.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico Anual
Juegos de Hermanos

Lucas Fernandez
109250

Nicolás Franco Celano Minig
107666

Ernesto Brian Fernández Ramírez
107326

Carla Mendoza
107011

1. Solución Greedy

1.1. Análisis del Problema

Para esta parte del trabajo, el objetivo es plantear un algoritmo Greedy que obtenga la solución óptima al problema planteado.

Repasando el problema en cuestión, tenemos dos jugadores, Sophia y Mateo, y una lista de n monedas con valores diferentes y sin un orden específico. En cada turno, comenzando por Sophia, los jugadores tienen que elegir entre la primera y última moneda de la lista, eventualmente agarrando todas las monedas. Finalmente, quien tenga más valor acumulado en monedas ganará el juego.

Como el turno inicial es de Sophia, el algoritmo comienza con Sophia eligiendo la mayor de las dos monedas para si misma, y la más pequeña para su hermano Mateo. Esto se repite hasta que ya no queden monedas.

1.2. Optimalidad del Algoritmo

Desestimando el caso en el que la cantidad de monedas es par y los valores son iguales, Sophia siempre ganará porque en cada turno su ganancia es mayor que la de Mateo, ya que no solo Sophia elige la moneda mas grande, sino que Mateo se ve obligado a recibir la moneda más pequeña. Como Sophia maximiza su ganancia en cada turno y Mateo la minimiza, la sumatoria final va a dar siempre mayor para Sophia (ignorando el caso especial mencionado).

1.3. Algoritmo Planteado

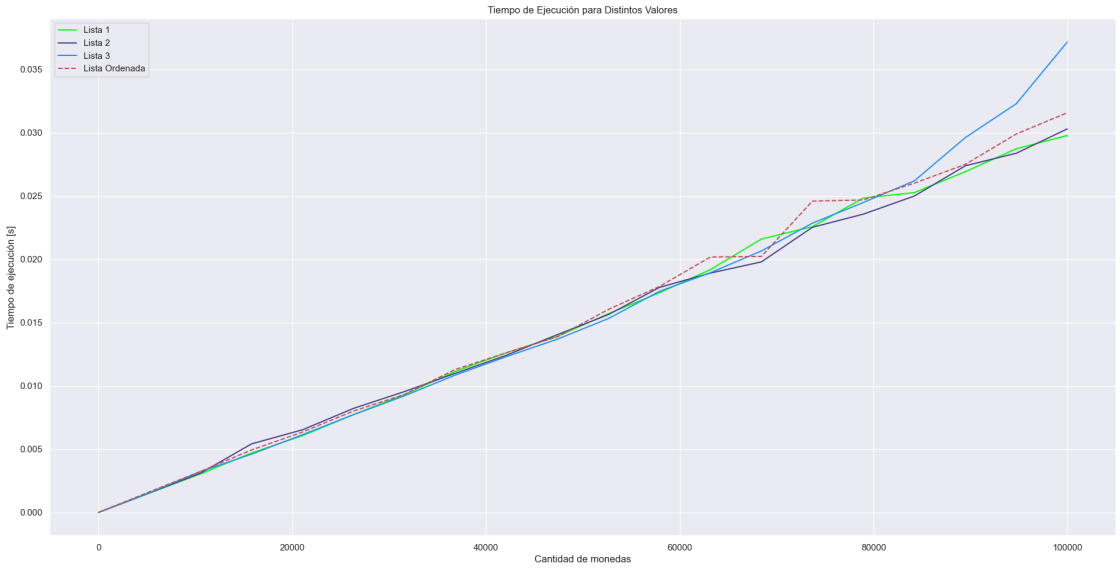
A continuación mostramos una implementación posible del algoritmo en Python

```python
from collections import deque

def elegir_sophia(monedas, decisiones):
    if monedas[0] < monedas[-1] or len(monedas)==1:
        decisiones.append(" ltima  moneda para Sophia")
        return -1
    else:
        decisiones.append("Primera moneda para Sophia")
        return 0


def elegir_mateo(monedas, decisiones):
    if monedas[0] > monedas[-1] or len(monedas)==1:
        decisiones.append(" ltima  moneda para Mateo")
        return -1
    else:
        decisiones.append("Primera moneda para Mateo")
        return 0


def monedas_greedy(monedas):
    monedas_sophia = []
    monedas_mateo = []
    decisiones = []
    monedas = deque(monedas)

    while len(monedas) > 0:
        i = elegir_sophia(monedas, decisiones)
        moneda_elegida = monedas.pop() if i<0 else monedas.popleft()
        monedas_sophia.append(moneda_elegida)
        if len(monedas) > 0:
            j = elegir_mateo(monedas, decisiones)
            moneda_elegida = monedas.pop() if j<0 else monedas.popleft()
            monedas_mateo.append(moneda_elegida)

    return decisiones, monedas_sophia, monedas_mateo
```

El algoritmo la recorre la fila de monedas de inicio a fin pasando una sola vez por cada elemento, por lo que la complejidad de la parte iterativa del algoritmo es $\mathcal{O}(n)$. Teniendo en cuenta que convertir la lista a un *deque* tiene complejidad $\mathcal{O}(n)$ y las operaciones *pop* y *popleft* son $\mathcal{O}(1)$, la complejidad total del algoritmo es de $\mathcal{O}(n)$.

1.3.1. Variabilidad del Tiempo de Ejecución

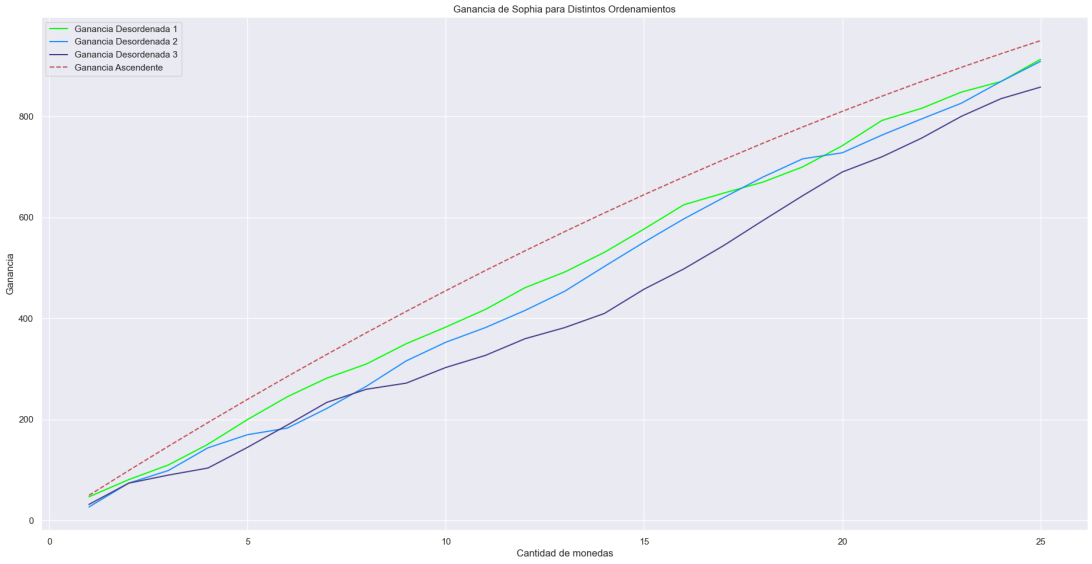
Finalizada esta parte del análisis, es importante saber que la notación Big O no es más que una herramienta conceptual para entender cómo se comporta un algoritmo frente al tamaño de un dato de entrada. En la práctica los tiempos de ejecución pueden variar por más que el tamaño de nuestra lista permanezca constante. En el siguiente ejemplo vamos a ver cómo se comporta nuestro algoritmo frente a variaciones en los valores de las monedas con un tamaño constante lo suficientemente grande (100000).



Viendo el gráfico se puede apreciar una variabilidad despreciable a efectos prácticos.

1.4. Variabilidad de la Optimalidad

Si bien Sophia ganará siempre que la lista no sea un número par de monedas iguales, es posible que la ganancia de Sophia se vea afectada por la variabilidad de los valores de las monedas. Para ver esto, el análisis realizado en el punto anterior no será de gran utilidad, ya que analizar un gráfico de la ganancia de Sophia para listas diferentes no nos permite sacar una conclusión satisfactoria. En cambio, el caso de la ganancia de Sophia para distintos ordenamientos de la misma lista resulta interesante. A continuación tenemos un gráfico de la ganancia de Sophia para distintos ordenamientos de la misma lista, incluyendo una lista de orden ascendente.



Se puede observar como se obtiene la mayor ganancia cuando las monedas están ordenadas. Cabe destacar que el resultado es el mismo para el ordenamiento descendente por la naturaleza del algoritmo.

