.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(TB024) Curso Buchwald - Genender

Trabajo Práctico: Juegos de Hermanos

02 de diciembre de 2024

Integrantes:

Kevin Emir Carbajal Robles
108304

Ramiro Mineldin
105876

Nicolas Chen
105907

Franco Corn
109025

Fabio Sicca
104892

Primera parte

1. Introducción

Los hermanos Sophia y Mateo comienzan un juego creado por su padre: el juego de las monedas. En este juego se dispone de una fila de monedas de diferentes valores, en donde los jugadores se turnan para seleccionar una moneda en cada ronda. La única regla que tiene el juego es que solo se pueden tomar las monedas de los extremos. El juego se termina cuando no haya mas monedas a disposición y el ganador de será quien tenga el mayor valor acumulado (por sumatoria).

Sophia es muy competitiva y está decidida a ganar, a pesar de que Mateo no sepa comó jugar. A lo largo del juego, Sophia intentará ayudar a Mateo, aunque su objetivo principal será hacerlo perder y asi, quedarse con la victoria.

2. Análisis del problema

El problema consiste en modelar una estrategia para que Sophia, elija las monedas de tal forma que la sumatoria de valores, sea mayor que la de Mateo. Para esto, consideramos que hay n monedas dispuestas en una fila, numeradas de 0 a $n-1$, y que cada jugador puede tomar una moneda ya sea del inicio o del final de la fila en su turno.

Dado que Sophia puede tomar una moneda tanto en su turno como determinar cuál tomará su hermano Mateo en el siguiente turno, ella seleccionará la moneda de mayor valor en su turno, y durante el turno de Mateo, elegirá la opción que le deje la moneda de menor valor.

Con esto, garantizamos que al final del juego, la suma de los valores acumulados de las monedas que eligió Sophia sea mayor que las de Mateo.

2.1. Estrategia de Sophia

Sophia busca tener la moneda de mayor valor en cada uno de sus turnos. La elección de Sophia se puede formalizar como:

$$S(i,j) = \text{máx}\,(v_i, v_j)$$

2.2. Estrategia de Mateo

Mateo, en su turno, seleccionará la moneda que tenga menor valor. La elección de Mateo se puede representar como:

$$M(i,j) = \text{mín}(v_i, v_j)$$

2.3. Notación

Donde:

- $S(i,j)$ representa el valor máximo que puede obtener Sophia al elegir entre las monedas posicionadas en i y j.
- $M(i,j)$ representa el valor mínimo que puede obtener Mateo al elegir entre las monedas posicionadas en i y j.
- v_i y v_j son los valores de las monedas en las posiciones i y j respectivamente.

3. Algoritmo Greedy

Proponemos el siguiente algoritmo greedy, que se basa en aplicar una regla sencilla en cada iteración. Esta regla establece que en cada turno, Sophia seleccionará la moneda de mayor valor disponible, mientras que Mateo elegirá la moneda de menor valor.

Al seguir esta regla, se busca maximizar la sumatoria del valor acumulado de las monedas obtenidas por Sophia en comparación con las monedas obtenidas por Mateo en cada turno del juego, lo que representa un óptimo local. Si esta regla se aplica de manera constante hasta que todas las monedas hayan sido seleccionadas, se logrará un óptimo global en términos de la sumatoria de los valores de las monedas acumuladas por Sophia al final del juego.

3.1. Implementación

El algoritmo alterna turnos entre Sophia y Mateo, determinados por el bool `es_turno_sophia`, quienes eligen entre las monedas en los extremos de la deque `monedas`, en donde son representadas como `primera_moneda` y `ultima_moneda`.

En la función `agarrar_moneda`, se comparan los valores de estas dos monedas: Sophia elige la de mayor valor y Mateo la de menor, mientras que la moneda no elegida se reintegra a `monedas`. La elección realizada se guarda en la deque `resultados` ya sea la primera o última moneda elegida.

Este proceso continúa hasta que solo queda una moneda, en cuyo caso se efectúa el último turno. Si desde el inicio solo hay una moneda, esta se considera en el primer y único turno.

```python
# Constantes
primera_moneda_sophia ="Primera moneda para Sophia"
ultima_moneda_sophia = "Última moneda para Sophia"
primera_moneda_mateo = "Primera moneda para Mateo"
ultima_moneda_mateo = "Última moneda para Mateo"

def agarrar_moneda(monedas, resultado, es_sophia):
    primera_moneda = monedas.popleft()
    ultima_moneda = monedas.pop()
    if es_sophia: # Turno de Sophia, elije siempre la mayor
        if primera_moneda > ultima_moneda:
            resultado.append(primera_moneda_sophia)
            monedas.append(ultima_moneda)
        else:
            resultado.append(ultima_moneda_sophia)
            monedas.appendleft(primera_moneda)
    else: # Turno de Mateo, elije siempre la menor
        if primera_moneda > ultima_moneda:
            resultado.append(ultima_moneda_mateo)
            monedas.appendleft(primera_moneda)
        else:
            resultado.append(primera_moneda_mateo)
            monedas.append(ultima_moneda)

def algoritmo_greedy(monedas):
    resultado = []
    es_turno_sophia = True
    while len(monedas) >= 2:
        agarrar_moneda(monedas, resultado, es_turno_sophia)
        es_turno_sophia = not es_turno_sophia  # Cambio de turno

    # Ultimo turno
    if es_turno_sophia:
        resultado.append(ultima_moneda_sophia)
    else:
        resultado.append(ultima_moneda_mateo)

    return resultado
```

3.2. Complejidad

Observamos que nuestro algoritmo, en `algoritmo_greedy`, itera sobre todas las monedas disponibles. Si llamamos n a la cantidad de monedas, obtenemos una complejidad temporal de $O(n)$ para esta parte del algoritmo, ya que solo hay asignaciones y condicionales adicionales restantes, los cuales no aumentan la complejidad.

En cada iteración del algoritmo, se llama a la funcion `agarrar_moneda` que tambien tenemos que analizar su complejidad.

En esta función, independientemente de si es el turno de Sophia o de Mateo, se extraen las monedas de los extremos y se comparan, seguidas de operaciones de almacenamiento y reinserción.

Cabe mencionar que las operaciones `append` y `appendleft` de una deque (o el `append` de una list) tienen complejidad $O(1)$, por lo que no afectan la complejidad temporal general.

Uniendo todo esto, concluimos que la complejidad total es $O(n)$.

3.3. ¿Es óptimo?

A continuación, demostraremos que el algoritmo es óptimo mediante el principio de inducción.

Sean S (Sophia), M (Mateo) y k la cantidad de turnos de cada uno:

Proposición: En cada turno, S toma la moneda de mayor valor disponible entre los dos extremos, y esta moneda siempre será mayor que la moneda que M pueda tomar en el siguiente turno.

Esto implica que la suma de los valores acumulados de las monedas que recoge S será mayor que la de M, independientemente del orden en el cual se presenten las monedas (desestimando el caso de una cantidad par de monedas de mismo valor).

Caso base (primer turno): En el primer turno, S elige entre las dos monedas de los extremos. Como S siempre toma la moneda de mayor valor en su turno, elige esa misma. Como M tomará una moneda en el siguiente turno, sera entre la moneda del otro extremo (que es menor en comparación con la que tomó S) o la que liberó S al momento de elegir su moneda (si es que aún quedan), la cual puede ser mayor o menor que la del extremo. En cualquier caso, M elegirá la moneda menor posible, asegurando que su elección nunca supere la moneda tomada por S.

Por lo tanto, la proposición se cumple en este turno inicial.

Paso inductivo: Supongamos que hasta el turno k, la proposición se cumple: cada moneda tomada por S ha sido mayor que cualquiera de las monedas que M ha tomado en cada turno.

Ahora, en el turno $k+1$, S debe elegir nuevamente entre las dos monedas de los extremos, y siguiendo la regla greedy, tomará la de mayor valor. En el siguiente turno de M elegirá la moneda de menor valor entre los dos extremos restantes. Esta elección de M garantiza que su moneda es menor que la moneda que acaba de tomar S.

Por lo tanto, la proposición se mantiene para el turno $k+1$: la moneda tomada por S sigue siendo mayor que la de M.

De esta manera, se concluye por inducción que, para cualquier cantidad de monedas, S siempre tendrá una suma de valores acumulados de monedas mayor que M, independientemente de la variabilidad en los valores de las monedas. Esto implica que el algoritmo es óptimo en todos los casos, salvo en aquellos donde el número de monedas es par y todas tienen el mismo valor.

4. Mediciones de tiempos y validación de complejidad

Para verificar el funcionamiento del algoritmo y confirmar que siempre proporciona una solución óptima, realizamos pruebas utilizando diferentes conjuntos de datos proporcionados por la cátedra.

