Trabajo Práctico 2
Programación Dinámica For The Win

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Nombre	Padrón
Denise Dall'Acqua	108645
Martín Alejo Polese	106808
Nicolás Agustín Riedel	102130

Índice

Consigna

Introducción

Seguimos con la misma situación planteada en el trabajo práctico anterior, pero ahora pasaron varios años. Mateo ahora tiene 7 años. Los mismos años que tenía Sophia cuando comenzaron a jugar al juego de las monedas. Eso quiere decir que Mateo también ya aprendió sobre algoritmos greedy, y lo comenzó a aplicar. Esto hace que ahora quién gane dependa más de quién comience y un tanto de suerte.

Esto no le gusta nada a Sophia. Ella quiere estar segura de ganar siempre. Lo bueno es que ella comenzó a aprender sobre programación dinámica. Ahora va a aplicar esta nueva técnica para asegurarse ganar siempre que pueda.

Consigna

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: Dada la secuencia de monedas m1,m2 .... mn,sabiendo que Sophia empieza el juego y que Mateo siempre elegirá la moneda más grande para sí entre la primera y la última moneda en sus respectivos turnos, definir qué monedas debe elegir Sophia para asegurarse obtener el máximo valor acumulado posible. Esto no necesariamente le asegurará a Sophia ganar, ya que puede ser que esto no sea obtenible, dado por cómo juega Mateo. Por ejemplo, para [1, 10, 5] , no importa lo que haga Sophia, Mateo ganará.

2. Demostrar que la ecuación de recurrencia planteada en el punto anterior en efecto nos lleva a obtener el máximo valor acumulado posible.

3. Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Analizar si (y cómo) afecta a los tiempos del algoritmo planteado la variabilidad de los valores de las monedas.

4. Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado. Adicionalmente, el curso proveerá con algunos casos particulares que deben cumplirse su optimalidad también.

5. 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.

6. Agregar cualquier conclusión que les parezca relevante.

Resolución

1. Ecuación de recurrencia

A continuación se mostrará la ecuación de recurrencia hallada para este problema

$$T(monedas, inicioFila, finFila) = \begin{cases} K_1 & si \quad K_1 > K_2 \\ K_2 & si \quad K_1 < K_2 \end{cases}$$

Siendo

$$K_1 = monedas[inicioFila] + T(monedas, S(monedas, inicioFila + 1, finFila))$$
$$K_2 = monedas[finFila] + T(monedas, S(monedas, inicioFila, finFila - 1))$$

donde

$$S(monedas, inicioFila, finFila) = \begin{cases} (inicioFila + 1, finFila) & si \quad monedas[inicioFila] > monedas[finFila] \\ (inicioFila, finFila - 1) & si \quad monedas[inicioFila] < monedas[finFila] \end{cases}$$

NOTA:

$monedas$ = El vector con los valores de las monedas

$inicioFila$ = Es el valor de la primera moneda del vector monedas

$finFila$ = Es el valor de la última moneda del vector monedas

Nuestra función $T(monedas, inicioFila, finFila)$ nos otorga la ganancia que consiguió Sophia al momento de jugar con una cantidad de n monedas contra Mateo. En esta, podemos observar como nuestras variables K_1 y K_2 realizan llamados recursivos a T teniendo en cuenta como variables $inicioFila$ y $finFila$ la salida de otra función llamada $S(monedas, inicioFila, finFila)$, que son las dos posibles decisiones que puede tomar mateo al elegir una moneda (recordemos que mateo sigue las reglas del juego estríctamente).

2. Demostración del Algortimo

Para que la ecuación de recurrencia sea la solución óptima, osea que nos de el máximo valor acumulado posible, basta con verificar que:

1. La solución óptima de un problema grande puede obtenerse combinando soluciones óptimas de subproblemas más pequeños.
2. Esta descomposición debe hacerse de tal manera que no se omita ninguna posibilidad relevante para la solución óptima.
3. Se debe tomar una decisión sobre cuál subproblema o combinación de subproblemas proporciona el valor máximo.

- Nuestro caso base sería: Si no hay monedas, Sophia tendría ganancia cero ya que no habría monedas para seleccionar.
- Si Sophia agarra la primera moneda, podemos calcular su ganancia como el valor de esa moneda, más lo que ella recolectará en turnos posteriores. Esto teniendo en cuenta que Mateo siempre sigue las reglas del juego de manera óptima. En este punto del análisis, se ve como el problema se divide en subproblemas, osea por cada turno de Sophia.

- Si Sophia agarra la última moneda, realizamos la misma lógica anterior, nada más que para la última moneda.
- Luego, elegimos cuál de los dos escenarios logra obtener la máxima ganancia. A medida que vamos maximizando los escenarios locales, finalmente llegaremos al óptimo global de nuestro problema, osea la ganacia máxima posible que puede obtener Sophia.