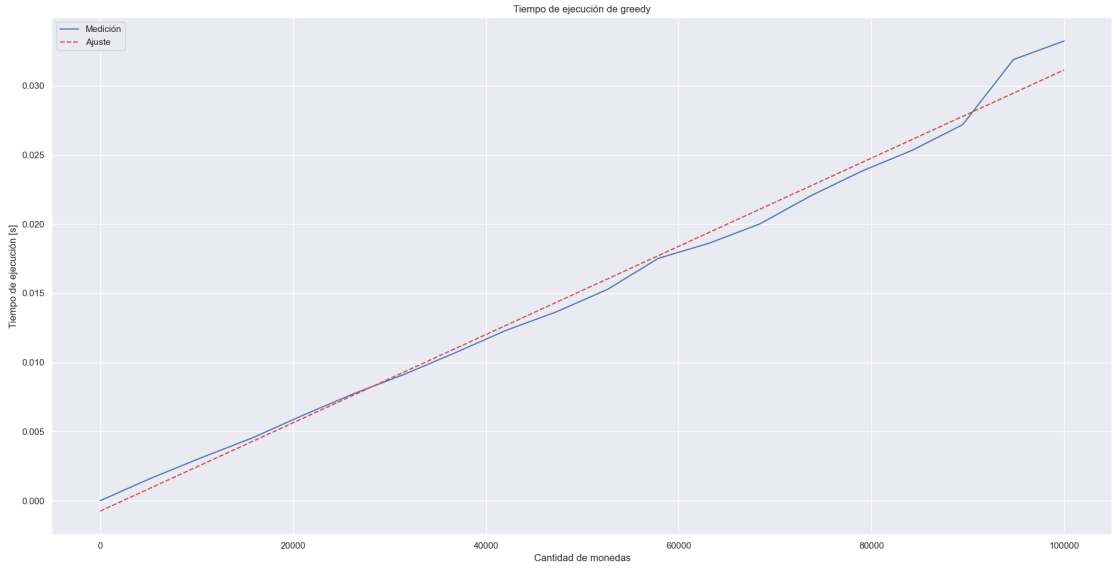
1.5. Ejemplos de Ejecución

Nos aseguramos que se cumpla la optimalidad para los casos brindados por la cátedra (20 elementos hasta 1000 elementos) y que las decisiones tomadas por los jugadores en cada turno sean iguales. Además, probamos los siguientes casos:

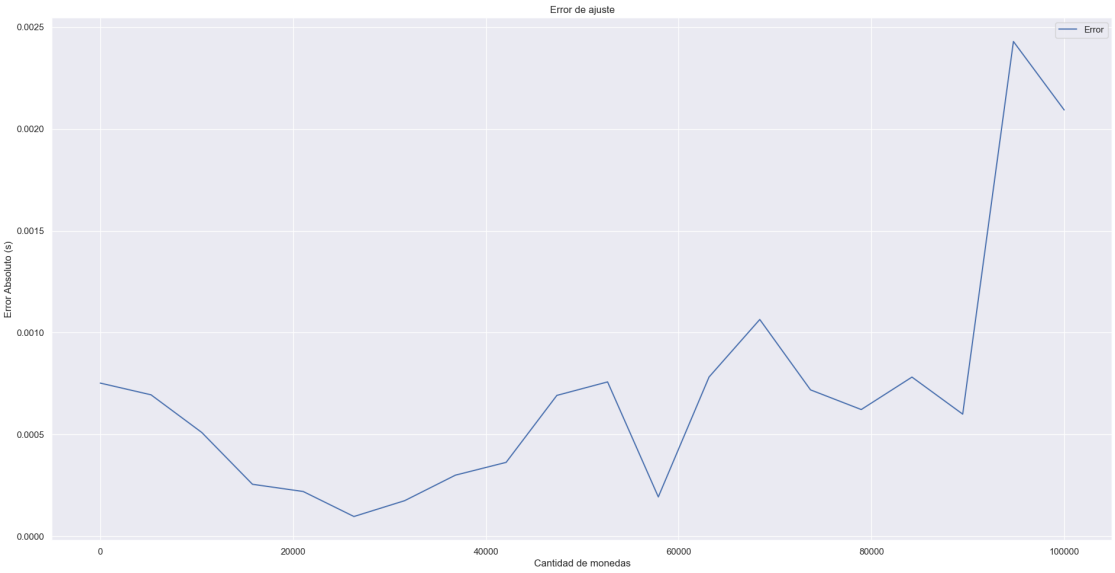
- Lista ordenada (ascendente y descendente).
- Lista par de valores iguales (para asegurarse de que funciona la condición de empate).
- Valores alternados.
- Lista simétrica.
- Caso de una moneda.

1.6. Mediciones de Tiempo

En este apartado se corrobora la complejidad $\mathcal{O}(n)$ indicada en la sección 1.3 de forma experimental y se realiza un ajuste por cuadrados mínimos para ver qué tan buena es la medición. Además, mostramos el error de ajuste para cada tamaño para concluir la efectividad del ajuste.



Podemos apreciar como la tendencia lineal del algoritmo es evidente.



Viendo el error de ajuste para cada tamaño, podemos concluir que el error es lo suficientemente pequeño para concluir que el algoritmo se comporta efectivamente con complejidad temporal lineal.

1.7. Anexo: Código

1.7.1. Variabilidad del Tiempo de Ejecución

```
from monedas_greedy import monedas_greedy
from random import seed
from matplotlib import pyplot as plt
import seaborn as sns
import numpy as np
import scipy as sp
from util import time_algorithm
```

```

def get_random_array(size: int):
    return list(np.random.randint(0, 100000, size))

if __name__ == '__main__':

    sns.set_theme()

    x = np.linspace(1,100000,20).astype(int)

    seed(12345)
    np.random.seed(12345)
    results = time_algorithm(monedas_greedy, x, lambda s: [get_random_array(s)])
    r_ordenado = time_algorithm(monedas_greedy, x, lambda s: [sorted(
    get_random_array(s))])

    seed(54321)
    np.random.seed(54321)
    r1 = time_algorithm(monedas_greedy, x, lambda s: [get_random_array(s)])
    seed(432535)
    np.random.seed(432535)
    r2 = time_algorithm(monedas_greedy, x, lambda s: [get_random_array(s)])
    seed(11214)
    np.random.seed(11214)
    r3 = time_algorithm(monedas_greedy, x, lambda s: [get_random_array(s)])

    ax: plt.axes
    fig, ax = plt.subplots()

    ax.plot(x, [r1[i] for i in x], label="Lista 1", color="lime")
    ax.plot(x, [r2[i] for i in x], label="Lista 2", color="darkslateblue")
    ax.plot(x, [r3[i] for i in x], label="Lista 3", color="dodgerblue")
    ax.plot(x, [r_ordenado[i] for i in x], "r--", label="Lista Ordenada")

    ax.set_title('Tiempo de Ejecuci n para Distintos Valores')
    ax.set_xlabel('Cantidad de monedas')
    ax.set_ylabel('Tiempo de ejecuci n [s]')
    plt.legend()
    plt.show()
```

1.7.2. Variabilidad de la Optimalidad

```
from monedas_greedy import monedas_greedy
import random
import seaborn as sns
import numpy as np
from matplotlib import pyplot as plt

#Numeros del 1 al 50 desordenados
monedas = [34, 7, 25, 2, 49, 16, 45, 12, 37, 1, 28, 20, 6, 31, 39, 46, 48, 23, 9,
    11, 30, 36, 15, 14, 21, 38, 44, 5, 10, 32, 24, 50, 42, 13, 22, 3, 17, 18, 8, 4,
     43, 26, 35, 33, 40, 19, 41, 29, 27, 47]
#Primero voy a ordenar y desordenar este array de forma diferente para ver como
    cambia el resultado
#El algoritmo siempre va a ser  ptimo  (siempre va a ganar Sophia), pero puede que
    la ganancia de Sophia dependa de como est  ordenado el array

monedas_asc = sorted(monedas)
monedas_dec = sorted(monedas, reverse=True)

decisiones_des, monedas_sophia_des, monedas_mateo_des = monedas_greedy(monedas)
decisiones_asc, monedas_sophia_asc, monedas_mateo_asc = monedas_greedy(monedas_asc)
decisiones_dec, monedas_sophia_dec, monedas_mateo_dec = monedas_greedy(monedas_dec)

ganancia_des = sum(monedas_sophia_des)
ganancia_asc = sum(monedas_sophia_asc)
ganancia_dec = sum(monedas_sophia_dec)

```

```
print(ganancia_des) #913
print(ganancia_asc) #950
print(ganancia_dec) #950

#Como veo que es igual de optimo si ordeno ascendente o descendente, voy a ver si cambia mucho entre diferentes arrays desordenados

#No uso random.randint porque me genera duplicados
def get_random_array():
    return random.sample(range(1,51), 50)


sns.set_theme()
x = np.linspace(1,25,25).astype(int)

random.seed(12345)
res1, y1, mateo1 = monedas_greedy(get_random_array())

random.seed(54321)
res2, y2, mateo2 = monedas_greedy(get_random_array())

ax: plt.axes
fig, ax = plt.subplots()
ax.plot(x, [sum(monedas_sophia_des[:i]) for i in x], label="Ganancia Desordenada 1", color="lime")
ax.plot(x, [sum(y1[:i]) for i in x], label="Ganancia Desordenada 2", color="dodgerblue")
ax.plot(x, [sum(y2[:i]) for i in x], label="Ganancia Desordenada 3", color="darkslateblue")
ax.plot(x, [sum(monedas_sophia_asc[:i]) for i in x], 'r--', label="Ganancia Ascendente")
#ax.plot(x, [sum(monedas_sophia_dec[:i]) for i in x], label="Ganancia Descendiente", color="pink")
#El grafico de los arrays ascendente y descendente son iguales. Me quedo con uno solo

ax.set_title('Ganancia de Sophia para Distintos Ordenamientos')
ax.set_xlabel('Cantidad de monedas')
ax.set_ylabel('Ganancia')
plt.legend()
plt.show()

#Ver que hay una leve variaci n en la ganancia final dependiendo de como est n ordenadas las monedas, pero no es significativo.
#Parece ser que la ganancia m xima se alcanza cuando el array est  ordenado ascendente o descendentemente, el resultado de cualquier otro ordenamiento va a ser menos  ptimo  (pero  ptimo  al fin).
```

1.7.3. Mediciones de Tiempo

```
from monedas_greedy import monedas_greedy
from random import seed
from matplotlib import pyplot as plt
import seaborn as sns
import numpy as np
import scipy as sp
from util import time_algorithm

def get_random_array(size: int):
    return list(np.random.randint(0, 100000, size))

if __name__ == '__main__':  #Este bloque es para que no explote time_algorithm
    seed(12345)
    np.random.seed(12345)

    sns.set_theme()

    x = np.linspace(1,100000,20).astype(int)
    results = time_algorithm(monedas_greedy, x, lambda s: [get_random_array(s)])
```

```
    f = lambda x, c1, c2: c1*x+c2 #forma de una funci n lineal
    c, pcov = sp.optimize.curve_fit(f, x, [results[n] for n in x])

    #Ajuste por cuadrados m nimos
    ax: plt.axes
    fig, ax = plt.subplots()
    ax.plot(x, [results[i] for i in x], label="Medici n")
    ax.plot(x, [c[0]*n+c[1] for n in x], 'r--', label="Ajuste")
    ax.set_title('Tiempo de ejecuci n de greedy')
    ax.set_xlabel('Cantidad de monedas')
    ax.set_ylabel('Tiempo de ejecuci n [s]')
    plt.legend()
    plt.show()

    #Error para cada tama o
    ax: plt.axes
    fig, ax = plt.subplots()
    errors = [np.abs(c[0]*n+c[1]-results[n]) for n in x]
    ax.plot(x, errors, label="Error")
    ax.set_title('Error de ajuste')
    ax.set_xlabel('Cantidad de monedas')
    ax.set_ylabel('Error Absoluto (s)')
    plt.legend()
    plt.show()
```

2. Solución Dinámica

2.1. Análisis del Problema

Para esta parte del trabajo el problema cambia ligeramente, ya que ahora Mateo es el que elige la mayor moneda. El objetivo es plantear un algoritmo por programación dinámica que obtenga la solución óptima al problema planteado y plantear su ecuación de recurrencia.

