.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico: Juego de Hermanos

28 de febrero de 2025

Integrante	Padrón	Mail
Mateo Lardiez	107992	mlardiez@fi.uba.ar
Tomás Vainstein	109043	tvainstein@fi.uba.ar

1. Primera parte: Algoritmos greedy

1.1. Análisis del problema

El juego de las monedas es un juego multijugador por turnos muy simple que consiste en que dada una fila de monedas, cada jugador en su respectivo turno deberá escoger una moneda de los extremos de la fila, dando lugar al turno del siguiente jugador. Una vez que se hayan agotado las monedas, el jugador que sume mayor valor es el ganador del juego.

En particular, se están estudiando el juego para el caso en el que se tiene a Sophia y Mateo como únicos jugadores. El objetivo es el de generar un algoritmo greedy que obtenga una solución que se asegure de que Sophia siempre gane el juego, teniendo en cuenta que como Mateo es muy chico Sophia juega por él (hace trampa).

1.2. Algoritmo greedy

Dado un arreglo de monedas sin ordenar, con una cantidad de monedas que puede ser par o impar, el siguiente algoritmo greedy devuelve la solución óptima al problema de las monedas devolviendo como salida una lista de las monedas de Sophia (las mejores) y una lista de las monedas de Mateo (las peores).

Para la implementación del algoritmo se tuvieron en cuenta las restricciones de que el juego es un juego por turnos y de que solo se pueden agarrar monedas de los extremos. Nótese que Mateo siempre tendrá las peores monedas para sí mismo.

```
def play_game(coins_list):
    sophia_coins = []
    mateo_coins = []

    sophia_is_playing = True

    beginning = 0
    end = len(coins_list) - 1

    while beginning <= end:

        selected_coin = -1

        if sophia_is_playing:
            if coins_list[beginning] <= coins_list[end]:
                selected_coin = coins_list[end]
                end -=1
            else:
                selected_coin = coins_list[beginning]
                beginning +=1

            sophia_coins.append(selected_coin)
            sophia_is_playing = False
        else:
            if coins_list[beginning] >= coins_list[end]:
                selected_coin = coins_list[end]
                end -=1
            else:
                selected_coin = coins_list[beginning]
                beginning +=1

            mateo_coins.append(selected_coin)
            sophia_is_playing = True

    return sophia_coins, mateo_coins
```

1.3. Solución óptima

Por definición, en un algoritmo greedy las soluciones óptimas se construyen a partir de una secuencia de óptimos locales que paso a paso van llevando a la mejor solución a un problema. La lógica greedy es la de tomar la mejor decisión en cada paso de un problema, y que el conjunto de las mejores decisiones lleven al óptimo.

Esta definición podría confundirse con la de la programación dinámica. Pero la diferencia principal es que los algoritmos greedy no tienen memoria.

Claramente no todo problema solucionado de forma greedy tiene solución óptima, sin embargo, para el caso del juego de las monedas con las restricciones definidas anteriormente sí se alcanza el óptimo.

Vamos a demostrarlo siguiendo la logica del algoritmo.

El algoritmo recibe una lista de monedas $C = [m_1, m_2, ..., m_n]$. En cada turno, se puede quitar las monedas en la posición $i = 0$ o $i = n-1$ de C, siendo n la cantidad de monedas en C. Llamemosle S a la ganancia de Sophia, y M a la ganancia de Mateo.

Nuestra regla greedy garantiza que en cada turno j ($0 \leq j \leq n$), se cumple $S[j] > M[j]$, ya que Sophia siempre opta por la moneda de mayor valor disponible, mientras que Mateo selecciona la de menor valor. (Es importante señalar que todas las monedas tienen valores únicos y no se repiten).

Para comprender mejor esta estrategia, observemos el primer turno ($j = 0$): Sophia escoge la moneda de mayor valor entre m_1 y m_n, es decir, máx(m_1, m_n). Luego, en el siguiente turno, Mateo elige la moneda de menor valor entre las dos restantes:

- Si Sophia tomó m_1, Mateo seleccionará mín(m_2, m_n).
- Si Sophia tomó m_n, Mateo elegirá mín(m_1, m_{n-1}).

Siguiendo esta lógica, Sophia optimiza su ganancia relativa en cada turno, asegurándose de mantener su ventaja sobre Mateo.

Demostraremos que el algoritmo devuelve un conjunto óptimo A, que maximiza la ganancia de Sophia, asegurando que en cada turno ella elige la moneda de mayor valor disponible.

Utilizaremos un razonamiento por contradicción. Supongamos que A no es óptimo, lo que implicaría la existencia de una estrategia O superior, que maximiza aún más la ganancia de Sophia, es decir, $S(O) > S(A)$. Si esto fuera cierto, en al menos un turno, Sophia habría elegido una moneda menos valiosa en A que en O, lo que reduciría su ganancia.

Dado que en la estrategia A, Sophia siempre selecciona máx(m_1, m_n), cualquier estrategia que la supere debería elegir una moneda diferente en algún turno. Como las únicas opciones en cada jugada son m_1 y m_n, la unica alternativa a la estrategia de A seria elegir mín(m_1, m_n) en al menos un turno. Sabemos que Sophia toma decisiones considerando también la jugada de Mateo, quien siempre elige la moneda de menor valor entre las disponibles. Para mayor claridad, en lo que sigue hablaremos de Mateo como si tomara sus propias decisiones en su turno.

Teniendo en cuenta que Sophia sacó la moneda de menor valor en el primer turno, en el próximo turno, Mateo podrá elegir entre las 2 monedas que quedan, habiendo la posibilidad de que una sea mayor que la moneda de Sophia, entonces tendria una mayor ventaja hasta ese momento. Si en un turno j, Sophia en O elige una moneda de menor valor que la elegida en A, entonces Mateo podrá elegir una moneda de mayor valor que en A, incrementando su ganancia respecto a Sophia. Por lo tanto, en cada turno, A siempre proporciona una mejor o igual ganancia a Sophia en comparación con O, lo que contradice nuestra suposición inicial de que $S(O) > S(A)$. Por lo tanto, la estrategia A no puede ser superada por ninguna otra estrategia O, lo que implica que A es óptima, y que cualquier desviación de A resulta en una peor ganancia para Sophia.

Por lo tanto, está probado que el algoritmo greedy de Sophia, donde selecciona la moneda de mayor valor entre las opciones disponibles y elige la de menor valor para Mateo, siempre es óptimo. Si optara por una polıitica diferente, terminarıia perjudicándose porque la relación de ganancia entre los dos es directa, lo que contradice la idea de que podria haber un mejor resultado. Si una

ganancia aumenta, la otra disminuye, por lo que Sophia siempre debe maximizar su ganancia en cada turno y eso lo hace eligiendo la moneda de mayor valor, pero también debe minimizar la ganancia de Mateo y eso lo hace eligiendo la moneda de menor valor.

Respecto al análisis de la complejidad algorítmica. Se formulo la hipótesis teórica de que el algoritmo propuesto tiene una complejidad $O(n)$. En cada paso del problema se está realizando un problema de decisión entre dos monedas, lo cual se resuelve en tiempo $O(1)$, por otra parte, la lista se itera alternando sus extremos de afuera hacia adentro, al iterar sobre todos los elementos se tendría una complejidad $O(n)$ (la complejidad de decidir cuál moneda tomar es despreciable comparada a la complejidad de moverse por la lista de monedas).

1.4. Variabilidad y optimalidad

La variabilidad de los valores no afecta la optimalidad del algoritmo, ya que Sophia siempre maximiza su ventaja. En su turno, elige la moneda más grande disponible entre los extremos de la lista, esto asegura que en cada decisión acumule el mayor valor posible. Otra variable a considerar es que Sophia controla las elecciones de Mateo. En el turno de su hermano, Sophia lo obliga a tomar la moneda de menor valor disponible en los extremos, minimizando el valor que puede acumular. Independientemente de la distribución de los valores, el algoritmo garantiza que Sophia obtendrá el máximo valor. En el caso extremo donde los valores de la lista sean todos iguales, la única influencia es la cantidad de monedas, dejando el siguiente caso:

- Si n es impar, Sophia siempre gana.
- Si n es par, se produce un empate.

n = cantidad de monedas.

Ejemplo de Ejecución: Tenemos un vector de monedas con los valores: [10, 5, 9, 11, 2, 6].

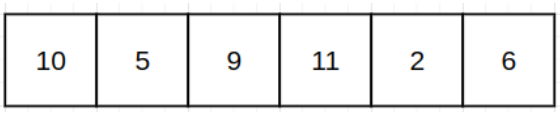


Figura 1: Vector

Iteramos el vector n veces, siendo n la longitud del vector de monedas y asignamos los índices primero y último del vector.

