

## 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

<b>1. Ecuación de recurrencia</b>	<b>4</b>
<b>2. Demostración del Algoritmo</b>	<b>4</b>
<b>3. Algoritmo planteado y complejidad</b>	<b>5</b>
3.1. Variabilidad . . . . .	6
<b>4. Ejemplos de ejecución</b>	<b>6</b>
<b>5. Medición empírica</b>	<b>9</b>
<b>6. Conclusiones</b>	<b>10</b>

## 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  $m_1, m_2, \dots, m_n$ , 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. 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:

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

Siendo

$$K_1 = \text{monedas}[\text{inicioFila}] + T(\text{monedas}, S(\text{monedas}, \text{inicioFila} + 1, \text{finFila}))$$

$$K_2 = \text{monedas}[\text{finFila}] + T(\text{monedas}, S(\text{monedas}, \text{inicioFila}, \text{finFila} - 1))$$

donde

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

**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  $T(\text{monedas}, \text{inicioFila}, \text{finFila})$  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  $S(\text{monedas}, \text{inicioFila}, \text{finFila})$ , que son las dos posibles decisiones que puede tomar mateo al elegir una moneda (recordemos que Mateo sigue una regla estricta de tomar la moneda de mayor valor entre la primera y la última).

### 2. Demostración del Algoritmo

En un problema que involucra programacion dinamica, para hallar la solución óptima, o sea que nos de el **máximo valor acumulado posible**, se debe cumplir 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 alcanzar la solución óptima.
3. Se debe tomar una decisión sobre cuál subproblema o combinación de subproblemas proporciona el valor máximo.

Vamos a demostrar entonces que nuestro algoritmo siempre encuentra la maxima ganancia que puede obtener Sophia:

- Nuestro caso base es el siguiente: Si no hay monedas, Sophia tendría ganancia cero ya que no habría monedas para agarrar.

- Un subproblema consiste en determinar la ganancia máxima que Sophia puede obtener en un intervalo de monedas dado por los índices  $i$  y  $j$  en el arreglo de monedas (donde  $i$  representa el índice de la primera moneda, y  $j$  el de la última). Para cada subproblema, Sophia debe tomar la decisión de sí:
  - Agarrar la primera moneda  $m_i$ , lo que lleva a resolver el subproblema de las monedas entre los índices  $i + 2$  y  $j$  (si Mateo también agarró la primera moneda) o  $i + 1$  y  $j - 1$  (si Mateo agarró la última moneda).
  - Agarrar la última moneda  $m_j$ , lo que lleva a resolver el subproblema de las monedas entre los índices  $i + 1$  y  $j - 1$  (si Mateo agarró la primera moneda) o  $i$  y  $j - 2$  (si Mateo también agarró la última moneda).
- Como se observa en el ítem anterior, nuestros subproblemas se vuelven cada vez más pequeños, hasta eventualmente alcanzar el caso base (enfoque Top Down). A medida que vamos resolviendo estos subproblemas, memorizamos sus resultados para no tener que recalcularlos. Al considerar todas las posibles decisiones que puede tomar Sophia (y las que va a tomar Mateo en consecuencia), hemos considerado (y memorizado) todas las soluciones a los subproblemas posibles, y al combinarlos hallamos la solución óptima para nuestro problema original.
- Es decir, dado que cada subproblema se resuelve de forma óptima y se usan esas soluciones para construir la solución final, el algoritmo garantiza que dicha solución será la ganancia máxima que Sophia puede obtener.

