.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

# Trabajo Práctico 2
# Aplicación de programación dinámica

14 de octubre de 2024

Inegrantes:
Noelia Salvatierra 100116
Fabiola V. Romero P. 107099
Franco Macke 105974

# 1. Análisis del problema

El enunciado tiene el mismo problema a resolver que tuvimos antes:

- Sophia y Mateo juegan un juego de monedas, donde en cada turno, uno de los dos tiene que elegir una moneda, ya sea la primera o la última de la fila de $n$ monedas.
- Al elegir, se remueve de la fila, y le toca luego al otro jugador, quien debe elegir otra moneda siguiendo la misma regla. Se siguen agarrando monedas hasta que no quede ninguna.
- Quien gane sería quien tenga el mayor valor acumulado.

Nosotros queremos darle a Sophia un algoritmo por programación dinámica que use una ecuación de recurrencia, el cual asegure que siempre gane ella (salvo casos excepcionales).

El funcionamiento completo y detalle paso a paso se especifican en sección 3 y la demostración de que la ecuación de recurrencia planteada en efecto nos lleva a obtener el máximo valor acumulado posible está en sección 2.

Nuestras consideraciones al plantear un algoritmo por progamación dinámica para resolver el problema fueron:

- Sophia siempre empieza jugando, por lo que agarra la primera moneda.
- Sophia no sabe que moneda agarrar. Para eso está el algoritmo que le vamos a proponer, el cual le dará las indicaciones de que moneda agarrar para ganar y con esto, tomará la decisión.
- Se considera que Mateo juega con un algoritmo Greedy, (que lo aprendió de Sophia), el cual calcula el máximo valor entre la primera y última moneda.

Se puede observar que el problema es el mismo pero con una dificultad extra. ¡Ahora Mateo sabe Greedy y nuestro algoritmo tiene que tener en cuenta eso y ANTICIPARSE a los juegos de Mateo para poder ganarle!

Acá solo mostraremos las indicaciones del algoritmo para Sophia:

Por ejemplo, si se tiene una fila de monedas de largo 4:

$$[1,\ 2,\ 20,\ 5]$$

Empieza Sophia agarrando una moneda. Tiene disponibles la primera y la última, en este caso [1, 5]. El algoritmo le recomendará a Sophia que tome la moneda 1. Sophia la agarra, teniendo un acumulado de 5. Mateo tiene disponibles ahora [2, 5] y como juega "Greedy", toma el 5, acumulando 5.

Siguiente turno:

$$[2,\ 20]$$

- Moneda que debe agarrar Sophia, según el algoritmo: 20.
- Moneda que agarra Mateo: Como no le queda otra, toma el 2.
- Acumulado de Sophia: $1 + 20 = 21$.
- Acumulado de Mateo: $5 + 2 = 7$.

Reconstruyendo las indicaciones para que Sophia gane:

Sophia debe agarrar 1; Mateo agarra 5; Sophia debe agarrar 20; Mateo agarra 2.

Podemos ver que el algoritmo funciona (para este caso), ya que el acumulado de Sophia es mayor que Mateo, aun así perdiendo en el primer turno, al final termina ganando.

**Observación**: No siempre ganará, ya que por ejemplo, si tenemos las monedas $[1, 10, 5]$, no importa lo que haga Sophia, Mateo ganará.

## 2. Demostración

Considerando los óptimos de Mateo después que Sophia haya agarrado su moneda, la ecuación de recurrencia queda:

$\acute{O}ptimo(izq, der) = máx$

$$\begin{cases} \text{Si Sophia elige la moneda izquierda: monedas(izq)} + \begin{cases} \text{Si monedas(izq+1)} > \text{monedas(der): Optimo(izq+2,der)} \\ \text{Caso contrario: Optimo(izq+1,der-1)} \end{cases} \\ \text{Si no, se elige derecha: monedas(der)} + \begin{cases} \text{Si monedas(izq)} > \text{monedas(der1): Optimo(izq+1,der1)} \\ \text{Caso contrario: Optimo(izq,der-2)} \end{cases} \end{cases}$$

Observar que en esta ecuación se simula un juego completo, anticipándose a los óptimos de Mateo (se explicará como se usan los últimos en sección 3).

Demostraremos que esta ecuación llega a la solución óptima, por el principio de inducción matemático y usando el método directo:

Sea $P(n)$ una proposición, que en nuestro caso es $\acute{O}ptimo(izq,\ der)$ con $izq$ y $der$ números naturales sin el 0.

**Paso base**: Una moneda con $\acute{O}ptimo(izq_0,\ der_0)$ nuestro $P(n_0)$: Si solo hay una moneda disponible, es decir, $izq = der$ entonces: $\acute{O}ptimo(izq_0,\ der_0) = monedas[izq]$ Como Sophia tiene una sola opción para elegir, va a tomar esa moneda, y su máximo es el valor de esa moneda y será el valor acumulado máximo total, ya que Mateo no agarra monedas y su acumulado es 0.