Para cada turno, Sophia tiene que elegir entre agarrar la primera o la última moneda de la fila. Si agarra la primera moneda, entonces la va a quitar de la fila y Mateo va a elegir la más grande disponible entre las otras dos que queden en los extremos. Esta logica se repite si Sophia decide agarrrar la última moneda, pero el subarray resultante va a ser diferente.

El caso base es cuando el subarray contiene un solo elemento, por lo que Sophia no tiene otra opción que tomarlo.

Si planteamos $dp[i][j]$ como el valor máximo que agarra Sophia para el subarray que va de i a j, entonces Sophia quiere quedarse con lo que maximice el valor de $dp[i][j]$ teniendo en cuenta que Mateo siempre va a hacer que ella tome el minimo valor entre $dp[i+2][j]$ y $dp[i+1][j-1]$. en caso de elegir la primera moneda, o $dp[i][j-2]$ y $dp[i+1][j-1]$, si elige la última, por lo que tiene que fijarse si le conviene elegir la moneda i o la j. Teniendo esto en cuenta, la ecuación de recurrencia tiene la siguiente forma.

$$dp[i][j] = max(monedas[i]+min(dp[i+2][j], dp[i+1][j-1]), monedas[j]+min(dp[i][j-2], dp[i+1][j-1]))$$

2.2. Demostración de la Efectividad del Algoritmo

Si bien el algoritmo puede no ser optimo en algunos casos (Por ejemplo, para la lista [1,10,5] Sophia gana 6 y Mateo 10), la ganancia de Sophia siempre será la máxima posible para las monedas en el orden dado. Esto se debe a que Sophia siempre está buscando la forma de maximizar su ganancia buscando la mejor decisión luego de optimizar los subproblemas resultantes de elegir la primera o la última moneda, y esto es posible ya que el comportamiento de Mateo es predecible, puesto que siempre busca elegir la moneda más grande para el y minimizar la ganancia posible de Sophia, por lo que Mateo puede ganar si la estructura de la lista lo favorece, pero Sophia siempre estará acumulando lo más que pueda.

2.3. Algoritmo Planteado

A continuación mostramos una implementación posible del algoritmo en Python, usando programación dinámica y memoization.

```
def reconstruccion(monedas, matriz):
    i, k = 0, len(monedas)-1
    instrucciones = []
    monedas_sophia = []
    if i == k:
        instrucciones.append(f"Sophia debe agarrar la ultima ({monedas[i]})")
    while k > 0:
        if matriz[k][i] == matriz[k - 2][i + 1] + monedas[i]:
            instrucciones.append(f"Sophia debe agarrar la primera ({monedas[i]})")
            monedas_sophia.append(monedas[i])
            i += 1
        elif (matriz[k][i] == matriz[k - 2][i + 2] + monedas[i]
              and (matriz[k][i] != matriz[k - 2][i] + monedas[k + i - 1]
                   or monedas[i + 1] <= (monedas[k + i - 1] and monedas[i]))):
            instrucciones.append(f"Sophia debe agarrar la primera ({monedas[i]})")
            monedas_sophia.append(monedas[i])
            i += 1
        else:
```

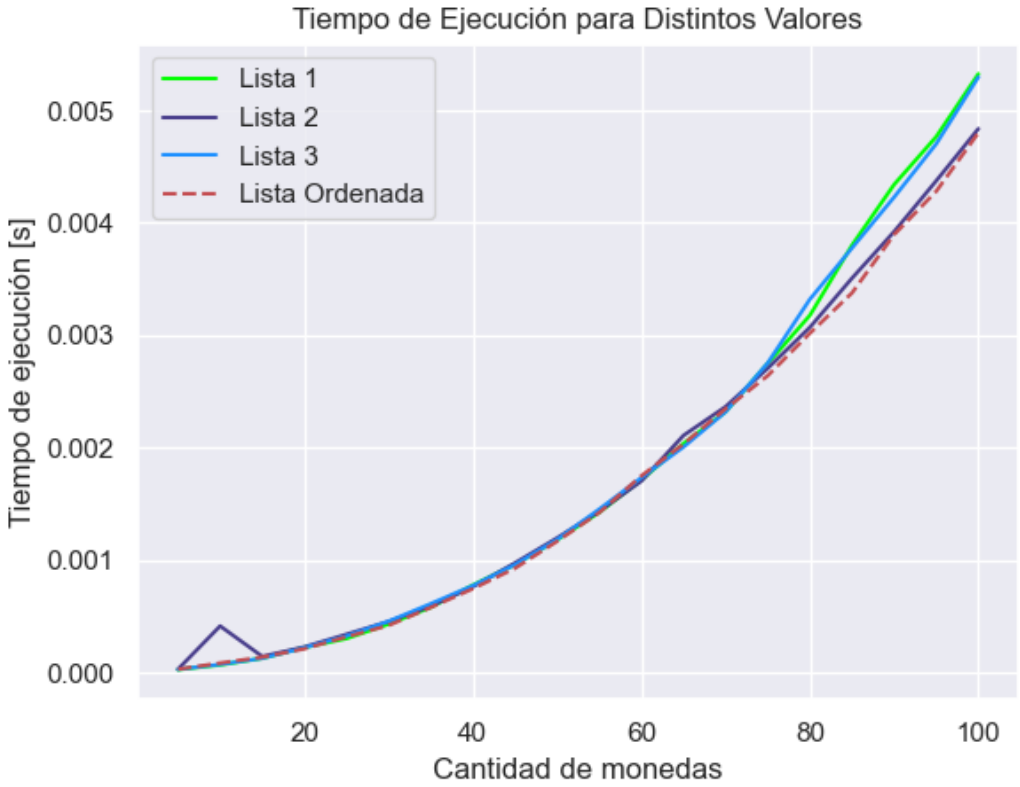
```
            instrucciones.append(f"Sophia debe agarrar la ultima ({monedas[k + i - 1]})")
            monedas_sophia.append(monedas[k+i-1])
        k -= 1

        if k > 0:
            if monedas[i] <= monedas[k + i - 1]:
                instrucciones.append(f"Mateo agarra la ultima ({monedas[k + i - 1]})")
            else:
                instrucciones.append(f"Mateo agarra la primera ({monedas[i]})")
                i += 1
            k -= 1
    return instrucciones, monedas_sophia


def elegir_izq(monedas, matriz, inicio):
    valor_izq = monedas[0]
    if len(monedas) <= 2:
        return valor_izq
    else:
        if monedas[1] < monedas[-1]:
            return valor_izq + matriz[len(monedas) - 2][inicio + 1]
        else:
            return valor_izq + matriz[len(monedas) - 2][inicio + 2]


def elegir_der(monedas, matriz, inicio):
    valor_der = monedas[-1]
    if len(monedas) <= 2:
        return valor_der
    else:
        if monedas[0] < monedas[-2]:
            return valor_der + matriz[len(monedas) - 2][inicio]
        else:
            return valor_der + matriz[len(monedas) - 2][inicio + 1]


def monedas_dinamicas(monedas):

    """Defino una matriz de el optimo de subarreglos que se divide por el largo de el subarreglo en
     filas y por la posicion del arreglo original por la que empiezo a construir el subarreglo en columnas"""
    arr = [0] * (len(monedas))
    matriz = []
    for i in range(len(monedas) + 1):
        matriz.append(arr.copy())

    for largo in range(1, len(monedas) + 1):
        for comienzo in range(0, len(monedas)):
            if largo + comienzo > len(monedas):
                continue
            max_izq = elegir_izq(monedas[comienzo:comienzo + largo], matriz, comienzo)
            max_der = elegir_der(monedas[comienzo:comienzo + largo], matriz, comienzo)
            matriz[largo][comienzo] = max(max_der, max_izq)

    total_monedas = sum(monedas)
    total_sophia = matriz[len(monedas)][0]
    total_mateo = total_monedas - total_sophia
    decisiones, monedas_sophia = reconstruccion(monedas, matriz)

    return decisiones, monedas_sophia, total_sophia, total_mateo
```

2.3.1. Complejidad del Algoritmo

En la primera parte del código llenamos la tabla *dp* para los distintos subarrays, por lo que tiene complejidad $\mathcal{O}(n^2)$. Por otro lado, la parte de memoization itera k veces ($\mathcal{O}(k)$) cargando cada decisión y las monedas que eligió Sophia, que son todas operaciones $\mathcal{O}(1)$. Finalmente, la complejidad del algoritmo es de $\mathcal{O}(n^2)$.

2.3.2. Variabilidad del Tiempo de Ejecución

Análogamente a la sección Greedy del trabajo, analizaremos cómo varía el tiempo de ejecución para diferentes listas de igual tamaño.



Como se puede observar en la figura, todas las mediciones tienen forma cuadrática, y presentan una variación despreciable.