Se presentan los resultados obtenidos en un cuadro donde incluye las ganancias (suma de los valores acumulados en monedas) de Sophia y Mateo, así como el tiempo de ejecución del algoritmo:

Archivo	Ganancia de Sophia	Ganancia de Mateo	Tiempo (s)
20.txt	7165	3474	$9{,}059906 \times 10^{-6}$
25.txt	9046	3577	$1{,}168251 \times 10^{-5}$
50.txt	17750	7651	$1{,}645088 \times 10^{-5}$
100.txt	35009	14641	$2{,}741814 \times 10^{-5}$
1000.txt	357814	148747	$1{,}900196 \times 10^{-4}$
10000.txt	3550095	1484229	0,001795053
20000.txt	7141395	2979133	0,003530979

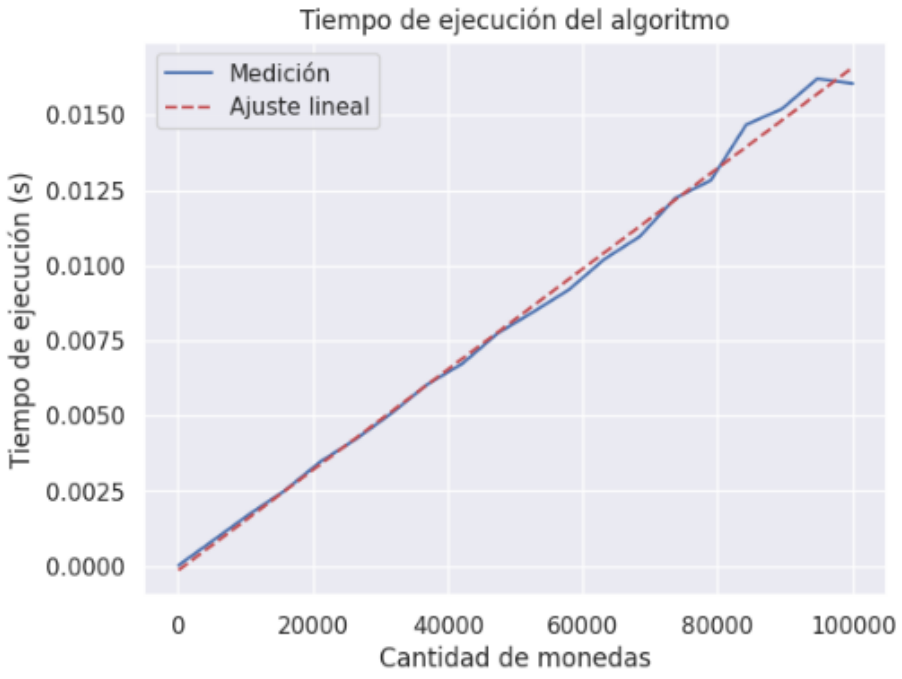
En todos los casos, la ganancia de Sophia es mayor que la de Mateo, lo que demuestra la optimalidad del algoritmo. Sin embargo, observamos que en algunos casos las ganancias obtenidas por Sophia no coinciden exactamente con los valores esperados por la cátedra, lo cual se debe a detalles específicos de la implementación del algoritmo.

4.1. Variabilidad de la cantidad y valor de las monedas

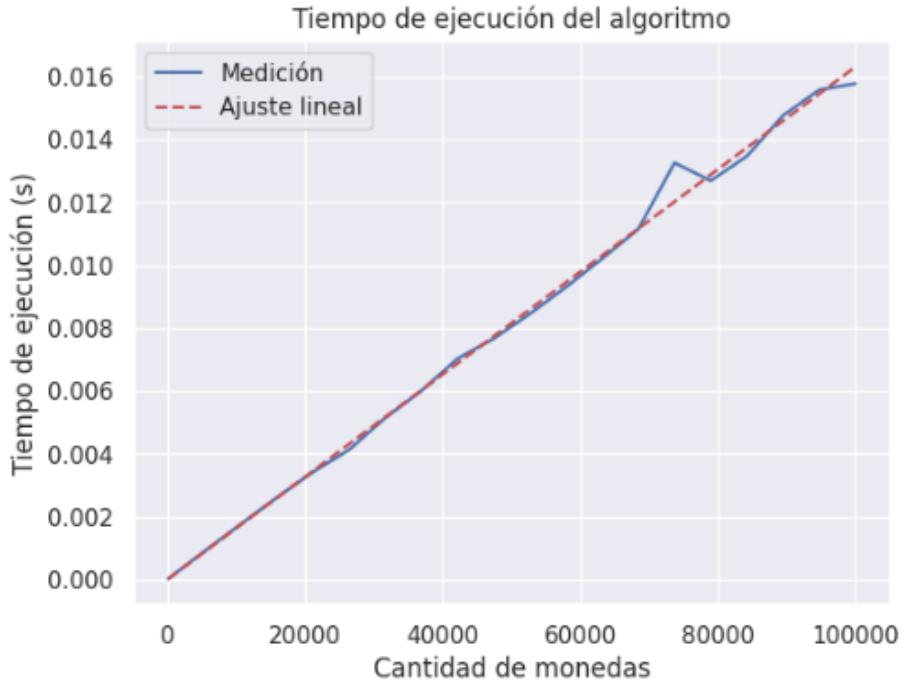
Se realizaron mediciones utilizando diferentes cantidades de monedas (y variando su valor) generados aleatoriamente con la biblioteca `random`. Para cada caso, se realizó un ajuste por mínimos cuadrados utilizando la biblioteca `scipy` y se calculó el error para verificar la complejidad temporal del algoritmo.

Para cada uno de los conjuntos se probaron con 20 arreglos de monedas con longitudes desde 100 hasta 100000 monedas. (Cada longitud equidistanciados igualmente)

4.1.1. Conjunto 1: Valores entre 0 y 1000



4.1.2. Conjunto 2: Valores entre 100000000 y 200000000



Se puede observar que al analizar distintas cantidades de monedas y variando los valores de las mismas el error del ajuste es significativamente pequeño. Esto sugiere que el algoritmo greedy propuesto presenta una complejidad temporal de $\mathcal{O}(n)$.

5. Conclusiones

En este trabajo, se propuso y se analizó un algoritmo greedy para resolver el problema de selección de monedas en el juego entre Sophia y Mateo. A través de un análisis formal se demostró que el algoritmo siempre garantiza la solución óptima en todos los casos, excepto cuando el número de monedas es par y todas tienen el mismo valor.

El algoritmo presentado es eficiente en términos de complejidad temporal operando en tiempo lineal en función de la cantidad de monedas. La variabilidad de los valores de las monedas no afecta la optimalidad del algoritmo, ya que siempre la suma de los valores acumulados de monedas de Sophia es mayor que la de Mateo, independientemente de los valores específicos.

Se realizaron mediciones para corroborar la complejidad teórica utilizando gráficos y la técnica de cuadrados mínimos para evaluar el comportamiento del algoritmo en diferentes escenarios. Los resultados empíricos confirmaron la validez de la complejidad calculada teóricamente y mostraron que el algoritmo sigue siendo lineal a medida que varia la cantidad de monedas y sus valores.

En fin el algoritmo propuesto no solo resuelve el problema de manera óptima, sino que también demuestra ser consistente ante variaciones en los valores de las monedas.

Segunda parte

1. Introducción

Con el paso del tiempo, Mateo aprendió a aplicar la misma estrategia que Sophia. Ahora Mateo ya conoce el Algoritmo de Greedy y el ganador del juego depende más de quién comience y un tanto de la suerte. Sin embargo, esto a Sophia no le gusto para nada. Lo bueno es que Sophia conoce una nueva táctica y decide aplicarlo para asegurar su victoria siempre que se pueda.

2. Análisis del problema

Al igual que en la primera parte, el problema consiste en modelar una estrategia para que Sophia elija las monedas de tal forma que la sumatoria de los valores sea mayor que la de Mateo. Para esto, consideramos que hay n monedas dispuestas en una fila, numeradas de 0 a $n-1$, y que cada jugador puede tomar una moneda ya sea del inicio o del final de la fila en su turno.

En este caso también tenemos que considerar que Mateo aprendió a aplicar la estrategia anterior. Por lo tanto, en cada turno de Mateo, Mateo elegirá la moneda de mayor valor de los extremos.

Dado que Sophia conoce la estrategia que aplicará Mateo, puede determinar cuál será la moneda que tomaría Mateo en su próximo turno. Por lo tanto, Sophia seleccionará en cada turno la moneda cuya sumatoria entre las primeras 2 monedas y las ultimas 2 monedas restantes de la fila sea la mayor posible incluyendo la proyección de la moneda que elija Mateo.

Con esto garantizamos que al final de juego, la sumatoria de los valores de las monedas que eligió Sophia sea el máximo valor acumulable posible. Esto no garantiza que siempre gane Sophia, ya que existen casos en donde no importa lo que haga Sophia, ganará Mateo. Por ejemplo, para las monedas `[4,20,50,1,3]`.