**Paso inductivo**: Sea la hipótesis VERDADERA: Los subproblemas (sublistas de la lista monedas) con tamaño menor o igual a $n$ (tamaño sublistas) siempre devuelven un $\acute{O}ptimo(izq,\ der)$. Es decir que para un rango $(izq,\ der)$ donde $der - izq$ menor que $n$, $\acute{O}ptimo$ devuelve el máximo valor que Sophia puede acumular.

Necesitamos demostrar la tesis, o sea que para el caso $der-izq$ menor que $n+1$ es VERDADERO TAMBIÉN

Consideremos dos elecciones posibles de Sophia:

1. Sophia elige la moneda en $izq$:
   - Su ganancia es $monedas[izq]$.
   - Mateo elegirá la moneda que maximiza su ganancia. Entonces, las opciones que quedan son $\acute{O}ptimo(izq + 1,\ der)$ u $\acute{O}ptimo(izq + 2,\ der)$ dependiendo de qué moneda elija Mateo.
2. Sophia elige la moneda en $der$:
   - Su ganancia es $monedas[der]$.
   - Mateo elegirá entre $\acute{O}ptimo(izq,\ der - 1)$ u $\acute{O}ptimo(izq,\ der - 2)$

Redefiniendo el $\acute{O}ptimo$:
$\acute{O}ptimo(izq,\ der) = máx(monedas[izq] + ganancia\ de\ Sophia\ después\ de\ la\ elección\ de\ Mateo,\ monedas[der] + ganancia\ de\ Sophia\ después\ de\ la\ elección\ de\ Mateo)$

Viendo esto, si Sophia elige $monedas[izq]$, su ganancia total es:

$$monedas[izq] + max \begin{cases} \acute{O}ptimo(izq + 2,\ der) \text{ (si Mateo elige } izq + 1) \\ \acute{O}ptimo(izq + 1,\ der - 1) \text{ (si Mateo elige } der) \end{cases}$$

Si Sophia elige $monedas[der]$:

$$monedas[der] + max \begin{cases} \acute{O}ptimo(izq + 1,\ der - 1) \text{ (si Mateo elige } izq) \\ \acute{O}ptimo(izq,\ der - 2) \text{ (si Mateo elige } der - 1) \end{cases}$$

Considerando la hipótesis como verdadera, (los óptimos que están dentro de los *máx* son óptimos de los subproblemas de tamaño menor que $n$, que ya los tenemos calculados anteriormente y los usamos directo, sabiendo que son óptimos válidos), concluimos que la tesis es verdadera para tamaños $n+1$, y probamos que la ecuación de recurrencia encuentra el valor máximo acumulado para tamaños $n+1$ de lista de monedas.

# 3. Algoritmo por Programación Dinámica

A continuación se muestra el algoritmo planteado para la solución del problema (*reconstruir_solucion*() la mostraremos en otro apartado):

```
def ecuacion_recurrencia(optimo, monedas, izq, der):

    if monedas[izq + 1] > monedas[der]:
        eleccion_izq = optimo[izq + 2][der] if izq + 2 <= der else 0
    else:
        eleccion_izq = optimo[izq + 1][der - 1] if izq + 1 <= der - 1 else 0

    if monedas[izq] > monedas[der - 1]:
        eleccion_der = optimo[izq + 1][der - 1] if izq + 1 <= der - 1 else 0
    else:
        eleccion_der = optimo[izq][der - 2] if izq <= der - 2 else 0

    return max(monedas[izq] + eleccion_izq, monedas[der] + eleccion_der)


/S1/ def jugar(monedas: list):

    optimo = [[0 for _ in range(len(monedas))] for _ in range(len(monedas))]

    /S2/ for i in range(len(monedas)):
        optimo[i][i] = monedas[i]

   /S3/ for cantidad in range(2, len(monedas) + 1):
        /S4/ for i in range(len(monedas) - cantidad + 1):
            j = i + cantidad - 1
            optimo[i][j] = ecuacion_recurrencia(optimo, monedas, i, j)


    /S5/ return reconstruir_solucion(optimo, monedas)
```

## 3.1. Funcionamiento del algoritmo

Teniendo en cuenta que las únicas monedas que pueden ser seleccionadas durante los turnos son los extremos, que Sophia siempre empieza y que Mateo juega con un algoritmo Greedy:

1. Función $jugar(monedas : list)$:
   - Inicializa una matriz *óptimo* donde $óptimo[i][j]$ almacenará el valor máximo que puede obtener el jugador (Sophia) al jugar con las monedas desde el índice $i$ hasta el índice $j$.
   - Se llena inicialmente con ceros, y luego se establece que cuando hay solo una moneda, el valor máximo que puede obtener es el valor de esa moneda: $óptimo[i][i] = monedas[i]$ (o sea, en la diagonal, tendremos la lista de monedas)
   - A continuación, se resuelven subproblemas de mayor longitud (de 2 *a n monedas*) usando la función $ecuacion\_recurrencia(óptimo,\ monedas,\ izq,\ der)$.
2. Función $ecuacion\_recurrencia(optimo, monedas, izq, der)$:
   - Recibe la matriz *óptimo*, la lista de monedas, y los índices *izq* y *der* que indican el rango de monedas en consideración.