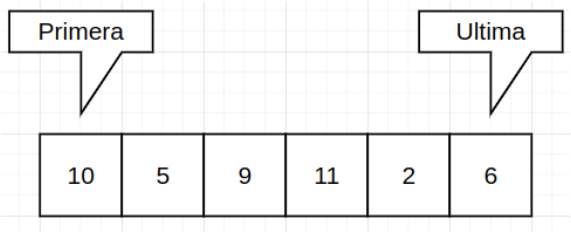


Figura 2: Extremos posibles del vector de monedas

Luego Sophia comienza eligiendo una moneda. Para ello compara los valores de la primera y última posición para así quedarse con la moneda de mayor valor. Como $10 > 6$, Sophia se queda con la moneda 10 y el índice se actualiza sumándole una unidad.

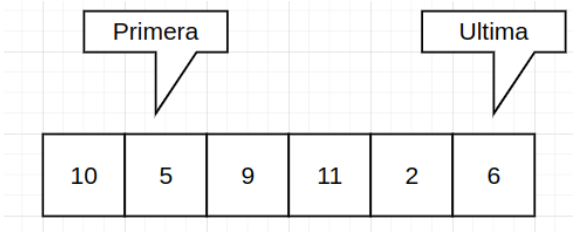


Figura 3: Indices en el turno de Mateo

En el siguiente turno, Sophia deberá elegir una moneda para Mateo que le permita ganar siempre a Sophia, por lo que le dará la moneda de menor valor. En el algoritmo planteado simplemente movemos en una unidad el índice correspondiente, en este caso será el índice "primera" ya que 5 ¡6. Luego nuevamente le tocara elegir la moneda de mayor valor a Sophia.

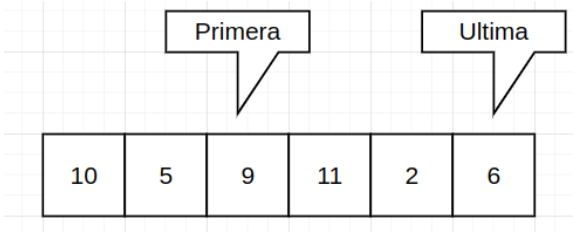


Figura 4: Indices en el 2do turno de Sophia

Y pasaremos al siguiente turno hasta que no queden más monedas. Al pasar los turnos el vector de monedas de Sophia queda de esta forma: [10,9,11]

Suma Máxima de Sophia: 30

Notar que 30 es la suma máxima del vector de monedas por lo que Sophia **SIEMPRE** obtendrá la mejor solución.

1.5. Ejemplos de ejecución

A modo de ejemplos, se utilizaron los datasets provistos por la cátedra para testear la optimalidad del algoritmo implementado.

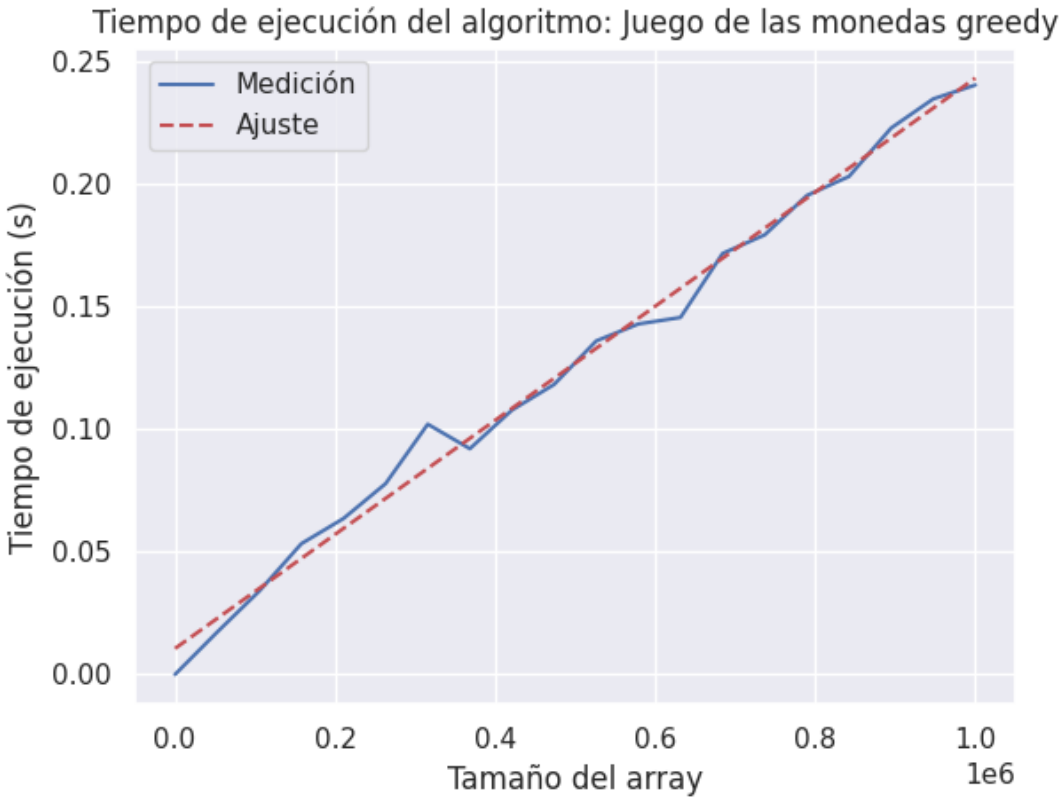
- 20.txt espera ganancia de 7165 para Sophia.
- 25.txt espera ganancia de 9135 para Sophia.
- 50.txt espera una ganancia de 17750 para Sophia.
- 100.txt espera una ganancia de 35009 para Sophia.
- 1000.txt espera una ganancia de 357814 para Sophia.
- 10000.txt espera una ganancia de 3550307 para Sophia.
- 20000.txt espera una ganancia de 7139357 para Sophia.

Al cumplir con la salida esperada de todos los ejemplos por se podría decir el algoritmo greedy implementado funciona correctamente.

Cabe aclarar que en los últimos 2 casos (10000.txt y 20000.txt) hay casos en que ambos extremos tienen el mismo valor, por lo que se decidió elegir que, en esta situación, si ambos valores de la primer moneda y la ultima son iguales, se tome la ultima moneda

1.6. Comprobación de la complejidad

Para corroborar empíricamente la complejidad teórica de $O(n)$, realizamos mediciones del tiempo de ejecución del algoritmo. Estos tiempos fueron luego graficados en función de n (el tamaño del problema) para observar la relación entre el tamaño de la entrada y el tiempo de ejecución.



Se puede ver que el ajuste lineal del tiempo de ejecucion sugiere que el algoritmo escala linealmente con el tamaño del array. Se puede apreciar que el gráfico corresponde con la hipótesis de la complejidad teórica.

1.7. Conclusiones y comentarios adicionales

Se utilizo un enfoque greedy para hallar la solución óptima al juego de las monedas entre Sophia y Mateo. Dadas las restricciones y a la implementación de la solución, Sophia siempre ganará el juego con el puntaje máximo y con complejidad $O(n)$.

Se comprobó que siempre se llega a la solución óptima esperada, utilizando diferentes datasets.

Respecto al proceso de implementación del algoritmo, inicialmente se consideró utilizar métodos de ordenamiento sobre las monedas antes de iniciar el juego, lo que hacía que la solución óptima se resumiese en siempre sacar monedas del mismo lugar hasta terminar el juego. Esta idea fue descartada inmediatamente al corroborar que rompía por completa con la lógica y esencia del juego.

En términos de implementación, no se presentaron grandes dificultades, salvo en la creación de los gráficos de complejidad, que requirieron del uso de librerías de las cuales se desconocía su uso. Estos gráficos confirmaron la relación lineal entre el tiempo de ejecución y el número de monedas, alineándose con la complejidad teórica calculada.

2. Segunda parte: Mateo empieza a jugar

2.1. Análisis del problema

Al igual que en la sección anterior se está jugando al juego de las monedas entre Sophia y Mateo, pero con un particular cambio, ahora Mateo empieza a jugar con lógica greedy (es decir, Sophia ya no escoge las monedas por él), lo cual hace que Sophia recurra a un cambio en la forma en que juega.

En esta ocasión, Sophia empieza a usar programación dinámica obtener las mejores monedas del juego, y de ser posible ganar (sí, ahora es posible que Sophia pierda). Si bien Sophia puede perder, si lo hace, lo hará teniendo la mejor colección de monedas.

Para diseñar el algoritmo se tuvo en cuenta que como ahora Mateo juega con lógica greedy siempre va a tomar las mejores monedas de los extremos.

Para lograr resolver este problema se pensó lo siguiente:

Como las monedas siempre se agarran de los extremos de la colección de monedas. Se definieron a estos extremos como **i** y **j**. Los cuales van variando a la medida que se van sacando monedas de la colección.

Por otra parte, si arma una matriz de nxn (siendo n la cantidad de monedas del juego). Y se definen los índices de las filas como **i** y a los índices de las columnas como **j**, es posible obtener una matriz con los óptimos para los distintos valores que pueden tomar los extremos de la colección de monedas.

