.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2

Programación Dinámica For The Win

14 de octubre de 2024

Alejo Fábregas	Camilo Fábregas	Juan Cruz Hernández
106160	103740	105711

Programación Dinámica For The Win

1. Análisis del problema

El problema que estamos buscando resolver en este segundo trabajo práctico consiste en hallar un algoritmo que, dado un set de monedas de diferentes valores, indique qué moneda debe elegir Sophia en cada turno para asegurarse **maximizar** su puntaje al finalizar la partida. Las reglas son que solamente se pueden elegir las monedas de los extremos (la primera o la última), y que Sophia es quien siempre comienza la partida. En este segundo trabajo práctico, la elección de Mateo ya no es más llevada a cabo por Sophia (que elegía la menor para su propio beneficio), sino que ahora elige en cada turno la de mayor valor de forma Greedy. Esto trae algunos cambios importantes en el desarrollo de la partida, ya que ahora existen casos en donde es incluso posible que Sophia pierda. Por eso es que el algoritmo que queremos hallar busca maximizar el puntaje de Sophia, sin importar quien es el que gana la partida.

Durante nuestro análisis inicial nos dimos cuenta que la forma en que Sophia puede "adelantarse" a elegir la mejor opción entre las dos monedas que se le presentan es observando también las monedas inmediatamente adyacentes a sus dos opciones, que son las que va a poder elegir Mateo en su turno. Es decir, observando la moneda a derecha de su opción izquierda, y la moneda a izquierda de su opción derecha. Esta aproximación inicial nos sirvió mucho para poder adentrarnos en el problema y para luego poder hacer un análisis más formal empleando los conceptos de programación dinámica vistos en clase.

Nuestro enfoque inicial para poder resolver este problema mediante programación dinámica consistió en utilizar un approach top-down, pero luego decidimos cambiar a bottom-up para alinearnos con las recomendaciones de la cátedra para resolver este tipo de problemas. Para esto, nuestro primer paso fue pensar en los **casos base** del problema:

- 1 moneda: como Sophia elige primero, la toma y maximiza su puntaje.
- 2 monedas: como Sophia elige primero, elige la de mayor valor y maximiza su puntaje.

El siguiente paso consistió en hallar los **subproblemas**: la forma de los mismos, como obtenerlos a partir de los casos base, y como combinarlos para poder utilizarlos para resolver subproblemas más grandes hasta llegar a la solución. Con los casos base ya tenemos resueltos los subproblemas para arreglos de largo 1 y 2 (1 y 2 monedas). Ahora queremos utilizar estas soluciones para resolver subproblemas de largo 3, y así luego poder extenderlos a subproblemas de cualquier tamaño. Para el caso de un subproblema de 3 monedas, nos dimos cuenta que tenemos que tomar dos decisiones:

- Sophia tiene que elegir entre la moneda de la izquierda o la moneda de la derecha.
- Luego Mateo tiene que elegir de forma Greedy la moneda más grande entre las dos que le haya dejado Sophia.
- Sophia toma la tercera y última moneda, que corresponde a un subproblema de largo 1 (ya resuelto por el caso base).

Como la idea es maximizar el puntaje de Sophia, habría que maximizar la suma entre la moneda 1 que eligió Sophia y la moneda 3, que le quedó luego de la elección de la moneda 2 de Mateo (de forma Greedy y por lo tanto, predecible). Ante esto se nos presentan 4 opciones:

1. Sophia elige la moneda izquierda (1) en su turno, y Mateo elige la moneda izquierda (2) en su turno. Sophia entonces elige la moneda restante (3).
2. Sophia elige la moneda izquierda (1) en su turno, y Mateo elige la moneda derecha (3) en su turno. Sophia entonces elige la moneda restante (2).
3. Sophia elige la moneda derecha (3) en su turno, y Mateo elige la moneda izquierda (1) en su turno. Sophia entonces elige la moneda restante (2).
4. Sophia elige la moneda derecha (3) en su turno, y Mateo elige la moneda derecha (2) en su turno. Sophia entonces elige la moneda restante (1).

La opción que maximice el puntaje de Sophia será la máxima entre estas 4 opciones.

Podemos extender esta lógica a subproblemas de 4 elementos, con la diferencia que luego de que Sophia y Mateo elijan sus respectivas monedas, quedarán dos monedas para que elija Sophia lo cual se resuelve con el caso base de dos monedas (Sophia elige la moneda más grande entre ambas opciones). Esto se puede extender a subproblemas de tamaño mayor aplicando las mismas elecciones (Sophia elige primero y luego Mateo elige entre sus dos opciones), ya que el arreglo de monedas restantes va a corresponder a un subproblema de tamaño menor que ya estará resuelto. Aquí es cuando nos damos cuenta que es conveniente ir guardando las resoluciones a los subproblemas más pequeños para poder utilizarlas en resolver subproblemas más grandes.

