Teoría de Algoritmos
(75.29) Curso Asincronico 2024

Trabajo Práctico: Juegos de Hermanos

2 de diciembre de 2024

Axel Alvarado
107679

Axel Apaza
105006

Facundo Fernández
109097

Gerónimo Fanti
109712

Gonzalo Oesterheld
110554

Primera parte: Introducción y primeros años

1. Análisis del problema - Propuesta de Algortimo Greedy

Se tiene una fila de n monedas y se conocen sus valores. Sophia debe elegir, para ella y para su hermano Mateo, la primera o la última de la fila en cada turno de modo tal que se asegure que sus monedas suman más que las de Mateo. Para resolver este problema dándole un enfoque greedy, se debe buscar el óptimo local dados los valores de las monedas que se encuentren en los extremos al momento de elegir. Para esto, en cada turno Sophia elige la moneda con el mayor valor entre las dos opciones posibles y como también está jugando por Mateo, elige la moneda de menor valor para él.

2. Algoritmo Greedy

2.1. Código

A continuación se muestra el código de solución Greedy del problema.

```
from collections import deque

ELECCION_ULTIMA_MONEDA = 'Ultima moneda para '
ELECCION_PRIMERA_MONEDA = 'Primera moneda para '
PRIMERA = True
ULTIMA = False
SOPHIA = 'Sophia'
MATEO = 'Mateo'

def elegir_moneda(elecciones, monedas, primera, nombre_hermano):
    if primera:
        elecciones.append(ELECCION_PRIMERA_MONEDA + nombre_hermano)
        monedas.popleft()
    else:
        elecciones.append(ELECCION_ULTIMA_MONEDA + nombre_hermano)
        monedas.pop()

def elecciones_sophia(monedas):
    elecciones = []
    turno_sophia = True

    while monedas:
        if turno_sophia:
            elegir_moneda(elecciones, monedas, PRIMERA, SOPHIA) if monedas[0] >
    monedas[-1] else elegir_moneda(elecciones, monedas, ULTIMA, SOPHIA)

        else:
        elegir_moneda(elecciones, monedas, PRIMERA, MATEO) if monedas[0] < monedas
    [-1] else elegir_moneda(elecciones, monedas, ULTIMA, MATEO)

        turno_sophia = not turno_sophia

    return elecciones
```

Dado los n valores de las monedas, se devuelve una lista de strings que representa las elecciones de monedas que Sophia debe ir haciendo para ganar el juego.

2.2. Análisis de complejidad

El algoritmo propuesto tiene una complejidad temporal de $\mathcal{O}(n)$, ya que en cada turno se toma una moneda de uno de los extremos de la fila, y el número total de monedas es n. En cada iteración del ciclo principal, se comparan las monedas en los extremos y se reduce el número de monedas restantes en uno, esto constituye una accion de costo $\mathcal{O}(1)$, y el ciclo se repite hasta que

no queden monedas. Cabe aclarar que se utiliza una deque de python para que quitar una moneda de la primera posición sea una operación $\mathcal{O}(1)$ utilizando popleft() (si se usaba una lista, hacer monedas.pop(0) tendría complejidad $\mathcal{O}(n)$ y haría que la complejidad resultante del algoritmo sea $\mathcal{O}(n^2)$).

2.3. Ejemplo de ejecución

Fila de monedas: [3, 9, 1, 2, 8, 5]

- En el primer turno, Sophia elige entre 3 y 5. Elige 5.
- En el turno de Mateo, Sophia elige entre 3 y 8. Elige 3 para Mateo.
- Sophia elige entre 9 y 8. Elige 9.
- Sophia elige entre 1 y 8 para Mateo. Elige 1 para Mateo.
- Sophia toma la moneda 8.
- Mateo toma la última moneda, que es 2.

Resultado final:

$$\text{Sophia} : 5 + 9 + 8 = 22$$
$$\text{Mateo} : 3 + 1 + 2 = 6$$

Sophia gana, como era de esperar.

2.4. Efecto de la variabilidad de los valores

La variabilidad de los valores de las monedas no afecta la complejidad del algoritmo, ya que la cantidad de monedas es el factor determinante del tiempo de ejecución. En términos de optimalidad, los valores de las monedas únicamente afectarían en el caso de que las n monedas sean iguales, haciendo que Sophia no pueda ganar y el juego termine empatado. Caso contrario, Sophia siempre ganará el juego siguiendo la regla greedy propuesta, como se demostrará a continuación.

3. Demostración de optimalidad del algoritmo

Sea s_i el valor de la elección de Sophia para ella misma en el turno i, m_i el valor de la elección de Sophia para Mateo en el turno i y C_j el valor de la moneda en la posición j de la fila de monedas.

Se desmostrará por inducción que $S(k) \geq M(k)\ \forall k \in \mathbb{N},\ k \leq n$, siendo:

$$S(k) = \sum_{i=1}^{k} s_i \quad \text{y} \quad M(k) = \sum_{i=1}^{k} m_i$$

k: número de turnos (elecciones de monedas) jugados del juego. Esto significa que m_i valdrá 0 en turnos impares (porque Sophia siempre empieza eligiendo) y s_i valdrá 0 en turnos pares.

Caso base ($k = 1$):

$s_1 = max\{C_1, C_n\}$ y $m_1 = 0 \implies S(1) = s_1 > m_1 = M(1) \implies$ se cumple que $S(1) \geq M(1)$.

Paso inductivo:

Sup. que $\forall l,\ 1 \leq l \leq h,\ S(l) \geq M(l)$. Se quiere ver que $S(h+1) \geq M(h+1)$.

Observación: La fila de monedas restante luego de h elecciones tiene longitud $n - h$, por lo que $h + 1$ puede ser menor o igual a n.

Sup. que en este paso se tiene una fila reducida de monedas que va desde la moneda en la posición r de la fila de monedas original hasta la moneda en la posición t de la fila de monedas original (es decir, C_r es el valor de la primera moneda en la fila actual y C_t el valor de la última moneda).

- Si $h + 1$ es impar, se tiene:

  Si $h + 1 = n$, $s_{h+1} = C_r = C_t > 0$. Si no, $s_{h+1} = max\{C_r, C_t\} > 0$

  $m_{h+1} = 0$

  $\implies s_{h+1} > m_{h+1}$.

  Por otro lado: $S(h + 1) = S(h) + s_{h+1}$, $M(h + 1) = M(h) + m_{h+1}$.

  Como $S(h) \geq M(h)$ por hipótesis inductiva, $S(h + 1) > M(h + 1)$ para $h + 1$ impar.

  (observación: si se tiene un número n impar de monedas, Sophia siempre ganará el juego, no habrá chances de empate).

- Si $h + 1$ es par, se tiene:

  Si $h + 1 = n$, $m_{h+1} = C_r = C_t$. Si no, $m_{h+1} = min\{C_r, C_t\}$

  $s_{h+1} = 0$.

  $m_h = 0$

  s_h puede ser $C_{r-1} = max\{C_{r-1}, C_t\}$ o $C_{t+1} = max\{C_r, C_{t+1}\}$, según como hayan sido las elecciones en los turnos previos del juego.