Los índices **i** y **j** indican en que posición de la colección de n monedas se encuentran los extremos actuales.

Esta matriz permitiría construir los óptimos para cada posible valor de **i** y de **j**, y así, basándose en una lógica inductiva y al uso de memoization, se podría obtener la solución óptima al juego de las monedas para Sophia.

Ejemplo de una matriz teniendo 5 monedas:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

El óptimo para el juego de las monedas se encontraría en la posición i=0, j=n-1. Estando las monedas almacenadas en una lista.

Esta descripción es solo un acercamiento inicial a la implementación del problema. Y a continuación se la describira en más detalle mediante la ecuación de recurrencia.

2.2. Ecuación de recurrencia

La siguiente ecuación de recurrencia es descrita basándose en lógica algorítmica de lenguajes de programación (en lo que se refiere a la forma de acceder a las monedas, a la variación de los extremos, y al acceso de la matriz de óptimos)

En cada paso se puede tomar una moneda de un extremo u otro, en base a cual moneda se tomé será necesario calcular un nuevo óptimo para los nuevos externos teniendo en cuenta que Mateo siempre elegirá las monedas de mayor valor entre los extremos.

$$\text{OPT}(i,j) = \text{máx}\left(\text{monedas}[i]+\text{mejor\_moneda\_siguiente}(i+1,j),\ \text{monedas}[j]+\text{mejor\_moneda\_siguiente}(i,j-1)\right)$$

$$\begin{aligned} &i = \text{inicio de la lista de monedas, varía según la moneda seleccionada} \\ &j = \text{fin de la lista de monedas, varía según la moneda seleccionada} \\ &\text{monedas}[i] = \text{valor de la moneda en la posición } i \\ &\text{monedas}[j] = \text{valor de la moneda en la posición } j \end{aligned} \tag{1}$$

$$\text{mejor\_moneda\_siguiente} = \begin{cases} 0, & \text{si } i+1 > j \text{ (no quedan monedas para Sofía)} \\ \text{OPT}(i+1, j), & \text{si monedas}[i] \geq \text{monedas}[j] \text{ (Mateo toma la izquierda)} \\ \text{OPT}(i, j-1), & \text{si monedas}[i] < \text{monedas}[j] \text{ (Mateo toma la derecha)} \end{cases}$$

2.3. Algoritmo planteado

El siguiente código muestra la implementación del algoritmo haciendo uso de la ecuación de recurrencia descrita anteriormente.

```
from libs import datasets_parser

def best_next_coin(i, j, gains_matrix, coins_list):
    if i+1 >j:
        return 0
    elif coins_list[i] >= coins_list[j]:
        return gains_matrix[i+1][j]
    else:
        return gains_matrix[i][j-1]


def get_gains_matrix(coins_list):

    coins_list_len = len(coins_list)

    gains_matrix = [[0 for _ in range(coins_list_len)] for _ in range(
    coins_list_len)]

    for i in range(coins_list_len-1, 0, -1):

        for j in range(0, coins_list_len):

            if  i > j:
                gains_matrix[i][j] = 0
            else:
                gains_matrix[i][j] = max(coins_list[i] + best_next_coin(i+1, j,
    gains_matrix, coins_list),
                                         coins_list[j] + best_next_coin(i, j-1,
    gains_matrix, coins_list))

    # La iteracion anterior llega hasta i=1 porque el cero del primer for (el que
    itera i) es excluyente. Por eso se hace este for aparte para abarcar el caso de
     la fila 0
    for j in range(0, coins_list_len):
        i = 0
        gains_matrix[i][j] = max(coins_list[i] + best_next_coin(i + 1, j,
    gains_matrix, coins_list),
                                 coins_list[j] + best_next_coin(i, j - 1, gains_matrix,
    coins_list))

    return gains_matrix
```

Este algoritmo utiliza una lógica Top Down para armar el espacio de soluciones requeridos para resolver el problema.

La matriz inicialmente tiene todos sus casilleros seteados en cero (0). La matriz se recorre de abajo hacia arriba y de izquierda a derecha, utilizando en cada paso la ecuación de recurrencia para obtener el óptimo local (notar que a diferencia de la solución greedy, acá la solución al problema sí tiene cierto nivel de memoria que usado para calcular los óptimos locales).

La función `get_gains_matriz()` se ocupa de rellenar la matriz como corresponde. Los for anidados permiten iterar sobre la matriz y calcular los óptimos para cada posición. La utilización de dos bloques de fors se debe particularidades de python en como genera los valores en un **range()**.

El primer bloque (el de los dos fors) itera desde la última fila a la segunda, y de izquierda a derecha. El segundo bloque itera sobre la primera fila, de izquierda a derecha.

La posición (0, n-1) contiene el óptimo buscado para esta versión del juego de las monedas.

Como comentario adicional, se menciona que en la matriz aquellas posiciones donde i ¿j tiene como valor óptimo cero (0) ya que no corresponden a un caso lógico del juego de las monedas.

Un ejemplo para una matriz de 4*4 es el siguiente:

13	14	15	16
0	10	11	12
0	0	7	8
0	0	0	4

Nuevamente, los valores de las monedas no afectan a los tiempos de ejecución del algoritmo (por el mismo motivo que el descrito en la sección de la solución greedy), como sí lo hacen la cantidad de monedas con las que se va a jugar. De hecho en esta ocasión el tiempo aumenta de forma cuadrática en base al tamaño del problema.

2.4. Demostracion de Optimalidad

La solución es óptima porque nos permite obtener el conjunto de monedas que maximiza el monto total obtenido por Sophia, es decir, la sumatoria de los valores de las monedas, teniendo en cuenta que Sophia siempre comienza jugando y que tiene que sacar una moneda de alguno de los dos extremos en cada turno.

Demostremos esto haciendo una prueba por induccion.

Para demostrar que la ecuación de recurrencia es óptima, utilizamos inducción sobre la longitud del intervalo de monedas.

Caso Base:

- **Caso 1 ($i = j$)**: Solo hay una moneda. Sophia la toma, y su ganancia es:

$$OPT(\text{i}, \text{j}) = \text{M[i]}$$

  lo cual es trivialmente óptimo.

- **Caso 2 ($i + 1 = j$)**: Hay dos monedas, entonces Sophia elige la de mayor valor:

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

  lo cual también es óptimo.

Hipótesis Inductiva: Sea $OPT(i, j)$ la función que representa la máxima ganancia posible que Sophia puede obtener al jugar con las monedas en el intervalo $[i, j]$. Supondremos que para cualquier subintervalo de tamaño k, con $k < |M|$, la ecuación de recurrencia proporciona correctamente la máxima ganancia de Sophia.

Queremos verificar que esta ecuación sigue siendo válida para un intervalo de tamaño $k + 1$.

Paso Inductivo: Consideremos un intervalo de $k + 1$ monedas, es decir, $j - i + 1 = k + 1$. En su turno, Sophia puede elegir entre dos opciones:

- **Si elige** $M[i]$, el conjunto de monedas restante es $[i+1, j]$, que tiene tamaño k. En el siguiente turno, Mateo tomará la moneda de mayor valor entre $M[i+1]$ y $M[j]$, reduciendo el problema a un nuevo intervalo de tamaño $k - 1$.

- **Si elige** $M[j]$, el conjunto de monedas restante es $[i, j-1]$, también de tamaño k. Mateo entonces tomará la moneda de mayor valor entre $M[i]$ y $M[j-1]$, reduciendo nuevamente el problema a un intervalo de tamaño $k-1$.

En ambos casos, la nueva instancia del problema es de tamaño $k-1$ monedas, y por la hipótesis inductiva, sabemos que la ecuación de recurrencia proporcionará la máxima ganancia posible para Sophia.

Dado que la ecuación de recurrencia cubre todos los casos posibles y considera correctamente la elección de Mateo (siempre tomando la moneda de mayor valor), se concluye que la solución obtenida para $k+1$ monedas también es correcta.

Por lo tanto, por el **principio de inducción matemática**, la ecuación de recurrencia permite calcular la ganancia máxima de Sophia para cualquier cantidad de monedas. Es decir, $OPT(i, j)$ garantiza que Sophia obtenga la máxima ganancia posible, para todo $i \leq j < |M|$.

Vale aclarar que esta solución nos permite conseguir el óptimo, es decir, el conjunto de monedas que maximiza el monto total obtenido por Sophia, pero no nos asegura que Sophia pueda ganar siempre. Esto se demuestra fácilmente con el siguiente ejemplo, si tenemos el conjunto de monedas [1,10,1], Sophia comenzaría sacando alguna de las monedas de valor 1 y Mateo sacaría la de valor 10, por lo que ganaría el juego independientemente de la estrategia tomada por Sophia.

2.5. Reconstruccion de la solucion

Ahora vamos a explicar la reconstruccion de una solucion, a partir de haber obtenido la matriz completa con todos sus valores calculados.