- Retorna la mejor opción de Sophia al elegir entre la primera moneda ($monedas[izq]$) y la última ($monedas[der]$), considerando cómo Mateo elegirá después.
- Utiliza la lógica de programación dinámica para calcular el máximo valor que puede obtener Sophia en función de las elecciones futuras de Mateo, usando *memorization* de las soluciones de los subproblemas y con todas estas, obtendremos la solución al problema más grande que es la lista de monedas y minimizando el valor que Mateo puede obtener en su próximo turno.

A continuación se hace una explicación más detallada de $ecuacion\_recurrencia(optimo, monedas, izq, der)$, haciéndose una simulación de un juego completo:

- Tomar la moneda en la posición izquierda ($monedas[izq]$).
- Tomar la moneda en la posición derecha ($monedas[der]$).

Mateo, en su turno, siempre elige la moneda más grande de los extremos restantes.

Si Sophia toma la moneda izquierda ($monedas[izq]$) (o sea, se entra en la condición $if$):

```
if monedas[izq + 1] > monedas[der]:
    eleccion_izq = optimo[izq + 2][der] if izq + 2 <= der else 0
else:
    eleccion_izq = optimo[izq + 1][der - 1] if izq + 1 <= der - 1 else 0
```

Aquí Sophia elige la moneda en la posición $izq$, y Mateo jugaría después.

Mateo elige según Greedy y luego "juega"Sophia:

- Si la moneda en $monedas[izq + 1]$ es mayor que la moneda en $monedas[der]$, Mateo tomará la moneda en $izq + 1$.

Después de eso, quedará la subcadena de monedas desde $izq + 2$ hasta $der$ para Sophia. Así, el valor acumulado que Sophia puede obtener será el que está en $óptimo[izq+2][der]$. Caso contrario, su óptimo será de $óptimo[izq + 1][der - 1]$.

- Si la moneda en $monedas[der]$ es mayor o igual que la moneda en $monedas[izq + 1]$, Mateo tomará la moneda en $der$.

Luego, la subcadena que queda es desde $izq + 1$ hasta $der - 1$. El valor que Sophia podría acumular en este caso sería $ótimo[izq + 1][der - 1]$.
Caso contrario, su óptimo será $óptimo[izq][der - 2]$.

Condición del límite de índices: Si los índices de las subcadenas se salen de los límites de la lista (es decir, si ya no hay monedas disponibles en los extremos), el valor será 0.

Ahora, si Sophia toma la moneda $der$ ($monedas[der]$) (o sea, entra en la condición $if$):

```
if monedas[izq] > monedas[der - 1]:
    eleccion_der = optimo[izq + 1][der - 1] if izq + 1 <= der - 1 else 0
else:
    eleccion_der = optimo[izq][der - 2] if izq <= der - 2 else 0
```

Se tiene el mismo análisis de la condición if anterior para este caso, la diferencia es que cambiarán los óptimos (valores).

Finalmente, Sophia elige la opción que maximiza su valor:

- Si ella toma la moneda de la izquierda ($monedas[izq]$), su valor será $monedas[izq]+eleccion\_izq$.
- Si ella toma la moneda de la derecha ($monedas[der]$), su valor será $monedas[der]+eleccion\_der$.

El máximo de ambos valores es lo que devuelve la función, ya que Sophia quiere maximizar su ganancia.

Ejemplo con monedas $[1, 2, 20, 5]$ (simularemos un juego completo):

En funcion $jugar(monedas : list)$:

Se recibe monedas, se inicializa óptimo (matriz de 4x4 para este caso) y se setean los valores de la diagonal de la matriz dentro del bucle:

- $óptimo[0][0] = 1$
- $óptimo[1][1] = 2$
- $óptimo[2][2] = 20$
- $óptimo[3][3] = 5$
- El resto de las celdas tienen valores iguales a 0.

Luego resolveremos los subproblemas, entrando en los dos bucles siguientes.

Para cantidad o subproblema de $tamaño = 2$ :

- Para $i = 0$, $j = 1$ (monedas $[1, 2]$):
- $monedas[izq] = 1$, $monedas[izq + 1] = 2$, $monedas[der] = 2$
- Elección Izquierda:
  - $monedas[1](2) > monedas[1](2)$: Es Falso
  - Entonces: $eleccionIzq = optimo[1][0]$ $(0)$ porque $izq + 1$ no es menor o igual a $der - 1$ (se pasó del límite)
- Elección Derecha:
  - $monedas[0](1) > monedas[0](1)$: Es Falso
  - Entonces: $eleccionDer = óptimo[0][-1]$ $(0)$ porque $izq$ no es menor o igual a $der - 2$.
  - $óptimo[0][1] = 2$.

