Trabajo Práctico: Juego de hermanos

Teoría de Algoritmos

Curso: 75.29 – Curso Buchwald - Annual

22 de diciembre de 2024

Alumno	Padron
Brayan Ricaldi	103344
Juan Francisco Gulden	107985
Theo Miguel	106438
Franco Altiera	105400
Daniela Gómez	107652

Institución de Estudios
Facultad de Ingeniería

1. Ejercicios

1.1. Ejercicio 1: Análisis y Propuesta de Algoritmo Greedy

1.1.1. Introducción y Primeros Años

Cuando Mateo nació, Sophia estaba muy contenta. Finalmente tendría un hermano con quien jugar. Sophi tenía 3 años cuando Mateo nació. Desde pequeños, jugaban mucho juntos.

Pasaron los años y los juegos fueron cambiando. Cuando Mateo cumplió 4 años, su padre les explicó un juego a Sophia: se disponía de una fila de n monedas de diferentes valores. En cada turno, un jugador debía elegir una moneda, pero solo podía elegir la primera o la última de la fila. La moneda elegida se removía y el turno pasaba al otro jugador. El juego continuaba hasta que no quedaran monedas, y ganaba quien tuviera el mayor valor acumulado.

Mateo era muy pequeño para entender cómo funcionaba el juego, así que Sophia debía elegir las monedas por él. A pesar de ser buena hermana, Sophia, competitiva por naturaleza y con sus 7 años de edad, quería ganar. Había aprendido algunos conceptos de algoritmos greedy en la primaria y los aplicaría aquí.

1.1.2. Análisis del Problema y Propuesta de Algoritmo Greedy

Descripción del Problema El ejercicio consiste en una selección de monedas de tal manera que Sophia siempre resultara ganadora, excepto en casos donde era imposible por razones lógicas, como el caso de tener n monedas de igual valor, siendo n un número par.

La solución al problema se abordó mediante un algoritmo greedy. Este algoritmo garantizaba que en cada jugada, Sophia eligiera la moneda de mayor valor y su hermano Mateo la de menor valor. De esta manera, se aseguraba que Sophia acumulara el mayor valor posible y Mateo el menor, en cada ronda del juego.

Algoritmo Greedy Propuesto Se propone un algoritmo que Sophia utilizaría para maximizar sus ganancias.

```python
# Turno Sophia: Agarra la moneda mas grande
# Turno Mateo: Agarra la moneda mas chica
# Path: source/excercise_1/greedy.py
def obtener_ganador(monedas):
    inicio = 0
    fin = len(monedas) - 1
    monedas_sofia = []

    for i in range(len(monedas)):
        primera = monedas[inicio]
        ultima = monedas[fin]
        if i % 2 == 0:
            if primera > ultima:
                monedas_sofia.append(primera)
                inicio += 1
            else:
                monedas_sofia.append(ultima)
                fin -= 1
        else:
            if primera >= ultima:
                fin -= 1
            else:
                inicio += 1
    return sum(monedas_sofia)
```

1.1.3. Demostración de Optimalidad

Supongamos, por reducción al absurdo, que el algoritmo greedy que Sophia emplea no es óptimo. Esto implica que existe una estrategia alternativa que proporcionaría una mayor ganancia para Sophia que la estrategia greedy.

Decisiones de Sophia: En cada par de turnos, Sophia elige la moneda de mayor valor, así maximiza su ganancia en cada jugada, pero no solo eso, en cada turno de ella también le quita la oportunidad a Mateo de aumentar su ganancia, dado que le impide tomar las monedas de mayor valor.

Decisiones de Mateo: : En cada turno, Sophia elige para Mateo la moneda de menor valor, así Sophia se asegura de minimizar la ganancia de Mateo y ella puede maximizar su puntuación en su turno.

Evaluación de Estrategias Alternativas: Si Mateo eligiera una moneda que no sea la de menor valor, influiría directamente en la puntuación de Sophia, dado que la ganancia total es la suma de todas las monedas, y si Mateo aumenta la suya, disminuiría la de Sophia por una relación directa. Lo mismo pasaría si Sophia dejara su política de tomar la de mayor valor, eso haría que su ganancia disminuya y Mateo pudiera elegir una de mayor valor.

Conclusión: 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 política diferente, terminaría perjudicándose porque la relación de ganancia entre los dos es directa, lo que contradice la idea de que podría 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.

1.1.4. Justificación de la Complejidad

El algoritmo iterativamente selecciona una moneda de los extremos de la lista en cada turno hasta que no queda ninguna. La complejidad se puede analizar considerando las operaciones realizadas:

Bucle Principal:

- El bucle `for i in range(len(monedas))` se ejecuta exactamente n veces, donde n es el número de monedas.

En cada iteración, se realizan operaciones que incluyen comparaciones, actualizaciones de índices, y en el turno de Sophia, la operación `append` para agregar una moneda a su lista. Todas estas operaciones tienen una complejidad constante $O(1)$ por iteración.

Complejidad Total:

- La complejidad total del bucle principal es $O(n)$.
- Operaciones adicionales como `sum(monedas_Sophia)` al final también son $O(n)$, ya que recorren la lista de monedas acumuladas por Sophia para calcular su suma total.

Por lo tanto, la complejidad total del algoritmo es $O(n)$.

Justificación mediante Ecuación de Recurrencia: La ecuación de recurrencia del algoritmo se puede representar considerando que en cada turno se reduce el tamaño de la lista de monedas al tomar una moneda de uno de los extremos. Es decir, cada turno procesa exactamente una moneda, reduciendo la longitud de la lista en uno.

Sea $T(n)$ el tiempo que tarda el algoritmo en procesar una lista de n monedas:

$$T(n) = T(n-1) + c$$

donde:

- $T(n-1)$ representa el tiempo necesario para procesar el subproblema restante después de elegir una moneda.

- c es el tiempo constante necesario para realizar las operaciones en un turno (comparar la primera y la última moneda, actualizar índices, agregar una moneda a la lista de Sophia).

La recurrencia se puede resolver de manera iterativa:

$$\begin{aligned} T(n) &= T(n-1) + c \\ T(n-1) &= T(n-2) + c \\ T(n-2) &= T(n-3) + c \\ &\vdots \\ T(1) &= c \end{aligned}$$

Sumando estas expresiones obtenemos:

$$T(n) = T(1) + c + c + \cdots + c \quad (n \text{ veces})$$

Por lo tanto:

$$T(n) = n \cdot c$$

Como c es constante:

$$T(n) \in O(n)$$

Influencia de la Variabilidad de los Valores de las Monedas en el Tiempo de Ejecución del Algoritmo: El algoritmo planteado no se ve afectado directamente por la variabilidad de los valores de las monedas porque las decisiones del algoritmo se basan únicamente en comparaciones locales entre dos valores (la primera y la última moneda), no realiza cálculos complejos ni depende de la distribución de los valores en la lista.