Para entender como es que se reconstruye la solucion, vamos a mostrar el siguiente codigo utilizado:

```
def reconstruct_solution(coins_list, gains_matrix):
    i = 0
    j = len(coins_list) - 1
    choices_sophia = []  # Lista para almacenar el orden de elecciones de monedas de Sophia
    choices_mateo = []

    while i <= j:
        left_option = coins_list[i] + best_next_coin(i+1, j, gains_matrix, coins_list)
        right_option = coins_list[j] + best_next_coin(i, j-1, gains_matrix, coins_list)

        if left_option >= right_option:
            choices_sophia.append(coins_list[i])
            i += 1
            if i > j: # No quedan mas monedas
                continue
            if coins_list[i] >= coins_list[j]:
                choices_mateo.append(coins_list[i])
                i += 1
            else:
                choices_mateo.append(coins_list[j])
                j-= 1
        else:
            choices_sophia.append(coins_list[j])
            j -= 1
            if i > j: #No quedan mas monedas
                continue
            if coins_list[i] >= coins_list[j]:
                choices_mateo.append(coins_list[i])
                i += 1
            else:
                choices_mateo.append(coins_list[j])
                j-= 1
```

```
33
34      return choices_sophia, choices_mateo
```

Lo que hace el codigo, que es lo que vamos a aplicarle a analizar para reconstruir la solucion es devolver el orden en el que Sophia toma sus monedas, y el orden en que Mateo toma las suyas. Y lo hace de la siguiente manera. Sabemos que nuestra solucion final se encuentra en la posicion final i=0, j=n-1 (n siendo el largo de la lista). Proponemos como condicion de corte que cuando el indice izquierdo (i) sea mayor al indice derecho (j) es cuando ya tomamos todas las monedas de la lista. Comenzamos nuestra iteracion, siendo Sophia, y buscamos que valor nos da mas ganancia. Si el de la izquierda o el de la derecha. Estos 2 calculos los hacemos a partir de la funcion bestNextCoin, la cual te devuelve el valor de la matrix de ganancia para [i+1,j], si es que Mateo tomaria la moneda que se encuentra en el inicio, o el valor de gainMatrix[i,j-1] si Mateo toma la que esta del lado derecho. Cabe aclarar que si Sophia agarra la moneda inicio, tambien se sumaria i+1, y si agarra la moneda fin, tambien se restaria j-1.

Cuando tenemos estos 2 valores, los comparamos y el que mayor valor nos devuelva es el camino que vamos a elegir para Sophia. Por ende tomamos la moneda en la posicion correspondiente y la agregamos a la lista de monedas de Sophia. Luego de este proceso es el turno de Mateo el cual, si es que siguen quedando monedas, tomara la mas alta y la guardara en su lista de monedas. Nuevamente le tocaria a Sophia. Esto se repite hasta que no haya mas monedas, entonces se devuelven las 2 listas completas de monedas.

2.6. Ejemplos de ejecución

Como ejemplos de ejecución se utilizaron los datasets provistos por la cátedra para verificar el correcto funcionamiento del algoritmo.

- 5.txt espera ganancia de 1483 para Sophia.
- 10.txt espera ganancia de 2338 para Sophia.
- 20.txt espera ganancia de 5234 para Sophia.
- 25.txt espera ganancia de 7491 para Sophia.
- 50.txt espera una ganancia de 14976 para Sophia.
- 100.txt espera una ganancia de 28844 para Sophia.
- 1000.txt espera una ganancia de 1401590 para Sophia.
- 10000.txt espera una ganancia de 2869340 para Sophia.
- 20000.txt espera una ganancia de 34107537 para Sophia.

2.7. Complejidad y medición

El bloque de los fors anidado tiene como complejidad algorítmica $O(N * (N-1))$.

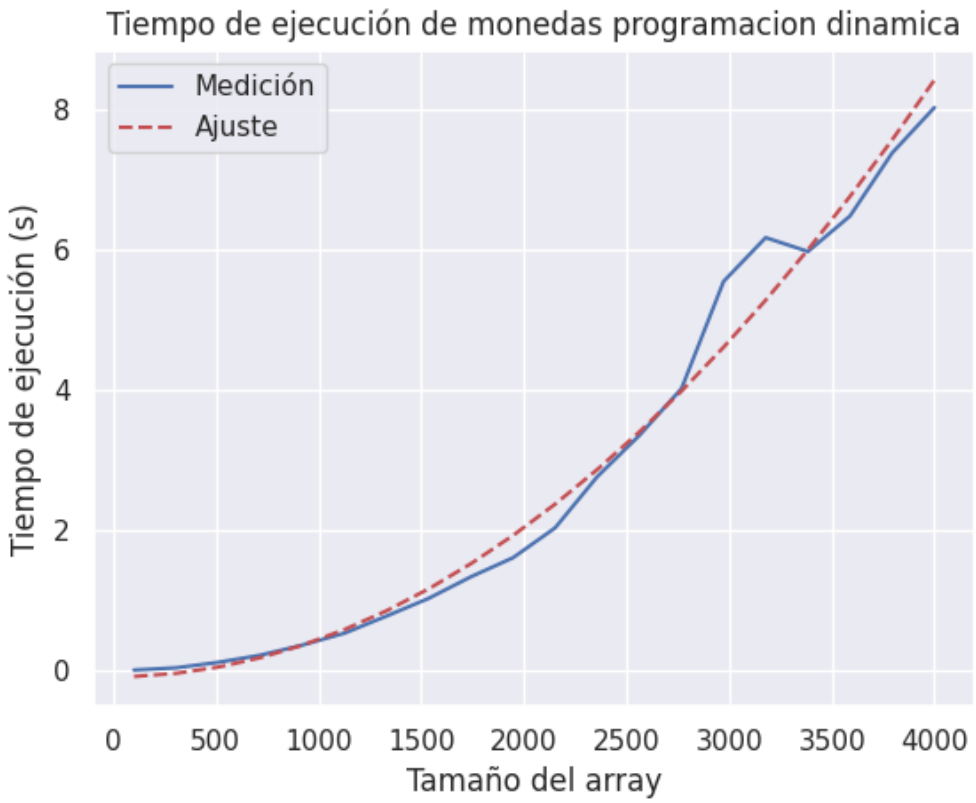
Por otro lado, el último for tiene como complejidad algorítmica $O(N)$

y el for de abajo tiene complejidad $O(N)$. Por lo que quedaría:

Por aproximación se simplifica a: $O(N^2) + O(N)$ y nuevamente se simplifica a $O(N^2)$

Por ende este algoritmo tiene complejidad: $O(N^2)$

A modo de validar la complejidad teórica se realizó el siguiente gráfico tiempo de ejecución VS tamaño del problema.



Lo cual es acorde con la hipótesis de la complejidad cuadrática.

2.8. Conclusiones y comentarios adicionales

La complejidad de esta sección fue mayor a la de la primera parte (complejidad de implementar una solución al problema). Fue necesario hacer una análisis exhaustivo del problema y de la forma en que se descomponía el problema en base a las monedas tomadas.

Inicialmente, se llevó la ecuación de recurrencia a un algoritmo mediante una lógica recursiva, que si bien era correcto lógicamente, en términos de eficiencia no lo era, ya que para tamaños muy grandes del problema se agotaba el stack asignado a la ejecución del algoritmo. Por lo tanto, se optó por una implementación iterativa.

Se comprobó la optimalidad de la implementación mediante la ejecución de los datasets provistos por la cátedra. Y se logró demostrar que la implementación de la solución es acorde con la complejidad algorítmica teórica.

3. Tercera parte: Cambios

3.1. Demostración de que el problema pertenece a NP

Para demostrar que el problema de la **Batalla Naval** pertenece a la clase **NP**, se debe probar que existe un *certificador* eficiente, es decir, una función que en tiempo polinomial pueda verificar que una salida del problema de la batalla naval es una solución válida.

3.1.1. Estructura del problema

El problema se define por los siguientes datos de entrada:

- **Tablero:** Una matriz $n \times m$ que representa el área donde se deben colocar los barcos.
- **Restricciones de filas:** Una lista de tamaño n que indica el número máximo de casillas que se pueden ocupar en cada fila.
- **Restricciones de columnas:** Una lista de tamaño m que indica el número máximo de casillas que se pueden ocupar en cada columna.
- **Barcos:** Una lista donde cada barco está definido por su tamaño. Los barcos deben colocarse sin superponerse ni estar adyacentes entre sí.

3.1.2. Proceso de verificación

El proceso de verificación consiste en generar un algoritmo que en tiempo polinomial analice si la solucion propuesta es la correcta. Para esto debe recorrer el tablero propuesto y comprobar que cumple con todas las reglas del problema.