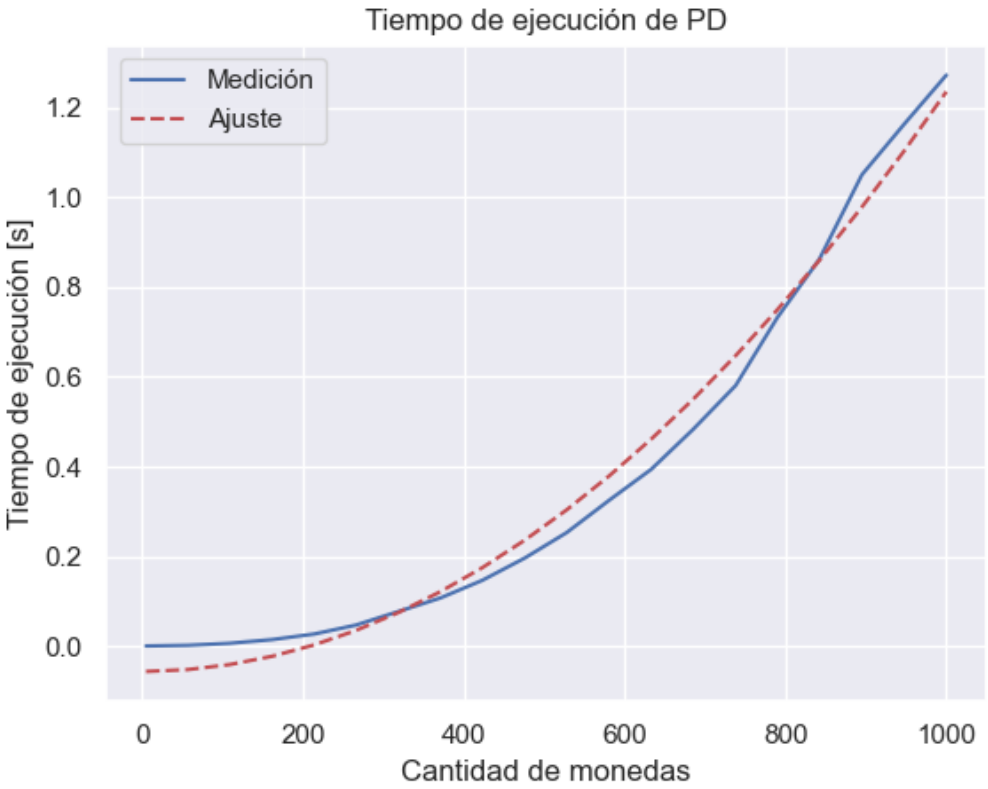
2.4. Ejemplos de Ejecución

Similar a la primera parte del trabajo, nos aseguramos que se cumplan los casos brindados por la cátedra, además de agregar algunos casos nuestros:

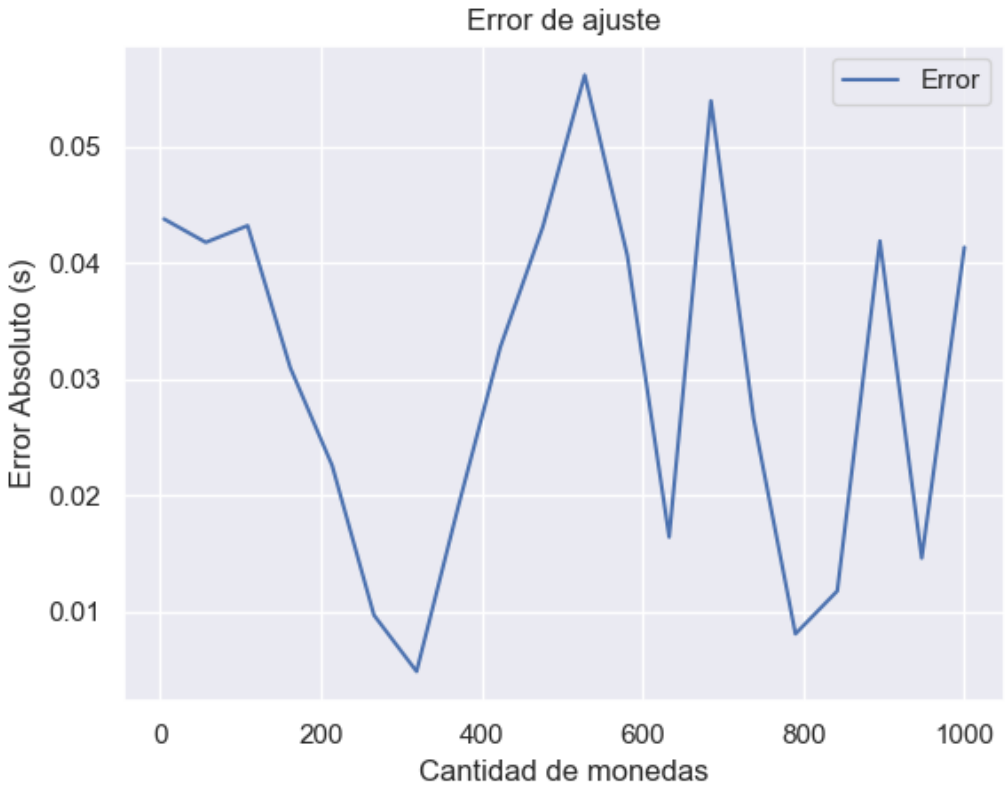
- Lista ordenada (ascendente y descendente).
- Lista [1,10,5] (para asegurarnos de que Mateo puede ganar).
- Lista simétrica.
- Caso de una moneda.

2.5. Mediciones de Tiempo

A continuación, realizamos mediciones de tiempo para corroborar que el algoritmo es realmente $\mathcal{O}(n^2)$ de forma experimental y ajustamos los resultados usando cuadrados mínimos, además de calcular el error de ajuste.



Se puede ver como la medición experimental tiene la misma forma cuadrática que el ajuste. A continuación vemos el error para asegurarnos que es una medición aceptable.



El error absoluto se mantiene acotado por $0{,}6 * 10^{-1}$ para cantidades menores a 1000. Consideramos que esto es un error aceptable y la medición es correcta.

2.6. Anexo: Código

2.6.1. Variabilidad de Tiempo de Ejecución

```
from monedas_dinamicas import monedas_dinamicas
from random import seed
from matplotlib import pyplot as plt
import seaborn as sns
import numpy as np
import scipy as sp
from util import time_algorithm

def get_random_array(size: int):
    return list(np.random.randint(0, 1000, size))

if __name__ == '__main__':  #Este bloque es para que no explote time_algorithm

    sns.set_theme()

    x = np.linspace(5,100,20).astype(int)

    seed(12345)
    np.random.seed(12345)
    results = time_algorithm(monedas_dinamicas, x, lambda s: [get_random_array(s)])
    r_ordenado = time_algorithm(monedas_dinamicas, x, lambda s: [sorted(
    get_random_array(s))])

    np.random.seed(2)
    r1 = time_algorithm(monedas_dinamicas, x, lambda s: [get_random_array(s)])
    np.random.seed(3)
```

```
    r2 = time_algorithm(monedas_dinamicas, x, lambda s: [get_random_array(s)])
    np.random.seed(54321)
    r3 = time_algorithm(monedas_dinamicas, x, lambda s: [get_random_array(s)])

    ax: plt.axes
    fig, ax = plt.subplots()

    ax.plot(x, [r1[i] for i in x], label="Lista 1", color="lime")
    ax.plot(x, [r2[i] for i in x], label="Lista 2", color="darkslateblue")
    ax.plot(x, [r3[i] for i in x], label="Lista 3", color="dodgerblue")
    ax.plot(x, [r_ordenado[i] for i in x], "r--", label="Lista Ordenada")

    ax.set_title('Tiempo de Ejecuci n para Distintos Valores')
    ax.set_xlabel('Cantidad de monedas')
    ax.set_ylabel('Tiempo de ejecuci n [s]')
    plt.legend()
    plt.show()
```

2.6.2. Mediciones de Tiempo

```
from monedas_dinamicas import monedas_dinamicas
from random import seed
from matplotlib import pyplot as plt
import seaborn as sns
import numpy as np
import scipy as sp
from util import time_algorithm

def get_random_array(size: int):
    return list(np.random.randint(0, 1000, size))

if __name__ == '__main__':  #Este bloque es para que no explote time_algorithm
    seed(12345)
    np.random.seed(12345)

    sns.set_theme()

    x = np.linspace(5,1000,20).astype(int)
    results = time_algorithm(monedas_dinamicas, x, lambda s: [get_random_array(s)])
    f = lambda x, c1, c2: c1*(x**2)+c2 #forma de una funci n cuadr tica
    c, pcov = sp.optimize.curve_fit(f, x, [results[n] for n in x])

    #Ajuste por cuadrados m nimos
    ax: plt.axes
    fig, ax = plt.subplots()
    ax.plot(x, [results[i] for i in x], label="Medici n")
    ax.plot(x, [c[0]*(n**2)+c[1] for n in x], 'r--', label="Ajuste")
    ax.set_title('Tiempo de ejecuci n de PD')
    ax.set_xlabel('Cantidad de monedas')
    ax.set_ylabel('Tiempo de ejecuci n [s]')
    plt.legend()
    plt.show()

    #Error para cada tama o
    ax: plt.axes
    fig, ax = plt.subplots()
    errors = [np.abs(c[0]*(n**2)+c[1]-results[n]) for n in x]
    ax.plot(x, errors, label="Error")
    ax.set_title('Error de ajuste')
    ax.set_xlabel('Cantidad de monedas')
    ax.set_ylabel('Error Absoluto (s)')
    plt.legend()
    plt.show()
```

3. Cambios

3.1. Análisis del Problema

Pasó mucho tiempo desde que Mateo y Sophia jugaban al juego de las monedas. Los hermanos ahora están en la etapa de la adolescencia y empiezan a tener gustos diferentes. Con respecto a los juegos, cada uno iba a por su lado. Centrándonos en Sophia, empezó a engancharse mucho con un juego creado en nuestro país por Jaime Poniachik denominado Batalla Naval Individual.