2. Ecuación de Recurrencia

Luego del análisis del problema que hicimos anteriormente, llegamos a la siguiente ecuación de recurrencia:

$$OPT(i,j) = \text{máx} \left(\begin{array}{l} monedas[i] + OPT(i+2,j) \text{ if } monedas[i+1] >= monedas[j], \\ monedas[i] + OPT(i+1,j-1) \text{ if } monedas[i+1] < monedas[j], \\ monedas[j] + OPT(i,j-2) \text{ if } monedas[i] < monedas[j-1], \\ monedas[j] + OPT(i+1,j-1) \text{ if } monedas[i] >= monedas[j-1] \end{array} \right)$$

Sea $OPT(i,j)$ la ganancia máxima que Sophia puede obtener si tiene acceso a las monedas en el rango $[i:j]$.

Las variables i, j representan los índices de las monedas que se están considerando en un turno de Sophia. 'i' representa el índice de la moneda en la parte izquierda y 'j' representa el índice de la moneda en la parte derecha.

Casos base:

1. **Una sola moneda**

$$OPT(i,j) = monedas[i] \quad \text{si } i = j$$

Hay una sola moneda. Como Sophia elige siempre primera, la elige, y ese es el óptimo.

2. **Dos monedas**

$$OPT(i,j) = \text{máx}(monedas[i], monedas[j]) \quad \text{si } i+1 = j$$

Hay dos monedas. Como Sophia elige siempre primera, elige la más grande, y ese es el óptimo.

Recurrencia:

Sophia tiene dos opciones:

1. **Tomar la moneda del extremo izquierdo** ($monedas[i]$):

   Después de esto, Mateo elige de forma greedy la máxima de las monedas restantes, por lo que la ganancia de Sophia es la moneda que elige más el óptimo del arreglo de monedas restante dado lo que eligió Mateo.

   Desglosamos los dos casos:

   $$opcion\_izq\_izq = monedas[i] + OPT(i+2, j) \text{ if } monedas[i+1] >= monedas[j] \text{ else } 0$$

   Donde $OPT(i+2, j)$ significa que en el turno siguiente Mateo va a elegir la moneda derecha y Sophia va a poder elegir entre la i+2 (nueva izquierda) y j (derecha).

   $$opcion\_izq\_der = monedas[i] + OPT(i+1, j-1) \text{ if } monedas[i+1] < monedas[j] \text{ else } 0$$

   Donde $OPT(i+1, j-1)$ significa que Mateo va a elegir la izquierda y Sophia va a poder elegir entre la i+1 (nueva izquierda) y j (nueva derecha).

2. **Tomar la moneda del extremo derecho** ($monedas[j]$):

   Luego, Mateo elige de forma greedy entre $monedas[i]$ y $monedas[j-1]$, lo que deja a Sophia con el subproblema de qué elegir después.

   Desglosamos los dos casos:

   $$opcion\_der\_der = monedas[j] + OPT(i, j-2) \text{ if } monedas[i] < monedas[j-1] \text{ else } 0$$

   si Mateo va a elegir la moneda derecha en el siguiente turno.

   $$opcion\_der\_izq = monedas[j] + OPT(i+1, j-1) \text{ if } monedas[i] >= monedas[j-1] \text{ else } 0$$

   si Mateo va a elegir la moneda izquierda en el siguiente turno.

Finalmente, Sophia opta por la mejor de las opciones, la que maximice su puntaje:

$$OPT(i, j) = \text{máx}(opcion\_izq, opcion\_der)$$

3. Algoritmo de Progamación Dinámica

3.1. Solución con Algoritmo de Programación Dinámica

Habiendo hecho este análisis al problema planteado, la solución que pensamos es la siguiente:

```python
def maximizar_ganancia(monedas):
    n = len(monedas)
    mem = [[-1] * n for _ in range(n)]
    elecciones = []

    for largo_subarreglo in range(1, n + 1):
        for i in range(n - largo_subarreglo + 1):
            j = i + largo_subarreglo - 1

            if i == j:
                mem[i][j] = monedas[i]
            elif i + 1 == j:
                mem[i][j] = max(monedas[i], monedas[j])
            else:
                opcion_izq_izq = monedas[i] + mem[i+2][j] if monedas[i+1] >= monedas[j] and i+2 <= j else 0
                opcion_izq_der = monedas[i] + mem[i+1][j-1] if monedas[i + 1] < monedas[j] and i+1 <= j-1 else 0
                opcion_izq = max(opcion_izq_izq, opcion_izq_der)

                opcion_der_der = monedas[j] + mem[i][j-2] if monedas[i] < monedas[j - 1] and i <= j-2 else 0
                opcion_der_izq = monedas[j] + mem[i+1][j-1] if monedas[i] >= monedas[j - 1] and i+1 <= j-1 else 0
                opcion_der = max(opcion_der_der, opcion_der_izq)

                mem[i][j] = max(opcion_izq, opcion_der)

    def reconstruir_elecciones(i, j):
        while i <= j:
            opcion_izq_izq = monedas[i] + mem[i+2][j] if monedas[i+1] >= monedas[j] and i+2 <= j else 0
            opcion_izq_der = monedas[i] + mem[i+1][j-1] if monedas[i+1] < monedas[j] and i+1 <= j-1 else 0
            opcion_izq = max(opcion_izq_izq, opcion_izq_der)

            opcion_der_der = monedas[j] + mem[i][j-2] if monedas[i] < monedas[j-1] and i <= j-2 else 0
            opcion_der_izq = monedas[j] + mem[i+1][j-1] if monedas[i] >= monedas[j-1] and i+1 <= j-1 else 0
            opcion_der = max(opcion_der_der, opcion_der_izq)

            if mem[i][j] == opcion_der:
                elecciones.append(f"Sophia debe agarrar la ultima ({monedas[j]})")
                j -= 1
            else:
                elecciones.append(f"Sophia debe agarrar la primera ({monedas[i]})")
                i += 1

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

    puntaje_sophia = mem[0][n-1]
    puntaje_mateo = sum(monedas) - puntaje_sophia

    reconstruir_elecciones(0, n-1)

    return puntaje_sophia, puntaje_mateo, "; ".join(elecciones)
```