3. Algoritmo planteado y complejidad

El algoritmo que decidimos utilizar para resolver el problema, es el siguiente:

- Mientras hayan monedas para elegir:
  - Vemos todas las monedas disponibles y en funcion de eso:
    - En el turno de Sophia, se elige la moneda que maximice la ganancia total, teniendo en cuenta que Mateo siempre va a elegir la moneda de mayor valor en todos los turnos posteriores.
    - En el turno de Mateo, se elige la moneda de mayor valor.
- Devolvemos la ganancia acumulada de Sophia.

Veamos el código:

```python
def juego_monedas(arr):
    memory = {}
    return _juego_monedas(arr, 0, len(arr), memory), memory

def _juego_monedas(arr, inicio = 0, fin = 0, memory = {}):
    if inicio >= fin:
        return 0

    key = (inicio, fin)
    if key in memory:
        return memory[key]

    primer_moneda = arr[inicio]
    ultima_moneda = arr[fin-1]

    decision_mateo_1 = decision_mateo(arr, inicio + 1, fin)
    decision_mateo_2 = decision_mateo(arr, inicio, fin-1)

    ganancia = max(primer_moneda + _juego_monedas(arr,decision_mateo_1[0], decision_mateo_1[1], memory),
                   ultima_moneda + _juego_monedas(arr, decision_mateo_2[0], decision_mateo_2[1], memory))

    memory[key] = ganancia

    return ganancia

def decision_mateo(arr, inicio = 0, fin = 0):
    if arr[inicio] > arr[fin - 1]:
        return (inicio + 1, fin)
    return (inicio, fin - 1)
```

Lo que estamos haciendo, es de forma recursiva observar todas las decisiones que puede tomar Sophia (sabiendo como se comporta Mateo) y quedarnos con la que maximice la ganancia total. Partimos de un caso base, donde si Sophia no tiene monedas para elegir, la ganancia es 0. Luego en cada llamado recursivo, vemos que moneda debemos elegir para maximizar la ganancia de ese subproblema, y guardamos en memoria el resultado para no tener que volver a calcularlo en caso de que se repita. De esta forma, combinando la solucion de los subproblemas, encontramos la solucion del problema original, que resulta en la maxima ganancia que puede obtener Sophia.

La complejidad temporal de este algoritmo en principio seria exponencial (O(2^n)), ya que en cada llamado recursivo se hacen dos llamados recursivos mas, y asi sucesivamente. Sin embargo, al utilizar memoria para guardar los resultados de los subproblemas, evitamos tener que recalcularlos, reduciendo la complejidad a cuadratica (O(n^2)). Esta complejidad surge de que la cantidad de subproblemas a resolver es n^2, ya que cada subproblema esta definido por un intervalo i,j donde i y j pueden tomar valores de 0 a n-1, siendo n la cantidad de monedas disponibles.

La complejidad espacial del algoritmo es O(n^2) ya que la memoria utilizada para guardar los resultados de los subproblemas es proporcional a la cantidad de subproblemas a resolver. Respecto al tamaño del stack es despreciable frente a este valor, ya que a lo sumo hay n llamados recursivos en simultaneo.

En resumen, la complejidad temporal y espacial del algoritmo es O(n^2).

Nota: En este analisis no se considera el algoritmo para la reconstruccion de la solucion.

3.1. Variabilidad

En relación a la complejidad algorítmica, la variabilidad de los valores de las monedas **no** afectan a los tiempos del algoritmo planteado, ya que los valores se utilizan para hacer comparaciones entre las monedas, y adiciones sobre el acumulado.

4. Ejemplos de ejecución

Se pueden encontrar encontrar multiples ejemplos de ejecución dentro del directorio *ejemplos* en el repositorio. Veamos algunos:

Supongamos que tenemos 5 monedas:

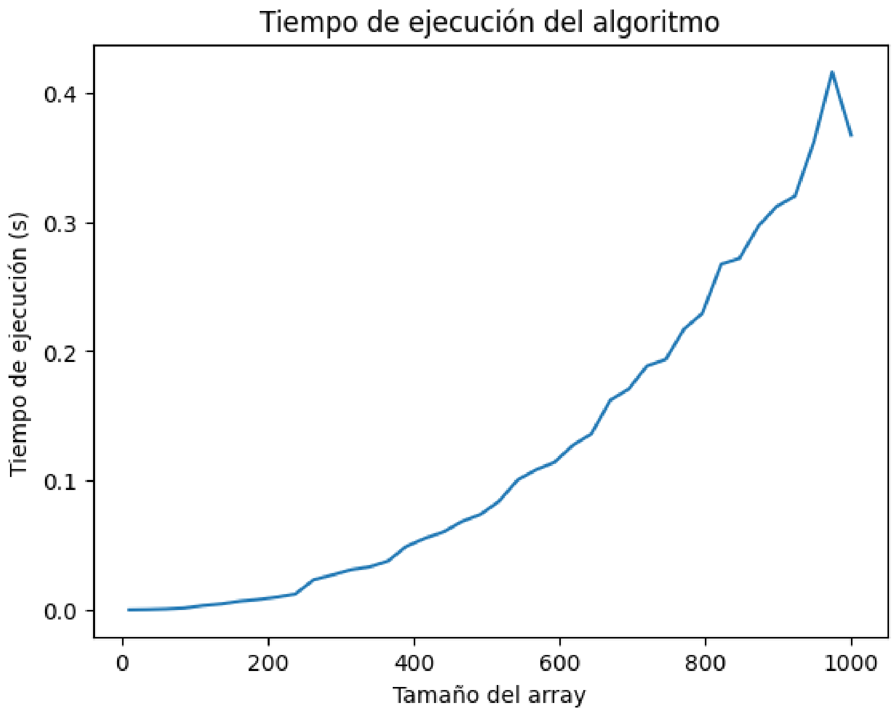
```
[96, 594, 437, 674, 950]
```

El primer turno es de Sophia, donde tiene 2 opciones: elegir la moneda del principio o la del final. Sabiendo que Mateo siempre va a elegir la moneda de mayor valor en los turnos posteriores, Sophia debe elegir la moneda que maximice su ganancia total.

5. Medición empírica

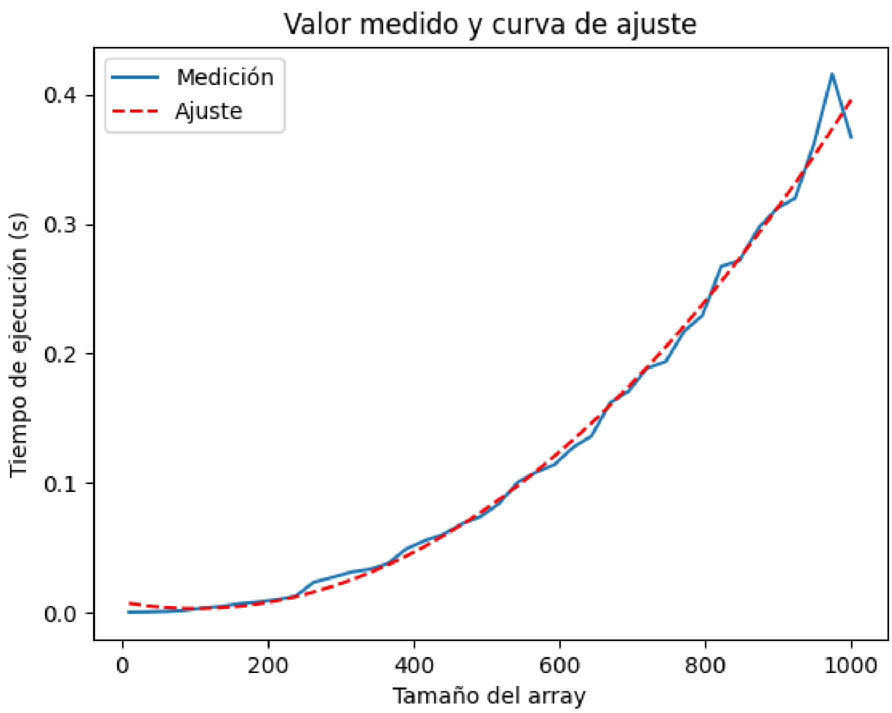
Para comprobar empíricamente la complejidad $\mathbf{O}(n^2)$ del algoritmo, se decidió ejecutar el mismo con distintos tamaños de entrada y medir el tiempo de ejecución. Se generaron muestras de tamaño n, las cuales varían desde 10 hasta 1000.

Para cada muestra se registró el tiempo de ejecución, obteniendo el siguiente gráfico:



A simple vista se puede observar un crecimiento cuadratico. Para confirmar esto, vamos a ajustar los datos a una recta mediante cuadrados mínimos. Esto lo realizamos con Python y la función *optimize.curve_fit* de la biblioteca *scipy*.

Obtenemos que el gráfico se puede ajustar a la curva $y = 4{,}86e^{-07}x^2 - 9{,}84e^{-05}x + 0{,}007$, con un error cuadrático medio de $9{,}37e^{-05}$. Por lo tanto, podemos verificar lo que ya vimos en la sección 3, que el orden es cuadratico $\mathbf{O}(n^2)$.



6. Conclusiones

Pudimos observar y verificar lo siguiente:

- Utilizando la tecnica de programacion dinamica con memorizacion, pudimos reducir la complejidad de un algortimo exponencial a uno cuadrático, al no tener que recalcular problemas anteriormente resueltos.
- Tras realizar un analisis empirico, pudimos confirmar que efectivamente la complejidad de nuestro algoritmo se vio beneficiada por la tecnica de programacion dinamica.

En conclusión, el presente trabajo permitió afianzar los conocimientos adquiridos en la materia de una manera práctica, donde desarrollamos