  De este modo, se tienen cuatro combinaciones posibles:

  - $m_{h+1} = C_r$ y $s_h = C_{r-1} \implies m_{h+1} = C_r \leq C_t \leq C_{r-1} = s_h$
  - $m_{h+1} = C_r$ y $s_h = C_{t+1} \implies m_{h+1} = C_r \leq C_{t+1} = s_h$
  - $m_{h+1} = C_t$ y $s_h = C_{r-1} \implies m_{h+1} = C_t \leq C_{r-1} = s_h$
  - $m_{h+1} = C_t$ y $s_h = C_{t+1} \implies m_{h+1} = C_t \leq C_r \leq C_{t+1} = s_h$

  $S(h + 1) = S(h - 1) + s_h + s_{h+1}$, $M(h + 1) = M(h - 1) + m_h + m_{h+1}$.

  En los cuatro casos, se da que $m_{h+1} \leq s_h$.

  Se llega entonces a que:

  $S(h + 1) = S(h - 1) + s_h + 0$ y $M(h + 1) = M(h - 1) + 0 + m_{h+1}$

  con $S(h - 1) \geq M(h - 1)$ (por hipótesis inductiva) y $s_h \geq m_{h+1}$

  $\implies S(h + 1) \geq M(h + 1)$ para h+1 par.

Luego, $S(k) \geq M(k)$ $\forall k \in \mathbb{N}$, con $k \leq n$. Es decir, la sumatoria de los valores de las monedas de Sophia es mayor o igual que la de Mateo al finalizar cualquier turno del juego. En particular, es mayor o igual al finalizar el último turno, es decir que el juego sólo puede terminar con Sophia como ganadora o con empate.

Observación: la igualdad entre m_{h+1} y s_h se da sólo en caso que $min\{C_r, C_t\} = C_r = C_t$ y que $max\{C_{r-1}, C_t\} = C_{r-1} = C_t$ o $max\{C_r, C_{t+1}\} = C_{t+1} = C_r$. Se puede concluir que Sophia sólo irá empatando con Mateo cuando todas las monedas elegidas hayan sido iguales. En particular, Mateo y Sophia sólo empatarán el juego cuando las n monedas sean iguales.

Consecuencias de la demostración:

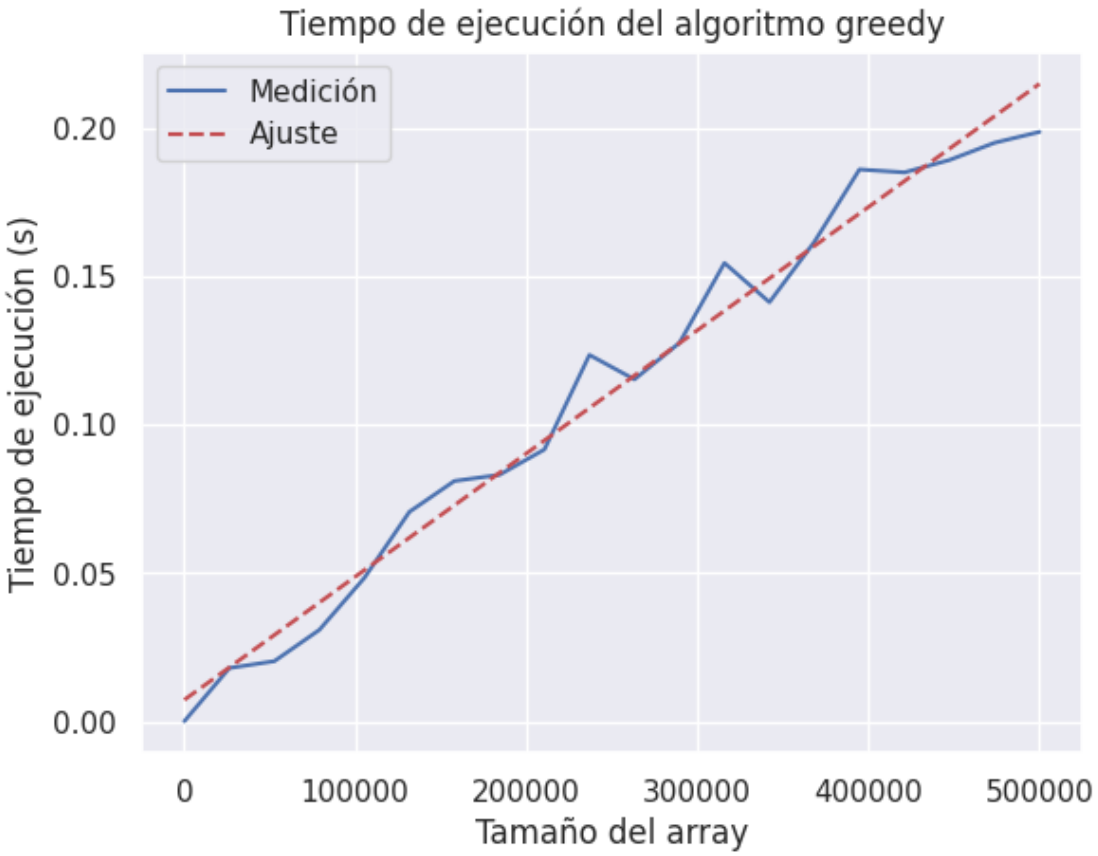
Notar que se demostró que Sophia siempre gana o empata el juego siguiendo el algoritmo propuesto.

Como se observó durante la demostració, Sophia siempre gana cuando la cantidad de monedas es impar, ya que aunque todas las monedas sean iguales, será la que elija última y se asegurará la victoria. Y para el caso en que la cantidad de monedas sea par, el empate sólo puede darse cuando en cada turno de Mateo la moneda que elige es igual a la moneda que eligió Sophia en el turno anterior. Esto sucede únicamente cuando todas las monedas son iguales. Si no, en algún turno par Mateo elegirá una moneda menor y a partir de ahí Sophia siempre mantendrá la ventaja aunque luego sólo haya empates nuevamente.

4. Mediciones

Se comprobará la complejidad teórica indicada mediante mediciones de tiempos ejecutando el algoritmo con distintas entradas.

