una tendencia efectivamente lineal en función del tamaño de la entrada, si bien el algoritmo iterativo es más veloz en cuestión de constantes.

5. Conclusiones

La conclusion que se pudo sacar en este informe es que, aunque el algoritmo disponga de una gran simplicidad y su entendimiento en sí es más fácil que otros alogritmos, no siempre se llega al óptimo (por el hecho de que hay casos los cuales se hace imposible a Sofia a ganar), la complejidad no es muy conveniente en casos los cuales hay mucho volumen en el problema. Aún así, existen muchos casos los cuales Sofia gana y el algoritmo cumple un rol significativo en ese proceso por el hecho de que le calcula el óptimo en cada caso y eso es clave en muchos casos (en muchos los cuales estuvieron plasmandos en los tests).Entonces, el algoritmo es conveniente para ser usado en muchos casos pero no puede ser considerado como el mejor algortimo posible para resolver el problema.