1.1.5. Análisis de la 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].

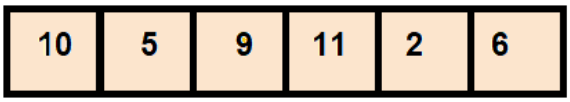


Figura 1: Vector

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

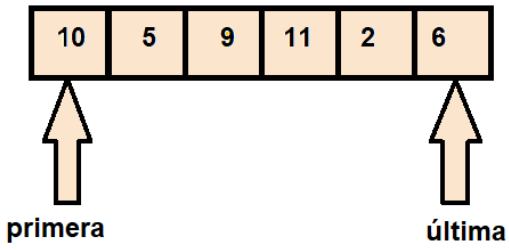


Figura 2: Extremos posibles del vector de monedas

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: Primera eleccion de Sophia

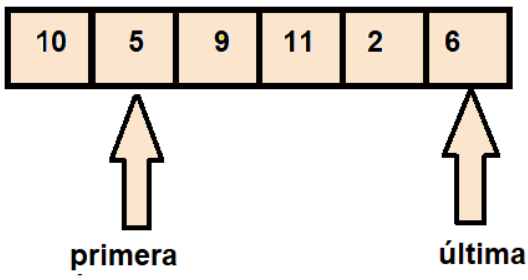


Figura 4: Indices en el turno de Mateo

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.

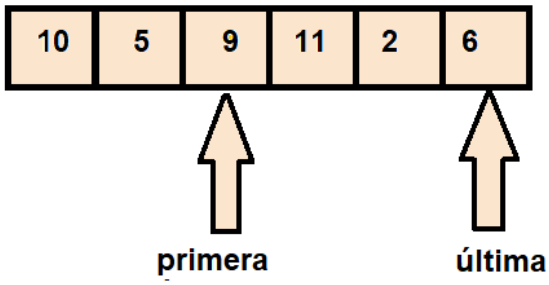


Figura 5: Indices en el 2do turno de Sophia

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:

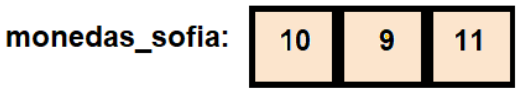


Figura 6: Monedas elegidas por Sophia

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.

1.1.6. Ejemplos de Ejecución y Verificación

El algoritmo `obtener_ganador` fue testeado con las pruebas provistas por la cátedra y superó todos los tests satisfactoriamente. Adicionalmente, desarrollamos tests con casos bordes para ilustrar y confirmar la optimización del algoritmo en todos los escenarios posibles, excepto en el caso de arrays con longitud par y monedas de igual valor, tal como se explico en el tp. A continuación, se describen los propósitos de cada test junto con los datos obtenidos.

Casos de Prueba

- **Arreglo Ordenado Ascendente:** `1;2;3;4;5;6;7`
  - *Descripción:* El array está ordenado de menor a mayor.
  - *Estrategia esperada:*Sophia elegirá las monedas del extremo derecho y Mateo las del izquierdo, asegurando que Sophia maximice su ganancia lo mayor posible.
  - *Resultado Esperado:* 22
  - *Resultado Obtenido:* Suma de las monedas seleccionadas por Sophia 22.
- **Arreglo Ordenado Descendente:** `10;9;8;7;6;5;4;3`
  - *Descripción:* El array está ordenado de mayor a menor.
  - *Estrategia esperada:* Sophia elegirá las monedas del extremo izquierdo y Mateo las del derecho, asegurando que Sophia maximice su ganancia lo mayor posible.
  - *Resultado Esperado:* 34
  - *Resultado Obtenido:* Sophia gana con una acumulacion de: 34.
- **Arreglo Valores Iguales Excepto Uno:** `1;1;1;1;1;1;1;1;1;1;1;1;1;1;2;1;1;1;1`
  - *Descripción:* Todos los valores son iguales excepto uno que es mayor.
  - *Estrategia esperada:* Sophia elegirá la única moneda diferente y por esa moneda ganará, al ser un vector de longitud par, la cantidad de monedas de Sophia y Mateo son iguales, la única diferencia entre las ganancias sera la moneda de valor 2.
  - *Resultado Esperado:* 11
  - *Resultado Obtenido:* Sophia gana con una acumulacion de: 11.
- **Arreglo Valores Máximos en Extremos:** `10;8;6;4;2;1;3;5;7;9`
  - *Descripción:* Los valores más altos están en los extremos del array.
  - *Estrategia esperada:* Sophia puede aprovechar su estrategia para siempre tomar el valor más alto y aumentar su ventaja sobre Mateo en cada turno.
  - *Resultado Esperado:* 34
  - *Resultado Obtenido:* Sophia gana con una acumulacion de: 34.

Los ejemplos de ejecución mencionados están disponibles en el repositorio dentro del archivo `test.py` ubicado en el directorio `exercise_1`.

1.1.7. Medición de Tiempos y Complejidad Empírica

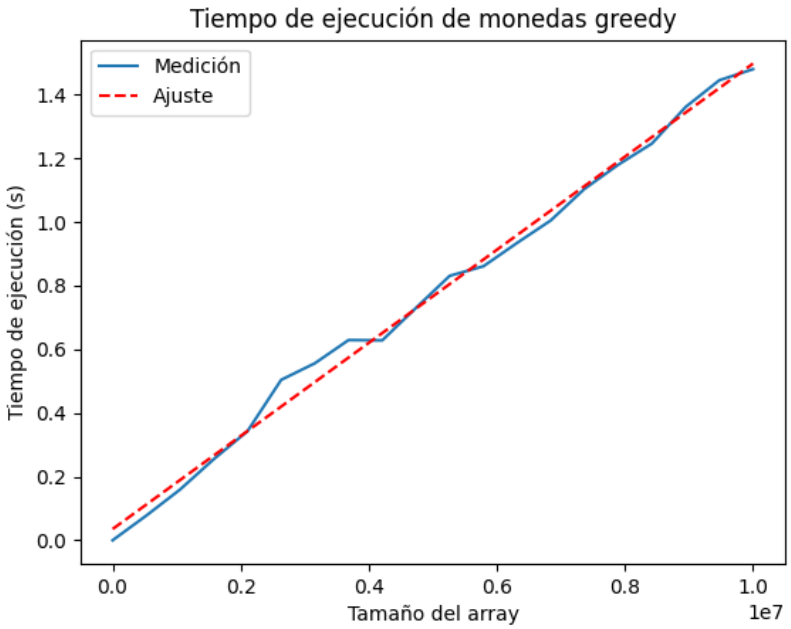


Figura 7: Mediciones

El gráfico muestra cómo el tiempo de ejecución del algoritmo `obtener_ganador` varía con diferentes tamaños del array, probando desde arrays de 100 elementos hasta 10,000,000 de elementos. La línea azul representa los tiempos medidos directamente, mientras que la línea roja discontinua indica un ajuste lineal de estos tiempos.

Análisis El ajuste lineal del tiempo de ejecución sugiere que el algoritmo escala linealmente con el tamaño del array, lo cual tiene sentido, porque la complejidad es O(n), con n la cantidad de elementos del array. El ajuste lineal del tiempo de ejecución sugiere que el algoritmo escala linealmente con el tamaño del array, lo cual es consistente por su complejidad de $O(n)$, donde n es el número de elementos del array.

1.1.8. Conclusiones

A lo largo de los items, se ha desarrollado y evaluado un algoritmo greedy para el juego de selección de monedas, diseñado específicamente para maximizar las ganancias de Sophia mientras juega con su hermano Mateo. Por las pruebas y demostraciones mencionadas se demuestra que el algoritmo es optimo, Sophia siempre va a maximizar sus ganancioas en cada iteracion.