En dicho juego, tenemos un tablero de nxm casilleros, y k barcos. Cada barco i tiene bi de largo. Es decir, requiere de bi casilleros para ser ubicado. Todos los barcos tienen 1 casillero de ancho. El tablero a su vez tiene un requisito de consumo tanto en sus filas como en sus columnas. Si en una fila indica un 3, significa que deben haber 3 casilleros de dicha fila siendo ocupados. Ni más, ni menos. No podemos poner dos barcos de forma adyacente (es decir, no pueden estar contiguos ni por fila, ni por columna, ni en diagonal directamente). Debemos ubicar todos los barcos de tal manera que se cumplan todos los requisitos.

En esta parte del informe vamos a explorar el problema que nos presenta este juego. Vamos a demostrar por qué el problema en su versión de decisión es NP y NP-Completo. Luego usaremos su versión de optimización para aplicar una solución que utilice un algoritmo de Backtracking y otra que empleé Programación Lineal, calcular sus mediciones de tiempos y comparar. Mas adelante, implementaremos y analizaremos en detalle un algoritmo de aproximación propuesto por John Jellicoe. Y por ultimo, propondremos un algoritmo de aproximación o un algoritmo Greedy, comparándolo con el algoritmo dicho anteriormente.

3.2. ¿El problema en su versión de decisión está en NP?

En principio, vamos a demostrar si el problema está en NP usando su versión de decisión:

Dado un tablero de nxm casilleros, y una lista de k barcos (donde el barco i tiene bi de largo), una lista de restricciones para las filas (donde la restricción j corresponde a la cantidad de casilleros a ser ocupados en la fila j y una lista de restricciones para las columnas (símil filas, pero para columnas), ¿es posible definir una ubicación de dichos barcos de tal forma que se cumplan con las demandas de cada fila y columna, y las restricciones de ubicación?

Como sabemos, un problema en NP es aquel que teniendo una posible solución al mismo, podemos comprobar su veracidad en un tiempo polinomial, basándonos en la teoría de complejidad algorítmica. Es decir, teniendo los datos que nos proporciona una instancia del problema junto con una posible solución a ella, debemos verificar que la función que reciba esos argumentos se ejecute en O(f(x)) siendo f(x) una función polinomial.

Ahora, que argumentos va a recibir esta funcion verificador? Bueno, a continuacion daremos una breve sintesis de lo que trata cada uno de ellos y luego mostraremos el codigo:

1) **tablero**: Una matriz de n filas por m columnas, donde sus valores están inicializados con `None`.

2) **longitud_barcos**: Una lista de las longitudes de los k barcos (el barco i tiene b_i de largo).

3) **restricciones_filas**: Una lista de restricciones para las filas (la fila i tiene el número de casilleros a ocupar en la misma).

4) **restricciones_columnas**: La lista de restricciones para las columnas (símil **restricciones_filas** pero para columnas).

5) **ubicacion_barcos**: Una lista de sublistas, donde cada sublista i representa las posiciones (f, c) del barco i.

```
1    ##Ejemplo de referencia:
2        ubicacion_barcos = [[(),(),...,()],
3                            [(),(),...,()],
4                            ...,
5                            [(),(),...,()]]
```

```python
def verificador_de_solucion(tablero,longitud_barcos,restricciones_filas,
    restricciones_columnas,
                ubicacion_barcos):

    n = len(tablero) ## cantidad de filas del tablero
    m = len(tablero[1]) ##cantidad de columnas del tablero
    k = len(longitud_barcos) ##cantidad de barcos que deber haber en el tablero

    if not es_valida_la_lista_ubicacion_barcos(n,m,longitud_barcos,k,
    ubicacion_barcos):
        return False


    ## Colocamos lo barcos en tablero
    id_barco = 0
    for posiciones_de_barco in ubicacion_barcos:

        for (f,c) in posiciones_de_barco:
            tablero[f][c] = id_barco

        id_barco += 1
    ####

    if not se_cumplen_requisitos_consumo_filas_y_columnas(tablero,n,m,
    restricciones_filas,restricciones_columnas):
        return False

    if hay_barcos_adyacentes(tablero,n,m,ubicacion_barcos,k):
        return False

    return True


def es_valida_la_lista_ubicacion_barcos(n,m,longitud_barcos,k,ubicacion_barcos):

    if len(ubicacion_barcos) != k:
        return False

    for i in range(0,k):
        posiciones_de_barco = ubicacion_barcos[i]
        if len(posiciones_de_barco) != longitud_barcos[i]:
            return False

        for (f,c) in posiciones_de_barco:

            if not (0 <= f <= n-1) or not (0 <= c <= m-1):
                return False

    return las_posiciones_de_cada_barco_son_verticales_u_horizontales(
    ubicacion_barcos)


def las_posiciones_de_cada_barco_son_verticales_u_horizontales(ubicacion_barcos):

    for posiciones_de_barco in ubicacion_barcos:
        if not barco_esta_posicionado_de_forma_vertical(posiciones_de_barco) and \
        not barco_esta_posicionado_de_forma_horizontal(posiciones_de_barco):
            return False
    return True


def barco_esta_posicionado_de_forma_vertical(posiciones_de_barco):

    pos_columna = posiciones_de_barco[0][1]
    for i in range(1,len(posiciones_de_barco)):
        _,c = posiciones_de_barco[i]
        if c != pos_columna:
            return False
    return True

```

```
67 def barco_esta_posicionado_de_forma_horizontal(posiciones_de_barco):
68
69     pos_fila = posiciones_de_barco[0][0]
70     for i in range(1,len(posiciones_de_barco)):
71         f,_ = posiciones_de_barco[i]
72         if f != pos_fila:
73             return False
74     return True
75
76
77 def se_cumplen_requisitos_consumo_filas_y_columnas(tablero,n,m,restricciones_filas,
       restricciones_columnas):
78
79     return se_cumplen_requisitos_consumo_filas(tablero,n,restricciones_filas) and \
80     se_cumplen_requisitos_consumos_columnas(tablero,n,m,restricciones_columnas)
81
82
83 def se_cumplen_requisitos_consumo_filas(tablero,n,restricciones_filas):
84
85     cant_elementos_en_fila = 0
86     for f in range(0,n):
87         fila = tablero[f]
88         for elemento in fila:
89             if elemento != None:
90                 cant_elementos_en_fila += 1
91
92         if cant_elementos_en_fila != restricciones_filas[f]:
93             return False
94
95         cant_elementos_en_fila = 0
96
97     return True
98
99 def se_cumplen_requisitos_consumos_columnas(tablero,n,m,restricciones_columnas):
100
101
102     cant_elementos_en_columna = 0
103
104     for c in range(0,m):
105         for f in range(0,n):
106
107             elemento = tablero[f][c]
108             if elemento != None:
109                 cant_elementos_en_columna += 1
110
111         if cant_elementos_en_columna != restricciones_columnas[c]:
112             return False
113
114         cant_elementos_en_columna = 0
115
116     return True
117
118
119 def hay_barcos_adyacentes(tablero,n,m,ubicacion_barcos,k):
120
121     id_barco = 0
122     for posiciones_de_barco in ubicacion_barcos:
123         for (f,c) in posiciones_de_barco:
124
125             for i in range(-1,2):
126                 for j in range(-1,2):
127
128                     offset_f = f+i
129                     offset_c = c+j
130
131                     if offset_f == f and offset_c == c:
132                         continue
133
134                     if posicion_de_desplazamiento_hay_posicion_de_otro_barco(
       tablero,n,m,offset_f,offset_c,id_barco):
```

```
                        return True
            id_barco += 1
    return False


def posicion_de_desplazamiento_hay_posicion_de_otro_barco(tablero,n,m,offset_f,
    offset_c,id_barco):

    ultima_pos_de_fila = n-1
    ultima_pos_de_columna = m-1

    if 0 <= offset_f <= ultima_pos_de_fila:
        if 0 <= offset_c <= ultima_pos_de_columna:
            return tablero[offset_f][offset_c] != None and tablero[offset_f][
    offset_c] != id_barco
                ##return True
            ##return False
        return False
    return False

###Este seria el tablero con las ubicaciones de los barcos que se muestra en imagen
     en el tp3
tablero = [[None for _ in range(10)] for _ in range(10)]
longitud_barcos =  [3,1,1,2,4,2,1,3,1,2]
restricciones_filas = [3,2,2,4,2,1,1,2,3,0]
restricciones_columnas = [1,2,1,3,2,2,3,1,5,0]
ubicacion_barcos = [[(0,4),(0,5),(0,6)],
                    [(1,0)],
                    [(1,8)],
                    [(2,2),(2,3)],
                    [(3,5),(3,6),(3,7),(3,8)],
                    [(4,1),(5,1)],
                    [(4,3)],
                    [(6,8),(7,8),(8,8)],
                    [(7,6)],
                    [(8,3),(8,4)]]

print(verificador_de_solucion(tablero,longitud_barcos,restricciones_filas,
    restricciones_columnas,ubicacion_barcos))
```

La función `verificador_solucion` es la función principal que recibe los argumentos mencionados. Se crean algunas variables que se ejecutan en tiempo constante. Luego se llama a la función `es_valida_la_lista_ubicacion_barcos()` que verifica, justamente, que la lista solución sea correcta. Esto es, que la cantidad de sublistas sea igual a k, que la longitud de cada sublista i sea igual a `longitud_barcos[i]`, que las posiciones (f, c) de cada sublista estén dentro del rango de la matriz y que se pueda comprobar que las posiciones de los barcos se reflejen en el tablero de forma vertical u horizontal, usando `las_posiciones_de_cada_barco_son_verticales_u_horizontales`. La complejidad hasta el momento es $O\left(\sum_{i=1}^{k} b_i\right)$ siendo k la cantidad de barcos y bi la longitud de los barcos. En el peor de los casos, vamos a recorrer toda la lista solucion y luego recorremos cada sub-lista.

Volviendo a la funcion principal, vamos agregando los barcos al tablero. Esto tambien es $O\left(\sum_{i=1}^{k} b_i\right)$.