2.1. Estrategia de Sophia

Sophia busca seleccionar la moneda que le permita garantizar la mayor sumatoria de valor acumulado incluyendo las posibles monedas de su próximo turno en cada uno de sus turnos. La elección de Sophia se puede formalizar con la siguiente ecuación de recurrencia:

$$S(arr,i,j) = \begin{cases} v_i & \text{si } len(arr) = 1 \\ \max(v_i, v_j) & \text{si } len(arr) = 2 \\ \max(F_1, F_2) & \text{si } len(arr) > 2 \end{cases}$$

siendo:

$$F_1 = arr[i] + \min(S(arr, i+2, j), S(arr, i+1, j-1))$$
$$F_2 = arr[j] + \min(S(arr, i+1, j-1), S(arr, i, j-2))$$

2.2. Estrategia de Mateo

Mateo, en su turno, seleccionará la moneda de mayor valor de los extremos en cada uno de sus turnos. La elección de Mateo se puede formalizar como:

$$M(i,j) = \max(v_i, v_j)$$

2.3. Notación

Donde:

- $S(arr, i, j)$ representa el valor máximo que puede obtener Sophia al elegir entre las monedas posicionadas en i y j.
- $M(i, j)$ representa el valor máximo que puede obtener Mateo al elegir entre las monedas posicionadas en i y j.
- arr representa la fila inicial de las monedas.
- v_i y v_j son los valores de las monedas en las posiciones i y j respectivamente.

3. Algoritmo Programación Dinámica

Proponemos el siguiente algoritmo de programación dinámica, que se basa en resolver subproblemas de manera eficiente mediante el uso de una tabla de resultados parciales. En cada paso, el algoritmo evalúa todas las posibles elecciones de Sophia y Mateo, tomando en cuenta no solo la jugada actual, sino también las consecuencias futuras de cada decisión. La regla básica es que, para cada subintervalo de monedas disponibles, Sophia elige la opción que maximiza su valor acumulado, considerando que Mateo sigue la regla greedy, es decir, elige siempre la moneda de mayor valor disponible en su turno.

La idea principal es almacenar, en una tabla $R[i][j]$, el valor máximo que Sophia puede obtener si juega entre las monedas m_i y m_j. A medida que el algoritmo llena la tabla, se considera el mejor valor que Sophia puede obtener, dado que después de cada jugada, Mateo elige de manera óptima siguiendo la estrategia mencionada. Este enfoque asegura que Sophia tome la mejor decisión en cada turno, anticipando las jugadas de Mateo y maximizando su valor final.

3.1. Implementación

El primer paso del algoritmo es inicializar una matriz de $n*n$ donde n es el número de monedas. El segundo paso es completar con los casos base $R[i][i] = v_i$ es decir cuando hay solo una moneda y cuando hay dos monedas $R[i][i+1] = \text{máx}(v_i, v_i + 1)$. Finalmente se completa la matriz con el resto de valores utilizando la ecuación de recurrencia.

Explicación de la ecuación:

Si Sophia elige la primera moneda v_i. Luego, Mateo juega de manera óptima entre las monedas restantes v_i+1 y v_j. Sophia obtiene v_i pero lo que le queda de valor depende de lo que Mateo deje disponible para ella. Es decir, si Mateo elige $m_i + 1$ entonces el intervalo se reduce a $[i + 2, j]$. Si Mateo elige m_j, el intervalo se reduce a $[i + 1, j - 1]$. Se usa el mínimo de estos dos posibles resultados garantizando el máximo acumulado para Sophia.

Inicio:

```
def obtener_cantidad_max(arr):
    n = len(arr)
    # Inicializamos la matriz con 0's
    optimos = [[0] * n for _ in range(n)]
    # Resolvemos todos los subprobelmas completando la tabla de optimos
    buscar_solucion_iterativa(arr, optimos)
    # Reconstruimos las elecciones de Sophia y Mateo
    return reconstruir_monedas(arr, optimos)
```

Resolución de los subproblemas:

```python
def buscar_solucion_iterativa(arr, dp):
    n = len(arr)
    # dp = [[0] * n for _ in range(n)]

    # Casos base
    for i in range(n):
        dp[i][i] = arr[i]  # Una sola moneda
    for i in range(n - 1):
        dp[i][i + 1] = max(arr[i], arr[i + 1])  # Dos monedas

    # Llenar la tabla para intervalos mayores
    for length in range(2, n):  # Longitud del intervalo
        for i in range(n - length):
            j = i + length
            # Calcular dp[i][j]
            # Si Sofia toma la moneda arr[i]
            if arr[i + 1] > arr[j]:
                take_first = arr[i] + dp[i + 2][j]  # Mateo elige arr[i + 1]
            else:
                take_first = arr[i] + dp[i + 1][j - 1]  # Mateo elige arr[j]

            # Si Sofia toma la moneda arr[j]
            if arr[i] > arr[j - 1]:
                take_last = arr[j] + dp[i + 1][j - 1]  # Mateo elige arr[i]
            else:
                take_last = arr[j] + dp[i][j - 2]  # Mateo elige arr[j - 1]

            # Sofia maximiza su ganancia
            dp[i][j] = max(take_first, take_last)

    # El resultado está en dp[0][n-1]
    return dp[0][n - 1]
```

Reconstrucción del camino:

```python
# Constantes
primera_moneda_sophia = "Primera moneda para Sophia"
ultima_moneda_sophia = "Última moneda para Sophia"
primera_moneda_mateo = "Primera moneda para Mateo"
ultima_moneda_mateo = "Última moneda para Mateo"


def reconstruir_monedas(arr, dp):
    n = len(arr)
    i, j = 0, n - 1
    camino_sofia = []
    camino_mateo = []
    resultado = []

    while i <= j:
        # Opción 1: Sofía toma arr[i]
        if arr[i + 1] > arr[j]:
            # Mateo toma arr[i + 1]
            take_first = arr[i] + (dp[i + 2][j] if i + 2 <= j else 0)
        else:
            # Mateo toma arr[j]
            take_first = arr[i] + (dp[i + 1][j - 1] if i + 1 <= j - 1 else 0)

        # Opción 2: Sofía toma arr[j]
        if arr[i] > arr[j - 1]:
            # Mateo toma arr[i]
            take_last = arr[j] + (dp[i + 1][j - 1] if i + 1 <= j - 1 else 0)
        else:
            # Mateo toma arr[j - 1]
            take_last = arr[j] + (dp[i][j - 2] if i <= j - 2 else 0)

```

```
        # Sofía elige la mejor opción
        if dp[i][j] == take_first:
            moneda_s = arr[i]
            # Sofía toma la moneda de la izquierda
            camino_sofia.append(moneda_s)
            resultado.append(primera_moneda_sophia % (moneda_s))
            if i + 1 <= j:  # Verificar si hay una elección válida para Mateo
                if arr[i + 1] > arr[j]:
                    moneda_m = arr[i + 1]
                    camino_mateo.append(moneda_m)
                    resultado.append(primera_moneda_mateo % (moneda_m))
                    i += 2  # Mateo toma arr[i + 1]
                else:
                    moneda_m = arr[j]
                    camino_mateo.append(moneda_m)
                    resultado.append(ultima_moneda_mateo % (moneda_m))
                    i += 1  # Mateo toma arr[j]
                    j -= 1
            else:
                i += 1  # Sofía toma la última moneda
        else:
            moneda_s = arr[j]
            # Sofía toma la moneda de la derecha
            camino_sofia.append(moneda_s)
            resultado.append(ultima_moneda_sophia % (moneda_s))
            if i <= j - 1:  # Verificar si hay una elección válida para Mateo
                if arr[i] > arr[j - 1]:
                    moneda_m = arr[i]
                    camino_mateo.append(moneda_m)
                    resultado.append(primera_moneda_mateo % (moneda_m))
                    i += 1  # Mateo toma arr[i]
                    j -= 1
                else:
                    moneda_m = arr[j - 1]
                    camino_mateo.append(moneda_m)
                    resultado.append(ultima_moneda_mateo % (moneda_m))
                    j -= 2  # Mateo toma arr[j - 1]
            else:
                j -= 1  # Sofía toma la última moneda

    return camino_sofia, camino_mateo, resultado
```

3.2. Complejidad

Sea n el numero de monedas. A continuación, se describen los pasos claves y su respectiva complejidad:

- **Inicio**

  Crear tabla de óptimos:

  $$O(n^2)$$

- **Resolución de los subproblemas:**

  Se itera $(n-2) + (n-3) + ... (n-(n-1))$ veces. Esto se puede escribir como $(n-2) + (n-3) + ... 1$. Utilizando la formula de la suma de una progresión aritmética, esto equivale a

  $$O\left(\frac{((n-2)+1) * (n-2)}{2}\right)$$

  Esto si lo desarrollamos un poco mas nos queda la siguiente expresión:

  $$O\left(\frac{(n-1) \cdot (n-2)}{2}\right)$$

Es decir que este resultado crece cuadráticamente a medida que aumenta n por lo tanto la complejidad total para la resolución de los subproblemas es:

$$O(n^2)$$

- **Reconstrucción del camino:**

  Iteración desde los extremos de las monedas:

$$O(n)$$

Cabe mencionar que tantos las operaciones `append` de una lista como las asignaciones en una matriz tienen complejidad $O(1)$, por lo que no afectan la complejidad temporal general.

En resumen, la complejidad total del algoritmo es:

$$O(n^2)$$

4. Mediciones de tiempos y validación de complejidad

Al igual que en la primera parte, para verificar el correcto funcionamiento del algoritmo se realizaron diversas pruebas utilizando los conjuntos de datos proporcionados por la cátedra.