### 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 función 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:

```
1 def juego_monedas(arr):
2     memory = {}
3     return _juego_monedas(arr, 0, len(arr), memory), memory
4
5 def _juego_monedas(arr, inicio = 0, fin = 0, memory = {}):
6     if inicio >= fin:
7         return 0
8
9     key = (inicio, fin)
10    if key in memory:
11        return memory[key]
12
13    primer_moneda = arr[inicio]
14    ultima_moneda = arr[fin-1]
15
16    decision_mateo_1 = decision_mateo(arr, inicio + 1, fin)
17    decision_mateo_2 = decision_mateo(arr, inicio, fin-1)
18
19    ganancia = max(primer_moneda + _juego_monedas(arr, decision_mateo_1[0],
20    decision_mateo_1[1], memory),
```

```
20         ultima_moneda + _juego_monedas(arr, decision_mateo_2[0],  
21         decision_mateo_2[1], memory))  
22     memory[key] = ganancia  
23  
24     return ganancia  
25  
26 def decision_mateo(arr, inicio = 0, fin = 0):  
27     if arr[inicio] > arr[fin - 1]:  
28         return (inicio + 1, fin)  
29     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 solución de los subproblemas, encontramos la solución del problema original, que resulta en la máxima ganancia que puede obtener Sophia.

La complejidad temporal de este algoritmo en principio sería exponencial  $O(2^n)$ , ya que en cada llamado recursivo se hacen dos llamados recursivos más, y así sucesivamente. Sin embargo, al utilizar memoria para guardar los resultados de los subproblemas, evitamos tener que recalcularlos, reduciendo la complejidad a cuadrática  $O(n^2)$ . Esta complejidad surge de que la cantidad de subproblemas a resolver es  $n^2$ , ya que cada subproblema está 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 simultáneo.

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

**NOTA:** En este análisis no se considera el algoritmo para la reconstrucción de la solución.

### 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 multiples ejemplos de ejecución dentro del directorio *ejemplos* en el repositorio. Veamos algunos:

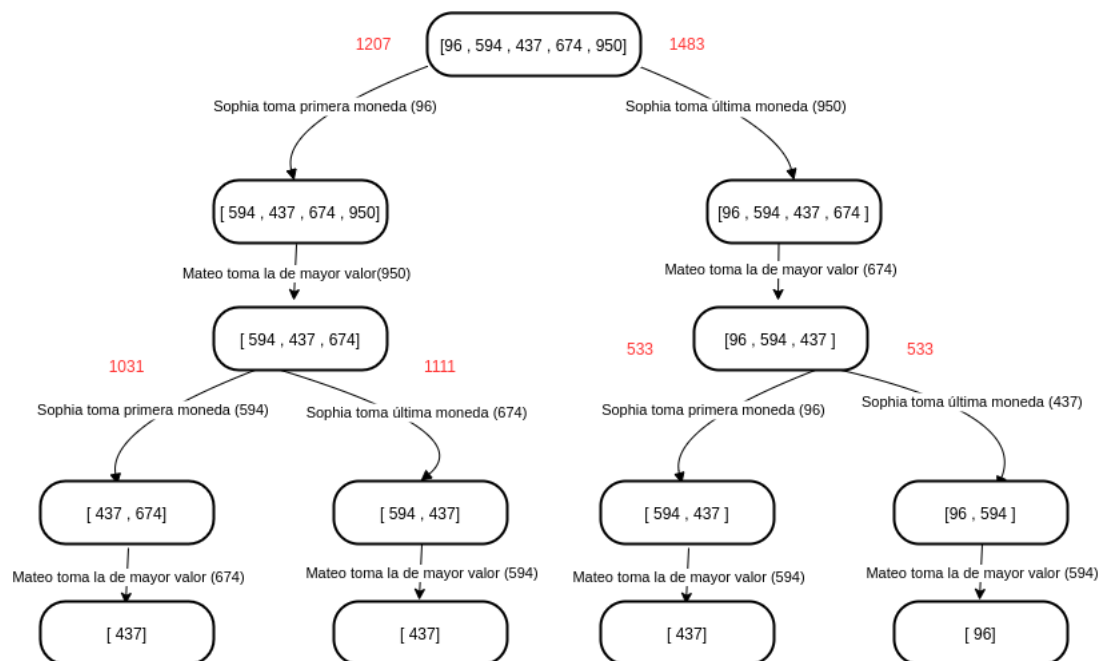
Supongamos que tenemos 5 monedas:

```
1 [96, 594, 437, 674, 950]
```

En cada turno, Sophia tiene la opción de elegir 2 monedas, la del principio o la del final. Esto implica que cada turno puede representarse como 2 opciones posibles, donde se debe elegir la opción que acumule el mayor valor.

Para esto se recorren las opciones de manera recursiva, y se va seleccionando el máximo de los valores agregados en cada turno.

De esta forma, como se puede observar en el siguiente gráfico, Sophia puede obtener un máximo de 1.483 puntos.



Veamos lo que obtenemos al ejecutar en código el ejemplo que acabamos de ver:

```
> python3 main.py ./ejemplos/5.txt
Ganancia Sophia: 1483
Ganancia Mateo: 1268

Movimientos:

Sophia debe agarrar la ultima (950)
Mateo agarra la ultima (674)
Sophia debe agarrar la primera (96)
Mateo agarra la primera (594)
Sophia debe agarrar la primera (437)
```

Veamos la ejecución de otros ejemplos brindados por la cátedra:

Este ejemplo es con 10 monedas:

```
> python3 main.py ./ejemplos/10.txt
Ganancia Sophia: 2338
Ganancia Mateo: 1780

Movimientos:

Sophia debe agarrar la ultima (145)
Mateo agarra la primera (520)
Sophia debe agarrar la primera (781)
Mateo agarra la primera (334)
Sophia debe agarrar la primera (568)
Mateo agarra la primera (706)
Sophia debe agarrar la primera (362)
Mateo agarra la primera (201)
Sophia debe agarrar la primera (482)
Mateo agarra la ultima (19)
```

Este ejemplo es con 25 monedas:

```
> python3 main.py ./ejemplos/25.txt
Ganancia Sophia: 7491
Ganancia Mateo: 6523

Movimientos:

Sophia debe agarrar la primera (704)
Mateo agarra la primera (752)
Sophia debe agarrar la primera (956)
Mateo agarra la primera (874)
Sophia debe agarrar la primera (790)
Mateo agarra la ultima (361)
Sophia debe agarrar la ultima (539)
Mateo agarra la ultima (323)
Sophia debe agarrar la ultima (623)
Mateo agarra la ultima (935)
Sophia debe agarrar la ultima (30)
Mateo agarra la ultima (390)
Sophia debe agarrar la ultima (367)
Mateo agarra la ultima (172)
Sophia debe agarrar la ultima (983)
Mateo agarra la ultima (427)
Sophia debe agarrar la ultima (266)
Mateo agarra la ultima (345)
Sophia debe agarrar la primera (157)
Mateo agarra la ultima (903)
Sophia debe agarrar la ultima (921)
Mateo agarra la ultima (636)
Sophia debe agarrar la ultima (931)
Mateo agarra la primera (405)
Sophia debe agarrar la primera (224)
```

Como podemos observar, los resultados obtenidos coinciden con los *Resultados esperados* compartidos por la cátedra.

Ademas de estos ejemplos, se pueden encontrar otros en el directorio *ejemplos* del repositorio. Los mismos pueden ser ejecutados con el comando `pytest -v -s` en la terminal.

```
riedel@riedel-ThinkBook-14-G4-IAP:~/Documents/fiuba/TDA/TDA-TP2$ pytest -v -s
===== test session starts =====
platform linux -- Python 3.12.3, pytest-7.4.4, pluggy-1.4.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/riedel/Documents/fiuba/TDA/TDA-TP2
collected 11 items

test_main.py::test_4_monedas PASSED
test_main.py::test_5_monedas PASSED
test_main.py::test_10_monedas PASSED
test_main.py::test_20_monedas PASSED
test_main.py::test_25_monedas PASSED
test_main.py::test_50_monedas PASSED
test_main.py::test_100_monedas PASSED
test_main.py::test_1000_monedas PASSED
test_main.py::test_2000_monedas PASSED
test_main.py::test_5000_monedas PASSED
test_main.py::test_10000_monedas PASSED

