Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica

24 de octubre de 2024

Jonathan Dominguez
110057

Manuel Pato
110640

Mateo Godoy Serrano
110912

1. 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.

1.1. 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 $m_1, m_2, ..., m_n$, 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.

6. Agregar cualquier conclusión que les parezca relevante.

2. Análisis del Problema

Para comenzar, tenemos que entender qué es la programación dinámica. Un algoritmo que utiliza programación dinámica divide un problema complejo en subproblemas más pequeños. A diferencia de otros enfoques, la programación dinámica resuelve cada subproblema una sola vez y almacena sus soluciones para que puedan reutilizarse en el futuro, en lugar de volver a calcularlas repetidamente. De este modo, se evitan cálculos innecesarios. Una vez tenemos las soluciones de los subproblemas, podemos reconstruir una solución óptima para el problema original.

Entonces, ¿por qué utilizar programación dinámica y no otro enfoque?

En retrospectiva, que Sophia eligiera tanto sus monedas como las de Mateo a conveniencia facilitaba la resolución y permitía tomar el acercamiento de un algoritmo greedy. Sin embargo, ahora que Mateo elige sus propias monedas, contamos con una restricción adicional que impide esa posibilidad. Una nueva estrategia, entonces, sería reducir el problema original en problemas más pequeños, con menos monedas, e intentar forzar situaciones donde Sophia salga más favorecida. Para eso, aplicamos la técnica de memorización de subproblemas que caracteriza a los algoritmos de programación dinámica, y planteamos una ecuación de recurrencia que implementaremos posteriormente en un algoritmo:

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

Donde M es el arreglo de monedas, e i y j son índices sobre este mismo arreglo; en particular, i es un índice inferior y j un índice superior, por eso mismo usamos como hipótesis que si $i < j$ entonces no pueden haber monedas. En síntesis, vamos a tomar la moneda que nos maximice la primera expresión y en caso de que reste nada más una moneda, solo resta tomarla.

3. Implementación y descripción del algoritmo

```python
def nuevos_indices(indices, monedas): # Funcion p
    if monedas[indices[0]] >= monedas[indices[1]]:
        return indices[0] + 1, indices[1]
    else:
        return indices[0], indices[1] - 1
```

En esta función se devuelven los nuevos pares de índices que representan el nuevo subproblema luego del turno de Mateo donde se aplica el algoritmo greedy de seleccionar siempre la de mayor valor entre los dos extremos.

```python
def matriz_de_optimos(monedas):
    opt = [[0] * len(monedas) for _ in range(0, len(monedas))]

    for j in range(0, len(monedas)):
        # Las diagonales son triviales
        opt[j][j] = monedas[j]
        for k in range(1, j + 1):
            # Definimos i. Con esta transformacion, i es el indice inferior del
    arreglo de monedas,
            # y j el indice superior.
            i = j - k

            # Definimos dos nuevos pares de indices. Para eso, consideramos que
            # Sophia puede tomar la primera o ultima moneda, y luego vemos que
            # decision tomaria mateo.
            nuevo_inf_1, nuevo_sup_1 = nuevos_indices((i + 1, j), monedas)
            nuevo_inf_2, nuevo_sup_2 = nuevos_indices((i, j - 1), monedas)

            max_tomando_inf = monedas[i]
            max_tomando_sup = monedas[j]
```

```
            # Siempre que los indices no se vayan de rango, sumamos el optimo del
            # par indicado.
            if nuevo_inf_1 <= nuevo_sup_1:
                max_tomando_inf += opt[nuevo_inf_1][nuevo_sup_1]

            if nuevo_inf_2 <= nuevo_sup_2:
                max_tomando_sup += opt[nuevo_inf_2][nuevo_sup_2]

            # Aplicamos la ecuacion de recurrencia directamente.
            opt[i][j] = max(
                max_tomando_inf,
                max_tomando_sup
            )

    return opt
```