En la siguiente tabla se muestran los resultados de la ejecución de cada uno de los archivos, incluyendo las ganancias de Sophia y Mateo, junto con el tiempo de ejecución del algoritmo:

Archivo	Ganancia de Sophia	Ganancia de Mateo	Tiempo (s)
5.txt	1483	1268	$4{,}053116 \times 10^{-5}$
10.txt	2338	1780	$3{,}337860 \times 10^{-5}$
20.txt	5234	4264	$9{,}179115 \times 10^{-5}$
25.txt	7491	6523	0,000194311
50.txt	14976	13449	0,000471592
100.txt	28844	22095	0,001815319
1000.txt	1401590	1044067	0,988255978
2000.txt	2869340	2155520	0,207950354
5000.txt	9939221	7617856	7,050170183
10000.txt	34107537	25730392	43,708832979

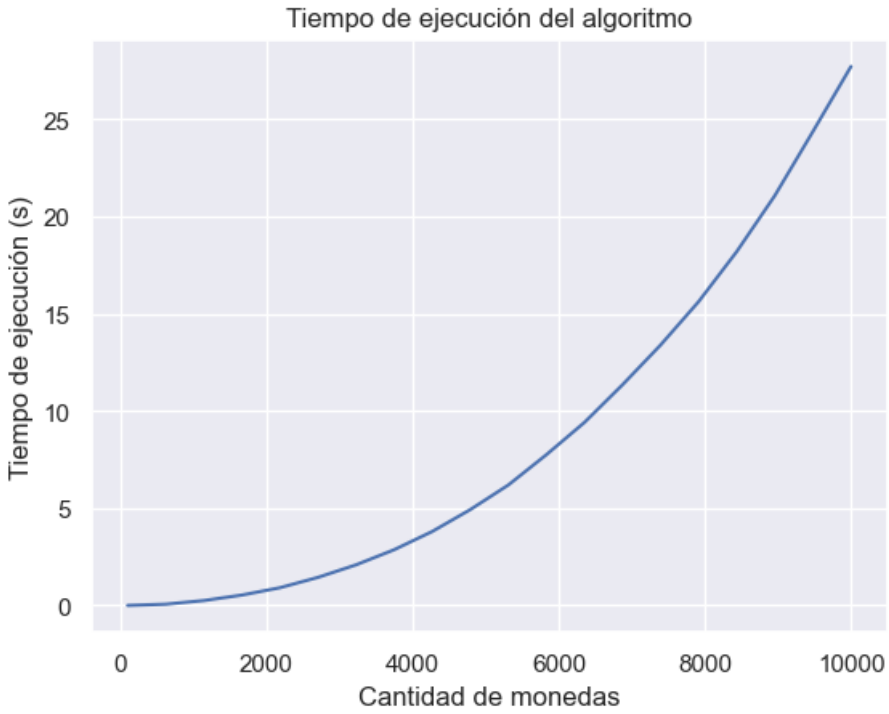
Como se puede ver en la tabla, en todas las pruebas la ganancia de Sophia fue superior a la de mateo, lo cual nos demuestra que la estrategia de Sophia es más eficiente que la de Mateo.

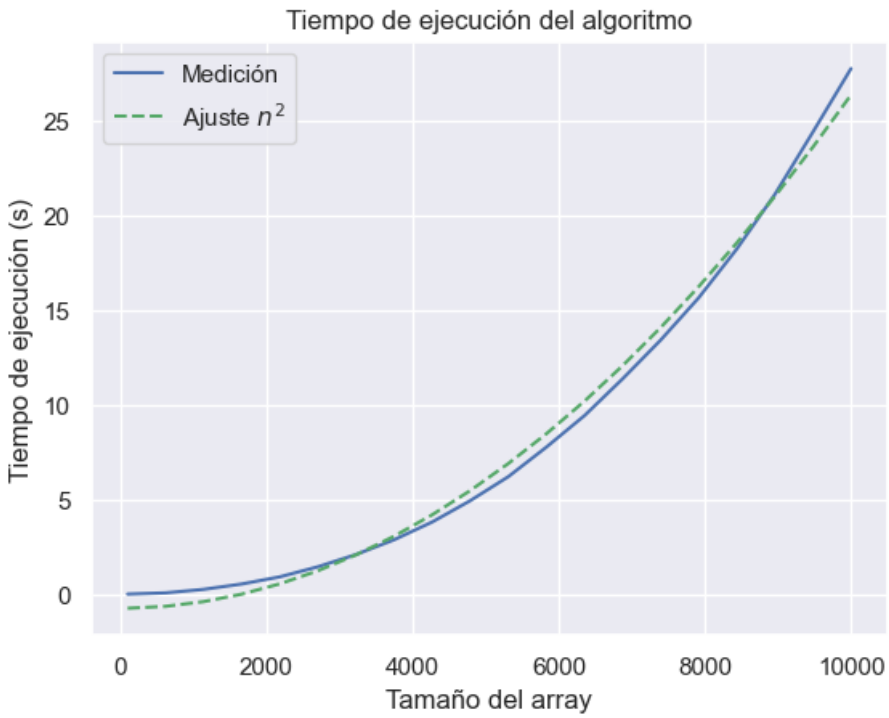
4.1. Variabilidad de la cantidad y valor de las monedas

Se realizaron mediciones utilizando diferentes cantidades de monedas (y variando su valor) generados aleatoriamente con la biblioteca `random`. Para cada caso, se realizó un ajuste por mínimos cuadrados utilizando la biblioteca `scipy` y se calculó el error para verificar la complejidad temporal del algoritmo.

Para cada uno de los conjuntos se probaron con 20 arreglos de monedas con longitudes desde 100 hasta 100000 monedas. (Cada longitud equidistanciados igualmente)

4.1.1. Conjunto 1: Valores entre 0 y 1000





Para analizar la relación entre el tiempo de ejecución y el tamaño de la entrada, se realizó un ajuste de curvas utilizando un modelo cuadrático. Como se puede observar, el tiempo de ejecución del algoritmo sigue una relación cuadrática con el tamaño de la entrada, lo cual concuerda con la complejidad teórica $O(n^2)$ del algoritmo de programación dinámica.

5. Conclusiones

En esta parte del trabajo se propuso buscar una solución eficiente para resolver el problema de Sophia garantizando el máximo valor acumulable posible para Sophia. A diferencia del primer problema ya no podemos garantizar que siempre salga ganando Sophia. Esto se debe a que existen casos como el que se menciono previamente, que sin importar lo que realice Sophia no puede ganarle a Mateo. En este caso se propuso usar un algoritmo de programación dinámica para resolverlo.

El algoritmo presentado resuelve correctamente lo buscado pero no es tan eficiente en términos de complejidad temporal como el algoritmo de la primera parte ya que este algoritmo lo resuelve en el orden cuadrático en función de la cantidad de monedas.

Se realizaron mediciones para corroborar la complejidad teórica utilizando gráficos y la técnica de cuadrados mínimos para evaluar el comportamiento del algoritmo en diferentes escenarios. Los resultados empíricos confirmaron la validez de la complejidad calculada teóricamente y mostraron que el algoritmo sigue siendo cuadrática a medida que varia la cantidad de monedas y sus valores.

Tercera parte

1. Introducción

El problema de *La Batalla Naval Individual* consiste en ubicar k barcos en un tablero de $n \times m$ casilleros, donde cada barco tiene una longitud b_i. El tablero presenta demandas en filas y columnas, y los barcos no pueden estar adyacentes, ni en filas, columnas ni diagonales.

El objetivo es determinar si es posible colocar todos los barcos de acuerdo con las restricciones de ubicación y satisfacer las demandas de cada fila y columna.

2. Análisis del Problema

Este problema consiste en ubicar k barcos en un tablero de $n \times m$ casilleros, donde cada barco i tiene una longitud b_i. Además, el tablero presenta demandas en las filas y columnas que especifican cuántos casilleros deben ser ocupados en cada una.

2.1. Restricciones de demanda

Cada fila y columna tiene una demanda de cuántos casilleros deben ser ocupados. Las longitudes de los barcos deben distribuirse de manera que la suma de los casilleros ocupados en cada fila y columna coincida con la demanda dada.

2.2. Restricciones de ubicación

Los barcos no pueden estar adyacentes entre sí, ya sea de manera horizontal, vertical o diagonal. Es fundamental tener en cuenta estas restricciones al colocar todos los barcos, asegurándose de que no se superpongan entre sí ni se salgan del tablero.

3. Problema en NP y NP-Completo

Consideremos el problema de *La Batalla Naval* en su versión de decisión:

Entrada: Un tablero de $n \times m$ casilleros y una lista de k barcos, donde el barco i tiene longitud b_i. Además, se dispone de dos listas de demandas: una para las n filas y otra para las m columnas.

- La lista de demanda para las filas es $filas = \{f_0, f_1, \ldots, f_{n-1}\}$, donde f_j indica la cantidad de demanda de la fila j.
- La lista de demanda para las columnas es $columnas = \{c_0, c_1, \ldots, c_{m-1}\}$, donde c_j indica la cantidad de demanda de la columna j.