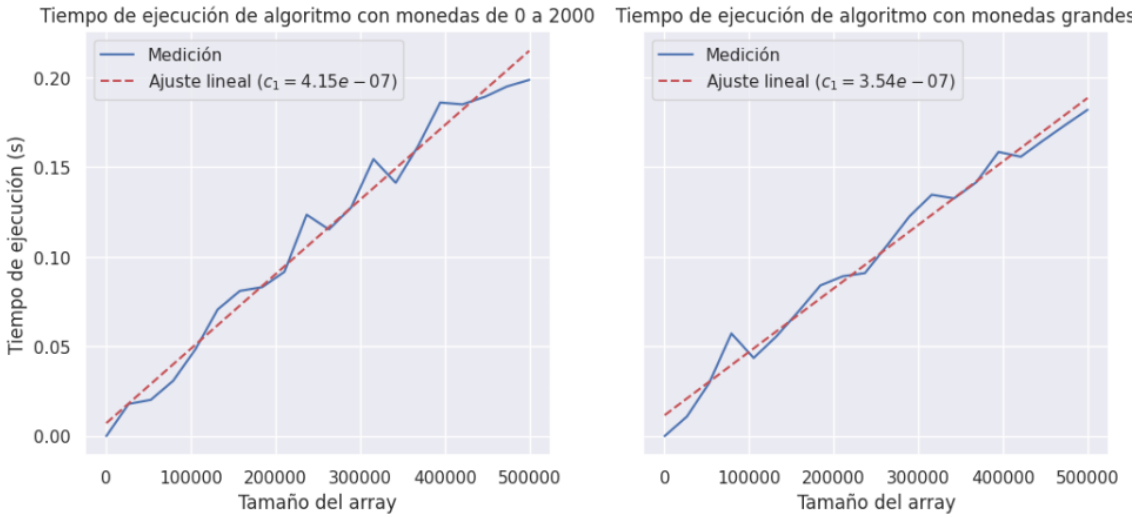
Midiendo los tiempos de ejecución con arreglos de monedas de valores entre 0 y 2000, se obtuvo lo siguiente:



Y se obtienen los errores, para cada tamaño:

Puede verse que para todos los tamaños de entrada el error del ajuste es considerablemente pequeño, lo cual nos lleva a pensar que efectivamente el algoritmo propuesto se comporta como $O(n)$.

Para verificar que la variabilidad de los valores no afecta la complejidad del algoritmo, se hicieron mediciones con tamaños muy grandes de monedas (entre 1000000000 y 1000002000), y se obtuvieron tiempos muy similares a los medidos anteriormente.



5. Conclusiones

El algoritmo greedy garantiza que Sophia siempre gana o empata el juego, teniendo en cuenta lo siguiente:

- 1. Optimalidad: Sophia maximiza su suma al elegir siempre la mejor moneda disponible, asegurando ventaja sobre Mateo.
- 2. Eficiencia: La complejidad es O(n), gracias al uso de una deque para operaciones rápidas en los extremos. Pudo verificarse mediante mediciones de tiempos que la complejidad efectivamente es O(n)
- 3. Empates: Solo ocurren si todas las monedas tienen el mismo valor, destacando la efectividad del enfoque.
- 4. Ventaja de Sophia: Sophia siempre gana cuando hay un número impar de monedas.
- 5. Generalidad: El algoritmo funciona óptimamente para cualquier distribución de valores en las monedas.
- 6. Flojeza de papeles: Sophia le está haciendo trampa a Mateo, eligiendo para él siempre la peor opción posible. Más adelante en la vida de los hermanos, Mateo comenzará a jugar...

6. Consideraciones

El algoritmo descripto en la sección Algoritmo Greedy es utilizado en nuesto programa (dentro del repositorio de github) del siguiente modo:

Está presente en el archivo juego_monedas.py, que tiene un main que recibe un archivo por parametro. El archivo debe ser un .txt con formato como en el siguiente ejemplo:

Primera linea: # Los valores de las monedas de la fila se muestran tal cual su orden correspondiente, separados por ;

Segunda linea: 3;9;1;2;8;5

De este modo, ubicados en la carpeta correspondiente a esta sección del trabajo (tp1) y corriendo por terminal el comando 'python3 juego_monedas.py archivo_monedas.txt' se simula un juego con las monedas presentes en el archivo de texto y como salida se guarda en una carpeta aparte otro archivo llamado 'resultados_archivo_monedas.txt' en donde puede verse la ganancia de Sophia y Mateo en el juego, y las elecciones que tuvo que hacer Sophia para llegar a eso.

Además, hay un archivo llamado 'tests.py' que simula juegos de monedas con distintos archivos .txt y testea que Sophia sea la ganadora (o haya empate en el caso de todas las monedas iguales) y que las ganancias de monedas sean las esperadas si se siguió el algoritmo propuesto. Entre los archivos para los que se testea se encuentran los provistos por la cátedra.

Segunda parte: Mateo empieza a Jugar

1. Análisis del problema

1.1. Planteo de la ecuación de recurrencia

Mateo aprendió sobre algoritmos greedy y comenzó a jugar, causando que Sophia no gane en el juego de monedas cada vez. Pero ella comenzó a aprender sobre programación dinámica y quiere asegurarse de obtener el máximo valor acumulado posible para tener más chances de ganar, sabiendo que ella siempre empieza eligiendo y que Mateo siempre eligirá la moneda más grande entre las dos posibles.

Dada una secuencia de valores de monedas $m_1, m_2, ..., m_n$, se busca que Sophia obtenga el máximo valor acumulado posible. Sea k el tamaño de una secuencia de valores de monedas contenida en la original, sea i el índice del valor de la primera moneda en esa secuencia y sea $OPT(k, i)$ el máximo valor acumulado posible para Sophia dada esa secuencia, entonces lo que se quiere obtener es $OPT(n, 1)$ (una secuencia que empieza por m_1 de tamaño n puede ser únicamente la secuencia de monedas original). Puede verse que $1 \leq k \leq n$ (los tamaños posibles de secuencias contenidas en la original) y que, por ende, $1 \leq i \leq n - k + 1$. Además, toda secuencia de monedas de tamaño k que empiece por m_i es de la forma $m_i, m_{i+1}, ..., m_{k+i-1}$

Si se tiene una única moneda, el máximo valor acumulado posible será el valor de esa moneda. En otras palabras, se tiene que $OPT(1, i) = m_i \ \forall \ 1 \leq i \leq n$.

Si se tienen dos monedas, el máximo valor acumulado posible será el de la moneda de moayor valor entre las dos. En otras palabras, se tiene que $OPT(2, i) = max(m_i, m_{i+1}) \ \forall \ 1 \leq i \leq n - 1$.

Para poder plantear una ecuación de recurrencia para llegar a la solución óptima buscada, vamos a suponer que al momento de elegir una moneda dada una secuencia de valores de tamaño k que empieza por m_i, Sophia conoce el máximo valor acumulado posible para todos los juegos de monedas con secuencia de valores de tamaño menor a k (empezando por cualquier moneda posible). Sophia debe elegir la moneda i o la moneda $k + i - 1$, según cuál sea el máximo entre:

- m_i más la suma máxima posible que puede obtener Sophia en un juego de monedas, dada una secuencia de monedas resultante luego de la elección de Sophia y la de Mateo.
- m_{k+i-1} más la suma máxima posible que puede obtener Sophia en un juego de monedas, dada una secuencia de monedas resultante luego de la elección de Sophia y la de Mateo.

Si Sophia elige la moneda i, la secuencia de monedas resultante será $m_{i+1}, ..., m_{k+i-1}$.

Si elige la moneda $k + i - 1$, la secuencia resultante será $m_i, m_{i+1}, ..., m_{k+i-2}$.