Luego llamamos a `se_cumplen_requisitos_consumo_filas_y_columnas()` que verifica si cada fila y columna cumple con las restricciones correspondientes. Todo esto se ejecuta en tiempo $O(n \times m)$, ya que recorremos todas las posiciones del tablero en el peor caso.

Por último, nos encontramos con el llamado a `hay_barcos_adyacentes()`. Si observan esta función, podrán notar que estamos recorriendo todas las posiciones (f, c) de cada barco, y por cada posición verificamos si las posiciones que están alrededor contienen alguna parte de otro barco. En ese caso, se devuelve `True` y, por lo tanto, la solución no es válida. Caso contrario, recorremos todas las posiciones, llegando a la conclusión de que esta función tiene complejidad

$$O\left(\sum_{i=1}^{k} b_i \cdot 9\right) = O\left(\sum_{i=1}^{k} b_i\right)$$

Si sumamos todas las complejidades....

$$f(n,m,k) = O\left(\sum_{i=1}^{k} b_i\right) + O\left(\sum_{i=1}^{k} b_i\right) + O(n \times m) + O\left(\sum_{i=1}^{k} b_i\right)$$

$$f(n,m,k) = O(n \times m) + O\left(3\sum_{i=1}^{k} b_i\right)$$

La complejidad total del algoritmo es ...

$$f(n,m,k) = O(n \times m) + O\left(\sum_{i=1}^{k} b_i\right)$$

Como ven, la funcion verificador es polinomica ,concluyendo que el problema en su version de decision de Batalla Naval está en NP.

3.3. ¿El problema en su versión de decisión esta en NP-Completo?

Debemos comprobar que el problema está en NP-Completo. Un problema X es NP-Completo si cumple dos condiciones:

1. X está en NP.
2. Para todo problema Y en NP, existe una reducción polinómica de Y a X.

La primera condición ya la hemos comprobado en la sección anterior. Para la segunda condición, sería exhaustivo demostrarlo para todos los problemas en NP. Sin embargo, podemos demostrar que el problema de *Batalla Naval* es NP-Completo si reducimos un problema NP-Completo conocido al nuestro.

En este caso, utilizaremos el problema denominado *Bin Packing* en su versión unaria:

Bin Packing (versión unaria) Dados:

- Un conjunto de números $U = \{u_1, u_2, \ldots, u_k\}$, cada uno representado en código unario.
- Una cantidad de contenedores B, también en código unario.
- Una capacidad C para cada contenedor, en código unario.

Se desea saber si es posible dividir U en B subconjuntos disjuntos (bins), tal que la suma de los elementos de cada subconjunto sea exactamente igual a C. Además, la suma total de los elementos de U es $C \times B$.

Reducción al problema de Batalla Naval :

Dado el conjunto U, transformamos cada elemento u_i en la longitud de un barco b_i y creamos una lista L que almacena las longitudes de los barcos. Cada u_i se convierte en b_i calculando la cantidad de unos que tiene.

Dimensiones del tablero: : - El tablero tendrá dimensiones $2 \times \text{len}(U)$ filas y C columnas:

- $2 \times \text{len}(U)$ filas: Cada fila impar va a contener un barco (representando un elemento de U), mientras que cada fila par se restringe a cero casilleros ocupados, reduciendo la probabilidad de adyacencia.
- C columnas: Si la longitud de un barco supera C, no va a poder guardarse en el tablero

Restricciones de filas y columnas: : - Las restricciones de filas alternarán entre consumir b_i casilleros (longitud del barco i) y cero casilleros para las filas siguientes. - Las restricciones de columnas serán de B casilleros por columna.

3.4. Backtracking

```python
def cumplir_maxima_demanda(datos, tablero, resultado_final, resultado_actual, indice_barco, posiciones_usadas):
    if len(resultado_actual["posiciones"]) == len(datos["barcos"]):
        return es_solucion(datos, resultado_final, resultado_actual)

    if cortar_si_no_va_superar_resultado_final(datos, resultado_final, resultado_actual, indice_barco):
        return False

    tamano_barco = datos["barcos"][indice_barco]

    if ( (indice_barco) and (tamano_barco != datos["barcos"][indice_barco-1]) ):
        for indice_x in posiciones_usadas[tamano_barco]["derecha"]:
            posiciones_usadas[tamano_barco]["derecha"][indice_x] = 0

        for indice_y in posiciones_usadas[tamano_barco]["abajo"]:
            posiciones_usadas[tamano_barco]["abajo"][indice_y] = 0
    else:
        posiciones_usadas = copy.deepcopy(posiciones_usadas)

    i_inicial_derecha = posiciones_usadas[tamano_barco]["derecha"]
    i_inicial_abajo = posiciones_usadas[tamano_barco]["abajo"]

    if datos["posibles_posiciones"][tamano_barco]["derecha"]:
        for indice_x in datos["posibles_posiciones"][tamano_barco]["derecha"]:
            if datos["filas"][indice_x] < tamano_barco:
                continue

            for i in range(i_inicial_derecha[indice_x], len(datos["posibles_posiciones"][tamano_barco]["derecha"][indice_x])):
                posiciones_usadas[tamano_barco]["derecha"][indice_x] += 1

                valores_y = datos["posibles_posiciones"][tamano_barco]["derecha"][indice_x][i]

                if es_valida_posicion_derecha(datos, tablero, tamano_barco, indice_x, valores_y):
                    poner_barco_derecha(datos, tablero, tamano_barco, indice_x, valores_y, resultado_actual)
                    if cumplir_maxima_demanda(datos, tablero, resultado_final, resultado_actual, indice_barco+1, posiciones_usadas):
                        return True
                    sacar_barco_derecha(datos, tablero, tamano_barco, indice_x, valores_y, resultado_actual)

    if datos["posibles_posiciones"][tamano_barco]["abajo"]:
        for indice_y in datos["posibles_posiciones"][tamano_barco]["abajo"]:
            if datos["columnas"][indice_y] < tamano_barco:
                continue

            for i in range(i_inicial_abajo[indice_y], len(datos["posibles_posiciones"][tamano_barco]["abajo"][indice_y])):
```

```
                posiciones_usadas[tamano_barco]["abajo"][indice_y] += 1

                valores_x = datos["posibles_posiciones"][tamano_barco]["abajo"][
    indice_y][i]

                if es_valida_posicion_abajo(datos, tablero, tamano_barco, indice_y,
     valores_x):
                    poner_barco_abajo(datos, tablero, tamano_barco, indice_y,
    valores_x, resultado_actual)
                    if cumplir_maxima_demanda(datos, tablero, resultado_final,
    resultado_actual, indice_barco+1, posiciones_usadas):
                        return True
                    sacar_barco_abajo(datos, tablero, tamano_barco, indice_y,
    valores_x, resultado_actual)

    resultado_actual["posiciones"].append([0])
    if cumplir_maxima_demanda(datos, tablero, resultado_final, resultado_actual,
    indice_barco+1, posiciones_usadas):
        return True
    resultado_actual["posiciones"].pop()
```

3.5. John Jellicoe

```

def get_index(lista, barco, aparicion):
    cont = 0
    for i, n in enumerate(lista):
        if n == barco:
            cont += 1
            if cont == aparicion:
                return i


def contar_lista(lista):
    total = 0
    for i in lista:
        total += i
    return total


def barcos_adyacentes(tablero, coordenada, inicio, barco):
    if coordenada[0] is None:
        coordenadas = (inicio, coordenada[1])
        for i in range(barco):
            for f in range(-1, 2):
                if coordenadas[0] + f < 0 or coordenadas[0] + f >= len(tablero):
                    continue
                for c in range(-1, 2):
                    if coordenadas[1] + c + i < 0 or coordenadas[1] + c + i >= len(
    tablero[0]):
                        continue
                    elif tablero[coordenadas[0] + f][coordenadas[1] + c + i] is not
     None:
                        return True
    else:
        coordenadas = (coordenada[0], inicio)
        for i in range(barco):
            for f in range(-1, 2):
                if coordenadas[0] + f + i < 0 or coordenadas[0] + f + i >= len(
    tablero):
                    continue
                for c in range(-1, 2):
                    if coordenadas[1] + c < 0 or coordenadas[1] + c >= len(tablero
    [0]):
                        continue
                    elif tablero[coordenadas[0] + f + i][coordenadas[1] + c] is not
     None:
                        return True
```

```
41
42      return False
43
44
45  def posicionar_barco(barcos, valor_restriccion, restriccion_min, tablero,
        coordenada):
46      for barco in barcos:
47          if barco <= valor_restriccion:
48              inicio = 0
49              while inicio + barco <= len(restriccion_min):
50                  entra = True
51                  for j in range(barco):
52                      if restriccion_min[inicio + j] <= 0:
53                          entra = False
54                          break
55
56                  if entra and not barcos_adyacentes(tablero, coordenada, inicio,
        barco):
57
58                      return barco, inicio
59                  inicio += 1
60      return -1, -1
61
62
63  def solucion_naval_jj(tablero, barcos, restricciones_f, restricciones_c):
64      barcos_ordenados = sorted(barcos, reverse=True)
65      solucion = {}
66      colocado = False
67      demanda_total = contar_lista(restricciones_f) + contar_lista(restricciones_c)
68      max_r_fil = sorted(restricciones_f, reverse=True)
69      max_r_col = sorted(restricciones_c, reverse=True)
70      while len(barcos_ordenados) > 0 and len(max_r_fil) > 0 and len(max_r_col) > 0:
71          coordenadas = []
72          if colocado:
73              max_r_fil = sorted(restricciones_f, reverse=True)
74              max_r_col = sorted(restricciones_c, reverse=True)
75          colocado = False
76          if max_r_fil[0] >= max_r_col[0]:
77              coordenada = restricciones_f.index(max_r_fil[0])
78              barco, inicio = posicionar_barco(barcos_ordenados, max_r_fil[0],
        restricciones_c,  tablero, (coordenada, None))
79              if barco == -1:
80                  max_r_fil.pop(0)
81                  continue
82              for i in range(barco):
83                  coordenadas.append((coordenada, inicio + i))
84                  restricciones_c[inicio + i] -= 1
85              restricciones_f[restricciones_f.index(max_r_fil[0])] -= barco
86          else:
87              coordenada = restricciones_c.index(max_r_col[0])
88              barco, inicio = posicionar_barco(barcos_ordenados, max_r_col[0],
        restricciones_f, tablero, (None, coordenada))
89              if barco == -1:
90                  max_r_col.pop(0)
91                  continue
92              for i in range(barco):
93                  coordenadas.append((inicio + i, coordenada))
94                  restricciones_f[inicio + i] -= 1
95              restricciones_c[restricciones_c.index(max_r_col[0])] -= barco
96
97          i = 1
98          while not colocado:
99              posicion = get_index(barcos, barco, i)
100             if posicion not in solucion:
101                 colocado = True
102                 solucion[posicion] = coordenadas.copy()
103                 for k in coordenadas:
104                     tablero[k[0]][k[1]] = posicion
105             i += 1
106         barcos_ordenados.remove(barco)
```

```
107     demanda_cumplida = 0
108     for i in solucion:
109         demanda_cumplida += len(solucion[i]) * 2
110     return solucion, demanda_cumplida, demanda_total
```

mathtools

4. Primera parte: Introducción y primeros años

4.1. Consigna 1

En un algoritmo Greedy aplicamos una regla para obtener optimos locales en el estado actual, esto lo hacemos de manera iterativa con el objetivo de alcanzar una solucion global optima.