Problema de Decisión: ¿Es posible ubicar los barcos en el tablero de modo que se cumplan las demandas en cada fila y columna, y se respeten las restricciones de ubicación de los barcos?

3.1. El problema se encuentra en NP

Para que el problema se encuentre en NP, debe existir un verificador eficiente. Es decir, un algoritmo que pueda verificar si una solución propuesta es válida en tiempo polinomial respecto a las variables del problema.

El problema consiste en asignar posiciones a barcos en un tablero de $n \times m$ casillas, cumpliendo con las demandas de filas y columnas, evitando superposiciones y respetando las restricciones de adyacencia. El verificador recibe las siguientes entradas:

- Una lista con las demandas de filas.
- Una lista con las demandas de columnas.
- Una lista con los tamaños de los barcos.
- Una solución propuesta, representada como un diccionario que asigna a cada barco una lista de posiciones.

El siguiente es el algoritmo del verificador eficiente:

```python
import math

def calcular_modulo(pos1, pos2):
    return math.sqrt((pos1[0] - pos2[0])**2 + (pos1[1] - pos2[1])**2)

def validador_eficiente(filas, columnas, barcos, solucion):
    # Verifico que los barcos sean los mismos, y su posicion se encuentren dentro del tablero
    for nro_barco, posiciones in solucion.items():
        if nro_barco > len(barcos):
            return False
        if len(posiciones) != barcos[nro_barco]:
            return False
        for pos_fil, pos_col in posiciones:
            if (pos_fil > len(filas) and pos_fil >= 0) or (pos_col > len(columnas) and pos_col >= 0):
                return False

    # Verifico que cumpla con la demanda, para esto calcula la demanda de la solucion y comparo
    demanda_fila = [0] * len(filas)
    demanda_columna = [0] * len(columnas)
    for posiciones in solucion.values():
        for pos_fil, pos_col in posiciones:
            demanda_fila[pos_fil] += 1
            demanda_columna[pos_col] += 1

    for pos_fil in range(len(filas)):
        if filas[pos_fil] != demanda_fila[pos_fil]:
            return False
        for pos_col in range(len(columnas)):
            if columnas[pos_col] != demanda_columna[pos_col]:
                return False

    # Verifica las restricciones de adyacencia, y de paso si esta superpuesto con otro barco
    for nro_barco, posiciones in solucion.items():
        for nro_otro_barco, posiciones_otro_barco in solucion.items():
            if nro_barco == nro_otro_barco:
                continue
            for posicion in posiciones:
                for otra_posicion in posiciones_otro_barco:
                    if calcular_modulo(posicion, otra_posicion) <= math.sqrt(2):
                        return False

    return True
```

3.1.1. Complejidad del validador

Antes de dar la complejidad temporal, definimos las variables de entrada del problema de la siguiente manera:

- n: Cantidad de filas del tablero.
- m: Cantidad de columnas del tablero.
- b: Cantidad de barcos.
- b_i: Largo del barco i-ésimo.

El validador realiza las siguientes operaciones:

1. **Verificar tamaños y posiciones de los barcos:** Se recorre la solución para comprobar que los barcos respeten su tamaño y posición válida. Dado que se verifican b barcos y cada uno tiene una longitud b_i, la complejidad es $O(b \cdot b_i)$.
2. **Verificar demandas de filas y columnas:** Para calcular las demandas de filas y columnas, se recorren las posiciones de los barcos. Finalmente, se recorren todas las filas y columnas por separado, verificando si la demanda ocupada por los barcos es igual a la demanda especificada para las filas y columnas. Esto resulta en una complejidad de $O(b \cdot b_i + n + m)$.
3. **Verificar restricciones de adyacencia:** Se verifica que ningún barco esté adyacente a otro (ni superpuesto). Esto implica comparar las posiciones de todos los pares de barcos. En el peor caso, la complejidad es $O(b^2 \cdot b_i^2)$.

En total, la complejidad del validador es:

$$O(b^2 \cdot b_i^2 + n + m)$$

Esto es polinomial respecto a las variables del problema (b, b_i, n, m). Por lo tanto, el problema se encuentra en NP.

3.2. Reducción de un problema NP-Completo

Ahora debemos elegir un problema NP-Completo para reducir a este. Tenemos dos opciones ofrecidas por la cátedra: el Problema de *3-Partición* o el *Bin Packing* (ambos en forma unaria).

Nosotros decidimos usar el problema de *Bin Packing*. Este problema se define de la siguiente manera:

> Dado un conjunto de números, cada uno expresado en código unario, una cantidad de bins B, expresada en código unario, y la capacidad de cada bin C, expresada en código unario, donde la suma del conjunto de números es igual a $C \cdot B$, se debe decidir si es posible dividir los números en B bins, formando subconjuntos disjuntos, tal que la suma de elementos de cada bin sea exactamente igual a la capacidad C.

El problema de *Bin Packing* en código unario es NP-Completo. Por lo tanto, tenemos que demostrar que:

$$\textit{Bin Packing} \text{ (en su forma unaria)} \leq_P \textit{Batalla Naval}.$$

3.2.1. Reducción planteada

Para resolver el problema de *Bin Packing* (BP) en su forma unaria utilizando una caja negra que resuelve el problema de *Batalla Naval* (BN), debemos mapear las variables de entrada del problema de BP a los parámetros que la caja negra de BN requiere. El problema de BP recibe los siguientes parámetros: una lista de n números con valores v_i expresados en su forma unaria, una cantidad de B bins también en forma unaria, y la capacidad C en forma unaria de cada bin.

Definimos entonces los siguientes elementos:

- Para cada valor v_i de la lista de números, se define un largo b_i. La cantidad de barcos será igual a la cantidad de números n, es decir, $b = n$. Se pasa una lista con estos largos de barco.
- La cantidad de filas será $B * 2 - 1$. Las demandas de cada fila se intercalan entre C y 0. Se proporciona una lista donde cada índice corresponde al número de fila y su valor es la demanda de esa fila.
- La cantidad de columnas será $B * C + n$. Las demandas de cada columna se intercalan entre 1 y 0. Se proporciona una lista donde cada índice corresponde al número de columna y su valor es la demanda de esa columna.

Ahora demostraremos que la reducción es correcta. Para esto, debemos demostrar que:

Hay solución de BP con n números con valores v_i y B bins con capacidad C (todos en forma unaria)

$$\Leftrightarrow$$

Hay solución de BN con b barcos de largo b_i ubicados en las n filas y m columnas posibles, cumpliendo con sus respectivas demandas y restricciones de ubicación en la reducción dada.

Demostramos ambas implicaciones.

3.2.2. Si hay BP, entonces hay BN

Si existe una solución para BP, donde los n números $v_1, v_2, \ldots, v_n$ pueden ser distribuidos en B bins de capacidad C, esto implica que se logró distribuir todos los números de manera que cada bin alcanza exactamente su capacidad.

Si consideramos que estos valores v_i son los largos de los barcos que deben ser ubicados en las filas y columnas de BN con sus respectivas demandas, se cumplirán las restricciones de ubicación y demanda. Esto es porque las restricciones de ubicacion y demanda por fila y por columna en BN fueron generadas directamente a partir de los bins y de la capacidad C en BP.

No puede suceder que algún casillero en BN esté ocupado por un barco que no cumpla con las demandas o las restricciones de adyacencia, ya que eso implicaría que los números no se distribuyeron correctamente en BP, contradiciendo la existencia de una solución para BP. Por lo tanto, si hay solución para BP, también la hay para BN.

3.2.3. Si hay BN, entonces hay BP

Ahora supongamos que existe una solución al problema de BN. Esto implica que es posible colocar los b barcos (donde para cada i-esimo barco, su largo corresponde a b_i) en un tablero de n filas y m columnas, respetando sus demandas y que cada posicion del barco no sea adyacente a otro distinto.

Dado esto, podemos construir una solución de BP de la siguiente manera:

- Los barcos de largo b_i corresponden a los números v_i.
- La cantidad filas corresponden a $B*2-1$, donde para cada demanda de una fila va intercalando entre 0 y C. Esto se puede representar de la siguiente manera:

  $$filas = \{C, 0, C, 0, \ldots, C\}$$

  Donde la cantidad de apariciones de la demanda C es B veces.
- La cantidad de columnas corresponden a $B*C+n$, donde para cada demanda de una columna vale entre 0 y 1. Esto se puede representar de la siguiente manera:

  $$columnas = \{v_1^1, v_1^2, \ldots, v_1^j, 0, v_2^1, v_2^2, \ldots, v_2^j, 0, v_i^1, v_i^2, \ldots, v_i^j, 0\}$$

  Donde habra 0 de demanda n veces, separados por j unos. Para cada v_i, su j-esimo uno (ya que esta en forma unaria) es v_i^j.

Si se cumplen las demandas y restricciones de ubicacion del tablero de BN, entonces los números pueden ser empaquetados correctamente en los bins de BP.