Para considerar las elecciones de Mateo, que siempre tomará la moneda de mayor valor que haya en los extremos, se define la siguiente función:

$$f_m(i, j) = \begin{cases} i + 1 & \text{si } m_i \geq m_j \\ i & \text{si } m_i < m_j \end{cases} \tag{1}$$

$$Dom(f_m) = \{(i, j) \in \mathbb{N}^2 \mid 1 \leq i < j \leq n\}$$

En donde los valores que recibe la función son los índices de los extremos de la secuencia de monedas de la que debe elegir Mateo, y el que devuelve es el índice de la primera moneda en la secuencia resultante luego de su elección (si la primera moneda es mayor que la última, la eligirá haciendo que a Sophia le quede como primera moneda la $i + 1$; sino elige la última y la primera moneda sigue siendo la misma). No se considera el caso en que Mateo elige para $i = j$ ya que representa cuando elige en el último turno, cosa que no la importa a Sophia porque ya no le quedan turnos.

Teniendo en cuenta todo lo dicho previamente, se define la siguiente ecuación de recurrencia:

$OPT(k,i) = max(m_i + OPT(k-2, f_m(i+1, k+i-1)), m_{k+i-1} + OPT(k-2, f_m(i, k+i-2)))$

$\forall\ 3 \leq k \leq n, \forall\ 1 \leq i \leq n-k+1$

Sophia elige la moneda según el máximo entre las dos opciones mostradas con anterioridad. En ambas se habla de una secuencia de monedas resultante luego de la elección de Sophia y Mateo. Esta secuencia resultante tendrá tamaño k-2, ya que cuando Sophia toma una moneda el tamaño de la fila se reduce en uno y cuando Mateo toma otra el tamaño vuelve a reducirse en uno. Y luego se debe buscar en qué moneda empezará esa secuencia de tamaño k-2: si Sophia elige la moneda m_i, Mateo debe elegir entre m_{i+1} y m_{k+i-1}, es decir que la primera moneda que quede luego de las elecciones de Sophia y Mateo estará determinada por el valor que devuelva $f_m(m_{i+1}, m_{k+i-1})$. Si Sophia elige la moneda m_{k+i-1}, Mateo debe elegir entre m_i y m_{k+i-2} y entonces la primera moneda que quede luego de las elecciones de los hermanos será la de índice $f_m(m_i, m_{k+i-2})$.

Obs 1: Cuando se tiene una cantidad n par de monedas, a Sophia sólo le tocará en turnos para los que la fila de monedas sea de tamaño par. Lo mismo sucederá para una cantidad n impar, siempre jugará con filas de monedas de tamaño impar. Es por esto que sólo será necesario calcular los óptimos para k pares cuando n sea par y sólo para k impares cuando n sea impar. Esto se utilizará en el algoritmo para optimizar el uso de espacio.

Obs 2: Notar que para $k = 1$ habrá n óptimos, para $k = 2$ habrá n-1, y así sucesivamente hasta que para $k = n$ haya un único óptimo que es el máximo valor que Sophia puede acumular en el juego.

2. Demostración de optimalidad

Se demostrará por inducción que la ecuación de recurrencia nos lleva a obtener el máximo valor acumulado posible para Sophia. Aclaración: En la sección anterior se explica cómo se llega a la ecuación de recurrencia, y esos argumentos se verán repetidos para la demostración formal.

Se demostrará que $OPT(k,i)$ es el máximo valor acumulado posible para cualquier secuencia de monedas de tamaño k que empieza por m_i $(m_i, m_{i+1}, ..., m_{i+k-1})$.

Casos base ($k = 1$ y $k = 2$)

$OPT(1,i) = m_i\ \forall 1 \leq i \leq n$. Es el máximo valor acumulado posible para cualquier secuencia de monedas de tamaño 1 al tener una única elección posible.

$OPT(2,i) = max(m_i, m_{i+1})\ \forall 1 \leq i \leq n-1$. Es el máximo valor acumulado posible para cualquier secuencia de monedas de tamaño 2 que empieza por m_i, al elegir la moneda de mayor valor posible entre las dos.

Paso inductivo:

Sup que $\forall 1 \leq l \leq h$, $OPT(l,i)$ es el máximo valor acumulado posible para cualquier secuencia de monedas de tamaño l que empieza por m_i. Se quiere ver que $OPT(h+1,i)$ es el máximo valor acumulado posible para cualquier secuencia de monedas de tamaño $h+1$ que empieza por m_i.

$OPT(h+1,i) = max(m_i + OPT((h+1)-2, f_m(i+1, (h+1)+i-1)), m_{(h+1)+i-1} + OPT((h+1)-2, f_m(i, (h+1)+i-2)))$

$\forall\ 3 \leq h+1 \leq n, \forall\ 1 \leq i \leq n-(h+1)+1$

$\rightarrow OPT(h+1,i) = max(m_i + OPT(h-1, f_m(i+1, h+i)), m_{h+i} + OPT(h-1, f_m(i, h+i-1)))$

En la última ecuación puede verse que $OPT(h+1,i)$ es el máximo entre m_i más un optimo de una fila de monedas de tamaño $h-1$ (que es óptimo por hipótesis inductiva) y m_{h+i} más un optimo de una fila de monedas de tamaño $h-1$ (¡que también es óptimo!). El valor que devuelva f_m no es relevante ya que sólo da el índice de una moneda, y por hipótesis sabemos el máximo acumulado posible por tamaño de la secuencia de monedas, cualquiera sea la primera moneda. Como se elige el máximo considerando el óptimo según cada elección posible para Sophia, $OPT(h+1,i)$ también será óptimo.

Luego, $OPT(k, i)$ es el máximo valor acumulado posible para cualquier secuencia de monedas de tamaño k que empieza por m_i. En particular, $OPT(n, 1)$ es la máximo valor acumulado posible en un juego de n monedas.

3. Algoritmo

3.1. Código

```
1
2  def primera_moneda_tras_eleccion_mateo(monedas, i, j):
3      if monedas[i] >= monedas[j]:
4          return i+1
5      else:
6          return i
7
8  def max_acumulados_sophia_desde(monedas, n, max_acumulados, k_inicial):
9      for k_actual in range(k_inicial, n+1, 2):
10         max_acumulados_k_actual = []
11         for i in range(n - k_actual + 1):
12             j = i + k_actual - 1
13
14             sig_i_eligiendo_i = primera_moneda_tras_eleccion_mateo(monedas, i+1, j)
15             max_acumulado_eligiendo_i = monedas[i] + max_acumulados[-1][
       sig_i_eligiendo_i]
16             sig_i_eligiendo_j = primera_moneda_tras_eleccion_mateo(monedas, i, j-1)
17             max_acumulado_eligiendo_j = monedas[j] + max_acumulados-1][
       sig_i_eligiendo_j]
18
19             max_acumulados_k_actual.append(max(max_acumulado_eligiendo_i,
       max_acumulado_eligiendo_j))
20         max_acumulados.append(max_acumulados_k_actual)
21
22 def max_acumulados_sophia_n_par(monedas, n, max_acumulados):
23     max_acumulados.append([monedas[i] if monedas[i] >= monedas[i+1] else monedas[i
       +1] for i in range(n-1)]) # k = 2
24     k_inicial = 4
25     max_acumulados_sophia_desde(monedas, n, max_acumulados, k_inicial)
26     return max_acumulados
27
28 def max_acumulados_sophia_n_impar(monedas, n, max_acumulados):
29     max_acumulados.append([monedas[i] for i in range(n)]) # k = 1
30     k_inicial = 3
31     max_acumulados_sophia_desde(monedas, n, max_acumulados, k_inicial)
32     return max_acumulados
33
34 def max_acumulados_sophia(monedas):
35     n = len(monedas)
36     max_acumulados = []
37     if n % 2 == 0:
38         return max_acumulados_sophia_n_par(monedas, n, max_acumulados)
39     else:
40         return max_acumulados_sophia_n_impar(monedas, n, max_acumulados)
```

Los óptimos se van guardando en una lista llamada max_acumulados.