Haciendo el mismo análisis para $i = 1$, $j = 2$ (monedas $[2, 20]$) y para $i = 2$, $j = 3$ (monedas $[20, 5]$), se resuelven todos los subproblemas de $len = 2$ y luego óptimo queda:

$$[[1,\ 2,\ 0,\ 0],\ [0,\ 2,\ 20,\ 0],\ [0,\ 0,\ 20,\ 20],\ [0,\ 0,\ 0,\ 5]]$$

Para subproblemas de $tamaño = 3$: Haciendo el mismo análisis, la matriz de óptimos nos queda:

$$[[1,\ 2,\ 21,\ 0],\ [0,\ 2,\ 20,\ 7],\ [0,\ 0,\ 20,\ 20],\ [0,\ 0,\ 0,\ 5]]$$

Y para subproblemas de $tamaño = 4$, (mismo análisis que antes):

$$óptimo = [[1,\ 2,\ 21,\ 21],\ [0,\ 2,\ 20,\ 7],\ [0,\ 0,\ 20,\ 20],\ [0,\ 0,\ 0,\ 5]]$$

Luego el valor máximo acumulado está en $óptimo[0][len(monedas) - 1] = óptimo[0][3]$ y es 21, esto es lo máximo que puede acumular Sophia, ganándole a Mateo.

Función $reconstruir\_solucion()$ :

```python
def reconstruir_solucion(optimo, monedas):
    izq = 0
    der = len(monedas) - 1
    elecciones = []
    turno_sophia = True
    print(tabulate(optimo))

    ganancia_sophia = 0
    ganancia_mateo = 0

    import pdb
    pdb.set_trace()


    while izq <= der:
        if turno_sophia:

            if monedas[izq + 1] > monedas[der]:
                eleccion_izq = optimo[izq + 2][der] if izq + 2 <= der else 0
            else:
                eleccion_izq = optimo[izq + 1][der - 1] if izq + 1 <= der - 1 else
    0
            if monedas[izq] > monedas[der - 1]:
                eleccion_der = optimo[izq + 1][der - 1] if izq + 1 <= der - 1 else
    0
            else:
                eleccion_der = optimo[izq][der - 2] if izq <= der - 2 else 0

            if monedas[izq] + eleccion_izq == optimo[izq][der]:
                elecciones.append(f"Sophia debe agarrar la primera ({monedas[izq]})
    ")
                ganancia_sophia += monedas[izq]
                izq += 1
            elif monedas[izq] + eleccion_der == optimo[izq][der]:
                elecciones.append(f"Sophia debe agarrar la ultima ({monedas[der]})"
    )
                ganancia_sophia += monedas[der]
                der -= 1
            elif monedas[der] + eleccion_der == optimo[izq][der]:
                elecciones.append(f"Sophia debe agarrar la ultima ({monedas[der]})"
    )
                ganancia_sophia += monedas[der]
                der -= 1
            elif monedas[der] + eleccion_izq == optimo[izq][der]:
                elecciones.append(f"Sophia debe agarrar la primera ({monedas[izq]})
    ")
                ganancia_sophia += monedas[izq]
                izq += 1

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

        turno_sophia = not turno_sophia


    print(tabulate(dict(Persona=["Sophia", "Mateo"], Ganancia=[ganancia_sophia,
    ganancia_mateo]), headers="keys"))

    return elecciones
```

- Recibe la matriz de óptimos ya calculada anteriormente y la lista de monedas.
- Su función es, justamente, saber que monedas tiene que tomar Sophia para ganar (obtener

el máximo).

- Devuelve una lista de las monedas que Sophia tiene que tomar, y las que tomó Mateo.

La manera que se obtienen las elecciones es la siguiente:

Como la matriz de óptimos, tiene todos los óptimos que Sophia puede tener según las jugadas de Mateo y sabemos que el máximo valor acumulado posible está en *óptimo*[0][3], entonces vuelvo para atrás, con la ecuación de recurrencia (por eso está en el código de *reconstruir_solucion*), tomo el óptimo anterior y le sumo la moneda (*izq* o *der*) para obtener el *óptimo*[0][3]. Una manera más fácil de pensarlo es:
Tengo el mejor valor (*óptimo*[0][3]), pero, de donde provine? De *izq* o *der*? Para resolver la pregunta, se suma el óptimo anterior a ese (para este caso es 20 si tomaba la moneda *izq* o 2 si tomaba la moneda *der*) más la incógnita (o sea la moneda de donde pude provenir). En el *if* de la línea 27 hasta la línea 42 se resuelve esto. Y el *else*, resuelve las elecciones de Mateo.