Este fragmento de código es el encargado de almacenar las soluciones óptimas a cada subproblema en la matriz *opt*, retornándola. Entrando más en detalle, generamos la matriz de óptimos con todos sus valores nulos donde la fila i representa el índice inferior i y cada columna j representa el índice superior j y la vamos llenando por columnas en is decrecientes. Para cada paso aplicamos la ecuación de recurrencia que discrimina entre el caso en el que Sophia agarre la moneda inferior y el caso en el que agarre la moneda superior y tomamos el valor máximo entre alternativas. Finalmente, el óptimo global al problema se encuentra en la fila $i = 0$ y columna $j = n - 1$.

```
def reconstruir_pasos(opt, monedas):
    i, j = 0, len(monedas) - 1

    decisiones_sophia = []
    decisiones_mateo = []

    while i <= j:
        # Para reconstruir, aplicamos la ecuacion de recurrencia a la inversa
        nuevo_inf_1, nuevo_sup_1 = nuevos_indices((i + 1, j), monedas)
        max_tomando_inf = monedas[i]
        if nuevo_inf_1 <= nuevo_sup_1:
            max_tomando_inf += opt[nuevo_inf_1][nuevo_sup_1]

        # Si el valor en la posicion corresponde a una eleccion, se toma esa, si no
    , la otra
        if opt[i][j] == max_tomando_inf:
            decisiones_sophia.append(f"Sophia debe agarrar la primera ({monedas[i
    ]})")
            i += 1
        else:
            decisiones_sophia.append(f"Sophia debe agarrar la ultima ({monedas[j]})
    ")
            j -= 1

        if i <= j:
            if monedas[i] > monedas[j]:
                decisiones_mateo.append(f"Mateo agarra la primera ({monedas[i]})")
                i += 1
            else:
                decisiones_mateo.append(f"Mateo agarra la ultima ({monedas[j]})")
                j -= 1

    return decisiones_sophia, decisiones_mateo
```

Finalmente, la función de reconstrucción toma como parámetros la matriz de óptimos y el arreglo de monedas y aplica la ecuación de recurrencia a la inversa para reconstruir los pasos partiendo de la posición $i = 0$ $j = n - 1$. Retorna un par de arreglos con las monedas que toma cada hermano.

4. Análisis de la solución

4.1. Análisis de la complejidad

Previamente, tuvimos que implementar un algoritmo greedy que logramos acotar a una complejidad de tan solo $O(n)$. Esta vez, el problema se nos presenta con un grado mayor de dificultad. Ahora Mateo también debe siempre agarrar la moneda más grande que pueda, como lo hacía Sophia. Sumado a esto se nos pidió resolverlo mediante programación dinámica.

En nuestro algoritmo, utilizamos una matriz de n x n posiciones, donde n es el tamaño del arreglo de monedas la cual almacena los valores óptimos para cada subproblema. Dada nuestra ecuación de recurrencia, esperábamos implementarlo con una complejidad de $O(n^2)$ siempre y cuando no fuera pseudo-polinomial. En este caso, logramos acotar la complejidad temporal por, justamente, $O(n^2)$, ya que no se trata de un algoritmo pseudo-polinomial debido a que su tiempo de ejecución depende pura y exclusivamente de la cantidad de monedas (n) y no del tamaño de las mismas.

A su vez, la reconstrucción tiene una complejidad $O(n)$ ya que la función trabaja con índices que representan sub arreglos del arreglo original hasta llegar a la reconstrucción total corriendo cada índice como mucho $n-1$ veces. Luego, la complejidad es, a lo sumo, $O(2(n-1))$, es decir $O(n)$.

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

El tamaño del arreglo moneda (denotado por n) afecta de manera cuadrática al tiempo de ejecución del algoritmo. La variabilidad en los valores de las monedas no influye en el tiempo de ejecución. Es importante señalar que los valores negativos, nulos (cero) y las monedas duplicadas no están permitidos. A modo de observación, nos gustaría mencionar que el valor de las monedas y su orden sí afectan al resultado del algoritmo, pero como ya mencionamos, no a su tiempo de ejecución. Un ejemplo de un arreglo de monedas con el que Sophia perdería podría ser: $[1, 5, 2]$, mientras que ganaría si se tomara esta permutación: $[5, 2, 1]$.

En definitiva, independientemente de los valores específicos en monedas, siempre y cuando se respeten las restricciones mencionadas y Mateo aplique el algoritmo greedy descrito en la introducción durante su turno, la suma de las monedas de Sophia va a ser máxima, lo que no implica que Sophia vaya a ganar.

4.3. Optimalidad del algoritmo

Para demostrar que el algoritmo es óptimo vamos a deducir la ecuación de recurrencia y verificar que es la misma que implementa el algoritmo.