Nuestra regla Greedy es que dado nuestro estado actual, de tener una fila de cierta cantidad de monedas, elegimos y sacamos una moneda de solo dos lugares, el inicio y el final, y la elección la hacemos basándonos en los turnos, si es el turno de Mateo elegimos la moneda de más bajo valor, si es el turno de Sophia elegimos la moneda de más alto valor. El algoritmo es Greedy porque aplicamos iterativamente esta regla hasta llegar al estado final, que es cuando la fila está vacía.

4.2. Consigna 2

Vamos a probar que el algoritmo es optimo por contradiccion. Queremos probar que al final del juego, es decir cuando ya no quedan monedas en la fila, Sophia tiene mas valor acumulado que Mateo.

Asi que supongamos que lo dicho arriba es falso, que en realidad, cuando ya no quedan mas monedas para elegir, Mateo tiene mas valor acumulado que Sophia. Tomemos una fila de 4 monedas, [8,3,4,6], el algoritmo dice que el primer turno es de Sophia y que siempre agarra la moneda que tiene mas valor, asi que se lleva la moneda de valor 8. Turno de Mateo, la fila esta asi [3,4,6] y, como siempre, agarra la moneda de menor que en este caso es la de valor 3. Va Sophia otra vez, agarra el 6 y al final Mateo agarra el 4.

Sin mas monedas ahora nos toca revisar el valor acumulado, Mateo tiene las monedas de valor 3 y 4, por lo cual tiene un valor acumulado de 7 y Sophia, habiendo agarrado las de valor 8 y 6, se queda con un 14.

Como sabemos, 7 es menor que 14 entonces podemos darnos cuenta de que Mateo no tiene mas valor acumulado que Sophia, que nuestra suposicion es falsa y que Sophia es la que realmente tiene mas valor acumulado.

Y probando esto podemos afirmar que nuestro algoritmo es optimo, logra su optima global de hacer que Sophia tenga mas valor acumulado al final del juego.

Y aqui abajo podemos observar una screenshot que nos muestra que 7 tests pasaron sin errores, y de esos 7 podemos ver 3 que se encargan de comparar los valores acumulados de Mateo y Sophia. Y para que esas 3 pruebas pasen si o si el valor acumulado de Sophia debe ser mayor al de Mateo.

```
25      def test_valores_alternados(self):
26          monedas = [1,10,1,10,1,10,1,10,1,10,1,10,1,10,1,10,1,10]
27          total_sophia, total_mateo, _ = greedy.monedas_greedy(monedas)
28          self.assertGreater(total_sophia, total_mateo)
29
30      def test_lista_simetrica(self):
31          monedas = [5,4,3,2,1,2,3,4,5]
32          total_sophia, total_mateo, _ = greedy.monedas_greedy(monedas)
33          self.assertGreater(total_sophia, total_mateo)
34
35      def test_20_elementos(self):
36          monedas = [72,165,794,892,880,341,882,570,679,725,979,375,459,603,112,436,587,699,681,83]
37          total_sophia, _, _ = greedy.monedas_greedy(monedas)
38          self.assertEqual(total_sophia, 7165)
39
40      def test_1000_elementos(self):
41          monedas = [79,755,257,648,721,209,542,766,864,687,957,461,484,122,604,498,572,181,967,521,
42          total_sophia, total_mateo, _ = greedy.monedas_greedy(monedas)
43          self.assertGreaterEqual(total_sophia, total_mateo)
44
45
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS  COMENTARIOS

PS C:\Users\Melanie\Desktop\carla\FIUBA\TDA\TP-Teoria-de-algoritmos> & C:/Users/Melanie/AppData/Local/Programs/P
la/FIUBA/TDA/TP-Teoria-de-algoritmos/parte_1/tests_greedy.py
.......
----------------------------------------------------------------------
Ran 7 tests in 0.001s

OK
PS C:\Users\Melanie\Desktop\carla\FIUBA\TDA\TP-Teoria-de-algoritmos>
```

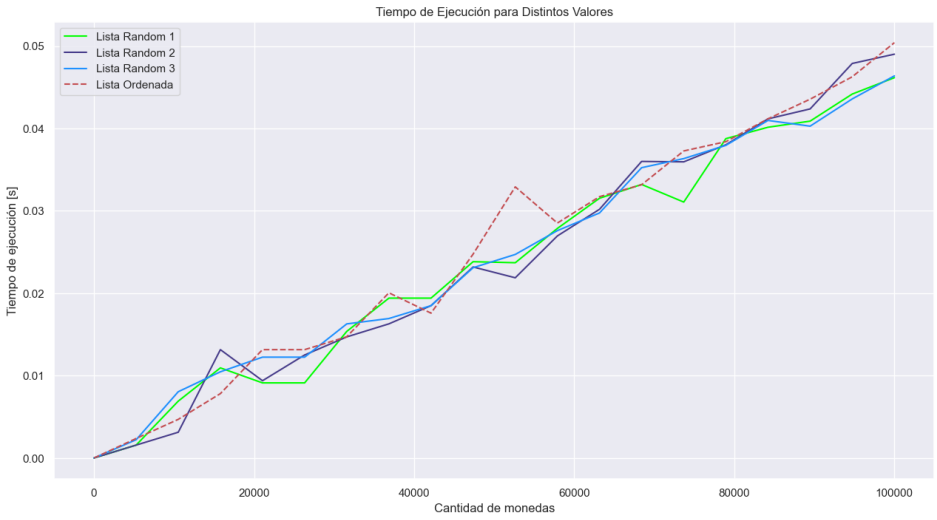
4.3. Consigna 3

El algoritmo fue modificado para dejar de agregar strings a un array, y si bien se ve diferente sigue teniendo la complejidad O(n).

```
1  from collections import deque
2
3  def elecciones(monedas, elecciones, turno_sophia):
4      if (len(monedas) == 0) or (turno_sophia and monedas[0] > monedas[-1]) or (not
       turno_sophia and monedas[0] < monedas[-1]):
5          elecciones.append(True)
6          return monedas.popleft()
7      else:
8          elecciones.append(False)
9          return monedas.pop()
10
11 def imprimir_datos(total_sophia, todas_elecciones):
12     jugadores = ["Sophia", "Mateo"]
13     for i in range(len(todas_elecciones)):
14         if todas_elecciones[i]:
15             print(f"Primera moneda para {jugadores[i%2]};", end=" ")
16         else:
17             print(f"Ultima moneda para {jugadores[i%2]};", end=" ")
18     print()
19     print(f"Ganancia de Sophia: {total_sophia}")
20
21 def monedas_greedy(monedas):
22     total_sophia = 0
23     total_mateo = 0
24     todas_elecciones = []
25     turno_de_sophia = True
26
27     monedas = deque(monedas)
28
```

```
    while len(monedas) > 0:
        if turno_de_sophia:
            total_sophia += elecciones(monedas, todas_elecciones, turno_de_sophia)
        else:
            total_mateo += elecciones(monedas, todas_elecciones, turno_de_sophia)

        turno_de_sophia = not turno_de_sophia

    print(imprimir_datos(total_sophia, todas_elecciones))
    return total_sophia, total_mateo, todas_elecciones
```

Otra parte de la consigna consistia en analizar la variabilidad de los valores de las diferentes monedas a los tiempos del algoritmo planteado. Y para esto decidimos crear un grafico.



En el grafico observamos los tiempos de ejecucion de 4 listas que contienen 100000 elementos cada una. Las 3 primeras tienen, como su nombre lo indica, un orden random, mientras que la ultima, como tambien su nombre lo indica, esta completamente ordenada.

Analizando el grafico podemos darnos cuenta de que la variabilidad y el orden que llevan las monedas realmente no genera mucho cambio en los tiempos de nuestro algoritmo. Maximo una variabilidad de 0.01 segundos.