Podemos tener una ejecución del paso a paso con números gracias al *debbugger pdb*:

```
C:\Users\admin_noe\Downloads>python TP2.py
-  -  --  --
1  2  21  21
0  2  20   7
0  0  20  20
0  0   0   5
-  -  --  --
> c:\users\admin_noe\downloads\tp2.py(63)reconstruir_solucion()
-> while izq <= der:
(Pdb) n
> c:\users\admin_noe\downloads\tp2.py(64)reconstruir_solucion()
-> if turno_sophia:
(Pdb) n
> c:\users\admin_noe\downloads\tp2.py(66)reconstruir_solucion()
-> if monedas[izq + 1] > monedas[der]:
(Pdb) n
> c:\users\admin_noe\downloads\tp2.py(69)reconstruir_solucion()
-> eleccion_izq = optimo[izq + 1][der - 1] if izq + 1 <= der - 1 else 0
(Pdb) n
> c:\users\admin_noe\downloads\tp2.py(70)reconstruir_solucion()
-> if monedas[izq] > monedas[der - 1]:
(Pdb) n
> c:\users\admin_noe\downloads\tp2.py(73)reconstruir_solucion()
-> eleccion_der = optimo[izq][der - 2] if izq <= der - 2 else 0
(Pdb) n
> c:\users\admin_noe\downloads\tp2.py(75)reconstruir_solucion()
-> if monedas[izq] + eleccion_izq == optimo[izq][der]:
(Pdb) print(eleccion_izq)
20
(Pdb) print(eleccion_der)
2
(Pdb) n
> c:\users\admin_noe\downloads\tp2.py(76)reconstruir_solucion()
-> elecciones.append(f"Sophia debe agarrar la primera ({monedas[izq]})")
(Pdb) print(monedas[izq])
1
(Pdb) n
> c:\users\admin_noe\downloads\tp2.py(77)reconstruir_solucion()
-> ganancia_sophia += monedas[izq]
(Pdb)
```

**Observación**: Para el caso de que el *len* de monedas es 3, Sophia tiene que elegir el máximo de los 2 extremos (igualmente el algoritmo contempla este caso). Y para el caso de *len* impar...

## 3.2. Justificación de la complejidad

**Observación**: Para el análisis, vamos a aplicar las reglas del modelo de análisis computacional (bibliografía Méndez). Y no vamos a considerar el procesamiento de archivos, ni todo lo que esté por fuera de la función.

Podemos dividir a esta función, es un bloque de 5 instrucciones ejecutables: $S_1$, $S_2$, $S_3$ , $S_4$ y $S_5$, donde $S_1$ empieza en la línea 16 hasta la 18 inclusive, $S_2$ empieza desde 20 hasta 21 inclusive, $S_3$ va desde 23 y dentro de él está $S_4$, y ambos terminan en la línea 26. $S_5$ es un *return* de la línea 29. Vemos que se tiene una secuencia de instrucciones. Por ende, la regla de cálculo del tiempo de ejecución de una secuencia, se basa en aplicar el teorema de aditividad:

$$O(S_1,\ S_2,\ S_3,\ S_4) = máx(O(S_1),\ O(S_2),\ O(S_3),\ O(S_5))$$

**Observación**: no agregamos $S_4$ en el teorema de aditividad, ya que cuando estudiamos $S_3$, antes tendríamos que estudiar $S_4$.

1. Vemos que en $S_1$ hay una asignación de una variable local *óptimo*, en el cual se inicializan todas las celdas de la matriz en 0, dentro de la inicialización tenemos dos $for$ ïndependientes". El tiempo de esta inicialización es $O(n)+O(n)$, luego la suma da en total $O(n)$ (explicaremos en el siguiente apartado por qué un $for$ de estas características tiene tiempo de ejecución $0(n)$).

2. En $S_2$ tenemos un bucle $for$ y para determinar su tiempo de ejecución se estudiará el peor de los casos posibles que puede ocurrir, esto es (el caso normal, justo en este caso) cuando: $i = len(monedas) = n$. Para ello hay que determinar la cantidad de iteraciones que se hace. Haciendo una tabla de iteraciones: Vemos que la cantidad de iteraciones para $i = n$ (peor caso) es $n$, por ende el tiempo de ejecución de $S_1$ es $O(n)$.
   **Observación**: En cada iteración, inicializamos una celda, esta operación cuesta $O(1)$ (asignaciones, $if$, $else$, $returns$ tiene tiempo de ejecución $O(1)$).

| i | iteraciones |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| ...n | ...n |

3. $S_3$ también tiene un bucle, pero sabemos que cuando tenemos dos $for$ anidados, tenemos que estudiar de adentro hacia afuera. Estudiaremos antes $S_4$ entonces.

   $S_4$ tiene ADENTRO del $for$ una asignación ($j = i+cantidad-1$), y sabemos que cuesta $O(1)$, luego tenemos un llamado a la función $ecuacion\_recurrencia()$, el cual si vemos su firma, tiene todas instrucciones de asignación, $if$, $else$ y $return$ y ya sabemos que todas estas

cuestan $O(1)$, luego la función cuesta $O(1)$. Ahora, si estudiamos el $for$ interno, y seguimos el mismo criterio anterior, el peor de los casos es cuando $i = len(monedas) - cantidad + 1 = n$ (si considero la cantidad un número pequeño comparado con $len(monedas)$), luego tenemos tiempo de ejecución de $S_4 = O(n)$. Ahora, estudiaremos $S_4$, siguiendo la regla de iteraciones anidadas: El tiempo total de una instrucción dentro de un grupo de iteraciones anidadas es el tiempo de ejecución de la multiplicado por la cantidad de iteraciones de cada iteración. La cantidad de iteraciones de $S_4$ ya la calculamos en el peor caso y es $n$ y la del $for$ de $S_3$, también es $n$ ($i = len(monedas) + 1 = n$). Luego, siguiendo la regla, cantidad total de iteraciones es $n * n = n^2$, y entonces, S4 costará $O(n^2)$.