3.2. Complejidad

El algoritmo propuesto tiene una complejidad $\mathcal{O}(n^2)$, tanto temporal como espacial, siendo n el tamaño del arreglo de monedas. Esto no es posible de demostrar mediante el Teorema Maestro, ya que la solución no es recursiva. De todos modos, podemos analizarlo paso a paso sobre el algoritmo.

Se crea una matriz de n*n, lo que tiene un costo de $\mathcal{O}(n^2)$, tanto en tiempo como en espacio. Luego vemos que tenemos dos bucles *for* anidados. El bucle externo itera todos los n tamaños de subarreglos posibles. El bucle interno va probando todos los subarreglos posibles de los tamaños 1 a n. Para tamaño 1, son n subarreglos; para tamaño 2, son n - 1 subarreglos; para tamaño n es un solo subarreglo, por lo que tenemos como máximo n iteraciones para cada tamaño de subarreglo. Para cada uno de ellos, se aplica la ecuación de recurrencia, lo cual es $\mathcal{O}(1)$ ya que son accesos a la matriz y simples comparaciones. Entonces, vemos que tenemos como máximo n iteraciones en los subarreglos para los n tamaños de subarreglos, haciendo operaciones constantes en cada uno de ellos. Entonces, en total el costo es $\mathcal{O}(n^2)$.

La reconstrucción de las elecciones es $\mathcal{O}(n)$, ya que se recorre el arreglo de n monedas evaluando en cada posición qué decisión se tomó, lo que implica realizar operaciones $\mathcal{O}(1)$. Como se repiten n veces, la complejidad es $\mathcal{O}(n)$.

Más adelante realizaremos un análisis empírico con mediciones de tiempo para verificar que el algoritmo efectivamente tiene esta complejidad teórica.

3.3. Uso de Programación Dinámica

La solución realizada aprovecha las bondades de la programación dinámica para resolver un problema en donde la programación greedy se queda corta. En este caso, al querer maximizar la puntuación obtenida por Sophia nuestra solución greedy del trabajo práctico anterior ya no es suficiente, sino que necesitamos ver más allá de cada iteración para maximizar el puntaje.

Se distinguen los casos base en los que hay una o dos monedas, para a partir de ellos construir soluciones a los subproblemas más pequeños. Luego utilizamos esas soluciones a subproblemas para solucionar subproblemas cada vez más grandes aplicando la ecuación de recurrencia en cada caso, para llegar a la solución final. Para guardar las soluciones a los subproblemas y no tener que recalcularlas hacemos uso de una matriz de *memoization* para ir guardándolos y utilizarlos cuando se necesiten.

En cada turno de Sophia aplicamos la ecuación de recurrencia para tomar la decisión que más la beneficie: si elegir la moneda izquierda o la derecha. Para ver esto hay que considerar varios factores: los valores de esas monedas, los valores de las monedas que agarraría Mateo de forma greedy en el turno siguiente, y el puntaje de Sophia sobre el subarreglo de monedas restante (que ya está calculado en la matriz de *memoization* por la programación dinámica).

Para la reconstrucción de las decisiones de los jugadores simplemente se aplica la misma lógica que se utilizó para decidir, sobre la matriz de *memoization*, para ir viendo qué monedas tomó Sophia y cuáles tomó Mateo.

3.4. Particularidades

El algoritmo maximiza el puntaje de Sophia, pero eso no garantiza que siempre va a ganar. Hay casos en los que no puede ganar por una combinación entre las reglas del juego y la naturaleza del arreglo de monedas.

Si todas las monedas son iguales, y hay una cantidad par de monedas, por ejemplo, al ser un juego por turnos ninguno de los dos va a ganar, sin importar las decisiones que tomen.

Similarmente, hay casos en los que Sophia pierde siempre, como el arreglo [3, 20, 2]. Sin importar lo que elija Sophia, Mateo gana siempre ya que puede elegir la moneda del medio, que tiene el mayor valor. Sin embargo, el algoritmo nos asegura maximizar el puntaje de Sophia.