Se tiene una función para cuando la cantidad de monedas es par y otra para una cantidad impar de monedas. En la de cantidad impar, se guarda en la primera posición de max_acumulados una lista con los valores de cada moneda, cubriendo el caso $OPT(1, i) = m_i \ \forall \ 1 \leq i \leq n$. En la de cantidad par, se guarda una lista con los valores de la moneda más grande entre cada par de moneras consecutivas posible, cubriendo el caso $OPT(2, i) = max(m_i, m_{i+1}) \ \forall \ 1 \leq i \leq n - 1$.

Luego, se sigue por el k que corresponda en cada caso: para n par, se sigue por $k = 4$ y para n impar, por $k = 3$.

En ambas se llama a una función (max_acumulados_sophia_desde) que aplicará la ecuación de recurrencia e irá completando para cada k que corresponda (recorre los k de 2 en 2 hasta n y

agrega a max_acumulados el óptimo por cada i posible). No se usa max_acumulados[k-2][i] porque se está avanzando de a dos en k_actual y no sería correcto acceder a esa posición, sino que hay que buscar para el último k que se consideró.

```
1
2  SOPHIA = 'Sophia'
3  MATEO = 'Mateo'
4  PRIMERA = True
5  ULTIMA = False
6  ELECCION_ULTIMA_MONEDA = ' agarra la ultima'
7  ELECCION_PRIMERA_MONEDA = ' agarra la primera'
8
9  def elegir_moneda(elecciones, i, j, primera, nombre_hermano):
10     if primera:
11         elecciones.append(nombre_hermano + ELECCION_PRIMERA_MONEDA)
12         i += 1
13     else:
14         elecciones.append(nombre_hermano + ELECCION_ULTIMA_MONEDA)
15         j -= 1
16     return i, j
17
18 def elegir_mejor_moneda(elecciones, monedas, i, j, nombre_hermano):
19     if monedas[i] >= monedas[j]:
20         return elegir_moneda(elecciones, i, j, PRIMERA, nombre_hermano)
21     else:
22         return elegir_moneda(elecciones,i, j, ULTIMA, nombre_hermano)
23
24 def recuperar_elecciones(monedas, max_ganancias_sophia):
25     elecciones = []
26     turno_sophia = True
27
28     i = 0
29     j = len(monedas) - 1
30     tam = len(max_ganancias_sophia) - 1
31     while i <= j:
32         if turno_sophia:
33             if tam == 0:
34                 i, j = elegir_mejor_moneda(elecciones, monedas, i, j, SOPHIA)
35             elif monedas[i] + max_ganancias_sophia[tam-1][
       primera_moneda_tras_eleccion_mateo(monedas, i+1, j)] == max_ganancias_sophia[
       tam][i]:
36                 i, j = elegir_moneda(elecciones, i, j, PRIMERA, SOPHIA)
37             else:
38                 i, j = elegir_moneda(elecciones, i, j, ULTIMA, SOPHIA)
39             tam -= 1
40         else:
41             i, j = elegir_mejor_moneda(elecciones, monedas, i, j, MATEO)
42         turno_sophia = not turno_sophia
43
44     return elecciones
45
46 def elecciones_hermanos(monedas):
47
48     max_ganancias_sophia = max_acumulados_sophia(monedas)
49     return recuperar_elecciones(monedas, max_ganancias_sophia)
```

Finalmente, se recuperan las elecciones usando los óptimos obtenidos y la lista de monedas original. Obviamente, se consideran también las elecciones de Mateo.

3.2. Análisis de complejidad

1. Cálculo de los máximos acumulados (función `max_acumulados_sophia`)

La función sigue un enfoque iterativo para calcular los valores óptimos en cada subproblema. Su análisis se divide en:

- **Generación de subproblemas base ($k = 1$ o $k = 2$):** Se guardan los óptimos para los

subproblemas de tamaño 1 o 2, que serán n o $n/2$ óptimos según el caso.

- **Generación de subproblemas por tamaños (k):** Se recorren todos los tamaños posibles de subproblemas desde $k = 4$ o $k = 3$ (según n sea par o impar) hasta $k = n$. Esto implica apróximadamente $n/2$ iteraciones, ya que se avanza de a dos.
- **Cálculo para cada tamaño k:** Para cada k, se procesan todas las posibles secuencias de monedas de longitud k, cuyo número es $n - k + 1$. En cada cálculo, se realizan dos accesos a valores previos de OPT para determinar el valor máximo.

Por lo tanto, la complejidad de esta parte:

$$T(n) = \mathcal{O}(n) \sum_{k=2,4,\ldots,n} \mathcal{O}(n - k + 1) \approx \mathcal{O}(n^2)$$

2. Reconstrucción de las elecciones (función `recuperar_elecciones`)

El proceso de reconstrucción recorre las monedas una vez mientras evalúa las elecciones de Sophia y Mateo. Esto tiene una complejidad lineal:

$$\mathcal{O}(n)$$

Complejidad Total

La complejidad total está dominada por el cálculo de los máximos acumulados, ya que es la parte más costosa. Por lo tanto, la complejidad general del algoritmo es:

$$\mathcal{O}(n^2)$$

Espacio

El espacio requerido se debe principalmente a la matriz de resultados intermedios `max_acumulados`:

- En el peor caso, esta matriz almacena $n/2$ listas de tamaño decreciente desde n hasta 1. Esto suma un total de:
$$\frac{n}{2} \cdot n \approx \mathcal{O}(n^2)$$

Por lo tanto, tanto el tiempo como el espacio son cuadráticos en el tamaño de la entrada n.