5. Segunda parte: Mateo empieza a Jugar

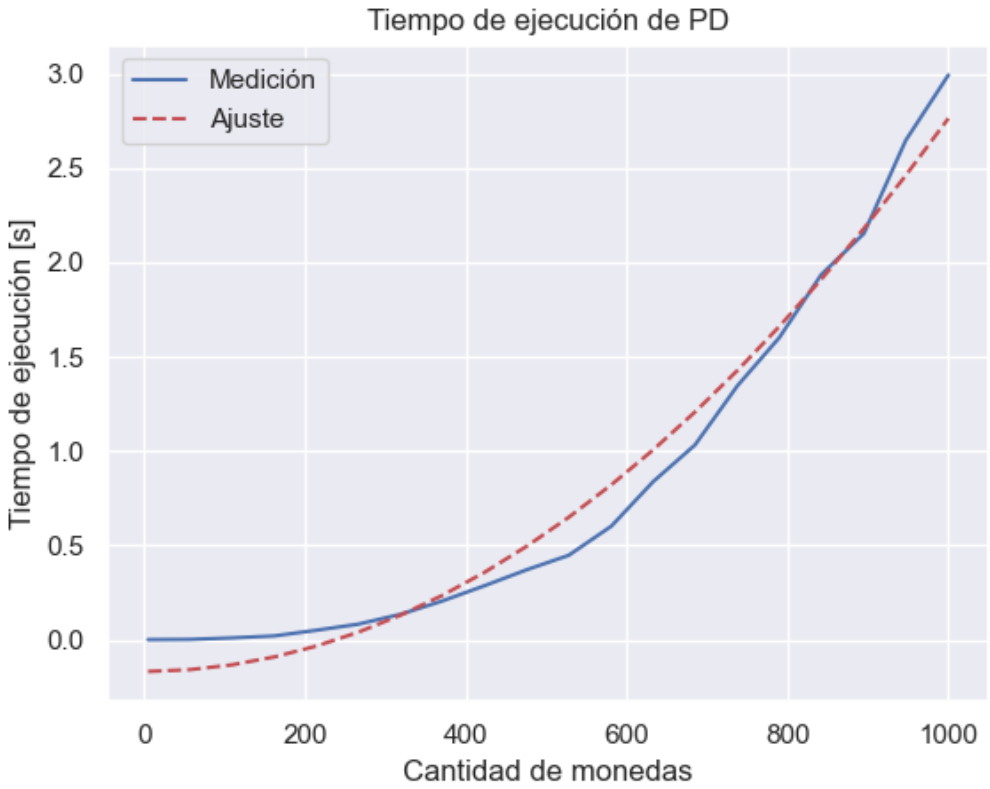
5.1. Consigna 1

Hubo, lamentablemente, un cambio de codigo para el ejercicio de la segunda parte. Lo bueno es que a pesar de que en apariencia parece diferente en realidad no cambia mucho su complejidad ni la variabilidad del tiempo de ejecucion por lo cual los graficos anteriores siguen siendo validos.

```
def valor_acumulado_mas_alto(monedas, i, j, datos, turno_actual):
    if j == i-1:
        return 0, {}

    if i == j:
        return monedas[i], {turno_actual: 0}

    if (i, j) in datos:
        return datos[(i, j)]["valor"], datos[(i, j)]["turnos"]

```

```
    turnos_primeros_actuales = {turno_actual: 0}
    i_luego_mateo, j_luego_mateo = indices_luego_turno_mateo(monedas, i+1, j,
    turnos_primeros_actuales, turno_actual+1)
    valor_acumulado, turnos_primeros_todos = valor_acumulado_mas_alto(monedas,
    i_luego_mateo, j_luego_mateo, datos, turno_actual+2)
    primero = monedas[i] + valor_acumulado

    turnos_ultimos_actuales = {turno_actual: -1}
    i_luego_mateo, j_luego_mateo = indices_luego_turno_mateo(monedas, i, j-1,
    turnos_ultimos_actuales, turno_actual+1)
    valor_acumulado, turnos_ultimos_todos = valor_acumulado_mas_alto(monedas,
    i_luego_mateo, j_luego_mateo, datos, turno_actual+2)
    ultimo = monedas[j] + valor_acumulado

    datos[(i, j)] = {}

    if primero >= ultimo:
        datos[(i, j)]["valor"] = primero
        turnos_finales = {**turnos_primeros_actuales, **turnos_primeros_todos}
    else:
        datos[(i, j)]["valor"] = ultimo
        turnos_finales = {**turnos_ultimos_actuales, **turnos_ultimos_todos}

    datos[(i, j)]["turnos"] = turnos_finales

    return datos[(i, j)]["valor"], turnos_finales

def monedas_dinamicas(monedas):
    datos = {}
    turno_actual = 0

    valor_acumulado_sophia, turnos = valor_acumulado_mas_alto(monedas, 0, len(
    monedas)-1, datos, turno_actual)
    valor_acumulado_mateo = sum(monedas) - valor_acumulado_sophia

    return valor_acumulado_sophia, turnos, valor_acumulado_mateo
```

Y aqui abajo podemos ver un grafico del tiempo de ejecucion y notar que la diferencia entre algoritmos no es mucha.



Ecuacion de recurrencia

$$dp(i,j) = \begin{cases} 0, & \text{if } j == i-1. \\ monedas[i], & \text{if } j == i. \\ max\left(\begin{array}{l} monedas[i] + \begin{cases} dp(i+1,j-1), & \text{if } monedas[i+1] <= monedas[j]. \\ dp(i+2,j), & \text{if } monedas[i+1] > monedas[j]. \end{cases}, \\ monedas[j] + \begin{cases} dp(i+1,j-1), & \text{if } monedas[i] > monedas[j-1]. \\ dp(i,j-2), & \text{if } monedas[i] <= monedas[j-1]. \end{cases} \end{array} \right) \end{cases} \quad (1)$$

Bueno, es verdad que esta ecuacion necesita algun que otro recorte pero igual funciona.

5.2. Consigna 2

Veamos con algunos ejemplos que de hecho la ecuacion escrita arriba es optima.

5.2.1. Ejemplo 1 - Fila sin monedas

El valor acumulado de Sophia tendria que ser cero si no hay monedas en la fila. Ahora vamos a ver si la ecuacion de recurrencia nos lleva a la misma respuesta.

Nuestro algoritmo hace uso de dos variables, i y j, donde la primera inicia con un valor de cero y la segunda inicia con la cantidad de monedas en la fila menos uno.

Entonces estariamos accediendo al algoritmo, en este caso, con un valor de i=0 y j=-1. Y si

miramos la ecuacion de recurrencia vemos que si ingresa con esos valores lo que le vamos a devolver es un cero.

Acabamos de comprobar que la ecuacion de recurrencia es optima para una fila vacia, aun faltan mas ejemplos.

5.2.2. Ejemplo 2 - Fila con una sola moneda

Como sabemos, Sophia tiene el primer turno asi que si la fila solo contiene una moneda el algoritmo nos tendria que devolver el valor acumulado de Sophia como el valor de la unica moneda disponible.

Accedemos al algoritmo con un valor de $i = 0$ y $j = 0$ ya que j esta definida por el tamaño de la fila (1) -1.

Podemos ver que la ecuacion de recurrencia devuelve monedas[i] cuando i==j, que es este caso y por lo tanto otra vez demostramos que esta ecuacion de recurrencia es optima.

5.2.3. Ejemplo 3 - Fila con 3 elementos

Ingresan, como ya sabemos, i y j con el valor 0 y 2 respectivamente. Entramos a la tercera condicion de la ecuacion de recurrencia, donde consigo el maximo entre dos opciones.

```
Primera opcion:
- Si monedas[0+1] <= monedas[2]:
    monedas[0] + dp(0+1, 2-1)
        dp(1, 1) = monedas[1]
    monedas[0] + monedas[1]
- Caso contrario:
    monedas[0] + dp(0+2, 2)
        dp(2,2) = monedas[2]
    monedas[0] + monedas[2]

Segunda opcion:
- Si monedas[2] > monedas[2-1]:
    monedas[2] + dp(0+1, 2-1)
        dp(1, 1) = monedas[1]
    monedas[2] + monedas[1]
- Caso contrario:
    monedas[2] + dp(0, 2-2)
        dp(0,0) = monedas[0]
    monedas[2] + monedas[0]
```

Ahora usemos unos ejemplos especificos:

Fila monedas 1 Consideremos $monedas = [2, 4, 5]$

Vamos a revisar la primera opcion:

```
monedas[0+1] <= monedas[2] -> 4 <= 5
    monedas[0] + monedas[1] = 2 + 4 = 6
```

Vamos a revisar la segunda opcion:

```
monedas[2] > monedas[2-1] -> 5 > 4
    monedas[2] + monedas[0] = 5 + 2 = 7
```

Bueno, ya tenemos los resultados de las dos opciones. Segun la ecuacion de recurrencia el maximo entre esas dos opciones es el resultado del valor acumulado de Sophia.

```
max(6, 7) = 7
```

Debido a que es una fila de monedas muy pequeña podemos darnos cuenta a simple vista que, de hecho, este resultado es correcto.

6. Tercera parte: Cambios

6.1. Reduccion

En la primera version de este archivo intentamos demostrar que el problema de Batalla Naval es NP-Completo si reducimos un problema NP-Completo conocido al nuestro, en ese caso se elijio el Bin Packing pero yo voy a elegir 3-Partition.

Para reducir el problema de 3-Partition al problema de Batalla Naval, primero hay que definir un conjunto de numeros: $A = a1, a2, ..., a3m$, donde cada valor a_i esta en codigo unario, y tambien definimos un tablero de Batalla Naval con 3 filas (n) y con un valor de columnas de m = $\sum_{i=1}^{3m} a_i$.

Las restricciones de las filas seran FR = [r, r, r], donde r = $\frac{m}{3}$. Esto asegura que cada fila contenga exactamente k casilleros ocupados por barcos.

Otra cosa a destacar es que las restricciones de las columnas son todas cero CR = [0, 0, . . . , 0], ya que realmente no nos importa cuantos casilleros se ocupan en las columnas.

Cada barco i tendra un tamaño b_i igual a a_i, por lo que tendremos 3m barcos con tamaños correspondiebtes a los elementos en A. De esta forma, si podemos colocar los barcos en el tablero cumpliendo todas las restricciones, y asi hemos encontrado una particion de los numeros en 3 subconjuntos con sumas iguales.

Entonces, para terminar, notamos que la asignacion de restricciones y la construccion del talbero, se realizan en tiempo polinomial respecto al tamaño de la entrada.

La reduccion del problema de 3-Partition al problema de la Batalla Naval queda demostrada en tiempo polinomial, demostrando asi que el problema de la Batalla Naval es un NP-Completo.