Sea $M \in (Z^+)^n, n > 0$ un arreglo de monedas distintas entre sí, sin un orden en particular. Para demostrar que OPT conduce al óptimo global del problema (esto es, que garantiza que la suma del sub arreglo de monedas de Sophia es máxima dado que Mateo toma siempre la moneda más grande de los extremos en su turno), vamos a hacer dos demostraciones planteando lo siguiente:

1. $OPT(i, j) \neq M[i] + OPT(p(i+1, j))$
2. $OPT(i, j) \neq M[j] + OPT(p(i, j-1))$

Luego, demostraremos que cada proposición implica la negación de la otra, implicando que son las únicas soluciones posibles. Finalmente, demostraremos, por reducción al absurdo, que la solución debe ser el máximo entre estas alternativas.

Primera demostración:

Asumimos $OPT(i, j) \neq M[i] + OPT(p(i+1, j))$. Es decir, dado un sub arreglo del arreglo original que comience por el índice i y termine en el índice j (lo que también incluye al arreglo

original, naturalmente), el óptimo global no puede ser $M[i] + OPT(p(i+1, j))$. Esto se traduce a que **Sophia no puede elegir la primera moneda**. Debido a esto, no hay otra opción más que **elegir la última moneda**, por lo que, necesariamente:

$$OPT(i, j) = M[j] + OPT(p(i, j-1))$$

Segunda demostración:

Análogamente, asumimos que $OPT(i, j) \neq M[j] + OPT(p(i, j-1))$. Es decir, dado un sub arreglo del arreglo original, incluyendo el arreglo total en sí, que comience por el índice i y termine en el índice j, el óptimo global no puede ser $M[j] + OPT(p(i, j-1))$. Esto se traduce a que, para alcanzar el óptimo global, **Sophia no puede tomar la última moneda**, implicando que **Sophia debe elegir la primera moneda**, por lo que:

$$OPT(i, j) = M[i] + OPT(p(i+1, j))$$

Finalmente, demostramos que:

$$OPT(i, j) \neq M[i] + OPT(p(i+1, j)) \rightarrow OPT(i, j) = M[j] + OPT(p(i, j-1))$$

$$\wedge$$

$$OPT(i, j) \neq M[j] + OPT(p(i, j-1)) \rightarrow OPT(i, j) = M[i] + OPT(p(i+1, j))$$

Por lo que:

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

Es decir, solo hay dos soluciones posibles. Ahora, si asumimos que:

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

Como el conjunto tiene solo dos elementos, es equivalente a decir:

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

Pero como OPT debe asegurar que la ganancia de Sophia sea máxima, elegir el mínimo entre dos alternativas violaría este supuesto, concluyendo indefectiblemente que:

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

Para todo sub arreglo de tamaño j-i+1 del arreglo original, incluyendo al arreglo original. Esto demuestra que la ecuación de recurrencia planteada efectivamente lleva a la solución óptima del problema, y, dado que nuestro algoritmo implementa la ecuación de recurrencia demostrada, **nuestra implementación alcanza el óptimo para el arreglo original**.

5. Mediciones

Para verificar la complejidad deducida de nuestro algoritmo realizamos una serie de mediciones empíricas, donde testeamos el algoritmo con distintos tamaños de arreglos y, luego, verificamos mediante el método de **cuadrados mínimos** que la complejidad es la indicada en las secciones anteriores. Si bien no es de interés particular detallar el mecanismo exacto por el cual aproximamos la complejidad a través de las mediciones, sí lo es observar cuán acertada fue la deducción; para ello, preparamos unos gráficos que ilustran los resultados de las mediciones reales, nuestra aproximación por la complejidad $O(n^2)$, y el error de cada aproximación dado el tamaño de entrada del arreglo.