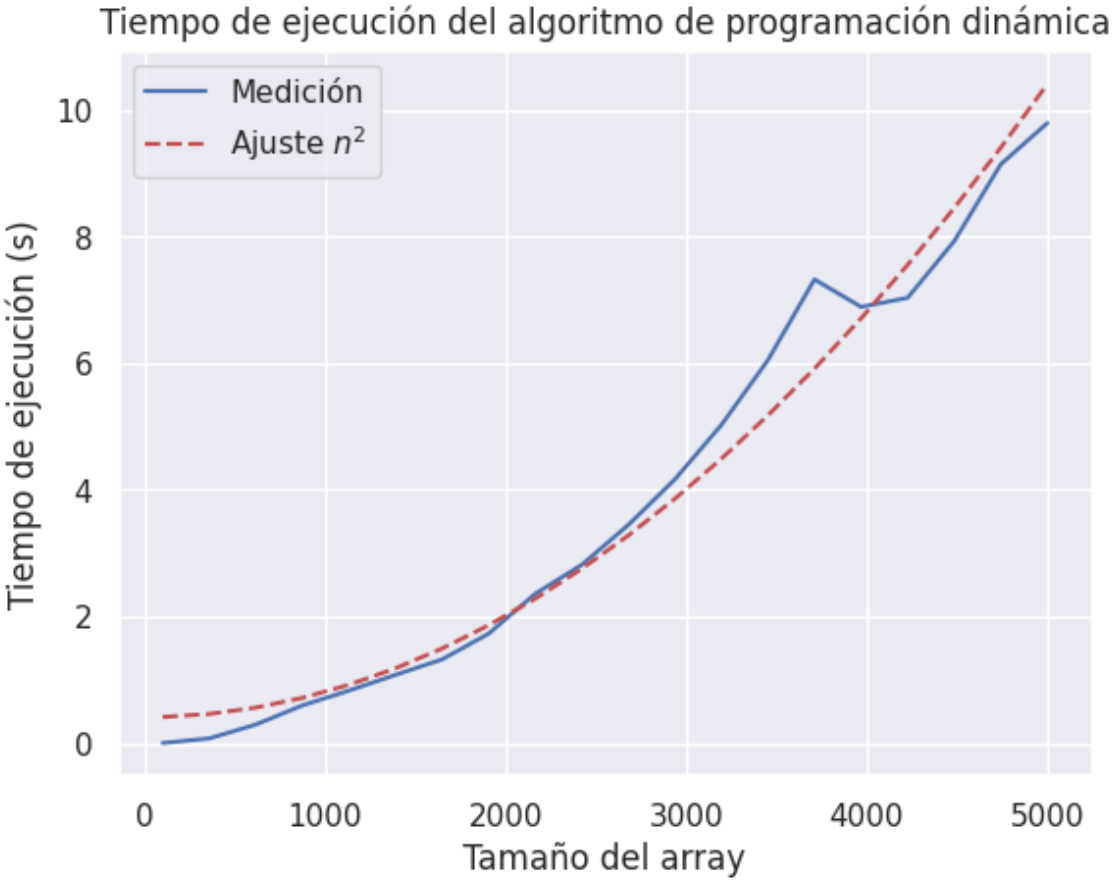
3.3. Efecto de la variabilidad de los valores

De manera similar a como ocurre en la parte 1, la variabilidad de los valores de las monedas no afecta la complejidad del algoritmo, ya que la cantidad de monedas es el factor determinante del tiempo de ejecución. En términos de optimalidad, se demostró que Sophia siempre obtiene la ganancia óptima a pesar que no siempre puede ganar el juego. Que gane el juego o no depende mucho de los valores de las monedas y la distribución dentro de las fila.

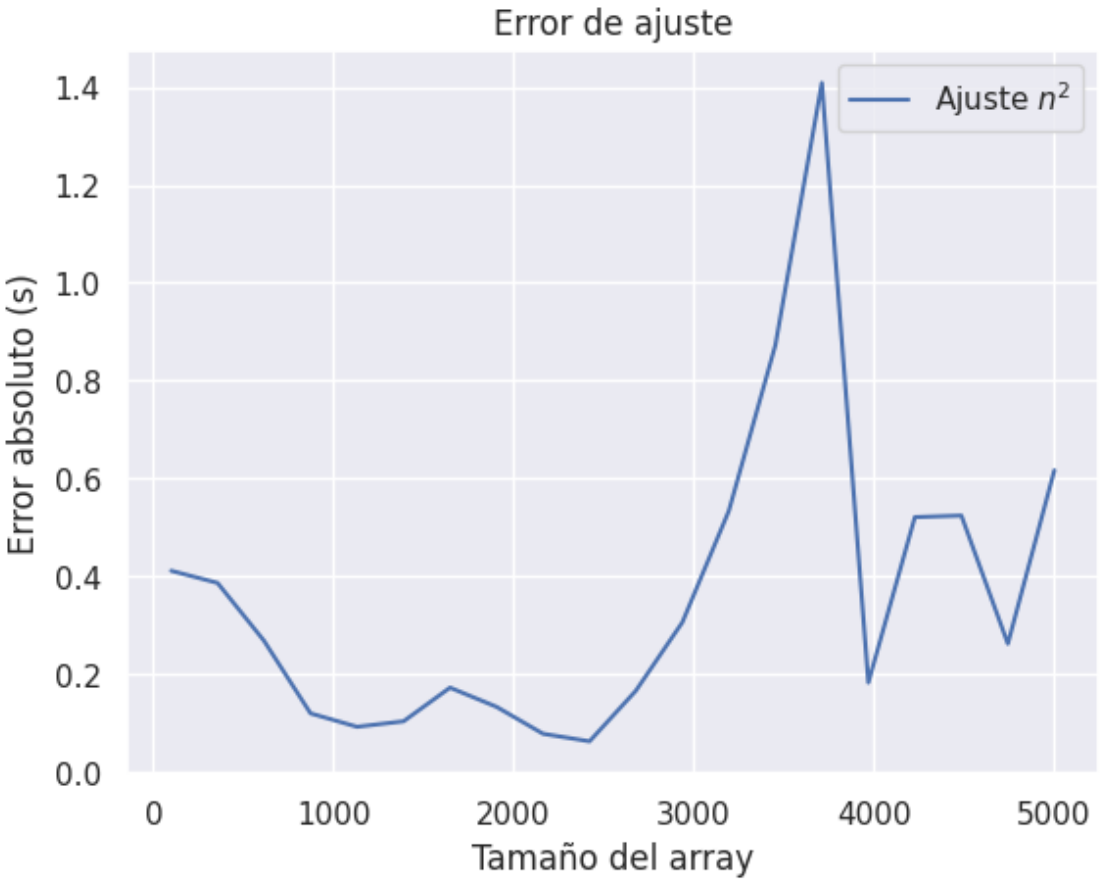
4. Mediciones

Se comprobará la complejidad teórica indicada mediante mediciones de tiempos ejecutando el algoritmo con distintas entradas.

Midiendo los tiempos de ejecución con arreglos (de tamaño entre 100 y 5000) de monedas de valores entre 0 y 2000, se obtuvo lo siguiente:



Y se obtienen los errores, para cada tamaño:



Puede verse que para todos los tamaños de entrada el error del ajuste no es grande, lo cual nos permite corroborar que la complejidad del algoritmo es $O(n^2)$

5. Conclusiones

El problema de maximización de las ganancias de Sophia frente a Mateo, quien siempre toma decisiones óptimas, se resuelve eficientemente mediante programación dinámica. La solución aprovecha la naturaleza recursiva del problema, descomponiéndolo en subproblemas más pequeños que se resuelven de manera iterativa para construir el resultado final. La ecuación de recurrencia planteada asegura que Sophia siempre toma la decisión más favorable considerando las elecciones futuras de Mateo.

Se optimiza tanto en tiempo como en espacio mediante observaciones clave sobre los tamaños pares e impares de las secuencias relevantes, evitando cálculos innecesarios.

Pudo corroborarse que la complejidad del algoritmo es $O(n^2)$ gracias a las mediciones hechas sobre distintos arreglos de números representando a la fila de monedas.