4. $S_5$ es un $return$, pero dentro de él se llama a la función $reconstruir\_solucion(..., ...)$ , entonces estudiaremos esta función: De la línea 2 hasta la 14, tenemos asignaciones y un $print$, luego sabemos que cuestan $O(1)$, luego sigue el ciclo $while$, donde recorre la lista de monedas, ya que justamente la idea de esta función era encontrar las elecciones de Sophia para poder ganar. Vemos después luego hay instrucciones $if$, $elif$, $else$ que cuestan $O(1)$. Si seguimos el criterio de 2), tenemos que calcular la cantidad de iteraciones de los $if$, $elif$ y $else$. El peor de los casos lo vamos a tener cuando $izq = der = len(monedas) - 1 = n$ que es recorrer toda la lista y luego $n$ es la cantidad de iteraciones y por ende el tiempo de ejecución es $O(n)$

Finalmente, se obtiene: $O(S_1,\ S_2,\ S_3,\ s_5) = máx(O(n),\ O(n),\ O(n^2),\ O(n))$ y por todo lo concluido, el orden de complejidad o tiempo de ejecución de $jugar(n)$ es $O(n^2 2)$.

Un caso no menos importante a observar es que para estudiar la optimalidad y los tiempos del algoritmo, jamás se tuvo en cuenta la variabilidad de los valores de las diferentes monedas, y podemos concluir que esto no afecta.

# 4. Ejemplos de ejecución

Tomaremos 2 casos particulares más, aparte de los proporcionados por la cátedra, y mostraremos las soluciones arrojadas, comprobando que obtendremos la solución óptima.

## 4.1. Ejemplo 1

Le pasamos al programa la siguiente lista:

$$\text{monedas} = [5,\ 10,\ 30,\ 1,\ 6]$$

y realizamos el seguimiento de la ejecución.

**Primera iteración**

$$\underset{\uparrow}{[5]}[10][30][1][6]\underset{\uparrow}{[10]}$$

El algoritmo le indica a Sophia que debe elegir 5 y Mateo va a tomar 10 y sus ganancias por el momento son:

```
> c:\users\admin_noe\downloads\tp2.py(76)reconstruir_solucion()
-> elecciones.append(f"Sophia debe agarrar la primera ({monedas[izq]})")
(Pdb) print(monedas[izq])
5
(Pdb) print(ganancia_sophia)
5
```

```
> c:\users\admin_noe\downloads\tp2.py(94)reconstruir_solucion()
-> elecciones.append(f"Mateo agarra la primera ({monedas[izq]})")
(Pdb) print(monedas[izq])
10
(Pdb) print(ganancia_mateo)
10
```

**Segunda iteración**

$$[30][1][6][10]$$
$$\uparrow \qquad \uparrow$$

Sophia debe elegir 30 y Mateo toma 6 y sus ganancias son:

```
> c:\users\admin_noe\downloads\tp2.py(76)reconstruir_solucion()
-> elecciones.append(f"Sophia debe agarrar la primera ({monedas[izq]})")
(Pdb) print(monedas[izq])
30
(Pdb) print(ganancia_sophia)
35
```

```
> c:\users\admin_noe\downloads\tp2.py(98)reconstruir_solucion()
-> elecciones.append(f"Mateo agarra la ultima ({monedas[der]})")
(Pdb) print(monedas[der])
6
(Pdb) print(ganancia_mateo)
16
```

**Tercera iteración**

Como solo tenemos una moneda, Sophia toma esa y finaliza el juego. Las ganancias totales son:

```
(Pdb) print(ganancia_sophia)
36
```

```
(Pdb) print(ganancia_mateo)
16
```

## 4.2. Ejemplo 2

Le pasamos al programa la siguiente lista:

$$[10, 5, 15, 90, 1, 40, 50, 9, 11, 70, 91, 20]$$

Y la salida del programa fue:

```
C:\Users\admin_noe\Downloads>python TP2.py
--  --  --  ---  --  ---  ---  ---  ---  ---  ---  ---
10  10  20  100  26  135   76  155  157  225  248  257
 0   5  15   95  95   56  145  146  156  216  247  248
 0   0  15   90  16  130   66  139  152  211  243  231
 0   0   0   90  90   91  130  141  141  152  212  243
 0   0   0    0   1   40   51   51   62  121  153  153
 0   0   0    0   0   40   50   49   61  129  152  151
 0   0   0    0   0    0   50   50   59   81  150  152
 0   0   0    0   0    0    0    9   11   79  102  111
 0   0   0    0   0    0    0    0   11   70  102  102
 0   0   0    0   0    0    0    0    0   70   91   90
 0   0   0    0   0    0    0    0    0    0   91   91
 0   0   0    0   0    0    0    0    0    0    0   20
--  --  --  ---  --  ---  ---  ---  ---  ---  ---  ---
Persona      Ganancia
---------  ----------
Sophia            257
Mateo             155
  Movimiento  Eleccion
------------  ------------------------------------
           1  Sophia debe agarrar la primera (10)
           2  Mateo agarra la ultima (20)
           3  Sophia debe agarrar la ultima (91)
           4  Mateo agarra la ultima (70)
           5  Sophia debe agarrar la ultima (11)
           6  Mateo agarra la ultima (9)
           7  Sophia debe agarrar la ultima (50)
           8  Mateo agarra la ultima (40)
           9  Sophia debe agarrar la primera (5)
          10  Mateo agarra la primera (15)
          11  Sophia debe agarrar la primera (90)
          12  Mateo agarra la ultima (1)
```