Para validar nuestro problema y demostrar que esta en NP, se utiliza el siguiente codigo:

```
def verificador_batalla_naval(tablero, restriccion_filas, restriccion_columnas,
    barcos):
    if (len(tablero) == 0) or len(tablero[0] == 0) or (len(barcos) == 0):
        return False

    # Si no hay lugar para insertar todos los barcos que hay
    if (len(tablero)*len(tablero[0])) < sum(barcos):
        return False

    # Validar que las rescricciones sean posiblesl
    if not validar_restricciones(restriccion_filas, restriccion_columnas, barcos):
        return False


    for i in range(len(tablero)):
        for j in range(len(tablero[0])):
            if tablero[i][j] == 1: # La casilla esta ocupada

                # Analiza que no haya barcos adyacentes
                if not validar_posicion_barco(tablero, barcos, i, j): #(O(m*n) + k)
                    return False

    if len(barcos) > 0:
        return False

    return True
```

Este codigo recibe el tablero, para el cual las celdas ocupadas valdran 1 y las libres 0, una lista de restriccion de filas, una lista de restriccion de columnas, y una lista de barcos cuyo valor es el largo del barco.

Este algoritmo se encarga de validar los siguientes requisitos: No pueden haber barcos adyacentes, los barcos tienen un ancho de una celda, se debe cumplir con las restricciones para filas y columnas y se deben colocar todos los barcos.

Analisis de complejidad:

- n: número de filas.
- m: número de columnas.
- k: número de barcos.
- La función `validar_restricciones` tiene una complejidad de $O(m \times n)$ por la validación de las restricciones de filas y columnas.
- La función `validar_posicion_barco` tiene una complejidad de $O(\text{máx}(m, n) + k)$. En el peor caso, se recorre toda la fila o columna, y se recorre la lista de barcos.

Para analizar la complejidad total de la función `verificador_batalla_naval`, se deben tener en cuenta los siguientes aspectos:

- La función `validar_restricciones` se ejecuta una sola vez, por lo que su complejidad no se multiplica por la cantidad de barcos.
- En el bucle, se recorren todas las filas y columnas del tablero. Podríamos llegar a pensar que en el peor de los casos, se haría $n \times m$ veces la llamada a `validar_posicion_barcos`, pero esto no es así.
- Dadas las restricciones de adyacencias impuestas, en un tablero podría haber como máximo x cantidad de barcos de longitud 1, siendo $x \approx n/2 \times m/2 + n/2$ (si n es impar) $+ m/2$ (si m es impar). Esto significa que se podría hacer como máximo x llamadas a `validar_posicion_barcos`. Para ilustrar esto, supongamos un tablero de 5x5 con barcos de tamaño 1:

  ```
  [1,0,1,0,1]
  [0,0,0,0,0]
  [1,0,1,0,1]
  [0,0,0,0,0]
  [1,0,1,0,1]
  ```

  La cantidad de barcos es de $5/2 \times 5/2 + 5/2 + 5/2 = 2 \times 2 + 2 + 2 \approx 8$, por lo que se harían 8 llamadas a `validar_posicion_barcos`, con un error de 1 para cuando n y m son impares.
- Como x depende de las dimensiones del tablero, al tender a infinito, podríamos decir que la complejidad de la función `verificador_batalla_naval` es de $O(m \times n \times (\text{máx}(m, n) + k))$. Pero hay que tener en cuenta que para que esto suceda, los barcos deben tener un tamaño de 1, y si lo tuvieran, las operaciones realizadas en `validar_posicion_barcos` se harían en $O(1)$ en lugar de $O(\text{máx}(m, n) + k)$.
- Dicho esto, concluimos en que el peor escenario posible se da cuando los barcos tienen un tamaño igual a la cantidad de filas o columnas del tablero. En este caso, $x = m/2 + 1$ (si m es impar) ó $x = n/2 + 1$ (si n es impar). Volvamos al ejemplo de la matriz de 5x5 con barcos de tamaño 5, pero teniendo en cuenta esto:

  ```
  [1,1,1,1,1]
  [0,0,0,0,0]
  [1,1,1,1,1]
  [0,0,0,0,0]
  [1,1,1,1,1]
  ```

ó

```
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
```

- Es decir, se ejecutaría $n/2$ veces la función `validar_posicion_barcos`, que tendría una complejidad de $O(m)$ ó $m/2$ veces la función `validar_posicion_barcos`, que tendría una complejidad de $O(n)$. Por lo tanto, considerando por ejemplo que la cantidad de barcos es $n/2$, la complejidad total de la función `verificador_batalla_naval` es bastante menor que $O(m \times n \times (\text{máx}(m,n)+k))$, siendo $O((m \times n) - n/2 + n/2 \times (m+k)) = O((m \times n) + (n \times m)) = O(m \times n)$.

Resumiendo: Nuestro validador verifica la configuración del tablero en tiempo polinomial, específicamente en $O(m \times n)$, donde m y n son las dimensiones del tablero. Esto demuestra que nuestro validador es eficiente y, por ende, **el problema la Batalla Naval es NP**.

3.1.3. Complejidad del algoritmo de verificación

La complejidad del proceso de verificación se descompone en los siguientes pasos:

- **Recorrer el tablero:** La operación de recorrer todas las casillas del tablero tiene una complejidad de $O(n \times m)$.
- **Validaciones locales:** Para cada casilla ocupada, se verifican la adyacencia y el tamaño del barco. Cada una de estas validaciones tiene una complejidad de $O(1)$ por casilla.
- **Verificación final de restricciones:** Al finalizar el recorrido, se comprueba que las restricciones de las filas y columnas se cumplen correctamente. Esta verificación tiene una complejidad de $O(n + m)$.

Por lo tanto, la complejidad total del proceso de verificación es:

$$O(n \times m) + O(n + m) + O(1) = O(n \times m) + O(n) + O(m) + O(1) = O(n \times m)$$

Dado que existe un certificador que puede verificar en tiempo polinómico, se concluye que el problema de la **Batalla Naval** pertenece a **NP**.

3.2. Demostración de que el problema es NP-Completo

Para demostrar que un problema pertenece a la clase de problemas NP-Completo, se deben cumplir con dos requisitos:

1. **Pertenencia a NP:** Mostrar que existe un certificador que pueda verificar una solución en tiempo polinómico. Esto fue demostrado en el punto anterior.
2. **Reducción desde un problema NP-Completo conocido:** Reducir un problema NP-Completo conocido al problema en cuestión, demostrando que si se puede resolver el segundo, también se puede resolver el primero.

El primer paso ya lo demostramos previamente.

Para el segundo paso debemos armar una reduccion que tome cualquier instancia del problema NP-Completo, en nuestro caso 3-Partition y la convierta en una instancia de Batalla Naval en tiempo polinomial. Luego demostrar que el problema 3-Partition tiene solución **si y sólo si** la instancia de Batalla Naval generada por la reducción tiene solución.

3.2.1. Reducción desde un problema NP-Completo conocido

Descripción del problema 3-Partition: Dado un conjunto S de $3m$ números enteros positivos y un entero B, ¿es posible dividir S en m subconjuntos disjuntos de 3 elementos cada uno, tal que la suma de los elementos de cada subconjunto sea exactamente B?

La conversión en tiempo polinomial del problema de 3-Partition a una instancia de Batalla Naval se realiza de la siguiente forma:

Sea C el conjunto de entrada del problema 3-Partition y K la suma de los valores de C. Armamos el tablero de Batalla Naval con 5 filas (3 + 2). Tres de las filas, no contiguas, deben tener como restricción (de cantidad de casilleros ocupados) una cantidad de $K/3$. Desde ya, si K no es divisible por 3, la respuesta al problema de decisión de 3-Partition es **falso**. De esta forma, quedarían la primera, tercera y quinta fila con restricción de $K/3$, y las filas segunda y cuarta con restricción 0.

Este paso tiene complejidad $O(K)$, ya que al estar los valores de C en unario, es necesario recorrer todos los 1 una vez para obtener K y dividirlo por 3.

Luego, para cada uno de los 1 en C, creamos una columna con restricción 1, separando estas secuencias de columnas con una columna de restricción 0 cuando se pasa de un elemento de C a otro. Así, si C tiene N elementos, quedarán $K + N - 1$ columnas: K con restricción 1 y $N - 1$ columnas con restricción 0. Este paso también tiene complejidad $O(K)$, ya que basta con recorrer todos los 1 de C una vez.

Por último, por cada elemento de C, creamos un barco de ese mismo largo que debe ser colocado en el tablero. Esto permite trazar una equivalencia 1 a 1 entre los valores de C y los barcos. Este paso también es $O(K)$.

Con esto tenemos el tablero de Batalla Naval armado en tiempo polinomial. Esta disposición de filas y columnas *permite* al algoritmo que resuelve la batalla naval (la caja negra de nuestra reducción) probar todas las combinaciones de barcos en 3 grupos con suma $K/3$. Los barcos se dispondrán necesariamente de forma horizontal.