6. Consideraciones

6.1. Otra forma de ver el problema

Inicialmente, habíamos desarrollado el análisis teniendo en cuenta la primera y la última moneda dentro de una fila de monedas, obteniendo así que:

- $OPT(i,i) = m_i \ \forall \ 1 \leq i \leq n$ (caso una sola moneda)
- $OPT(i,j) = 0 \ \forall \ 1 \leq j < i \leq n$ (caso no existente en la realidad)
- $OPT(i,j) = max(m_i + OPT(f_m(i+1,j)),\ m_j + OPT(f_m(i,j-1)))$
  $\forall \ 0 \leq i < j \leq n-1$

con f_m (representando las elecciones de Mateo) de la forma:

$$f_m(i,j) = \begin{cases} (i+1,j) & \text{si } m_i \geq m_j \wedge i \neq n \\ (i,j-1) & \text{si } m_i < m_j \vee i = n \end{cases} \tag{2}$$

$$Dom(f_m) = \{(i,j) \in \mathbb{N}^2 \mid 0 \leq i \leq j \leq n\}$$

Donde $OPT(i,j)$ es el máximo valor acumulado posible para Sophia dada una secuencia de valores de monedas $m_i, m_{i+1}, ..., m_{j-1}, m_j$. Entonces, lo que se quería obtener es $OPT(1,n)$.

No se desarrollará la justificación de por qué se obtiene el óptimo buscado con esta ecuación de recurrencia, sino que se muestra ya que fue el planteo inicial que se realizó y por el que se llego al planteo final del trabajo. El problema de este planteo es que, a la hora de hacer el algoritmo, se tenía una matriz (cada posición [i][j] representaba $OPT(i,j)$) que quedaba con muchísimos ceros. Todo el triángulo inferior quedaba en ceros por el caso $j < i$ y, además, como no es necesario calcular para todos los tamaños de secuencias sino que sólo para los tamaños pares cuando n es par y para los tamaños impares cuando n es impar, se tenía la mitad de las diagonales con ceros.

Con el planteo elegido no se tiene una matriz innecesaria, sino se calculan y guardan sólo los óptimos que serán necesarios para llegar al óptimo buscado.

6.2. Cómo ejecutar

Ejecutar juego_monedas.py y tests.py del mismo modo que para la primera parte de este trabajo, pero accediendo a la carpeta tp2 en vez de tp1. Los archivos .txt deben tener el mismo formato que para la parte uno. Entre los archivos que se testean en tests.py se encuantran los provistos por la cátedra.

Tercera parte: Cambios

1. Análisis del problema

Los hermanos siguieron creciendo, Mateo aprendió sobre programación dinámica y el juego de las monedas se tornó aburrido. Los años pasaron y comenzaron a jugar juegos individuales. En particular, Sophia estaba muy enganchada con la Batalla Naval Individual.

En dicho juego, tenemos un tablero de nxm casilleros, y k barcos. Cada barco i tiene b_i de largo. Es decir, requiere de 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. No se pueden poner dos barcos de forma adyacente (es decir, no pueden estar contiguos ni por fila, ni por columna, ni en diagonal directamente). Se deben ubicar todos los barcos de tal manera que se cumplan todos los requisitos.

En este informe se demostrará que la versión de decisión de este problema es un problema NP-Completo. Luego se escribirá un algoritmo de backtracking que obtenga la solución óptima al problema en su versión de optimización. Por último se mostrará un algoritmo de aproximación para resolver este problema.

1.1. La versión de decisión del problema - Demostración de que es NP-completo

La versión de decisión del problema de La Batalla Naval es la siguiente: dado un tablero de n×m casilleros, y una lista de k barcos (donde el barco i tiene b_i de largo), una lista de restricciones para las filas (donde la restricción j corresponde a la cantidad de casilleros a ser ocupados en la fila j) y una lista de restricciones para las columnas (símil filas, pero para columnas), ¿es posible definir una ubicación de dichos barcos de tal forma que se cumplan con las demandas de cada fila y columna, y las restricciones de ubicación?

El problema se encuentra en NP:

Queremos ver qe existe un certificador eficiente para este problema (es decir, un verificador que dada una solución al problema, pueda validar en tiempo polinomial que efectivamente es una solución).

Se considera un candidato a solución al problema como una lista de tuplas, en las que se indica la posición inicial y la final de cada barco.

En primer lugar, se verifica que los barcos en la solución sean válidos y los mismos k barcos de la lista original. La solución no es válida si:

- El tamaño de la lista solución no es k.
- Los tamaños de los barcos no coinciden con los de la lista original.
- Si hay algún barco para el que no se puede calcular el tamaño por no estar ubicado horizontal o verticalmente.
- Si alguna posición tiene un valor que se sale del rango del tablero.

Todo esto puede hacerse en O(k).

Luego, se inicializa una matriz de nxm y dos listas de tamaño n y m respectivamente, para ir ubicando los barcos en la matriz y sumando las demandas que cubren. Se recorre la lista de barcos en la solución y se van ubicando en la matriz. Al ocupar una posición (i,j), se chequea que no este ocupada ni haya ninguna adyacente ocupada por otro barco. Por cada posición (i,j) que este barco

pasa a ocupar, se suma 1 en la posición i de la lista de tamaño n y 1 en la posición j de la lista de tamaño m. La solución no es válida si:

- No se logró ubicar los k barcos porque se encontraron barcos que ocupaban la misma posición o adyacentes entre sí.
- Al finalizar de ubicar los barcos, la lista de tamaño n no es igual a la de las restricciones para las filas.
- Al finalizar de ubicar los barcos, la lista de tamaño m no es igual a la de las restricciones para las columnas.

Un barco tiene tamaño de a lo sumo max(n,m), por lo que ubicar un barco en la matriz cuesta O(max(n,m)). Verificar adyacencias por posición (se chequea cada posición contra sus 8 posiciones adyacentes) es O(1), al igual que sumar 1 en las listas de demandas. Recorrer la lista de barcos es O(k). Entonces, cuesta O(nxm) + O(n) + O(m) generar el tablero y las listas de demandas y O(k*max(n,m)) ubicar los barcos en el tablero.

Si se pudieron ubicar en el tablero todos los barcos (ya sabiendo que son los k barcos originales) y las demandas cubiertas coinciden, la solución es válida.

Luego, se logró verificar la solución en tiempo polinomial y, por lo tanto, el problema está en NP.

El problema es NP-Completo:

Para demostrar esto, se reducirá un problema NP-Completo al problema de la Batalla Naval: el problema de Bin Packing en su versión unaria.

El Problema de Bin Packing en código unario: Dados un conjunto de n 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*B, debe decidir si puede cumplirse que los números pueden dividirse en B bins, subconjuntos disjuntos, tal que la suma de elementos de cada bin es exactamente igual a la capacidad C.

En primer lugar se mostrará que el problema de Bin-Packing este problema es NP-Completo.

Existe un certificador eficiente: Dada una solución, puede chequearse que la solución tiene B subconjuntos de números (O(B)), los números en cada subconjnto suman C (O(n)) y que la unión de los B subconjuntos es igual a la entrada original de n números (pueden unirse los subconjuntos, ordenar esta unión y la entrada original y compararlas, O(n*logn)).