Por otro lado, en un momento consideramos que quizás se podría optimizar el uso de memoria guardando únicamente los tres valores de la matriz que se utilizan en la ecuación de recurrencia en cada iteración: mem[i+2][j], mem[i+1][j-1] y mem[i][j-2]. Esto sería algo similar a lo que se hace en el ejemplo de Fibonacci, para no tener que guardar toda la matriz. Sin embargo, surge el problema de cómo reconstruir las elecciones de la solución sin tener la matriz, por lo que descartamos esta opción.

4. Demostración de optimalidad

Es necesario demostrar que el algoritmo propuesto maximiza el puntaje de Sophia en una partida. Para esto, tenemos que demostrar la solución óptima para el problema sigue la ecuación de recurrencia del algoritmo. Vamos a hacerlo mediante el principio de inducción matemática.

Hipótesis de inducción: suponemos que la ecuación de recurrencia $OPT(i, j)$ da la solución óptima para cualquier subproblema de k monedas, es decir que devuelve el máximo puntaje que Sophia puede obtener para ese subconjunto de k monedas.

Recordemos la ecuación de recurrencia en cuestión:

$$OPT(i,j) = \text{máx} \left(\begin{array}{l} monedas[i] + OPT(i+2,j) \text{ if } monedas[i+1] >= monedas[j], \\ monedas[i] + OPT(i+1,j-1) \text{ if } monedas[i+1] < monedas[j], \\ monedas[j] + OPT(i,j-2) \text{ if } monedas[i] < monedas[j-1], \\ monedas[j] + OPT(i+1,j-1) \text{ if } monedas[i] >= monedas[j-1] \end{array} \right)$$

Casos base: tenemos los dos casos base ya vistos, por lo que sabemos que la ecuación es válida para los casos de $k = 1$ (una moneda) y $k = 2$ (dos monedas).

- $k = 1$ **(una moneda)**: Como Sophia elige siempre primera, elige la única moneda, y ese es el óptimo.
$$OPT(i,j) = monedas[i] \quad \text{si } i = j$$
- $k = 2$ **(dos monedas)**: Como Sophia elige siempre primera, elige la moneda más grande, y ese es el óptimo.
$$OPT(i,j) = \text{máx}(monedas[i], monedas[j]) \quad \text{si } i + 1 = j$$

Paso inductivo: ahora vamos a considerar un subproblema de $k + 1$ monedas, y queremos ver que la ecuación también sea óptima para este caso. Si antes teníamos un rango (i, j) con k monedas, ahora vamos a tener un rango $(i, j + 1)$ con $k + 1$ monedas. Esto equivale a decir que antes teníamos $k = j - i$ monedas, y que ahora vamos a tener $k + 1 = j + 1 - i$ monedas.

Siguiendo la ecuación de recurrencia, sabemos que tenemos 4 casos posibles:

$$OPT(i,j+1) = \text{máx} \left(\begin{array}{l} monedas[i] + OPT(i+2,j+1) \text{ if } monedas[i+1] >= monedas[j+1], \\ monedas[i] + OPT(i+1,j) \text{ if } monedas[i+1] < monedas[j+1], \\ monedas[j+1] + OPT(i,j-1) \text{ if } monedas[i] < monedas[j], \\ monedas[j+1] + OPT(i+1,j) \text{ if } monedas[i] >= monedas[j] \end{array} \right)$$

- Si Sophia toma la moneda del lado izquierdo ($monedas[i]$), sabemos que Mateo elige de forma greedy la mayor de las monedas $monedas[i + 1]$ y $monedas[j + 1]$.
  - Si Mateo elige $monedas[i+1]$, Sophia queda con el subproblema $OPT(i+2, j+1)$, que es de tamaño $k - 1$ (ya que $(j + 1) - (i + 2) = j - i - 1 = k - 1$), y por la hipótesis de inducción, sabemos que la solución para este subproblema es correcta.
  - Si Mateo elige $monedas[j + 1]$, Sophia queda con el subproblema $OPT(i + 1, j)$, que también es de tamaño $k - 1$ y por hipótesis también es correcto.
- Si Sophia toma la moneda del lado derecho ($monedas[j + 1]$), sabemos que Mateo elige de forma greedy la mayor de las monedas $monedas[i]$ y $monedas[j]$.
  - Si Mateo elige $monedas[j]$, Sophia queda con el subproblema $OPT(i, j - 1)$, que es de tamaño $k - 1$ y es correcto por la hipótesis.
  - Si Mateo elige $monedas[i]$, Sophia queda con el subproblema $OPT(i+1, j)$, que también es de tamaño $k - 1$.

Como cada una de las 4 posibilidades que se evalúan al plantear un subproblema de $k+1$ monedas se reduce a otro subproblema menor de $k-1$ monedas, que por hipótesis inductiva se resuelve de forma óptima, podemos afirmar que también es óptimo para $k+1$ monedas. Por lo tanto, completamos el paso inductivo, por lo que podemos concluir que la ecuación de recurrencia es óptima para cualquier número de monedas.