Dada una solución para esta versión de Batalla Naval, puede trazarse una equivalencia uno a uno entre cada barco colocado y cada número de C. Además, cada barco tendrá su propia secuencia *dedicada* de columnas de su mismo largo y sólo un barco puede ser colocado en esa secuencia de columnas al mismo tiempo, gracias a la restricción de una celda ocupada por columna. El barco podrá colocarse dentro de este espacio dedicado en cada una de las 3 filas con restricción, según lo requiera la solución.

Gracias a esto, la demostración de que la instancia de 3-Partition tiene solución **si y sólo si** la instancia de Batalla Naval la tiene es bastante directa.

Si 3-Partition tiene solución ⇒ Batalla Naval tiene solución

Si la instancia de 3-Partition tiene solución, entonces para cada uno de los subconjuntos podremos tomar sus elementos y ubicarlos en una de las filas con restricción $K/3$, cumpliendo así con estas mismas restricciones. Por otro lado, debido a que cada barco tiene asignado su propio espacio vertical de columnas, necesariamente las restricciones de columnas estarán satisfechas, ya que ese barco debe ser colocado en ese rango de columnas. Por lo tanto, habiéndose satisfecho todas las restricciones, la instancia de Batalla Naval tiene solución.

Si Batalla Naval tiene solución ⇒ 3-Partition tiene solución

Al igual que en la demostración anterior, el mapeo 1 a 1 entre los barcos y los elementos de C permite tomar una solución del problema de Batalla Naval y convertirla fácilmente en una solución para el problema de 3-Partition. Para ello, tomamos cada una de las 3 filas con restricciones y, a partir de los barcos colocados, formamos uno de los 3 subconjuntos con elementos de igual tamaño a los barcos. Dado que la restricción de las filas es exactamente $K/3$ y estas restricciones están satisfechas, los subconjuntos también sumarán $K/3$, resolviendo así el problema de 3-Partition.

Queda demostrado entonces que 3-Partition tiene solución si y sólo si su versión adaptada por

la reducción de Batalla Naval la tiene.

Lo cual, a su vez, completa la demostración de que 3-Partition (un problema NP-Completo) es al menos tan difícil como Batalla Naval, haciendo así a Batalla Naval también un problema NP-Completo.

3.3. Algoritmo Backtracking

El problema consiste en colocar una serie de barcos de longitudes específicas en un tablero de $n \times m$, cumpliendo con las restricciones de demanda de ocupación para cada fila y columna, mientras se minimiza la cantidad de demanda incumplida. La solución se implementa mediante un algoritmo de *backtracking* que explora todas las posibles configuraciones válidas para los barcos.

El algoritmo se divide en múltiples funciones que colaboran entre sí para construir la solución. A continuación, se detalla cada componente clave del algoritmo:

3.3.1. Generación de combinaciones de posiciones válidas

Para colocar un barco horizontalmente o verticalmente, se generan todas las combinaciones posibles que cumplan con las restricciones de ocupación de filas y columnas. Por ejemplo, la función `generate_rows_combinations` encuentra las posiciones horizontales válidas para un barco en el tablero.

```
def generate_rows_combinations(board, rows_restrictions, rows_occupation,
                               columns_restrictions, columns_occupation,
    current_ship_len):
    rows_combinations = []
    rows_amount = len(rows_restrictions)
    columns_amount = len(columns_restrictions)

    for i, row in enumerate(board):
        if (rows_occupation[i] + current_ship_len) > rows_restrictions[i]:
            continue

        adjacent_grids = 0
        for j in range(columns_amount):
            if (board[i][j] == 0) and (columns_occupation[j] < columns_restrictions
    [j]):
                adjacent_grids += 1
            else:
                adjacent_grids = 0

            if adjacent_grids == current_ship_len:
                ship_begin_position = j - current_ship_len + 1

                if ships_operations.is_valid_position(board, i, ship_begin_position
    , current_ship_len, HORIZONTAL_DIRECTION):
                    rows_combinations.append((i, ship_begin_position))
                adjacent_grids -= 1

    return rows_combinations
```

Esta función examina cada fila del tablero, verificando si es posible colocar un barco horizontalmente sin exceder las restricciones.

3.3.2. Intentos de colocación en direcciones horizontales y verticales

Una vez generadas las combinaciones, el algoritmo intenta colocar el barco en cada posición válida, evaluando el impacto en la solución global. Por ejemplo, la función `try_horizontal_combinations` intenta colocar el barco horizontalmente y actualiza el estado del tablero.

```
def try_horizontal_combinations(
    ships, current_ship_index, board, rows_restrictions, columns_restrictions,
    rows_occupation, columns_occupation, local_solution, global_solution
```

```
):
    current_ship_len = ships[current_ship_index]
    ship_id = current_ship_index + 1  # IDs para distinguir a los barcos.
    best_board = copy.deepcopy(board)

    rows_combinations = generate_rows_combinations(
        board=board, rows_restrictions=rows_restrictions,
        rows_occupation=rows_occupation, columns_restrictions=columns_restrictions,
        columns_occupation=columns_occupation, current_ship_len=current_ship_len
    )

    for row, ship_begin_position in rows_combinations:
        can_put_ship = ships_operations.put_ship_on_row(
            game_board=board, current_row=row, current_column=ship_begin_position,
            ship_len=current_ship_len, ship_id=ship_id, rows_occupation=
    rows_occupation,
            columns_occupation=columns_occupation
        )
        if can_put_ship:
            local_solution_aux = local_solution + (ships[current_ship_index] * 2)

            board_aux, local_solution_aux = build_game_board_bt(
                ships=ships, current_ship_index=current_ship_index+1, game_board=
    board,
                rows_restrictions=rows_restrictions, columns_restrictions=
    columns_restrictions,
                rows_occupation=rows_occupation, columns_occupation=
    columns_occupation,
                local_solution=local_solution_aux, global_solution=global_solution
            )

            if local_solution_aux > global_solution:
                global_solution = local_solution_aux
                best_board = copy.deepcopy(board_aux)

            ships_operations.remove_ship_on_row(
                board, row, ship_begin_position, current_ship_len,
                rows_occupation, columns_occupation
            )

    return best_board, global_solution
```

La lógica es similar para las combinaciones verticales, utilizando una función separada `try_vertical_combinations`.

3.3.3. Función principal BT

La función `build_game_board_bt` se utiliza para resolver el problema de Batalla Naval utilizando un enfoque de backtracking. A continuación, se proporciona una explicación detallada de cada parte del código.

```
def build_game_board_bt(ships, current_ship_index, game_board, rows_restrictions,
                        columns_restrictions, rows_occupation, columns_occupation,
                        local_solution, global_solution):

    best_game_board = copy.deepcopy(game_board)
```

La función `build_game_board_bt` se define con varios parámetros, entre ellos:

- `ships`: lista de barcos con sus longitudes.
- `current_ship_index`: índice del barco actual que se intenta colocar.
- `game_board`: tablero actual.
- `rows_restrictions`, `columns_restrictions`, `rows_occupation`, `columns_occupation`: restricciones y ocupación en filas y columnas.

- `local_solution`: solución parcial.
- `global_solution`: mejor solución global encontrada hasta ahora.

Al inicio, se hace una copia profunda del tablero `game_board` para trabajar con una copia sin modificar el original.

```
    if local_solution > global_solution:
        global_solution = local_solution

    if current_ship_index >= len(ships):
        return best_game_board, global_solution
```

Aquí, se verifica si la `local_solution` es mejor que la `global_solution`, y si es así, se actualiza la solución global. Además, si el índice del barco actual es mayor o igual al número total de barcos (`len(ships)`), significa que todos los barcos han sido colocados, por lo que se retorna la mejor solución.

```
    if local_solution + sum(ships[current_ship_index:]*2) <= global_solution:
        return best_game_board, global_solution
```

Este bloque evalúa si la `local_solution` más el "potencial"de los barcos restantes (el cálculo `sum(ships[current_ship_index:] * 2)`) no puede superar la solución global. Si esto es cierto, se retorna la mejor solución encontrada hasta ahora sin seguir intentando colocar más barcos.

```
best_game_board = copy.deepcopy(game_board)

game_board_aux, local_solution_aux = try_horizontal_combinations(
    ships=ships,
    current_ship_index=current_ship_index,
    board=game_board,
    rows_restrictions=rows_restrictions,
    columns_restrictions=columns_restrictions,
    rows_occupation=rows_occupation,
    columns_occupation=columns_occupation,
    local_solution=local_solution,
    global_solution=global_solution
)
```

Aquí se llama a la función `try_horizontal_combinations`, que intenta colocar el barco en combinaciones horizontales en el tablero. La función devuelve un tablero actualizado (`game_board_aux`) y una nueva solución parcial (`local_solution_aux`).

```
    if local_solution_aux > global_solution:
        global_solution = local_solution_aux
        best_game_board = copy.deepcopy(game_board_aux)
```