Resumen de Resultados

- El algoritmo greedy propuesto ha demostrado ser capaz de seleccionar de manera estratégica las monedas de mayor y menor valor en los turnos de Sophia y Mateo, respectivamente. Esto asegura que Sophia pueda acumular la mayor cantidad de ganancia posible, excepto en casos donde todas las monedas tienen igual valor y su número es par.
- La variabilidad en los valores de las monedas no afecta la optimalidad del algoritmo, ya que este se basa en decisiones locales que siempre buscan el mejor resultado inmediato para Sophia.
- Las pruebas de ejecución y las mediciones de tiempo han confirmado la complejidad teórica del algoritmo.

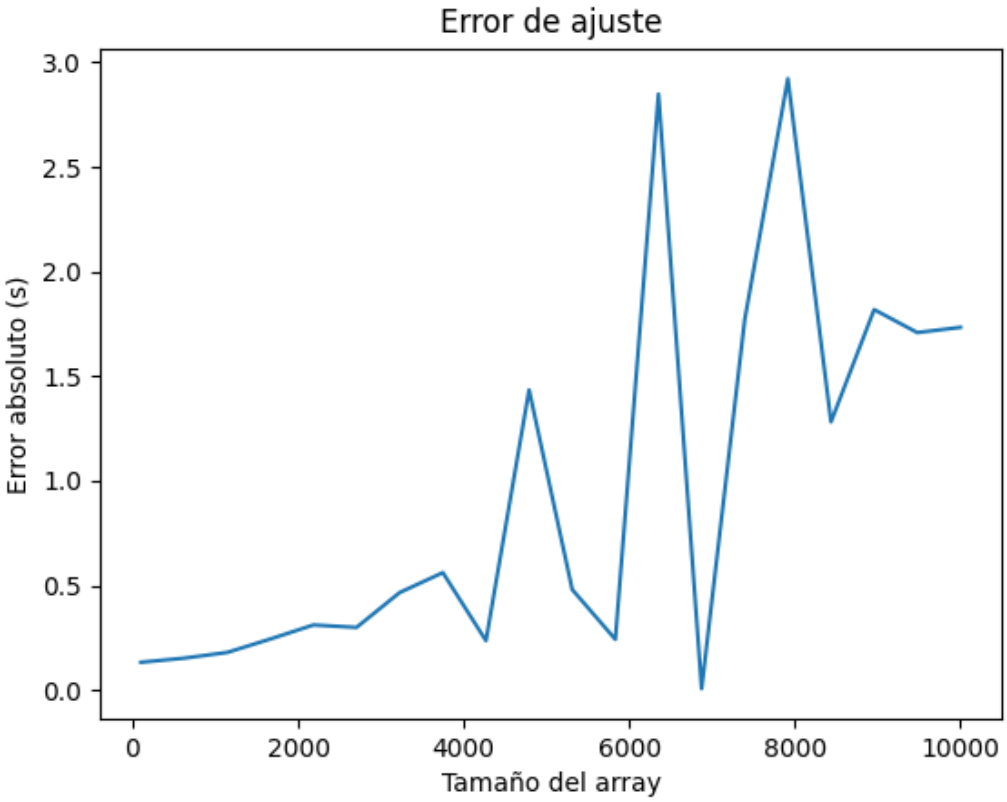


Figura 8: Error Ajuste Greedy

1.2. Ejercicio 2: Algoritmo de Programación Dinámica

1.2.1. Introducción

Pasan los años. Mateo ahora tiene 7 años, la misma edad que tenía Sophia cuando comenzaron a jugar al juego de las monedas. Mateo ha aprendido sobre algoritmos greedy y comenzó a aplicarlos, lo que hace que el resultado del juego dependa de quién comience y un poco de suerte. Sophia, decidida a ganar siempre que pueda, ahora aplicará técnicas de programación dinámica que ha aprendido recientemente.

1.2.2. Análisis del Problema

A diferencia del problema anterior, si Sophia simplemente optara por sacar la moneda más grande en cada turno, su estrategia no sería óptima, porque Mateo haría lo mismo y el resultado dependería de las monedas proceden a las que van sacando en cada turno, aquellas que no se tienen en cuenta en la estrategia anterior.
Por esta razón, habría que considerar el valor de cada una de las monedas que se encuentran entre la primera y la última de cada turno. Para entenderlo mejor, veamos un ejemplo: Supongamos que tenemos el siguiente conjunto de monedas: [6, 20, 1, 5]. Teniendo en cuenta que siempre comienza Sophia, si optara por sacar la moneda más grande en su primer turno, sacaría la de valor 6. Luego, Mateo, siguiendo la misma estrategia, sacaría la de valor 20, y Sophia la de valor 5. Finalmente, Mateo sacaría la moneda restante de valor 1 y sumaría un total de 21, mientras que Sophia sumaría 11 y perdería el juego. Si, en este caso, Sophia hubiera considerado que si sacaba la moneda de valor 6, en el siguiente turno Mateo iba a sacar la moneda de valor 20, hubiera optado por sacar la de menor valor en el primer turno y hubiese ganado el juego. Pero solo hubiese podido hacerlo si tenía en cuenta que detrás de la moneda de valor 6, se encontraba una moneda de valor 20, y que detrás de la moneda de valor 5, se encontraba otra de valor 1.

1.2.3. Algoritmo de Programación Dinámica Propuesto

Teniendo en cuenta el análisis anterior, nos dimos cuenta de que, para conseguir la solución óptima, íbamos a tener que considerar el resultado de subproblemas más pequeños dentro de nuestro conjunto de monedas. Es así que decidimos implementar un algoritmo de Programación Dinámica, que comienza calculando la solución al caso base (aquel problema en el que sólo hay una moneda) y continúa calculando la solución a subproblemas más grandes en base a las soluciones guardadas para los subproblemas más pequeños, hasta que llega al problema que queremos resolver. El cuál se calcula teniendo en cuenta la resolución de todos los subproblemas más pequeños que parten de este. Dicho esto, decidimos plantear nuestro problema de Programación Dinámica como un problema de dos variables: *inicio* y *fin*. Estas variables representan la posición de inicio y la posición de fin del vector, respectívamente.
Al estar trabajando con dos variables, decidimos guardar los óptimos para cada subproblema en una matriz. De dicha matriz, únicamente vamos a trabajar con la matriz triangular inferior, porque no nos interesa el caso en el que inicio sea mayor a fin.
Comenzamos planteando el óptimo para los subproblemas más pequeños, es decir, para cuando inicio y fin tienen el mismo valor. Por lo que iteramos sobre la diagonal de la matriz y vamos guardando el óptimo, que será el valor de la moneda en esa posición en la lista (inicio o fin), y luego seguimos a los subproblemas de longitud 2, cuyos óptimos serán los máximos entre ambas monedas.
Una vez que tenemos los óptimos para los subproblemas de longitud 1 y 2, comenzamos a aplicar la ecuación de recurrencia, pero antes veamos como obtenerla.
Si tenemos un conjunto de 3 monedas, y teniendo en cuenta que siempre comienza Sophia, si Sophia saca la primera moneda, la próxima que debería sacar sería el mínimo entre las dos restantes, ¿Por qué? Porque como dijimos antes, Mateo va a sacar la moneda de mayor valor en cada turno. Esto es lo mismo que plantear que si Sophia saca la primera moneda, la próxima que podría sacar sería el óptimo de la solución para la segunda moneda si la tercera moneda es mayor que la segunda, o el óptimo de la solución para la tercera moneda en caso contrario, que para ambos casos correspondería al valor de las respectivas monedas. Por lo tanto, el óptimo para este caso sería el máximo entre sacar la primera moneda o sacar la última, que en pseudo-código se traduce en:

```
max((monedas[0] + (monedas[1] if monedas[2] > monedas[1] else monedas[2])), (
    monedas[2] + (monedas[1] if monedas[0] > monedas[1] else monedas[0])))
```