Un ejemplo ilustrativo de esto:

- números = [111, 1, 11, 1, 1, 1]
- B = 3
- C = 3

⇒

- barcos = [3, 1, 2, 1, 1, 1]
- filas = [3, 0, 3, 0, 3]
- columnas = [1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0]

	1	1	1	0	1	0	1	1	0	1	0	1	0	1	0
3					1					1		1			
0															
3	1	1	1												
0															
3							1	1						1	

Volviendo a la demostracion, no puede no estar todos los numeros distribuido en cada bins, debido al mapeo de numero a barcos ($v_i = b_i$). Luego, todos los numeros estaran distribuido en los bins cumpliendo con la capacidad, porque como dijimos, si no hay una distribucion que cumpla con la capacidad, implica que esos numeros no cumplen con la demanda pedida del casillero que se encuentra. Esto indicaria que la solucion brindada por BN no es correcta, lo cual es un absurdo ya que partimos la hipotesis de ahi. Entonces, si hay BN, hay BP.

3.2.4. Conclusiones

Habiendo demostrado que, si la reducción es correcta, el problema de *Batalla Naval* es un problema NP-Completo.

Aclaracion: cabe recalcar que estamos usando el problema de *Bin Packing* en su **forma unaria**, ya que sino, la reduccion seria pseudopolinomial, debido a que las listas de filas y columnas a construir, no dependerian del largo de bits que tiene el numero, sino de su valor numerico.

3.3. Bin Packing es NP-Completo

Para demostrar que el problema de *Bin Packing* es NP-Completo, utilizaremos una reducción desde un problema que ya hemos visto en clase, el *2-Partition*, el cual es NP-Completo.

Aunque no es estrictamente necesario en esta demostración, es importante mencionar que utilizamos un problema NP-completo previamente no visto en clase, el cual nos permitió demostrar que el problema de *Batalla Naval* también pertenece a NP-Completo.

Nos enfocaremos en una reducción más directa y corta desde *2-Partition* a *Bin Packing*, lo que nos permitirá concluir que *Bin Packing* también es NP-completo.

3.3.1. 2-Partition

Este problema se define de la siguiente manera:

> Dado un conjunto de números m numeros con valores $\{x_1, x_2, \ldots, x_m\}$ y un número E, decidir si es posible dividir este conjunto en dos subconjuntos disjuntos de tal manera que la suma de los elementos en cada subconjunto sea exactamente E.

3.3.2. Bin Packing esta en NP

Planteamos un validador eficiente que verifica en tiempo polinomial si una solución es válida o no.

Este validador recibe como parámetros:

- Una lista de n strings que representan los números v_i en su forma unaria.
- Una cantidad B de bins.
- Una capacidad C de cada bin.
- Una solución representada como un diccionario, donde cada clave es un bin y su valor es una lista con los números distribuidos en ese bin.

El validador realiza las siguientes verificaciones:

- Que la suma de las longitudes de los números en la lista de cada bin sea exactamente C.
- Que todos los números estén distribuidos en un solo bin.
- Que haya exactamente B bins.

La complejidad de este validador es $O(n + B \cdot C)$. Por lo tanto, como se ejecuta en tiempo polinomial respecto a las variables del problema, esto demuestra que el problema pertenece a la clase NP.

3.3.3. Reducción de un problema NP-Completo

Como dijimos antes, utilizaremos el problema *2-Partition* para hacer la reducción. Queremos hacer lo siguiente:

$$\textit{2-Partition} \leq_P \textit{Bin Packing (en su forma unaria)}$$

Primero, debemos definir una caja negra que resuelve el problema de *Bin Packing* (BP), para que podamos utilizarla para resolver el problema de *2-Partition* (2P).

Dado los parámetros de entrada de *2-Partition*:

- Una lista de tamaño m que contiene el conjunto de números.
- Un valor E que representa la capacidad de los dos subconjuntos disjuntos.

Estos parámetros se transforman para adaptarse a las entradas del problema *Bin Packing* (BP) de la siguiente manera:

- Cada número x_i del conjunto de números se mapea a un valor v_i correspondiente en su forma unaria. En términos de cantidad, se mantienen iguales ($n = m$).
- La capacidad E se asigna como la capacidad C.
- La cantidad de bins se fija como una constante, igual a 2.

Ahora, explicamos de forma breve la demostración:

Si existe una solución para 2P con m elementos, donde es posible dividir el conjunto en 2 subconjuntos disjuntos de manera que la suma de los elementos en cada subconjunto sea exactamente E, entonces existirá una solución para el problema de BP con n números de valores v_i, $B = 2$ bins y capacidad $C = E$.

Si hay solución para 2P, significa que todos los números x_i se pudieron ubicar en 2 subconjuntos disjuntos de capacidad E.

Si definimos que estos números sean equivalentes a v_i del BP, cumplirían con la misma restricción de subconjuntos disjuntos, solo que ahora ocuparían 2 bins. Como los 2 bins con capacidad C son equivalentes a los 2 subconjuntos disjuntos de capacidad E, no es posible que algún v_i quede fuera de los bins, porque sino, implicaria que algún x_i no estuviese en uno de los 2 subconjuntos, donde no se cumpliria la demanda E, lo cual es un absurdo.

Si hay solución para BP, significa que los n números se distribuyeron en B bins, de manera que cada bin llenó exactamente su capacidad C.

Podemos definir que los números x_i de 2P están representados por estos n números en BP y al ser 2 subconjuntos disjuntos en 2P, se correspone a B como 2 con capacidad $E = C$. Dado que $n = m$, los números en ambos problemas deben coincidir, y no puede haber más ni menos elementos.

Si existiera algún número x_i que no estuviera ubicado en alguno de los 2 subconjuntos, esto implicaría que los v_i correspondientes no se distribuyeron correctamente en los 2 bins con capacidad C, lo que contradice la solución válida de BP.

Este razonamiento lleva a un absurdo, ya que asumimos que BP tiene una solución válida. Por lo tanto, si hay una solución para BP, también existe una solución para *2-Partition*.

Habiendo demostrado brevemente que la reducción es correcta, queda demostrado también que el problema de *Bin Packing* es un problema NP-Completo.

4. Backtracking

4.1. Problema de optimización

Para implementar el algoritmo de *backtracking* (y el de aproximación también), se utilizó el problema de la batalla naval en su versión de optimización.

Dado un tablero de $n \times m$ casilleros y una lista de k barcos, donde el barco i tiene b_i de largo, junto con una lista de demandas de las n filas y una lista de demandas de las m columnas, el objetivo es asignar las posiciones de los barcos de tal manera que se minimice la cantidad de demanda incumplida.

Específicamente, si una columna que debería tener d_j casilleros ocupados tiene d'_j, la demanda incumplida será $d_j - d'_j$. Del mismo modo, no está permitido exceder la cantidad demandada en ninguna fila o columna.

Además, se permite que no todos los barcos sean utilizados si no es necesario para satisfacer las demandas. La estrategia busca minimizar la cantidad de demanda incumplida en el tablero.

Podemos pensar al problema como:

$$\text{mín}(\sum_{j=1}^{m} d_j - d'_j + \sum_{i=1}^{n} d_i - d'_i)$$

Siendo que:

$$0 \leq d'_j \leq d_j \quad \text{para cada columna}$$
$$0 \leq d'_i \leq d_i \quad \text{para cada fila}$$

4.2. Algoritmo de Backtracking

El algoritmo de *Backtracking* implementado se basa en ubicar los barcos de manera parcial. Si en algún punto se determina que una combinación parcial no llevará a un mejor resultado, se realiza una poda y el algoritmo retrocede para probar con una alternativa diferente.

El proceso avanza generando combinaciones de ubicación. Si se detecta que una combinación no es válida, el algoritmo retrocede y prueba con otra opción.

4.3. Implementación

Primero, en la función `backtracking`, los barcos son ordenados de mayor a menor tamaño, lo que optimiza la búsqueda. Luego, se llama a la función `ubicaciones_barcos`, que se encarga de ubicar los barcos según las condiciones del problema.

En términos generales, esta función recorre todos los casilleros del tablero e intenta ubicar los barcos, ya sea de forma horizontal o vertical. Si no es posible colocar un barco en una ubicación, se intenta con el siguiente casillero, cambiando la dirección.

Para verificar si un barco puede ser colocado en una determinada posición, se llama a la función `tratar_de_ubicar_barco`, que determina si es posible ubicarlo. Esta función, a su vez, invoca a `colocar_barco`, que asegura que el barco no esté fuera de los límites del tablero.

Si el barco se coloca correctamente, se verifica que no haya superposición con otros barcos, que no estén adyacentes y que no se exceda la demanda de las posiciones ocupadas. Si cumple todas estas condiciones, el barco se coloca en la variable `puestos`, y se intenta ubicar el siguiente barco, llamando nuevamente a `ubicaciones_barcos`.

El algoritmo comienza ubicando un barco en el casillero $(0, 0)$ y continúa hasta llegar al último casillero en la fila $n - 1$ y columna $m - 1$, tratando de colocar todos los barcos.

Para facilitar la verificación de las restricciones, se implementó una clase `Barco`, que incluye métodos para comprobar las condiciones mencionadas.