Si la solución obtenida con las combinaciones horizontales es mejor que la solución global, se actualiza `global_solution` y `best_game_board` con los valores obtenidos.

```
game_board_aux, local_solution_aux = try_vertical_combinations(
    ships=ships,
    current_ship_index=current_ship_index,
    board=game_board,
```

```
    rows_restrictions=rows_restrictions,
    columns_restrictions=columns_restrictions,
    rows_occupation=rows_occupation,
    columns_occupation=columns_occupation,
    local_solution=local_solution,
    global_solution=global_solution
)
```

De manera similar a las combinaciones horizontales, se llaman a las combinaciones verticales utilizando la función `try_vertical_combinations`. Esta función trata de colocar el barco de manera vertical en el tablero.

```
    if local_solution_aux > global_solution:
        global_solution = local_solution_aux
        best_game_board = copy.deepcopy(game_board_aux)
```

Si la solución obtenida con las combinaciones verticales es mejor que la global, se actualizan `global_solution` y `best_game_board`.

```
game_board_aux, local_solution_aux = build_game_board_bt(
    ships=ships,
    current_ship_index=current_ship_index+1,
    game_board=game_board,
    rows_restrictions=rows_restrictions,
    columns_restrictions=columns_restrictions,
    rows_occupation=rows_occupation,
    columns_occupation=columns_occupation,
    local_solution=local_solution,
    global_solution=global_solution
)
```

Si no se ha llegado a la solución final, se realiza una llamada recursiva a `build_game_board_bt`, pasando el siguiente barco (`current_ship_index + 1`). Esta recursión permite explorar todas las posibles configuraciones de los barcos en el tablero.

```
    if local_solution_aux > global_solution:
        global_solution = local_solution_aux
        best_game_board = copy.deepcopy(game_board_aux)
```

Tras la llamada recursiva, si la solución obtenida es mejor que la global, se actualizan `global_solution` y `best_game_board`.

```
    return best_game_board, global_solution
```

Finalmente, la función retorna el `best_game_board`, que contiene la mejor configuración encontrada, y la `global_solution`, que es la mejor solución global.

Construcción inicial del tablero Finalmente, la función `build_naval_battle_game_board` se encarga de iniciar el proceso, preparando las estructuras necesarias para el algoritmo de *backtracking*.

```python
def build_naval_battle_game_board(rows_restrictions, columns_restrictions, ships ):
    game_board = [[0 for _ in range(len(columns_restrictions))] for _ in range(len(
    rows_restrictions))]
    rows_occupation = [0]*len(rows_restrictions)
    columns_occupation = [0]*len(columns_restrictions)
    ships.sort(reverse=True)

    return build_game_board_bt(ships,0, game_board, rows_restrictions,
    columns_restrictions,
                               rows_occupation, columns_occupation, 0, 0)
```

3.4. Optimalidad

El algoritmo de backtracking implementado en la resolución de la Batalla Naval es óptimo porque:

Evalúa todas las configuraciones posibles de los barcos en el tablero, asegurando que ninguna solución válida sea omitida.

Utiliza poda basada en cotas inferiores, descartando configuraciones parciales que no pueden superar la mejor solución global, reduciendo su tiempo de ejecucion y complejidad. Esta es una cota:

```python

if local_solution + sum(ships[current_ship_index:]*2) <= global_solution:
        return best_game_board, global_solution
```

Mantiene un registro de la mejor solución encontrada hasta el momento, lo que garantiza que al finalizar la búsqueda, el tablero resultante corresponde a la mejor disposición de barcos que minimiza la demanda incumplida. Este enfoque permite que el algoritmo encuentre la configuración óptima en términos de cumplimiento de restricciones y ocupación del tablero, dentro del tiempo factible para su ejecución.

3.5. Mediciones

Para medir la efectividad del codigo procedemos a hacer algunas mediciones con los distintos datasets provistos por la catedra.

- El dataset `3_3_2.txt` se ejecuta en: 0,0001 segundos
- El dataset `5_5_6.txt` se ejecuta en: 0,0455 segundos
- El dataset `8_7_10.txt` se ejecuta en: 0,0045 segundos
- El dataset `10_3_3.txt` se ejecuta en: 0,0002 segundos
- El dataset `10_10_10.txt` se ejecuta en: 0,9951 segundos
- El dataset `12_12_21.txt` se ejecuta en: 1,9863 segundos
- El dataset `15_10_15.txt` se ejecuta en: 0,0109 segundos
- El dataset `20_20_20.txt` se ejecuta en: 1,1200 segundos
- El dataset `20_25_30.txt` se ejecuta en: 0,1037 segundos

Con estos tiempos podemos observar la efectivdad de nuestro codigo, ya que resuelve muy rapidamente el problema, siendo el dataset que mas se demora el `12_12_21.txt` con un tiempo de 1,9863 sgundos lo cual es muy poco

3.6. Algoritmo de aproximación

Implementamos un algoritmo de aproximación para resolver el problema de la Batalla Naval. Este algoritmo se usa para aproximar una solución al problema de La Batalla Naval de forma rápida y práctica. La idea es simple: se identifica la fila o columna con mayor demanda insatisfecha y se intenta colocar allí el barco más largo que quepa. Si ese barco no entra porque supera la demanda, se lo pasa por alto y se prueba con el siguiente en tamaño. Este proceso se repite hasta que no queden barcos para ubicar o demandas por satisfacer.

La ventaja de este enfoque es que, aunque no garantiza la solución óptima, permite obtener un resultado bastante razonable en mucho menos tiempo. Es útil para comparar cómo se acerca esta aproximación a la solución óptima, sobre todo en casos donde resolver el problema de forma exacta sería demasiado lento o directamente impracticable.

El algoritmo es el siguiente:

```
HORIZONTAL = "horizontal"
VERTICAL = "vertical"

def horizontal_pos_available(board, row, col, ship_len):
    if col + ship_len > len(board[0]):
        return False
    for c in range(col, col + ship_len):
        if board[row][c] != 0:
            return False
    return True

def vertical_pos_available(board, row, col, ship_len):
    if row + ship_len > len(board):
        return False
    for r in range(row, row + ship_len):
        if board[r][col] != 0:
            return False
    return True

def position_horizontal(board, row, col, ship_len, ship_id, row_demand, col_demand)
    :
    for c in range(col, col + ship_len):
        board[row][c] = ship_id
        col_demand[c] -= 1
    row_demand[row] -= ship_len

def position_vertical(board, row, col, ship_len, ship_id, row_demand, col_demand):
    for r in range(row, row + ship_len):
        board[r][col] = ship_id
        row_demand[r] -= 1
    col_demand[col] -= ship_len

def battleship_approximation(n, m, row_demand, col_demand, ships):
    game_board = [[0 for _ in range(m)] for _ in range(n)]
    row_demand_available = row_demand[:]
    col_demand_available = col_demand[:]
    ships_in_game = []

    ship_id = 1

    for ship in ships:
        in_game = False

        possible_positions = []
        for i in range(n):
            if row_demand_available[i] >= ship:
                possible_positions.append((i, HORIZONTAL, row_demand_available[i]))
        for j in range(m):
            if col_demand_available[j] >= ship:
                possible_positions.append((j, VERTICAL, col_demand_available[j]))

        possible_positions.sort(key=lambda x: x[2], reverse=True)

```

```
        for position, orientation, _ in possible_positions:
            if orientation == HORIZONTAL:
                for col in range(m - ship + 1):
                    if horizontal_pos_available(game_board, position, col, ship):
                        position_horizontal(game_board, position, col, ship,
    ship_id, row_demand_available, col_demand_available)
                        ships_in_game.append(ship)
                        in_game = True
                        break
            elif orientation == VERTICAL:
                for row in range(n - ship + 1):
                    if vertical_pos_available(game_board, row, position, ship):
                        position_vertical(game_board, row, position, ship, ship_id,
     row_demand_available, col_demand_available)
                        ships_in_game.append(ship)
                        in_game = True
                        break
            if in_game:
                break

        if not in_game:
            continue

        ship_id += 1

    aproximation = sum(ships_in_game)
    return game_board, aproximation
```

Vayamos separando el algoritmo para entender su complejidad.

- Se inicializa el tablero. $O(n.m)$

- Generacion de `global_solution` . En el peor de los casos, si todas las filas y columnas son validas, este bloque toma $O(n+m)$

- Luego se ordenan las posibles soluciones. $O((n+m)*log(n+1))$

- Luego la verificacion de posiciones y colocacion de barcos:

```
    for position, orientation, _ in possible_positions:
    if orientation == HORIZONTAL:
        for col in range(m - ship + 1):
            if horizontal_pos_available(game_board, position, col, ship):
                position_horizontal(game_board, position, col, ship, ship_id,
    row_demand_available, col_demand_available)
                ships_in_game.append(ship)
                in_game = True
                break
```

- Las funciones `horizontal_pos_available` y `vertical_pos_available` verifican disponibilidad con complejidad $O(S)$, siendo S la cantidad de barcos. En el pero de los casos ocurre cuando todas las posiciones se revisan antes de encontrar una valida o descartar la colocacion, resultado en: $O((n+m)*max(n,m))$

- El bucle principal itera sobre los S barcos, por ende la complejidad total es: $O(k((n+m)log(n+m)+(n+m)max(n,m)))$.