Siguiendo la misma lógica, si pasamos a un conjunto de 4 monedas, el óptimo se podría calcular como el máximo entre sacar la primera moneda sumada al óptimo entre el subconjunto restante luego de quitar la primera moneda y la moneda de mayor valor del subconjunto resultante, y la última moneda sumada al óptimo entre el subconjunto restante luego de quitar la ultima moneda y la moneda de mayor valor del subconjunto resultante.
Teniendo en cuenta todo esto, ya podemos plantear la ecuación de recurrencia, que es la siguiente:

$$\text{Si inicio} == \text{fin}:$$
$$OPT(\text{inicio}, \text{fin}) = \text{monedas[inicio]}$$

$$\text{Si inicio} + 1 == \text{fin}:$$
$$OPT(\text{inicio}, \text{fin}) = \text{monedas[inicio]}$$

$$\text{Si inicio} + 1 > \text{fin}:$$
$$OPT(\text{inicio}, \text{fin}) = \text{máx} \left(\begin{array}{l} \text{monedas[inicio]} + \begin{cases} OPT(\text{inicio}+1, \text{fin}-1), & \text{si monedas[inicio}+1] < \text{monedas[fin]} \\ OPT(\text{inicio}+2, \text{fin}), & \text{e.o.c.} \end{cases} \\ \\ \text{monedas[fin]} + \begin{cases} OPT(\text{inicio}+1, \text{fin}-1), & \text{si monedas[inicio]} < \text{monedas[fin}-2] \\ OPT(\text{inicio}, \text{fin}-2), & \text{e.o.c.} \end{cases} \end{array} \right)$$

Tener en cuenta que la variable *monedas* es una lista de monedas, y que las variables *inicio* y *fin* corresponden a primera y ultima posición en la lista, respectivamente. El óptimo del problema que queremos resolver será OPT(0, n-1), que es el óptimo de la lista de monedas de tamaño n.

```
def maxima_ganancia_sofia(monedas):
    n = len(monedas)

    matriz_solucion = [[None] * n for _ in range(n)]

    for i in range(n):
        matriz_solucion[i][i] = monedas[i]

    for i in range(n - 1):
        matriz_solucion[i+1][i] = max(monedas[i], monedas[i+1])

    for i in range(2, n):
        k = i
        for j in range(n - i):

            opt_1 = monedas[k] + (matriz_solucion[k-2][j] if monedas[k-1] >=
    monedas[j] else matriz_solucion[k-1][j+1])
            opt_2 = monedas[j] + (matriz_solucion[k-1][j+1] if monedas[k] >=
    monedas[j+1] else matriz_solucion[k][j+2])

            matriz_solucion[k][j] = max(opt_1, opt_2)
            k += 1

    solucion = []
    reconstruir_solucion(monedas, matriz_solucion, 0, n-1, solucion)
    return solucion
```

Listing 1: Algoritmo de Programación Dinámica Propuesto

Complejidad del Algoritmo Para analizar la **complejidad temporal** del algoritmo, vamos a ir analizando cada paso por separado.

- Inicializar la matriz supone una complejida de $O(n^2)$.
- Cargar los valores óptimos para los subproblemas de longitud 1, tiene una complejidad de $O(n)$.
- Cargar los valores óptimos para los subproblemas de longitud 2, tiene una complejidad temporal de $O(n-1) = O(n)$
- La función *maxima_ganancia_sofia* tiene un ciclo for que itera n veces, y dentro de este ciclo for hay otro ciclo for que itera n - 1 veces, y todas las operaciones que se realizan dentro de este bucle tienen complejidad O(1). La complejidad, entonces, sería del orden de la serie aritmética:

$$\sum_{i=2}^{n-1}(n-i) = (n-2) + (n-3) + ... + 1 = \frac{(n-2)(n-1)}{2} = \frac{n^2-3n+2}{2}$$

  Esto nos da una complejidad temporal de $O(n^2)$.

- La función *reconstruir_solucion* va a hacer como maximo n/2 + 1 llamadas recursivas, ya que Sophia agarra la mitad de las monedas en total, y si el total de monedas es impar, podria agarrar una moneda más que la otra mitad en el peor de los casos, por lo tanto la complejidad temporal es de $O(n)$.

Finalmente, la **complejidad temporal** de nuestro algoritmo sería:

$$O(n^2)$$

Por su parte, la **complejidad espacial** también es de $O(n^2)$ porque utilizamos una matriz para guardar los óptimos.

1.2.4. Ejemplo de ejecución

Supongamos que tenemos la siguiente lista de monedas: [8, 5, 20 ,9]. El primer paso es inicializar la matriz, que tendrá una dimensión de 4x4:

fin\inicio	0	1	2	3
0	-	-	-	-
1	-	-	-	-
2	-	-	-	-
3	-	-	-	-

En segundo lugar, se cargan los óptimos para los subproblemas de longitud 1:

fin\inicio	0	1	2	3
0	8	-	-	-
1	-	5	-	-
2	-	-	20	-
3	-	-	-	9

Luego, se cargan los óptimos para los subproblemas de longitud 2, que serán el máximo entre cada par de monedas consecutivas:

fin\inicio	0	1	2	3
0	8	-	-	-
1	8	5	-	-
2	-	20	20	-
3	-	-	20	9

Por último, se entra al tercer bucle, donde se cargan los óptimos para los subproblemas de longitud 3 y para el problema de longitud 4. El óptimo para nuestro problema, por lo tanto, se encontrará en la posición (0,3) de la matriz de óptimos.

fin\inicio	0	1	2	3
0	8	-	-	-
1	8	5	-	-
2	25	20	20	-
3	-	14	20	9

fin\inicio	0	1	2	3
0	8	-	-	-
1	8	5	-	-
2	25	20	20	-
3	28	14	20	9

1.2.5. Demostración de la Optimalidad

Tal como se mencionó anteriormente, 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.
Para probar que el algoritmo encuentra la solución óptima:

- **Base del caso:**
  - Si inicio == fin: Solo hay una moneda disponible, por lo que OPT(inicio, fin) = monedas[inicio]. Esto es trivialmente óptimo.
  - Si inicio + 1 == fin: Hay dos monedas, y Sophia elige la de mayor valor, asegurando el máximo beneficio. Esto también es óptimo.
- **Hipótesis inductiva:** Supongamos que el algoritmo calcula correctamente la solución óptima para todos los subrangos [inicio, fin] con longitud $k < n$.
- **Paso inductivo:**
  - Para un rango [inicio, fin] de longitud $k = n$:

- El algoritmo evalúa las dos posibles elecciones iniciales de Sophia y calcula la ganancia futura usando las soluciones óptimas de los subrangos más pequeños.
- Como las soluciones de los subrangos son óptimas (hipótesis inductiva), el algoritmo asegura que la decisión de Sophia en [inicio, fin] también es óptima.