Queda demostrado que el algoritmo de programación dinámica que utiliza esta ecuación de recurrencia es óptimo para resolver el problema de maximizar el puntaje de Sophia en una partida.

4.1. Variabilidad

En la demostración no se tuvo en cuenta los valores particulares de las monedas, por lo que la variabilidad de los valores no afecta la optimalidad de la solución. Anteriormente vimos ciertos casos en los que Sophia no puede ganar, como el caso de monedas iguales que termina en empate, o el caso de 3 monedas con una más valiosa en el medio, que siempre gana Mateo. A pesar de que Sophia no tenga la posiblidad de ganar, se sigue maximizando su puntaje dentro de las posibilidades que se plantean en estos casos.

5. Análisis de complejidad

5.1. Introducción

Vamos a realizar mediciones de tiempo de nuestro algoritmo para comprobar empíricamente que tiene la complejidad teórica indicada anteriormente: $\mathcal{O}(n^2)$.

Como nuestra función *maximizar_ganancia* toma como parámetro un arreglo de n números (que representan monedas), podemos crear arreglos de distinto tamaño para crear simulaciones que nos servirán para medir el tiempo de ejecución de la función.

Correremos estas simulaciones para obtener los tiempos de ejecución de la función para distintas entradas, y luego haremos un ajuste por cuadrados mínimos para ver si nos alejamos de una tendencia cuadrática. Si el error es bajo, significa que la función es de complejidad cuadrática.

5.2. Dataset de prueba

Para probar y medir nuestra función utilizaremos distintos tamaños de entrada para evaluar cómo evoluciona el tiempo de ejecución. Tendremos 40 arreglos distintos, con tamaños variando entre 10 elementos y 1.000 elementos, y los valores de esos arreglos serán números enteros entre 0 y 1000.

Idealmente hubiésemos querido probar la función con arreglos con muchos más elementos, pero en este caso nos limita la complejidad de la función ya que correr esas pruebas implicaría un tiempo muy largo y un consumo de memoria muy alto.

Más adelante realizaremos otro análisis con valores del arreglo más variados, para ver si la variabilidad de los valores de las monedas afecta a la complejidad.

5.3. Tiempo en función de la entrada

A continuación graficamos el tiempo de ejecución de *maximizar_ganancia* en función al tamaño de la entrada que probamos anteriormente. Podemos ver que para 400 elementos tarda aproximadamente 0,05 segundos, y que para 800 de elementos tarda aproximadamente 0,20 segundos, lo que nos estaría indicando que el algoritmo es cuadrático.

5.4. Ajuste por cuadrados mínimos

Ahora vamos a ajustar por cuadrados mínimos a los resultados que obtenimos de los tiempos de ejecución de nuestra función. En nuestro caso vamos a ajustar contra la siguiente parábola, para comprobar la tendencia cuadrática del algoritmo:

$$y = c_0 x^2 + c_1 x + c_2$$

Podemos ver que el error es bajo, por lo que ajusta muy bien a la función cuadrática:

- Valores de los coeficientes:

$$c\_0 = 3{,}8123117907217056e-07$$

$$c\_1 = -4{,}776115603692792e-06$$

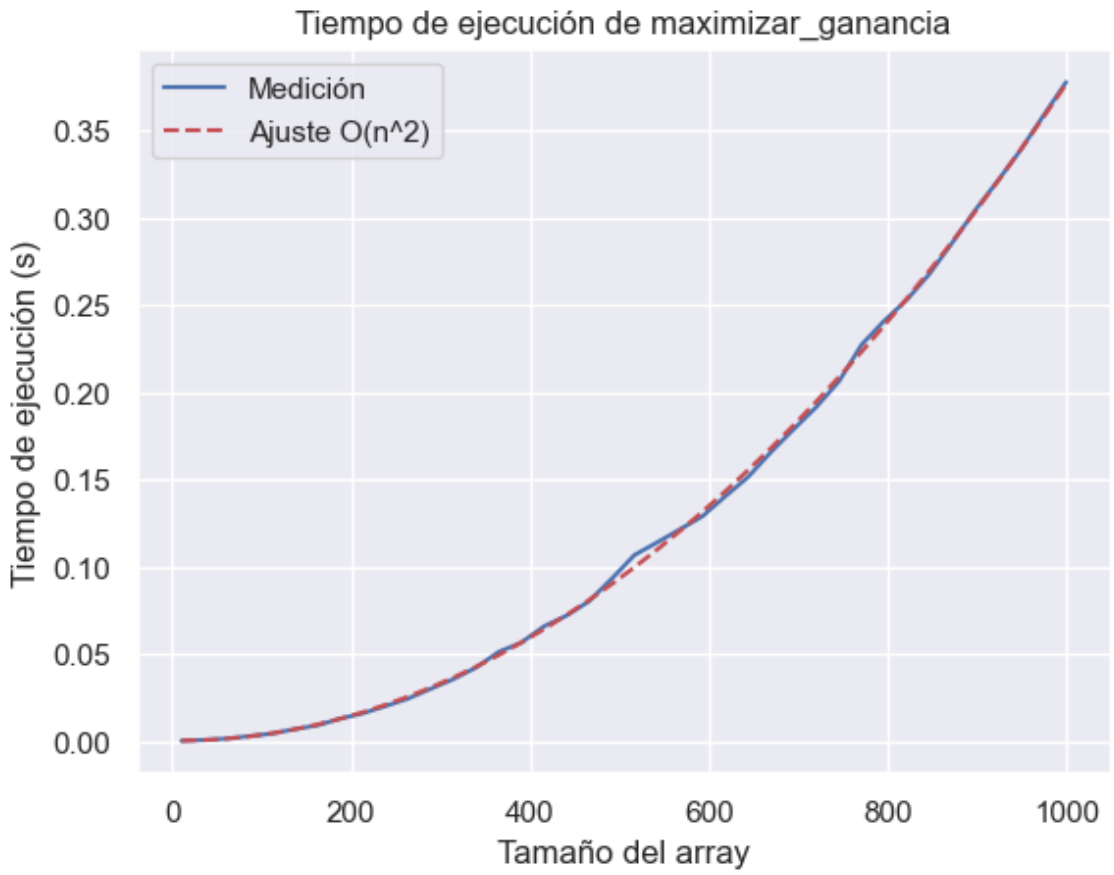
$$c\_2 = 0{,}00017773984349076397$$

- Error cuadrático total:

$$0{,}00017502168342102293$$

5.5. Gráfico de los datos y el ajuste

Se observa que los datos ajustan muy bien contra la parábola.



5.6. Error de ajuste en función del tamaño de la entrada

En este gráfico se observa que el error de ajuste según el tamaño de la entrada es muy bajo en todos los casos, y es parejo a lo largo de todos los tamaños. Esto es otro indicio de que el algoritmo es cuadrático.

5.7. Variabilidad

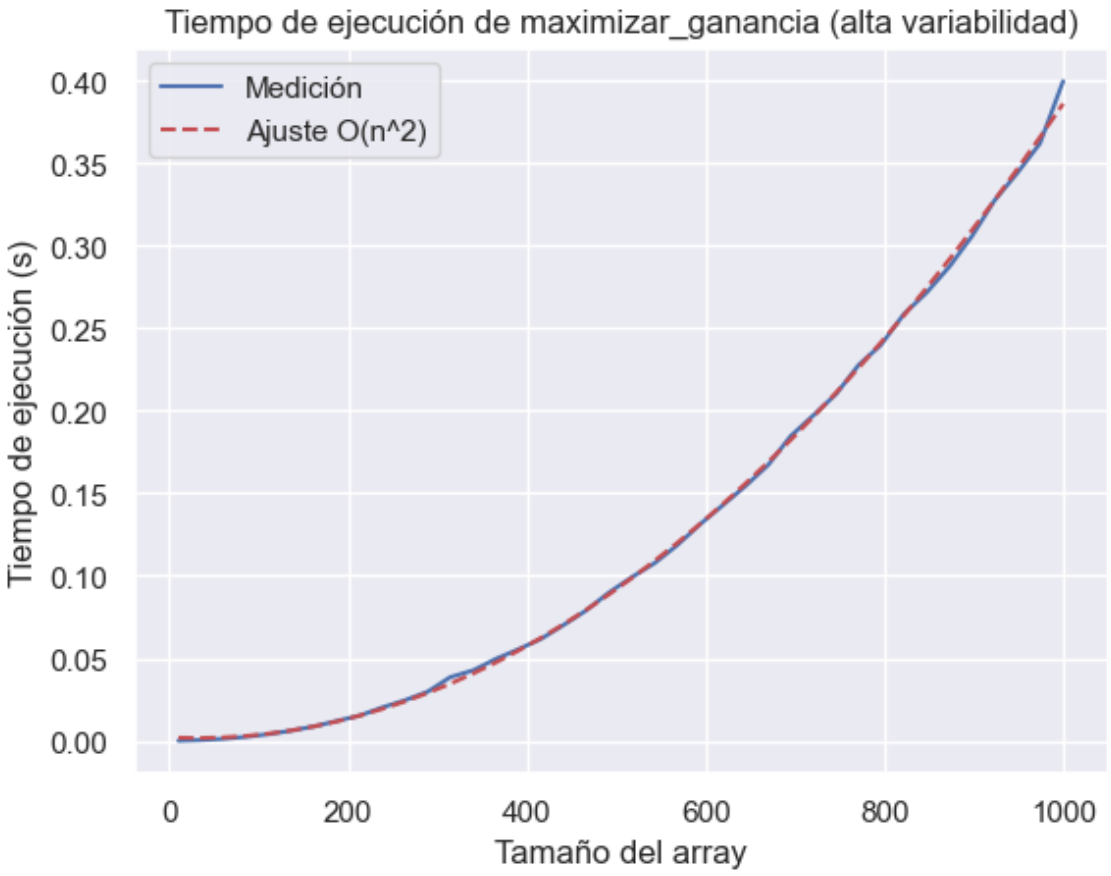
Ahora vamos a realizar un análisis extra para observar si la variabilidad de los datos que utilizamos afecta a la complejidad del algoritmo. Es decir, si cómo varían los valores de las monedas unos respecto a otros modifica el tiempo de ejecución de la función.