### 4.3. Caso 25.txt

Este ejemplo es uno de los dos provistos en el enunciado. La salida del programa fue:

```
Persona      Ganancia
---------  ----------
Sophia           7491
Mateo            6523
  Movimiento  Eleccion
------------  ------------------------------------
           1  Sophia debe agarrar la primera (704)
           2  Mateo agarra la primera (752)
           3  Sophia debe agarrar la primera (956)
           4  Mateo agarra la primera (874)
           5  Sophia debe agarrar la primera (790)
           6  Mateo agarra la ultima (361)
           7  Sophia debe agarrar la ultima (539)
           8  Mateo agarra la ultima (323)
           9  Sophia debe agarrar la ultima (623)
          10  Mateo agarra la ultima (935)
          11  Sophia debe agarrar la ultima (30)
          12  Mateo agarra la ultima (390)
          13  Sophia debe agarrar la ultima (367)
          14  Mateo agarra la ultima (172)
          15  Sophia debe agarrar la ultima (983)
          16  Mateo agarra la ultima (427)
          17  Sophia debe agarrar la ultima (266)
          18  Mateo agarra la ultima (345)
          19  Sophia debe agarrar la primera (157)
          20  Mateo agarra la ultima (903)
          21  Sophia debe agarrar la ultima (921)
          22  Mateo agarra la ultima (636)
          23  Sophia debe agarrar la ultima (931)
          24  Mateo agarra la primera (405)
          25  Sophia debe agarrar la primera (224)
```

## 5. Cálculo de complejidad por cuadrados mínimos

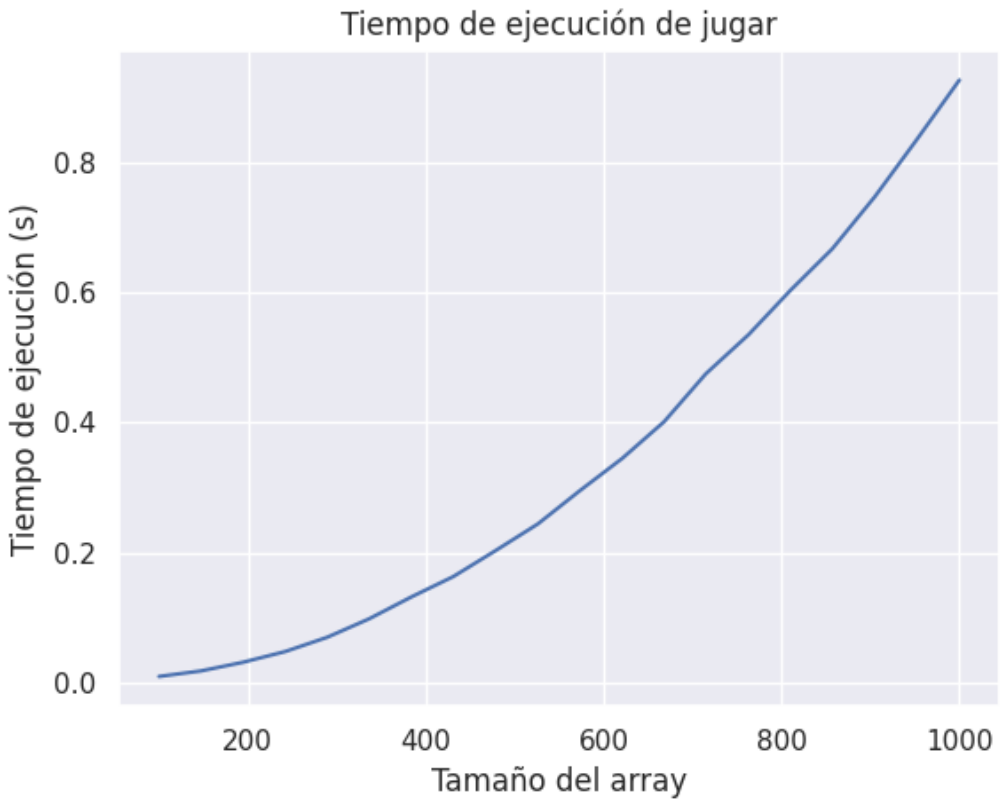
Para este caso, vamos a corroborar la complejidad calculada de manera teórica haciendo mediciones para el algoritmo formulado con programación dinámica de la función jugar, para posteriormente realizar un ajuste por cuadrados mínimos.

```python
def get_random_array(size: int):
    return np.random.randint(0, 100.000, size)

x = np.linspace(100, 1_000, 20).astype(int)

results = time_algorithm(jugar, x, lambda s: [get_random_array(s)])
```