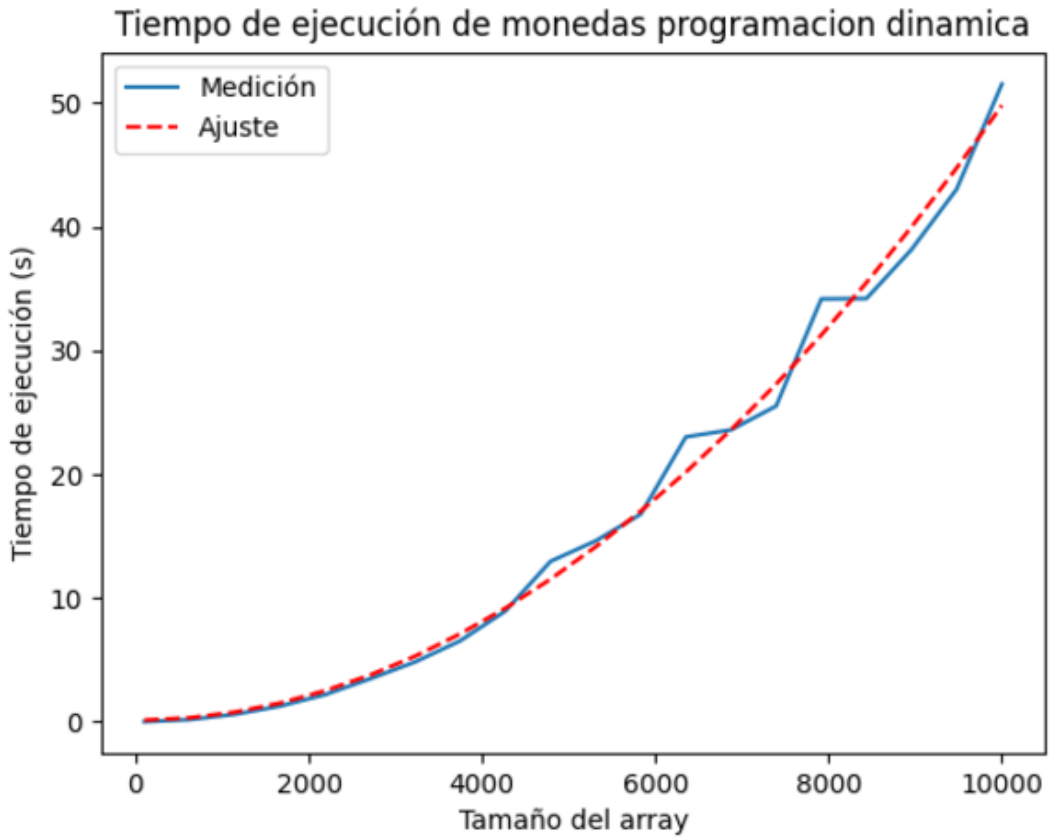
Por lo tanto, el algoritmo construye soluciones óptimas para rangos más grandes a partir de subproblemas más pequeños, cumpliendo con el principio de optimalidad.

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 [1,100,1], Sophia comenzaría sacando alguna de las monedas de valor 1 y Mateo sacaría la de valor 100, por lo que ganaría el juego independientemente de la estrategia tomada por Sophia.

1.2.6. Medición de Tiempos y Complejidad Empírica

Se realizaron mediciones utilizando distintos conjuntos de datos con tamaños variados para evaluar cómo la variabilidad en los tamaños de las monedas impacta en la complejidad del algoritmo.

Antes de estas pruebas, se llevó a cabo un análisis teórico de la complejidad del algoritmo propuesto. Para validar que la complejidad estimada teóricamente corresponde con el comportamiento empírico, ajustamos los tiempos obtenidos para cada conjunto de datos mediante el método de mínimos cuadrados a una curva representativa. Este enfoque nos permitió analizar con mayor precisión la relación entre el tamaño de los datos y el tiempo de ejecución del algoritmo.



Como podemos observar en los gráficos, el tiempo de ejecución se ajusta correctamente a una curva cuadrática a medida que la cantidad de monedas en el problema aumenta. Esto nos reafirma que la complejidad calculada en el punto 3 es correcta.

El error de ajuste, medido como la diferencia absoluta entre los tiempos reales y los estimados, muestra valores bajos para tamaños pequeños del array, pero incrementa su variabilidad a medida que crece el tamaño del array.

1.2.7. Conclusiones

A través del análisis del problema y la implementación de una solución basada en programación dinámica, queda claro que este enfoque resulta indispensable cuando se trata de problemas donde las decisiones locales no aseguran una solución global óptima. El caso del juego de las monedas ilustra perfectamente esta situación: una estrategia çodiciosa", como elegir siempre la moneda de mayor valor disponible, puede ser fácilmente derrotada por un adversario que piense varios movimientos hacia adelante.

En cambio, la programación dinámica permite evaluar todas las posibilidades de forma sistemática, garantizando que Sophia tome las decisiones más inteligentes en cada etapa del juego, anticipándose a las posibles respuestas de Mateo. La implementación desarrollada demostró ser eficiente en términos de tiempo y espacio, con una complejidad de $O(n^2)$, y permitió resolver el problema de manera estructurada y clara.

Más allá de este ejercicio específico, lo interesante de haber resuelto este problema es que los principios aplicados (descomposición del problema en subproblemas, reutilización de soluciones parciales y optimización global) son herramientas muy potentes que se pueden trasladar a la resolucion de otros problemas.

1.3. Ejercicio 3:Cambios

1.3.1. Introducción

Los hermanos siguieron creciendo. Mateo también aprendió sobre programación dinámica, y cada uno aplicaba la lógica sabiendo que el otro también lo hacía. El juego de las monedas se tornó aburrido en cuanto notaron que siempre ganaba quien empezara o según la suerte. Los años pasaron, llegó la adolescencia y empezaron a tener gustos diferentes. En general, jugaban a juegos individuales. En particular, Sophia estaba muy enganchada con un juego inventado en Argentina por Jaime Poniachik en 1982: La Batalla Naval Individual.

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

1.3.2. Demostración de que el Problema de la Batalla Naval se Encuentra en NP

Para demostrar que el problema se encuentra en NP, construimos un validador eficiente. Es decir, un validador cuya complejidad temporal sea polinomial. El problema recibe una lista de listas que representa al tablero, junto con una lista que contiene las restricciones de filas y otra que contiene las restricciones de columnas, y otra lista que contiene a todos los barcos, donde cada numero representa la longitud de un barco.
Dicho tablero estará compuesto $0s$ para las celdas vacías y $1s$ para las celdas que son ocupadas por barcos.

```
def naval_battle_validator(board: List[List[int]], boats: List[int], res_rows: List
    [int], res_cols: List[int]):
    """
    Complejidad: O(m x n) + O(m x n + n/2 x (m + k)) = O(m x n)
    """

    if len(board) == 0:
        return False

    if len(board[0]) == 0:
        return False

    if len(res_rows) == 0 or len(res_cols) == 0 or len(boats) == 0:
        return False

    if sum(boats) > len(board) * len(board[0]): # Si la dimension del tablero es
    menor que la cantidad de posiciones a ocupar, no es valido.
        return False

    if not validate_restrictions(board, boats, res_rows, res_cols): # O(m x n)
        return False

    for i in range(len(board)): # O(n)
        for j in range(len(board[i])):  # O(m)
            if board[i][j]:
                if not validate_boat(board, boats, i, j):   # O(max(m, n) + k)
                    return False

    if len(boats) > 0:  # Si no se colocaron todos los barcos
        return False

    return True
```

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 `validate_restrictions` tiene una complejidad de $O(m \times n)$ por la validación de las restricciones de filas y columnas.
- La función `validate_boat` 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 `naval_battle_validator`, se deben tener en cuenta los siguientes aspectos:

- La función `validate_restrictions` 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 `validate_boat`, 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 `validate_boat`. 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 `validate_boat`, 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 `naval_battle_validator` 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 `validate_boat` 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 `validate_boat`, que tendría una complejidad de $O(m)$ ó $m/2$ veces la función `validate_boat`, 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 `naval_battle_validator` 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**.