La mejor ubicación de los barcos se guarda en la variable `mejores_puestos`, y solo se actualiza si la demanda de los barcos puestos es mayor que la demanda obtenida en la mejor ubicación encontrada hasta el momento.

Finalmente, el algoritmo cuenta con varias podas de interés que permiten descartar ubicaciones que no sean mejores que la mejor ubicación encontrada hasta el momento:

- **Uso de todos los barcos:** Si ya se han utilizado todos los barcos, no tiene sentido seguir buscando combinaciones, por lo que se detiene la búsqueda.
- **Demanda total:** Se calcula la demanda de los barcos restantes y se suma con la demanda actual. Si la demanda total alcanzable es menor que la mejor demanda encontrada hasta el momento, se poda esa rama de la búsqueda.
- **Poda por tamaño de barco:** Si no se pudo colocar un barco de cierto tamaño, no se podrá colocar otro barco de igual tamaño, por lo que se salta la búsqueda de barcos de ese tamaño.

```python
# Devuelve las posiciones del barco solo si las mismas no esten afuera del tablero
def colocar_barco(direccion, pos_fil, pos_col, filas, columnas, longitud_barco):
    (fila_act, columna_act) = (pos_fil, pos_col)
    posiciones_ocupadas = set()
    se_pudo = False

    # Caso borde, posicion invalida (no tendria que ocurrir... pero por las dudas)
    if fila_act >= len(filas) or columna_act >= len(columnas):
        return posiciones_ocupadas, se_pudo

    # Trato de colocarlo horizontal o verticalmente
    if direccion == 'horizontal':
        # Me fijo si el largo excede las filas
        if columna_act + (longitud_barco-1) < len(columnas):
            for m in range(columna_act, columna_act + longitud_barco):
                posiciones_ocupadas.add((fila_act, m))
            se_pudo = True
    else:
        # Idem arriba
        if fila_act + (longitud_barco-1) < len(filas):
            for n in range(fila_act, fila_act + longitud_barco):
                posiciones_ocupadas.add((n, columna_act))
            se_pudo = True

    return posiciones_ocupadas, se_pudo
```

```
1
2  # Devuelve si se pudo ubicar un barco
3  def tratar_de_ubicar_barco(puestos, filas, columnas, barcos, barco_act,
       direccion, pos_fil, pos_col, mejores_puestos):
4      posiciones, se_pudo = colocar_barco(direccion, pos_fil, pos_col, filas,
       columnas, barcos[barco_act][1])
5      if se_pudo:
6          barco = Barco(barcos[barco_act][0], posiciones, direccion)
7          # Verifico que no se superponga con los demas, o que sea adyacente, o
        que no exceda demanda
8          if estan_superpuesto(puestos, barco) or estan_adyacentes(puestos,
       barco) or barco.excede_demanda(filas, columnas, puestos):
9              return False
10         puestos.add(barco)
11         ubicaciones_barco(puestos, filas, columnas, barcos, barco_act+1,
       mejores_puestos)
12         puestos.remove(barco)
13         return True
14     else:
15         return False
16
17
18
19 # Ubica a los barcos tratando de minimizar la demanda incumplida, obteniendo
       asi un optimo
20 def ubicaciones_barco(puestos, filas, columnas, barcos, barco_act,
       mejores_puestos):
21     # La mejor solucion es usar todos los barcos, ya no sigo buscando
22     if len(mejores_puestos) == len(barcos):
23         return
24
25     demanda_actual = demanda(puestos, filas, columnas)
26     demanda_mejor = demanda(mejores_puestos, filas, columnas)
27     demanda_faltante = demanda_barcos_faltantes(barcos, barco_act)
28
29     # Si con los barcos que tengo puestos (y los que me faltan colocar)
30     # no llego a cubrir la maxima demanda que obtuve, no sigo
31     if demanda_actual + demanda_faltante <= demanda_mejor:
32         return
33
34     # Llegue a una mejor solucion, es decir, tengo mayor demanda cumplida que
        antes
35     if demanda_actual > demanda_mejor:
36         mejores_puestos[:] = puestos
37
38     # Me fijo que no me exceda por las dudas
39     if barco_act >= len(barcos):
40         return
41
42
43     se_pudo_colocar = False
44     for n in range(len(filas)):
45         # Salteo las filas con demanda 0 o las que tengan su demanda al
       maximo
46         if filas[n] == 0 or demanda_por_fila(n, puestos) == filas[n]:
47             continue
48
49         for m in range(len(columnas)):
50             if tratar_de_ubicar_barco(puestos, filas, columnas, barcos,
       barco_act, 'vertical', n, m, mejores_puestos):
51                 se_pudo_colocar = True
52             # Para barcos de largo 1, basta con colocarlo horizontal o
       vertical
53             if barcos[barco_act][1] == 1:
54                 continue
55             if tratar_de_ubicar_barco(puestos, filas, columnas, barcos,
       barco_act, 'horizontal', n, m, mejores_puestos):
56                 se_pudo_colocar = True
57
58
```

```
59      # Si no pude colocar un barco de cierta longitud, no voy a poder colocar
        el siguiente si tiene la misma longitud
60      if not se_pudo_colocar and barco_act != len(barcos)-1:
61          barco_act += 1
62          while barcos[barco_act-1][1] == barcos[barco_act][1]:
63              barco_act += 1
64              if barco_act == len(barcos):
65                  return
66          barco_act -= 1
67
68      # No tengo en cuenta este barco (ya sea si lo pude colocar o n o), asi
        que sigo con el siguiente
69      ubicaciones_barco(puestos, filas, columnas, barcos, barco_act+1,
        mejores_puestos)
70
71  def backtracking(filas, columnas, barcos):
72      mejores_puestos = [Barco(-1, set(), True)]
73
74      # Ordeno los barcos de mayor a menor longitud
75      tuplas = [(i, largo) for i, largo in enumerate(barcos)]
76      barcos_ordenados = sorted(tuplas, key=lambda x: x[1], reverse=True)
77
78      ubicaciones_barco(set(), filas, columnas, barcos_ordenados, 0,
        mejores_puestos)
79      return mejores_puestos, demanda(mejores_puestos, filas, columnas)
```

Aclaración: Se colocaron solo las funciones que son relevantes para el algoritmo.

4.4. Complejidad

Sea n el número de filas, m el número de columnas, y b el número de barcos, donde el i-ésimo barco tiene una longitud b_i (suponiendo que es la mayor).

A continuación, se describen los pasos clave y su respectiva complejidad:

- **Ordenar los barcos:** $O(b \log b)$
- **Recorrer la matriz:** $O(n \cdot m)$
- **Colocar el barco:** $O(b_i)$
- **Verificar superposición y adyacencia:** $O(b \cdot b_i)$
- **Verificar que no se exceda la demanda:** $O(n + m + b \cdot b_i)$
- **Calcular la demanda actual:** $O(n + m + b \cdot b_i)$
- **Calcular la demanda faltante:** $O(b)$, suponiendo que faltan todos los barcos.
- **Actualizar la mejor ubicación de los barcos:** $O(b)$
- **Evitar colocar barcos de cierta longitud si no fueron colocados previamente:** $O(b)$, suponiendo que todos los barcos tienen la misma longitud.

En resumen, la complejidad total del algoritmo es:

$$O(b \log b + n \cdot m \cdot (b \cdot b_i + n + m))$$

La parte más costosa es recorrer el tablero de $n \times m$. Los otros términos, como el ordenamiento de los barcos o las verificaciones de superposición, afectan principalmente en función de la cantidad de barcos y su tamaño, pero no cambian significativamente el orden global de la complejidad.

5. Algoritmo de Aproximacion

El algoritmo de aproximación propuesto por John Jellicoe para el problema de La Batalla Naval sigue los siguientes pasos:

1. **Selección de fila/columna de mayor demanda:** En cada paso, se elige la fila o columna con mayor demanda de casilleros ocupados.
2. **Ubicación del barco más grande:** Se coloca el barco de mayor longitud en un lugar válido dentro de la fila o columna seleccionada. Si el barco es más largo que la demanda de la fila o columna, se omite y se pasa al siguiente barco.
3. **Repetir hasta que se agoten los barcos o las demandas:** Este proceso se repite hasta que no queden más barcos o no haya más demandas a satisfacer.

5.1. Análisis de Aproximación

Para medir la calidad de la aproximación del algoritmo, se define la relación entre la solución óptima $z(I)$ y la aproximada $A(I)$ como:

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

donde $r(A)$ es la *cota de aproximación* del algoritmo. El objetivo es calcular $r(A)$ y demostrar que esta cota es correcta, evaluando la calidad de la solución aproximada en comparación con la solución exacta.