5.1. Medición de nuestro algoritmo

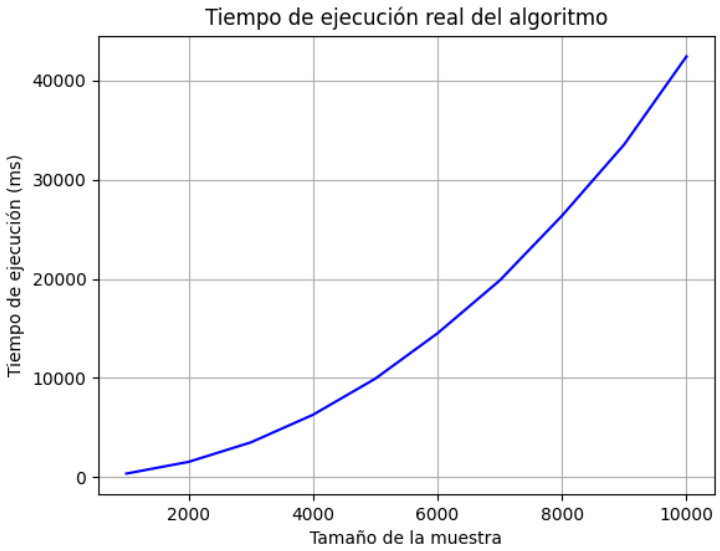

Figura 1: Mediciones

Para este caso usamos una muestra inicial de 2 mil elementos, hasta 10 mil, escalando el tamaño del arreglo de a 2 mil elementos. A su vez, se le asignó en el otro eje su tiempo de ejecución en milisegundos.

5.2. Comparación con el ajuste

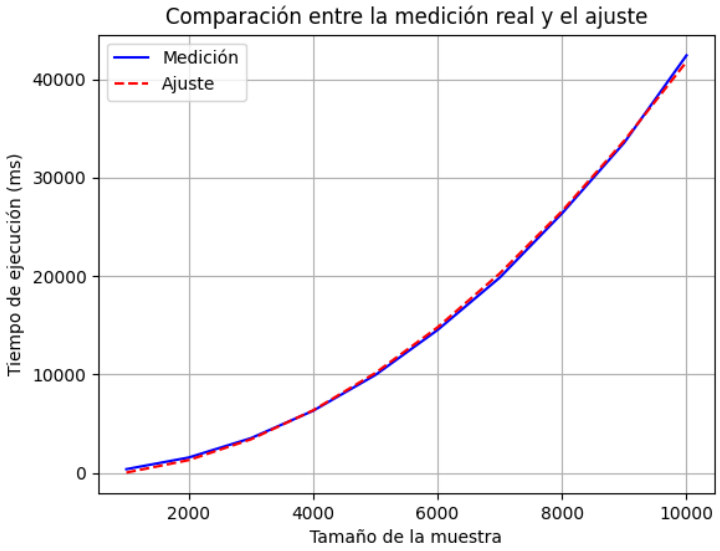

Figura 2: Comparación

Como se puede apreciar, la recta ajusta óptimamente a las mediciones realizadas, indicando que el algoritmo es, efectivamente, de complejidad $O(n^2)$.

5.3. Errores de la comparación

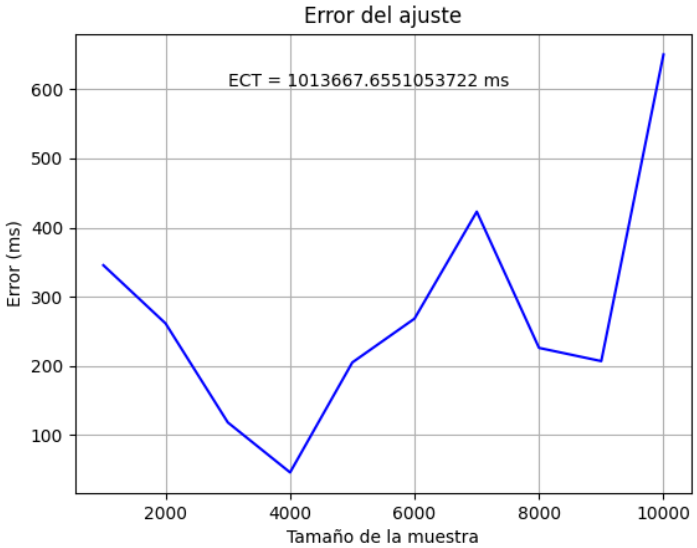

Figura 3: Errores y ECT

Finalmente, el **error cuadrático total** que proporciona la función elegida (la función que coincide con la complejidad el algoritmo, $f(n) = n^2$) para nuestra aproximación es, con estas mediciones, 1013667,655...ms^2.