1.3.3. Demostración de que el Problema de la Batalla Naval es NP-Completo

Para demostrar que el problema de **Batalla Naval** en su versión de decisión es NP-Completo, debemos seguir los siguientes pasos:

1. Probar que Batalla Naval está en NP (esto ya fue demostrado anteriormente).
2. Tomar un problema que sabemos de antemano que es NP-Completo.
   - Utilizamos el problema **3-Partition** en su versión unaria y de decisión.(Demostración de NP-Completo al final de la sección)
3. Armar una **reducción** que tome cualquier instancia del problema NP-Completo y la convierta en una instancia de Batalla Naval en tiempo polinomial.
4. Demostrar que el problema 3-Partition tiene solución **si y sólo si** la instancia de Batalla Naval generada por la reducción tiene solución.

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.

Nota Adicional: Demostración de NP-Completitud de 3-Partition Para demostrar que 3-Partition es NP-Completo debemos reducir un problema que sabemos que es NP-Completo a 3-Partition. En este caso utilizaremos Partition (o 2-Partition) que, como vimos en clase, es NP-Completo.

En primer lugar demostramos que 3-Partition es NP. La demostración es trivial, ya que dada una posible solución de 3-Partition, basta con obtener la suma de cada uno de los 3 conjuntos y verificar que sea igual a la suma de todos los elementos originales dividido 3.

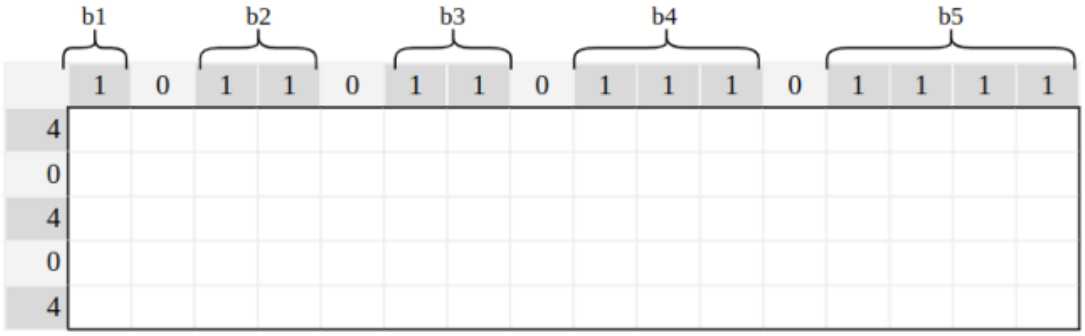
Ahora debemos transformar una instancia de Partition y convertirla en una instancia de 3-Partition en tiempo polinomial. Dado un conjunto S = s1, s2, ..., sN tal que sum(S) = T, quiero hayar dos subconjuntos disjuntos de S tal que la suma de cada uno sea T/2. Para convertirlo en una instancia de 3-Partition basta con armar un conjunto S*, que contenga todos los elementos de S, más un elemento X de valor T/2. De esta forma, el problema se trata de encontrar 3 subconjuntos disjuntos de S* de forma que la suma de cada uno valga T/2. Por último demostramos que la instancia de Partition tiene solución si y sólo si la instancia de 3-Partition tiene.

Si Partition tiene solución entonces 3-Partition la tiene: Si el conjunto original S tiene solución entonces se pueden armar dos subconjuntos cuya suma sea T/2. Entonces, en la versión de 3-Partition podremos armar los mismos subconjuntos y un tercer subconjunto que contiene al elemento X, obteniendo así 3 subconjuntos de suma T/2, que solucionan el problema.

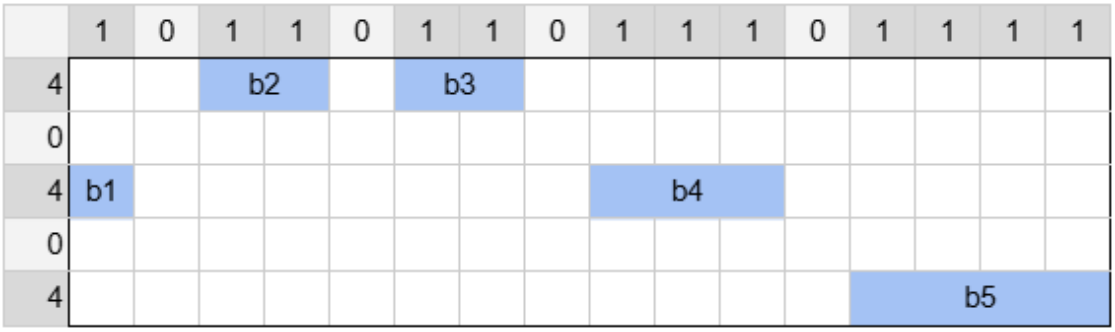
Si 3-Partition tiene solución entonces Partition la tiene: Una solución de la instancia de 3-Partition está compuesta por 3 subconjuntos de suma T/2. Al ser X de ese mismo valor, uno de los 3 contiene sólo ese elemento. Por lo tanto, los otros dos subconjuntos tendrán los elementos originales S y cada uno sumaría T/2. Entonces estos otros dos subconjuntos serían la solución de Partition.

Supongamos que tenemos la siguiente instancia de 3 partition: S = 1, 2, 2, 3, 4. Dado que sum(S) = 12, queremos probar si es posible armar 3 subconjuntos disjuntos de S cuya suma individual sea 4.

Transformamos este problema en un problema de Batalla Naval siguiendo los pasos descritos anteriormente. Obteniendo el siguiente tablero:



Se puede observar cómo hay 3 filas con demanda 4 y en las columnas se puede apreciar cómo cada barco tiene un espacio "dedicado" para que pueda colocarse horizontalmente en cualquiera de las 3 filas. Ahora solo resta pasar esa instancia de Batalla Naval a nuestro resolvedor de backtracking y obtendremos el siguiente resultado:



Podemos ver que la instancia de Batall naval tiene solución, colocando todos los barcos y satisfaciendo toda la demanda. Entonces, por lo demostrado anteriormente, el problema original de 3-Partition también la tiene y es: S1 = 1, 3, S2 = 2,2 y S3 =4.

1.3.4. Algoritmo de Backtracking Propuesto

Descripción del Algoritmo El algoritmo de Backtracking propuesto recibe una lista de barcos a colocar, una lista de demandas de filas y otra de demandas de columnas, y devuelve la solución óptima al problema de la Batalla Naval. Es decir, aquella que minimiza la demanda incumplida.
Este algoritmo comienza por ordenar los barcos de mayor a menor tamaño, e intenta colocar cada barco en cada posición del tablero, probando, por cada posición, colocar el barco en alguna orientación (horizontal o vertical). Una vez probadas las distintas variantes con el barco actual puesto, se prueba la opción de no colocar el barco actual y pasar al siguiente. El algoritmo termina cuando la solución encontrada es óptima. Es decir, cuando puede asegurar que no se puede mejorar la mejor solución actual obtenida.
¿Cómo se logra esto? Mediante podas. Las podas son fundamentales para no iterar sobre alternativas que sabemos que no van a llegar a la solución óptima. Son los elementos clave que distinguen a un algoritmo de Backtracking de un algoritmo de Fuerza Bruta.
En nuestro algoritmo, utilizamos distintas podas. Estas son:

1. Para cada posición, antes de colocar el barco, se chequea si es posible colocarlo, de acuerdo a las restricciones de filas, de columnas, y de adyacencias. Esta es la poda más trivial y a su vez, la más importante.