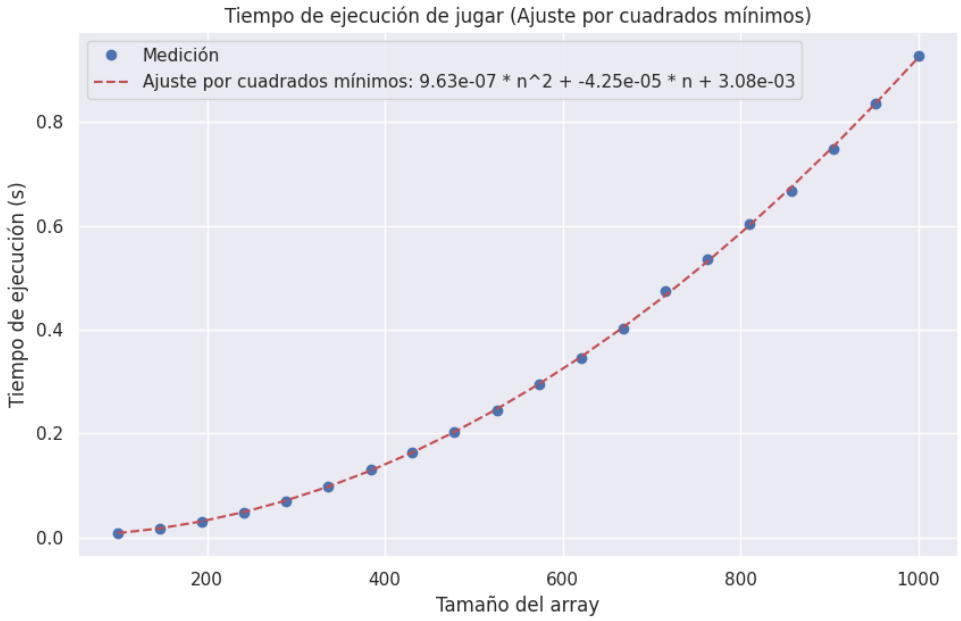
Siendo $x$ los valores de las abscisas, y el de las ordenadas: $[results[i]\ for\ i\ in\ x]$, se obtuvo:



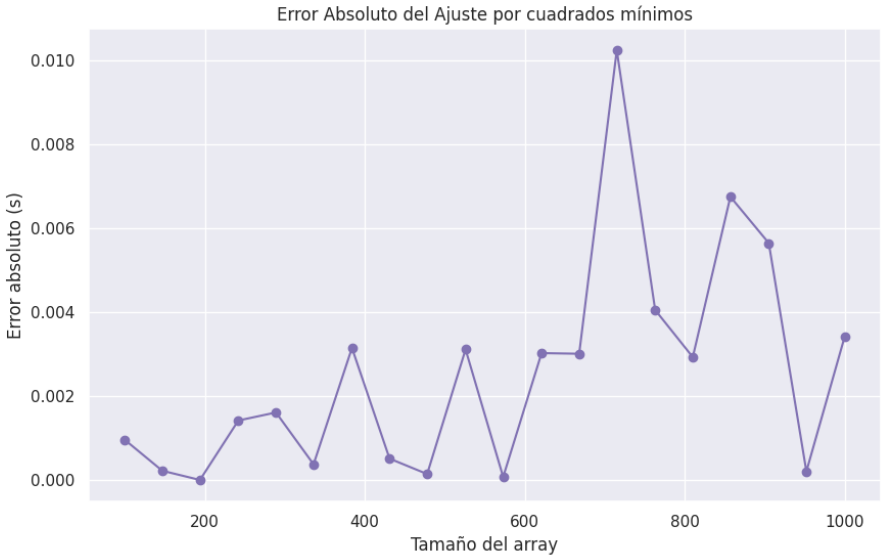

Calculando el ajuste por cuadrados mínimos (ajustamos a una recta $c_0 * n_2 + c_1 * n + c_2$) se obtuvo el siguiente error:

$$c_0 = 9{,}63 * 10^{-7}$$
$$c_1 = -4{,}25 * 10^{-5}$$
$$c_2 = 3{,}08 * 10^{-3}$$

$$\textit{Error cuadrático total} = 2{,}63 * 10^{-4}$$




Y para finalizar, graficamos el ajuste y el error para cada tamaño del array:




Se puede apreciar que para todos los tamaños del array el error de nuestro ajuste es bastante pequeño, lo cual nos lleva a concluir que nuestro algoritmo se comporta como $O(n^2)$.

# 6. Bibliografía

1. https://github.com/algoritmos-rw/tda_ejemplos/tree/main/analisis_complejidad
2. Apuntes Méndez: ejerResueltosAnalisisAlgoritmos.pdf y Análisis de Algoritmos.pdf