Para esto, vamos a repetir las mismas mediciones que hicimos antes pero con datasets de prueba distintos. Tendremos un dataset con alta variabilidad de los valores de las monedas, y otro con baja variabilidad.

5.7.1. Alta variabilidad

En este caso utilizamos arrays de monedas cuyos valores varían aleatoriamente entre 0 y 10.000.000, para testear una alta variabilidad de valores.

A continuación podemos ver el resultado del ajuste por cuadrados mínimos:



Podemos observar que el ajuste sigue siendo muy bueno, y el error cuadrático total es un poco mayor pero similar a nuestras mediciones anteriores:

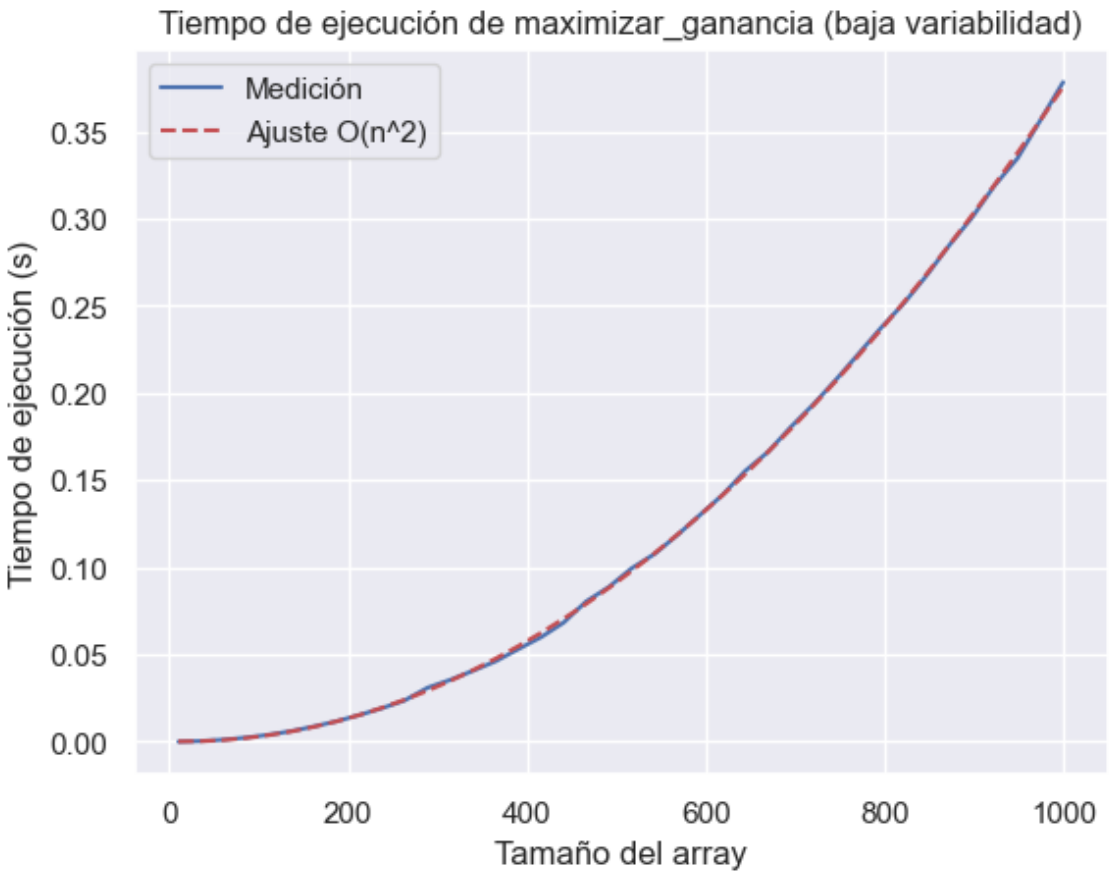
- Error cuadrático total:

  0,00030495440945438605

5.7.2. Baja variabilidad

En este caso utilizamos arrays de monedas cuyos valores varían aleatoriamente entre 0 y 10, para testear una baja variabilidad de valores. Como son valores enteros, significa que sólo hay 10 valores posibles para las monedas.