2. Por cada barco sobre el que iteramos, chequeamos que la diferencia entre demanda incumplida conseguida hasta este barco (no necesariamente la correspondiente a la mejor solución actual) sumada a la suma del tamaño de los barcos restantes sobre los que queda iterar, sea menor a la demanda incumplida obtenida en la mejor solución actual. De esta forma, nos ahorramos tener que iterar sobre posibles soluciones que ya sabemos que no van a ser óptimas.

```
    def is_better_solution_possible(board, ships, current_ship, best_solution)
    :
    remaining_ships = sum(ships[current_ship:])
    best_atteinable_solution = board.get_available_demand() - remaining_ships
    * 2
    if (
        len(best_solution.ocuppied_boxes) > 0
        and best_atteinable_solution >= best_solution.remaining_demand
    ):
        return False

```

3. Por cada barco sobre el que iteramos, si ese barco tiene el mismo tamaño que el anterior, y el anterior se omitió, entonces omitimos el barco actual también y pasamos al siguiente. Si no hicieramos esto, volveríamos a probar muchas variantes posibles que ya se habían probado con el barco anterior puesto.

4. Si el tamaño del barco es mayor que el máximo entre las demandas de filas y de columnas para cualquier fila y columna, se omite, ya que sabemos de antemano que no va a ser posible colocarlo en ninguna orientación.

5. Si el barco actual tiene el mismo tamaño que el anterior, y el anterior se colocó, entonces vamos a comenzar a iterar sobre el tablero desde la posición en que se colocó ese barco, porque si sabemos que en las posiciones anteriores no fue posible colocar ese barco, tampoco lo será para el barco siguiente si tiene el mismo tamaño.

```
# Algoritmo de backtracking para resolver el problema de la Batalla Naval
def resolver_batalla_naval(tablero, barcos, restricciones_filas,
    restricciones_columnas):
    # Implementacion del algoritmo de backtracking
    pass
```

Listing 2: Algoritmo de Backtracking para la Batalla Naval

Complejidad del Algoritmo La complejidad de este algoritmo depende de la cantidad de barcos y la cantidad de filas y columnas del tablero. En el peor caso, el algoritmo intentará colocar cada barco en cada fila o columna, lo cual resulta en una complejidad de O(n * m * k), donde n y m son las dimensiones del tablero y k es la cantidad de barcos. En la práctica, la complejidad que se manifiesta es mucho menor, ya que el algoritmo no intentará colocar cada barco en cada fila o columna, sino que intentará colocar los barcos en las filas o columnas con mayor demanda. Por lo tanto, la complejidad real del algoritmo es mucho menor que O(n * m * k), pero no se puede determinar con exactitud.

1.3.5. Modelo de Programación Lineal

Implementamos un modelo de programación lineal que resuelve el problema de batalla naval de forma óptima. El modelo utiliza dos tipos principales de variables booleanas:

1. Variables para las celdas del tablero: Estas variables representan si una celda específica del tablero está ocupada (**11**) o no (**00**). Para su creación y almacenamiento, utilizamos la función *LpVariable.dicts()* de la librería PuLP. Este conjunto de variables garantiza que las celdas ocupadas correspondan a la colocación de barcos válidos.

2. Variables para la colocación de barcos: Definimos un conjunto de variables booleanas que indican si un barco específico está colocado comenzando en una posición dada (fila y columna)

y con una orientación específica (horizontal o vertical). Hay una variable de este tipo para cada barco, en cada posición válida del tablero y para ambas orientaciones. También empleamos diccionarios de PuLP para gestionar estas variables de manera estructurada.

Restricciones del modelo El modelo está regido por un conjunto de restricciones diseñadas para cumplir con las reglas del juego y las demandas de filas y columnas:

1. Restricciones de colocación de barcos: - Cada barco debe estar colocado a lo sumo una vez. Esto asegura que no haya duplicación en la colocación de barcos. - La colocación de un barco debe garantizar que las celdas que ocupa estén marcadas como ocupadas en el tablero. - Un barco no puede estar adyacente a otro (ya sea en celdas verticales, horizontales o diagonales).

2. Restricciones de las celdas del tablero: - Cada celda puede estar ocupada por lo sumo un barco. - Una celda marcada como ocupada debe estar asociada a la colocación válida de un barco.

3. Restricciones de demanda por filas y columnas: - La cantidad de celdas ocupadas en cada fila y columna no puede exceder la demanda máxima especificada. - Se minimiza la diferencia entre la demanda máxima y la cantidad real de celdas ocupadas, con el fin de satisfacer la mayor cantidad posible de la demanda.

Funcion objetivo La función objetivo está diseñada para maximizar la ocupación efectiva del tablero, minimizando la diferencia entre las demandas especificadas y las celdas ocupadas. En términos matemáticos, buscamos minimizar la suma de las demandas insatisfechas de todas las filas y columnas. Resolución del modelo

Resolución del modelo El modelo fue implementado utilizando la librería PuLP, que permite definir, resolver y optimizar problemas de programación lineal. Una vez definido el problema y sus restricciones, se utilizó el solver CBC para encontrar la solución óptima. Como resultado, obtenemos:

- El tablero con la disposición óptima de los barcos.
- Las celdas ocupadas en cada fila y columna.
- La demanda satisfecha e insatisfecha para cada fila y columna.

Conclusiones Al correr los casos de prueba otorgados por la cátedra, en los casos con menor tamaño de entrada, obtuvimos tiempos de ejecución competitivos con los de *backtracking*. Sin embargo, a medida que el tamaño de los datos (más filas, más columnas y más barcos) crece, se hace notar la complejidad exponencial de los modelos de programación lineal. A los últimos casos de prueba les lleva mucho tiempo finalizar la ejecución.

A continuación, mencionamos algunas posibles optimizaciones para contrarrestar esto que no llegamos a implementar:

- Posiblemente la estructura `LpVariable.dicts()` no sea la más eficiente para guardar las variables booleanas, ya que, si su implementación por debajo es una tabla de hash, entonces el costo de sus operaciones es $O(\log(n))$. En cambio, si guardásemos las variables en una matriz, el acceso sería en $O(1)$.
- Por otro lado, podríamos reducir bastante la cantidad de variables booleanas, que justamente es el único parámetro que afecta la complejidad (ya que los modelos de programación lineal son exponenciales sobre la cantidad de variables booleanas). Una forma de lograr esto es no crear las variables de colocación (barco, posición y orientación) para las posiciones donde ese barco con esa orientación no entra en el tablero.

- Otra forma de reducir la cantidad de variables es no tener en cuenta cada barco individualmente, sino tomarlos por su largo. De esta forma, si hay muchos barcos de largo repetido, habrá menos variables innecesarias. Luego, por ejemplo, para la restricción de que un barco de cierto largo pueda ser colocado a lo sumo una vez, se cambiaría por que pueda ser colocado tantas veces como barcos de ese tamaño haya.
- Además, se podrían eliminar algunas de las restricciones que pusimos que sean redundantes.