5.4. Aclaraciones sobre las mediciones

Una aclaración que queremos dar antes de cerrar es que los gráficos que se usaron para demostrar el funcionamiento de nuestro algoritmo es uno de tantos que estuvimos creando en tres computadoras totalmente distintas. Esto se debe a que el algoritmo siempre va a depender del equipo/sistema en el que se trabaje, como también los componentes que lleve. Esos factores pueden generar que sea más o menos rápido frente a una muestra. Si bien no se notó un cambio en la complejidad del mismo, si hubo cotas más o menos grandes sobre el ECT.

6. Conclusiones

Haciendo una comparación con el algoritmo implementado en el anterior trabajo práctico notamos que la nueva restricción, que es que Mateo puede elegir la moneda de mayor denominación entre los extremos, implicó una desmejora sustancial en la cota de la complejidad temporal del algoritmo (pasando de $O(n)$ a $O(n^2)$), aunque no así en el impacto de la variabilidad de los argumentos, y requirió del uso de una técnica más sofisticada como lo es la programación dinámica.

Luego de analizar tanto el problema como la solución pudimos notar que la cuestión radicaba en el uso de la memorización de soluciones a problemas menores, y planteamos una ecuación de recurrencia que posteriormente implementamos en un algoritmo en Python que aplica la programación dinámica en su forma bottom-up.

Finalmente, y luego de pruebas empíricas, propinamos una demostración de la optimalidad del algoritmo haciendo uso de la exclusión mutua y reducción al absurdo, con lo que podemos afirmar que, dada nuestra implementación, Sophia siempre maximizará la suma de sus monedas.

7. Anexos

7.1. Correcciones de la primera reentrega

7.1.1. Corrección de la ecuación de recurrencia

En el capítulo 2, *Análisis del problema*, se enuncia incorrectamente la ecuación de recurrencia del problema. En realidad, la ecuación de recurrencia real, que es la demostrada en la sección 4.3, *Optimalidad del algoritmo*, hace uso de una función p, que determina el corrimiento de los índices sobre el arreglo según la moneda que toma Mateo. Funciona de la siguiente forma:

$$p(i,j) = \begin{cases} (i+1,j) \text{ si Mateo toma la moneda } M[i] \\ (i,j-1) \text{ si Mateo toma la moneda } M[j] \end{cases}$$

Ahora sí, la ecuación de recurrencia del problema corregida es la siguiente:

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

En síntesis, se evalúa qué moneda toma Sophia: si toma la moneda del índice i, entonces se corre ese índice (positivamente), y si toma la moneda de índice j, se corre ese otro índice (negativamente). Luego, ese nuevo par de índices determina las opciones de Mateo, por lo que se las pasa por la función p para determinar qué moneda toma, e invocar recursivamente la función con esa información.

7.1.2. Análisis de la complejidad de la reconstrucción

Dado un análisis insuficiente de la reconstrucción en la sección 4.1, *Análisis de la complejidad*, nos vemos en la necesidad de darle un breve repaso para determinar su complejidad acertadamente. El algoritmo recibe una matriz de óptimos y un arreglo con las monedas del juego, y se propone reconstruir la solución óptima para Sophia dados estos argumentos; para ello, se aplica la ecuación de recurrencia a la inversa, es decir, deducimos qué moneda tuvo que haber tomado Sophia en cada paso dado que el resultado es óptimo. Luego, dependiendo de qué moneda haya tomado Sophia, añadimos la información a un arreglo que lleva este registro.

Dada esta implementación, donde se desplazan los índices sobre el arreglo hasta converger a la solución óptima, el proceso del bucle *while* tiene una complejidad temporal de $O(n)$, ya que sus operaciones internas son todas de complejidad $O(1)$, dado que se trata de cálculos aritméticos e inserciones al final del array (aunque en Python las listas se comportan como arreglos dinámicos, la inserción al final de ellos, se dice, está *amortizada*, por lo que la complejidad promedio es $O(1)$). Finalmente, La complejidad del algoritmo de reconstrucción es $O(n)$.

Una nota final importante es que las mediciones realizadas en la sección 5, *Mediciones*, miden tanto el tiempo del cálculo del óptimo final como el tiempo de reconstrucción. Aun así, la complejidad de la ejecución secuencial de estos dos algoritmos sigue siendo $O(n^2)$