A continuación podemos ver el resultado del ajuste por cuadrados mínimos:



Observamos que el ajuste también es muy bueno, y el error cuadrático total es apenas menor:

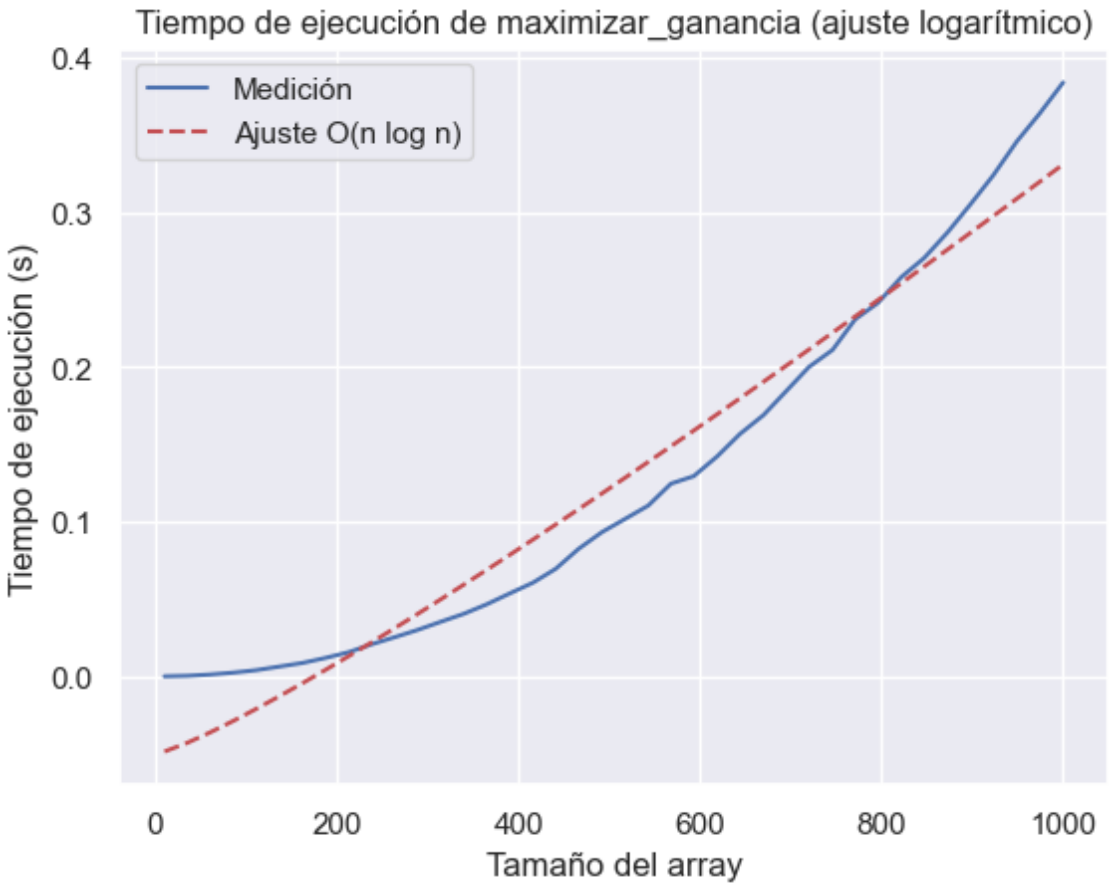
- Error cuadrático total:

$$5{,}2303129792970856e-05$$

5.8. Comparación con otras curvas

Podemos comparar nuestro algoritmo contra otras curvas para ver cómo ajusta. Nos puede servir para ver si ajusta peor con otras curvas, lo que indicaría que la curva que elegimos originalmente para ajustar es la correcta.

En el siguiente gráfico ajustamos contra la función $n \cdot log(n)$:



Vemos que el ajuste no es bueno, lo que indica que el algoritmo no es $n \cdot log(n)$ sino que es cuadrático como suponíamos. En consecuencia, también aumenta el error cuadrático total:

- Error cuadrático total:

  0,025122474710382037

Si probáramos con tamaños de entrada más grandes, las diferencias serían más notorias y los errores serían más importantes.

5.9. Resultados del análisis

Basándonos en nuestro análisis teórico de la complejidad temporal, y en este análisis empírico, podemos afirmar categóricamente que nuestro algoritmo de programación dinámica es cuadrático, de complejidad $\mathcal{O}(n^2)$.

Además, por nuestro estudio de la variabilidad de los valores de las monedas, podemos sostener que la complejidad no se ve afectada por las diferencias en la variabilidad.

6. Conclusiones

Como conclusión a este trabajo práctico, queríamos destacar las bondades de la programación dinámica para poder encontrar la solución al problema planteado. Hallar un algoritmo que maximice el puntaje de Sophia no hubiera sido posible con un enfoque greedy. Por eso creemos que fue muy importante nuestro enfoque inicial, explorando las decisiones que Sophia debe tomar en cada turno considerando las elecciones futuras de Mateo. Analizar los casos base y luego los subproblemas de tamaño chico nos permitieron hallar una ecuación de recurrencia mediante el enfoque bottom-up. Haberla encontrado con este enfoque nos trajo además el beneficio de un código muy legible y fácil de entender. Además, gracias a nuestras demostraciones teóricas y empíricas pudimos confirmar la complejidad temporal y espacial de nuestra solución, además de asegurar que Sophia siempre va a maximizar su puntaje con este algoritmo. Este análisis también nos permitió demostrar que el algoritmo es sensible al tamaño de la entrada, pero no así a la variabilidad en los valores de las monedas.