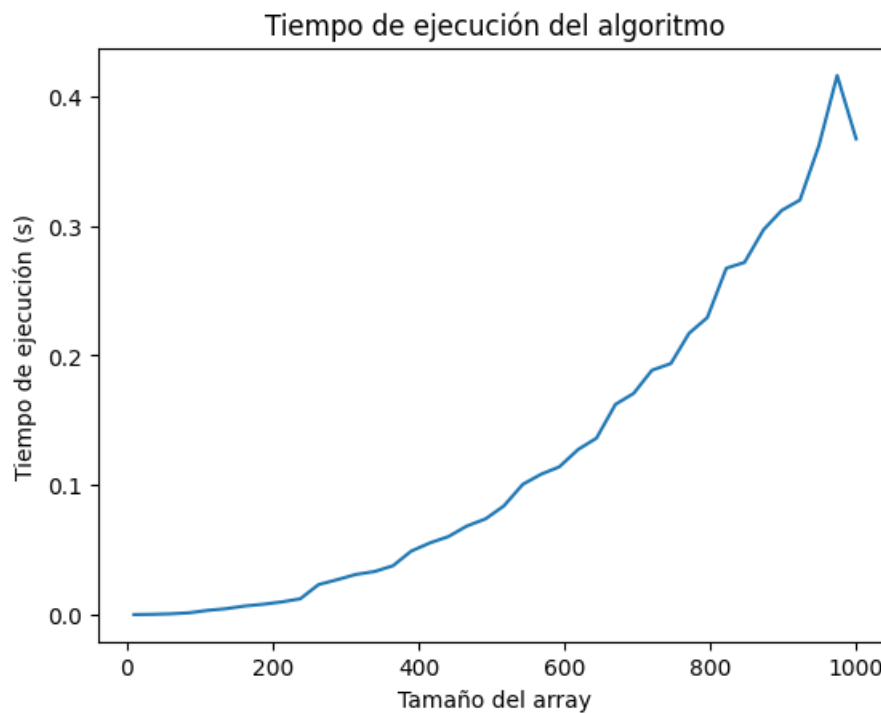
===== 11 passed in 40.65s =====
riedel@riedel-ThinkBook-14-G4-IAP:~/Documents/fiuba/TDA/TDA-TP2$
```



## 5. Medición empírica

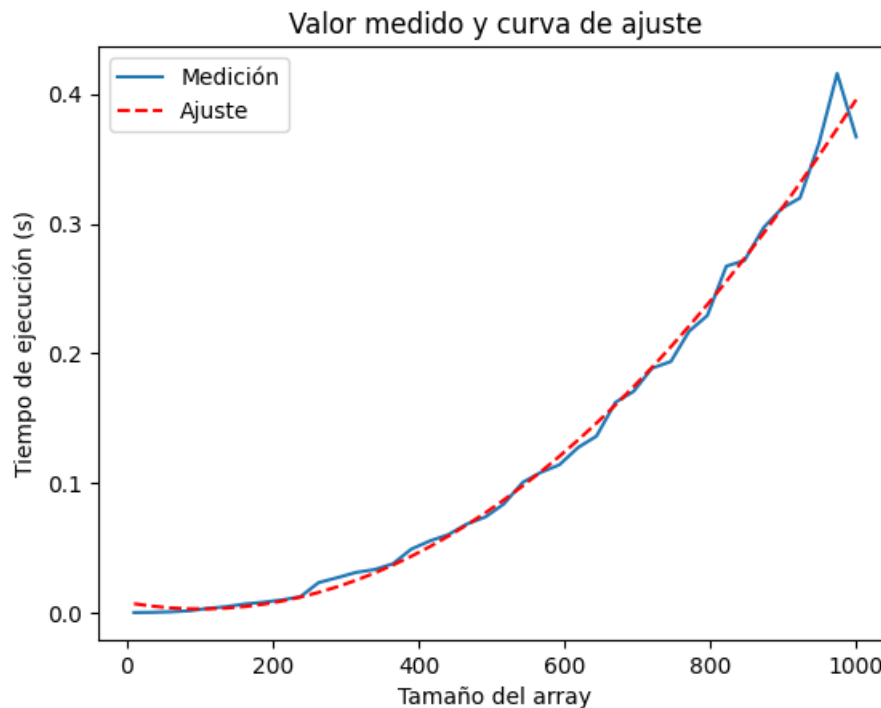
Para comprobar empíricamente la complejidad  $O(n^2)$  del algoritmo, se decidió ejecutar el mismo con distintos tamaños de entrada y medir el tiempo de ejecución. Se generaron muestras de tamaño  $n$ , las cuales varían desde 10 hasta 1000.

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



A simple vista se puede observar un crecimiento cuadrático. 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 cuadrático  $O(n^2)$ .



## 6. Conclusiones

Pudimos observar y verificar lo siguiente:

- Utilizando la técnica de programación dinámica con memorización, pudimos reducir la complejidad de un algoritmo exponencial a uno cuadrático, al no tener que recalcular problemas anteriormente resueltos.
- Tras realizar un análisis empírico, pudimos confirmar que efectivamente la complejidad de nuestro algoritmo se vio beneficiada por la técnica de programación dinámica.
- La variabilidad de los valores de las monedas **no** afecta en el tiempo de ejecución del algoritmo.

En conclusión, el presente trabajo permitió afianzar los conocimientos adquiridos en la materia de una manera práctica, donde desarrollamos un algoritmo con programación dinámica para resolver el problema planteado, con una complejidad cuadrática, obteniendo de esta manera la ganancia máxima que puede adquirir Sophia.