1.3.6. Algoritmo de Aproximación Propuesto por John Jellicoe

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.

```python
def naval_approximation(demands_rows, demands_cols, ships):

    n = len(demands_rows)
    m = len(demands_cols)
    board = np.zeros((n, m), dtype=int)
    ships = sorted(ships, reverse=True)

    while ships:
        max_row_demand = max((d, i) for i, d in enumerate(demands_rows) if d > 0)
        max_col_demand = max((d, i) for i, d in enumerate(demands_cols) if d > 0)

        if max_row_demand[0] >= max_col_demand[0]:
            index = max_row_demand[1]
            is_row = True
        else:
            index = max_col_demand[1]
            is_row = False

        placed = False
        for ship in ships:
            if ship <= (demands_rows[index] if is_row else demands_cols[index]):

                if is_row:
                    for col in range(m - ship + 1):
                        if is_valid_placement(
                            board, index, col, ship, True, demands_rows,
    demands_cols
                        ):
                            place_ship(board, index, col, ship, True)
                            demands_rows[index] -= ship
                            for c in range(col, col + ship):
                                demands_cols[c] -= 1
                            ships.remove(ship)
                            placed = True
                            break
                else:
                    for row in range(n - ship + 1):
                        if is_valid_placement(
                            board, row, index, ship, False, demands_rows,
    demands_cols
                        ):
                            place_ship(board, row, index, ship, False)
                            demands_cols[index] -= ship
                            for r in range(row, row + ship):
                                demands_rows[r] -= 1
                            ships.remove(ship)
                            placed = True
```

```
                                break
            if placed:
                break

        if not placed:
            break

    return board
```

Listing 3: Algoritmo de Aproximación para la Batalla Naval

Análisis de Complejidad y Cota de Aproximación La complejidad de este algoritmo depende de la cantidad de barcos y la cantidad de filas y columnas del tablero. En el peor caso, el algoritmo intentará colocar cada barco en cada fila o columna, lo cual resulta en una complejidad de O(n * m * k), donde n y m son las dimensiones del tablero y k es la cantidad de barcos. En la práctica, la complejidad que se manifiesta es mucho menor, ya que el algoritmo no intentará colocar cada barco en cada fila o columna, sino que intentará colocar los barcos en las filas o columnas con mayor demanda. Por lo tanto, la complejidad real del algoritmo es mucho menor que O(n * m * k), pero no se puede determinar con exactitud.

Se realizaron distintas pruebas con el algoritmo para evaluar su desempeño y establecer una cota que permita analizar la calidad de la aproximación.

A partir de los datos recopilados, se concluyó que la cota de aproximación es de **1.6**.

Tests	15_30_10	18_28_10	20_12_18	22_25_17	25_20_15	26_30_10	30_15_10
Óptimo (*z(I)*)	346	308	238	180	225	424	342
Obtenido (*A(I)*)	380	340	306	230	359	468	388
Diferencia	34	32	68	50	134	44	46
Coeficiente	1595 ≈ 1.6						

	Tests	15_30_10	18_28_10	20_12_18	22_25_17	25_20_15	26_30_10	30_15_10
Backtracking	Resultado	346	308	238	180	225	424	342
	Tiempo	0.017787s	0.014263s	0.009359s	258.474202s	0.054013s	0.035311s	0.195845s
Greedy	Resultado	380	340	306	230	359	468	388
	Tiempo	0.000596s	0.000510s	0.001398s	0.008107s	0.003883s	0.002133s	0.000346s
	Diferencia	34	32	68	50	134	44	46
	Coeficiente	1595 ≈ 1.6						

Figura 9: Comparacion de Tiempos de Ejecución entre Algoritmo de Aproximación y Greedy

Como vemos, un test que a Backtracking le toma más de 4 minutos en ejecutar, se ejecuta en 8 milisegundos con el algoritmo de aproximación.

1.3.7. Algoritmo de Greedy propuesto

Para lograr una aproximación al óptimo con una complejidad polinomial, implementamos un algoritmo Greedy para resolver el problema de la Batalla Naval. Este algoritmo trabaja con tres listas: una con las demandas de filas, otra con las demandas de columnas y una más con los barcos. Su objetivo es minimizar la demanda incumplida de forma aproximada respecto a la solución óptima.

El enfoque comienza ordenando los barcos de mayor a menor tamaño. Luego, itera sobre cada barco y recorre el tablero buscando la posición con menor demanda en la que se pueda colocar. Una vez encontrada, coloca el barco en esa posición.

La demanda de cada posición se calcula según la orientación del barco. Para la orientación horizontal, se suma la demanda de las columnas ocupadas por el barco más la demanda de la fila correspondiente. En el caso de la orientación vertical, se calcula sumando las demandas de las filas ocupadas más la demanda de la columna donde se colocará.

La idea de ubicar cada barco en la posición con menor demanda incumplida tiene como objetivo que cada barco ocupe el espacio más justo posible. Esto puede ahorrar espacio en el tablero que podría ser aprovechado por los barcos restantes, intentando optimizar así la distribución general.

Análisis Cota de Algoritmo Greedy Se hicieron distintas pruebas con el algoritmo para determinar una cota y poder analizar que tan buena es la aproximación. Algunas resultaron en la solución óptima y otras, se alejaron un poco del resultado.

En base a los datos tomados, podemos determinar que la cota es de **1.4**.

Tests	15_30_10	18_28_10	20_12_18	22_25_17	25_20_15	26_30_10	30_15_10
Óptimo ($z(I)$)	346	308	238	180	225	424	342
Obtenido ($A(I)$)	346	308	238	240	251	424	362
Diferencia	0	0	0	60	26	0	20
Coeficiente	1.333 ≈ 1.4						

1.3.8. Conclusiones

El problema de la Batalla Naval, como otros problemas complejos, tiene varios desafíos cuando buscamos la mejor solución. En este trabajo, se propusieron diferentes métodos para resolverlo, cada uno con sus ventajas y limitaciones.

El algoritmo de backtracking resultó útil para explorar distintas formas de colocar los barcos, aprovechando las restricciones para reducir el número de opciones a revisar. Aunque este método da la solución óptima, es un poco lento por la cantidad de posibilidades que hay que evaluar.

El modelo de programación lineal nos permitió encontrar la mejor disposición de los barcos con seguridad, pero también es un proceso que puede tomar mucho tiempo cuando el tablero y el número de barcos son grandes. Aunque ofrece una solución precisa, puede ser costoso en términos de tiempo de cálculo.

El enfoque de aproximación propuesto por John Jellicoe es más rápido y sencillo, pero no garantiza la mejor solución posible. Es útil cuando no se necesita una precisión exacta y cuando el tiempo es un factor importante. Este método es adecuado cuando queremos una solución rápida sin preocuparnos tanto por la exactitud.

Por último, el algoritmo greedy también ofrece una forma rápida de encontrar una solución y obtiene resultados más precisos que el algoritmo de aproximación propuesto por John Jellicoe.

En resumen, cada método tiene su lugar dependiendo de lo que necesitamos. Si lo más importante es la precisión y el tiempo no es un problema, los métodos de backtracking o programación lineal pueden ser mejores. Si necesitamos algo rápido y aproximado, los métodos de aproximación o greedy pueden ser más adecuados. La elección depende de lo que prioricemos entre precisión y tiempo de ejecución.