5.2. Implementación

Para implementar este algoritmo de aproximación, simplemente modificamos algunas partes del algoritmo de *backtracking*, de la siguiente manera:

- Ordenamos las filas y columnas de menor a mayor demanda, de manera similar a como se ordenan los barcos por su longitud. Entonces, antes de llamar al metodo `ubicaciones_barco`, agregamos lo siguiente:

```
    # Ordeno las filas por mayor demanda
    dict_filas = {i: demanda for i, demanda in enumerate(filas)}
    dict_filas_ordenado = dict(sorted(dict_filas.items(), key=lambda
    item: item[1], reverse=True))

    # Ordeno las columnas por mayor demanda
    dict_columnas = {i: demanda for i, demanda in enumerate(columnas)}
    dict_columnas_ordenado = dict(sorted(dict_columnas.items(), key=
    lambda item: item[1], reverse=True))

```

- Antes de tratar de ubicar el barco, verificamos que su longitud no exceda la demanda de la fila ni la de la columna. Si el barco excede alguna de estas demandas, se pasa al siguiente barco. Entonces, dentro de la función `ubicaciones_barco` (en los for's anidados), cuando estamos recorriendo los casilleros, agregamos lo siguiente:

```
    while barcos[barco_act][1] > demanda_fila and barcos[barco_act][1] >
     demanda_columna:
        barco_act += 1
        if barco_act == len(barcos):
            barco_act -= 1
            return

```

5.3. Complejidad

El bucle `while` agregado sumaría complejidad temporal $O(b)$ cuando se recorre la matriz. Sin embargo, dado que ya tenemos $b \cdot b_i$ dentro del recorrido, este término no agrega complejidad adicional, ya que $O(b)$ está acotado por $O(b \cdot b_i)$.

Lo único que sí sumaría es el ordenamiento de las filas y columnas, el cual tendría una complejidad logarítmica, es decir, $O(m \log m)$ para las columnas y $O(n \log n)$ para las filas.

En total, la complejidad se expresa como:

$$O(m \log m + n \log n + b \log b + n \cdot m \cdot (b \cdot b_i + n + m))$$

5.4. Cálculo de la aproximación

Para calcular $r(A)$, se realizaron varias corridas del algoritmo de aproximación y del backtracking sobre diferentes instancias del problema, donde se variaba la cantidad de filas y columnas (con su respectiva demanda), así como la cantidad de barcos y su longitud.

En general, nos concentramos mayormente en mantener constante la cantidad de filas y columnas, y variar la cantidad de barcos.

Estas instancias se encuentran en la carpeta `test_data_aprox`, y los resultados obtenidos fueron los siguientes:

Archivo	A	Z	Cota
15_10_5.txt	4	16	0.25
15_10_10.txt	14	26	0.53
15_10_15.txt	28	40	0.70
20_15_15.txt	32	34	0.94
20_15_20.txt	46	48	0.95
20_15_25.txt	50	52	0.96
25_25_20.txt	38	50	0.76
25_25_20_barcos_largos.txt	22	34	0.64

Cota del peor caso: 0.25, por lo que la cota de la aproximación es aproximadamente $r(A) \approx 0{,}25$.

6. Mediciones

Se realizaron varias mediciones sobre los conjuntos de datos propios y proporcionados por la cátedra para los algoritmos de backtracking y de aproximación implementados.

6.1. Conjuntos de Datos Dados por la Cátedra

A continuación se muestran los tiempos de ejecución resultantes para el algoritmo de *backtracking* y de *aproximacion* con los conjuntos de datos proporcionados por la cátedra:

Archivo	**Tiempo de Ejecución (segundos)**
3_3_2.txt	0.000088
5_5_6.txt	0.040815
8_7_10.txt	0.002957
10_3_3.txt	0.000248
10_10_10.txt	0.018660
12_12_21.txt	10.707238
15_10_15.txt	0.007973
20_20_20.txt	0.071148
20_25_30.txt	0.552365

Cuadro 1: Tiempos para el algoritmo de backtracking

Archivo	**Tiempo de Ejecución (segundos)**
3_3_2.txt	0.000110
5_5_6.txt	0.030082
8_7_10.txt	0.003285
10_3_3.txt	0.000029
10_10_10.txt	2.874485
12_12_21.txt	0.573498
15_10_15.txt	0.007404
20_20_20.txt	0.063305
20_25_30.txt	0.193623

Cuadro 2: Tiempos para el algoritmo de aproximación

6.2. Conjuntos de Datos del Algoritmo de Aproximación

A continuación, se muestran los tiempos de ejecución obtenidos para el algoritmo de *backtracking* y de *aproximación* con los conjuntos de datos creados para el de aproximacion:

Archivo	**Tiempo de Ejecución (segundos)**
15_10_5.txt	0.003262
15_10_10.txt	0.007218
15_10_15.txt	0.012659
20_15_15.txt	0.111740
20_15_20.txt	0.123451
20_15_25.txt	0.255786
25_25_20.txt	0.239438
25_25_20_barcos_largos.txt	0.205760

Cuadro 3: Tiempos para el algoritmo de backtracking

Los tiempos de ejecución del algoritmo de aproximación se presentan en la siguiente tabla:

Archivo	**Tiempo de Ejecución (segundos)**
15_10_5.txt	0.000854
15_10_10.txt	0.002228
15_10_15.txt	0.008480
20_15_15.txt	0.016582
20_15_20.txt	0.029108
20_15_25.txt	0.032732
25_25_20.txt	0.076022
25_25_20_barcos_largos.txt	0.025334

Cuadro 4: Tiempos para el algoritmo de aproximacion

Como resultado vemos que los tiempos de ejecución del `algoritmo de aproximación` son considerablemente menores que los del `algoritmo de backtracking`. Esto se debe a que el algoritmo de `backtracking` busca siempre la solución óptima, evaluando todas las posibles soluciones y podando ramas cuando es posible. Este enfoque, aunque garantiza encontrar la mejor solución, puede ser muy costoso en términos de tiempo de ejecución debido a la necesidad de explorar un gran número de combinaciones.

En contraste, el `algoritmo de aproximación` busca una solución rápida sin asegurar que sea la óptima. Este enfoque reduce significativamente el tiempo de ejecución, pero sacrifica la garantía de encontrar la mejor solución posible, lo que puede llevar a obtener una solución no optima en algunos casos.

7. Conclusiones

En esta tercera parte, abordamos diversas consignas que nos proporcionaron valiosos conocimientos sobre los primeros y últimos temas vistos en la materia.

Comenzamos demostrando que el problema de la *Batalla Naval* (BN, en su versión de decisión) es NP-completo, asegurando primero que pertenece a NP. Para ello, verificamos si una solución es verificable en tiempo polinómico (con respectos a las variables del problema) y luego realizamos una reducción desde el problema de *Bin Packing* (BP, que es NP-Completo) ofrecido la cátedra. Explicamos cómo los parámetros de entrada de BP se transforman a los de BN y desarrollamos una exhaustiva demostración utilizando el método directo para confirmar la validez de la reducción.

A lo largo del proceso, pudimos demostrar que si existe una solución en BP, también la habrá en BN (y viceversa). Esta reducción nos permitió reforzar nuestras habilidades para demostrar que un problema está en NP y NP-Completo, además de fortalecer nuestra comprensión del procedimiento para abordar este tipo de problemas.

Seguimos luego con la parte del algoritmo *backtracking* para obtener una solución al problema de *Batalla Naval* en su versión de optimización. El mismo se encarga de recorrer todos los casilleros, tratando de colocar los barcos de forma horizontal o vertical, verificando que siempre se cumplan las restricciones de ubicación y demanda pedidas.

Pudimos observar empíricamente cómo son los tiempos para un algoritmo de backtracking usando pocas/muchas podas, demostrando la enorme diferencia que habría si lo hubiésemos implementado por fuerza bruta. Explorando podas de interés, íbamos viendo cómo el algoritmo ejecutaba los tests de la cátedra en un tiempo bastante menor de lo que teníamos al principio, dándonos a entender que la poda evita explorar soluciones que no van a ser mejores que las que ya veníamos obteniendo. De esta forma, el algoritmo para y retrocede sobre la recursividad, explorando nuevas soluciones posibles que tengan chances de ser mejores a las que teníamos en el menor tiempo posible.

Finalmente, dimos con el algoritmo de aproximación, el cual trataba de ir a la fila/columna con más demanda, y colocar los barcos con mayores longitudes ahí. Si no podíamos colocarlo, pasábamos al siguiente, y así hasta encontrar alguno que pudiese entrar.

Al saltearnos posibles barcos largos, no dábamos con la solución óptima al problema, pero sí una muy cercana, y en un menor tiempo que el algoritmo de backtracking, donde la diferencia de tiempo era notable.

Pudimos calcular una cota empíricamente probando varias instancias del problema, donde calculábamos la demanda obtenida tanto para el algoritmo de backtracking como para el de aproximación, así dando con la cota. Esta cota nos dice que, de la mayoría de las instancias (si fuese posible, en todas), el algoritmo de aproximación no va a ser peor que esa cota, dándonos así una referencia de qué tan malo puede ser este algoritmo al momento de encontrar una solución al problema de la batalla naval.

A todo esto, hicimos varias mediciones sobre distintas instancias del problema aplicando tanto el algoritmo de backtracking como el de aproximación, obteniendo resultados interesantes con respecto a los tiempos y soluciones encontradas.

La mayoría de estas instancias del problema fueron dadas por la cátedra, y otras creadas por nosotros mismos para probar diferentes escenarios, donde podíamos modificar a gusto la cantidad de filas/demanda, dejando fija la cantidad de barcos y su longitud, o viceversa.