Con el termino dominante, la complejidad del algoritmo implementado es $O(n.m.S)$, donde S es el número de barcos (ships).

Para analizar qué tan buena es la aproximación en cada instancia I del problema, calcularemos el factor de aproximación que cumple con

$$\frac{A(I)}{z(I)} \leq r(A)$$

para todas las instancias.

Comprobamos que $r(A) \leq r$, siendo $r = 1$ la constante que representa la cota de aproximación, en todas las instancias. Esto significa que el valor calculado de cada $r(A)$ oscila entre 0 y 1, debido

a que si el valor es mayor que 1 significaría que el algoritmo de aproximación encontró mejores resultados que el algoritmo de backtracking usando las pruebas locales.

Instancia	**Óptimo**	**Aproximación**	$r(A)$
3_3_2.txt	4	6	1,5
5_5_6.txt	12	36	3
8_7_10.txt	26	36	1,384615
10_3_3.txt	6	20	3,333333
10_10_10.txt	40	46	1,15
12_12_21.txt	46	218	4,739130
15_10_15.txt	40	52	1,3
20_20_20.txt	104	244	2,346153
20_25_30.txt	172	926	5,383720

Cuadro 1: Valores de r(A) para cada instancia del problema de Batalla Naval

Como podemos ver, para cada instancia I del problema, se encontró que la cota $r(A)$ es de

$$\frac{A(I)}{z(I)} \leq 5,383720$$

.

El algoritmo alternativo que usamos para calcular la aproximación es "peor.ªl algoritmo implementado de backtracking ya que z(I) es menor o igual a la demanda incumplida por este mismo.

Conclusiones

En esta parte del trabajo abordamos la resolución del problema de la Batalla Naval, un problema NP-completo, utilizando dos enfoques: un algoritmo exacto basado en *backtracking* y un algoritmo de aproximación. El *backtracking* demostró ser efectivo al encontrar soluciones óptimas, pero su escalabilidad se ve limitada en instancias más grandes debido a su alta complejidad. Los tiempos de ejecución obtenidos fueron razonables para los datasets analizados, siendo el más exigente el archivo `12_12_21.txt` con un tiempo de 1,9863 segundos.

El algoritmo de aproximación, por su parte, proporcionó soluciones más rápidas pero con menor precisión. Analizamos su rendimiento mediante el factor de aproximación $r(A)$, que osciló entre 0,375 y 1,0, cumpliendo con los límites establecidos. Aunque en algunos casos $r(A)$ mostró un rendimiento limitado, el método de aproximación se presenta como una alternativa válida cuando el tiempo es una restricción crítica.

En resumen, la comparación entre ambos enfoques resalta la necesidad de un balance entre precisión y eficiencia según el contexto. Este análisis demuestra cómo técnicas exactas y aproximadas pueden complementarse para abordar problemas complejos de manera efectiva.

Correcciones

Correccion a: "El certificador nuevamente tiene funciones sin describir por ejemplo `"validar_restricciones"` y `"validar_posicion_barco"`. Tienen su complejidad pero no dice cómo hace lo que tiene que verificar. Dónde está el pseudocódigo de estas funciones? o al menos una explicación de como logran su objetivo?"

Se escribe archivo validador.py en la carpeta `parte_III` del repositorio. Ahi se muestra la complejidad de cada ecuacion utilizadad, a travez de su codigo.

Correccion a: En la demostración de optimalidad faltaron los índices entre las opciones de selección. sino "Sophia escoge la moneda de mayor valor entre m y m"no se entiende.

Se escriben correctamente los caracteres especiales para Latex. Asi queda la explicacion con los valores correctos:

3.7. Solución óptima

Por definición, en un algoritmo greedy las soluciones óptimas se construyen a partir de una secuencia de óptimos locales que paso a paso van llevando a la mejor solución a un problema. La lógica greedy es la de tomar la mejor decisión en cada paso de un problema, y que el conjunto de las mejores decisiones lleven al óptimo.

Esta definición podría confundirse con la de la programación dinámica. Pero la diferencia principal es que los algoritmos greedy no tienen memoria.

Claramente no todo problema solucionado de forma greedy tiene solución óptima, sin embargo, para el caso del juego de las monedas con las restricciones definidas anteriormente sí se alcanza el óptimo.

Vamos a demostrarlo siguiendo la logica del algoritmo.

El algoritmo recibe una lista de monedas $C = [m_1, m_2, ..., m_n]$. En cada turno, se puede quitar las monedas en la posición $i = 0$ o $i = n-1$ de C, siendo n la cantidad de monedas en C. Llamemosle S a la ganancia de Sophia, y M a la ganancia de Mateo.

Nuestra regla greedy garantiza que en cada turno j ($0 \leq j \leq n$), se cumple $S[j] > M[j]$, ya que Sophia siempre opta por la moneda de mayor valor disponible, mientras que Mateo selecciona la de menor valor. (Es importante señalar que todas las monedas tienen valores únicos y no se repiten).

Para comprender mejor esta estrategia, observemos el primer turno ($j = 0$): Sophia escoge la moneda de mayor valor entre m_1 y m_n, es decir, máx(m_1, m_n). Luego, en el siguiente turno, Mateo elige la moneda de menor valor entre las dos restantes:

- Si Sophia tomó m_1, Mateo seleccionará mín(m_2, m_n).
- Si Sophia tomó m_n, Mateo elegirá mín(m_1, m_{n-1}).

Siguiendo esta lógica, Sophia optimiza su ganancia relativa en cada turno, asegurándose de mantener su ventaja sobre Mateo.

Demostraremos que el algoritmo devuelve un conjunto óptimo A, que maximiza la ganancia de Sophia, asegurando que en cada turno ella elige la moneda de mayor valor disponible.

Utilizaremos un razonamiento por contradicción. Supongamos que A no es óptimo, lo que implicaría la existencia de una estrategia O superior, que maximiza aún más la ganancia de Sophia, es decir, $S(O) > S(A)$. Si esto fuera cierto, en al menos un turno, Sophia habría elegido una moneda menos valiosa en A que en O, lo que reduciría su ganancia.

Dado que en la estrategia A, Sophia siempre selecciona máx(m_1, m_n), cualquier estrategia que la supere debería elegir una moneda diferente en algún turno. Como las únicas opciones en cada jugada son m_1 y m_n, la unica alternativa a la estrategia de A seria elegir mín(m_1, m_n) en al menos un turno. Sabemos que Sophia toma decisiones considerando también la jugada de Mateo, quien siempre elige la moneda de menor valor entre las disponibles. Para mayor claridad, en lo que sigue hablaremos de Mateo como si tomara sus propias decisiones en su turno.

Teniendo en cuenta que Sophia sacó la moneda de menor valor en el primer turno, en el próximo turno, Mateo podrá elegir entre las 2 monedas que quedan, habiendo la posibilidad de que una sea mayor que la moneda de Sophia, entonces tendria una mayor ventaja hasta ese momento. Si en un turno j, Sophia en O elige una moneda de menor valor que la elegida en A, entonces Mateo podrá elegir una moneda de mayor valor que en A, incrementando su ganancia respecto a Sophia. Por lo tanto, en cada turno, A siempre proporciona una mejor o igual ganancia a Sophia en comparación con O, lo que contradice nuestra suposición inicial de que $S(O) > S(A)$. Por lo tanto, la estrategia A no puede ser superada por ninguna otra estrategia O, lo que implica que A es óptima, y que cualquier desviación de A resulta en una peor ganancia para Sophia.

Por lo tanto, está probado que el algoritmo greedy de Sophia, donde selecciona la moneda de mayor valor entre las opciones disponibles y elige la de menor valor para Mateo, siempre es óptimo. Si optara por una poliitica diferente, terminariia perjudicándose porque la relación de ganancia entre los dos es directa, lo que contradice la idea de que podria haber un mejor resultado. Si una ganancia aumenta, la otra disminuye, por lo que Sophia siempre debe maximizar su ganancia en cada turno y eso lo hace eligiendo la moneda de mayor valor, pero también debe minimizar la ganancia de Mateo y eso lo hace eligiendo la moneda de menor valor.

Respecto al análisis de la complejidad algorítmica. Se formulo la hipótesis teórica de que el algoritmo propuesto tiene una complejidad $O(n)$. En cada paso del problema se está realizando un problema de decisión entre dos monedas, lo cual se resuelve en tiempo $O(1)$, por otra parte, la lista se itera alternando sus extremos de afuera hacia adentro, al iterar sobre todos los elementos se tendría una complejidad $O(n)$ (la complejidad de decidir cuál moneda tomar es despreciable comparada a la complejidad de moverse por la lista de monedas).