Se vio que existe un certificador eficiente y por lo tanto Bin-Packin está en NP.

Puede reducirse el problema de 2-Partition a Bin-Packing en su versión unaria:

Se define el problema 2-Partition de la siguiente manera: Se cuenta con un conjunto de n elementos. Cada uno de ellos tiene un valor asociado. Se desea separar los elementos en 2 subconjuntos tal que la suma de los valores de cada subconjunto sea igual para ambos. Se vio en clases que 2-Partition es un problema NP-Completo.

La reducción es la siguiente. El conjunto de n elementos con valor asociado se transforma en el conjunto de números que recibe Bin-Packing. Se generan 2 Bins (B=2) con capacidad C=$\sum_{i=1}^{n} \frac{i}{2}$. Si Bin-Packing puede resolverse con estos datos de entrada, significa que 2-Partition también (si se pueden dividir los números en dos bins de capacidad =$\sum_{i=1}^{n} \frac{i}{2}$, significa que puede dividirse los n elementos en dos subconjuntos para los que sus valores sumen lo mismo).

Sabiendo que Bin-Packing es NP-Completo, se reducirá la versión unaria de este problema al problema de la batalla naval.

Se tiene conjunto de n números expresados en código unario, una cantidad de bins B, expresada en código unario, y la capacidad de cada bin C, expresada en código unario.

Como se tienen B bins con capacidad C, se tendrán B filas que tengan demanda C, haciendo que cada fila del tablero represente a un Bin. Como no se pueden poner barcos adyacentes en las

filas y no hay una restricción similar para los Bins, simplemente se tendrán B-1 filas con demanda 0 para separar las de demanda B.

Los n números que se quieren ubicar en los bins serán los barcos. Se tendrán columnas que hagan que los barcos se ubiquen horizontalmente para cubrir la demanda C de las filas. Un barco no puede ubicarse dos veces en un tablero, por lo que en la reducción se asegura que un número no puede estar dos veces en un bin.

Cualquier barco debería tener libertad para poder ubicarse en cualquier fila, ya que un número puede meterse en cualquier bin. Entonces deben tenerse tantas columnas de demanda 1 como la suma de los tamaños de los barcos, agregando columnas con demanda 0 en medio para que los barcos puedan ir separados (para bin packing no hay restricciones de adyacencia dentro de los bins, debe evitarse eso en la construcción del tablero de la batalla naval). Se dará un ejemplo para entender lo que se busca: Si se tiene C=111111 (6 en unario), B=111 (3 en unario) y n elementos [111111, 1111, 11, 111, 111] ([6, 4, 2, 3, 3]). Entonces se tendrán barcos [6, 4, 2, 3, 3], una demanda por filas [6,0,6,0,6] y una demanda por columnas [1,1,1,1,1,1,0,1,1,1,1,0,1,1,0,1,1,1,0,1,1,1].

6																					
0																					
6																					
0																					
6																					
	1	1	1	1	1	1	0	1	1	1	1	0	1	1	0	1	1	1	0	1	1	1

Traduciéndose esto en el siguiente tablero:

En donde claramente una solución al problema de la batalla naval es:

6	B	B	B	B	B	B															
0																					
6																					
0								B	B	B	B		B	B							
6																B	B	B		B	B	B
	1	1	1	1	1	1	0	1	1	1	1	0	1	1	0	1	1	1	0	1	1	1

que sería lo mismo que meter el número 6 en un bin, los números 4 y 2 en otro y los números 3 y 3 en el último.

Construyendo las entradas al problema de la batalla naval de esta manera, partiendo de las entradas de un problema de Bin-Packing, nos aseguramos que si el de la batalla naval tiene solución con esas entradas construidas, el de Bin-Packing también lo tiene. Esto se cumple ya que una solución al problema de la batalla naval dados los barcos y las demandas por filas y columnas generadas:

- Usará todos los barcos una sola vez, por definición del problema. Es decir, se usan todos los n números.
- Los barcos estarán ubicados horizontalmente (no entran verticales ya que cada columna tiene demanda 1 o 0) y cada barco va en las columnas de 1s que le corresponden (si un barco de tamaño b_i está en un grupo de b_j columnas de 1s y el barco de b_j es el más grande, este no encajaría en ningún otro lugar del tablero y ni siquiera sería solución; si un barco b_i esta

en el grupo de columnas que le corresponde, no entrará otro barco por tener las columnas demanda 1). Es decir, cada elemento se ubicará en un bin sí o sí.

- Siempre cubrirá las demandas de las B filas que tienen demanda positiva. Es decir, se llenaran los B bins.

Notar que la reducción es polinomial: Se leen los 1s que representan a B y C para generar las filas que correspondan y luego se leen los 1s que representan a los números para generar las columnas que correspondan. Las filas y columnas con 0s que se agregan en el medio no suman complejidad (son B-1 filas y n-1 columnas). Si se hubiera usado el problema de Bin-Packing pero con números en decimal, la cantidad de columnas con 1s sería exponencial con respecto al valor de los números, lo que haría que la reducción fuese pseudo-polinomial.

Finalmente, Batalla Naval $\leq_p$ Bin-Packing en código unario.

El problema de la Batalla Naval en su versión de decisión está en NP y puede reducirse un problema NP-Completo al mismo. Luego, es un problema NP-Completo.

2. La versión de optimización del problema

Dado un tablero de n×m casilleros, y una lista de k barcos (donde el barco i tiene b_i de largo) una lista de las demandas de las n filas y una lista de las m demandas de las columnas, dar la asignación de posiciones de los barcos de tal forma que se reduzca al mínimo la cantidad de demanda incumplida. Pueden no utilizarse todos los barcos. Si simplemente no se cumple que una columna que debería tene 3 casilleros ocupados tiene 1, entonces contará como 2 de demanda incumplida. Por el contrario, no está permitido exceder la cantidad demandada.

2.1. Algoritmo de backtracking - Código

```
def batalla_naval_bt(tablero, dem_fil, n, dem_col, m, barcos, k, barco_actual,
    pos_actual, orientacion_actual, solucion_parcial, mejor_solucion,
    menor_demanda_incumplida):

    if pos_actual is None or barco_actual == k: #probe con todos los barcos o no
    hay lugar en tablero
        if sum(dem_fil) + sum(dem_col) < menor_demanda_incumplida[0]:
            mejor_solucion[:] = solucion_parcial.copy()
            menor_demanda_incumplida[0] = sum(dem_fil) + sum(dem_col)
        return mejor_solucion, menor_demanda_incumplida

    if sum(dem_fil) < barcos[-1] or sum(dem_col) < barcos[-1]: #no puedo meter ni
    el barco mas chico
        if sum(dem_fil) + sum(dem_col) < menor_demanda_incumplida[0]:
            mejor_solucion[:] = solucion_parcial.copy()
            menor_demanda_incumplida[0] = sum(dem_fil) + sum(dem_col)
        return mejor_solucion, menor_demanda_incumplida

    if menor_demanda_incumplida[0] <= menor_demanda_incumplida_posible(barcos,
    barco_actual, sum(dem_fil) + sum(dem_col)): #no puedo mejorar solucion con los
    barcos que quedan
        return mejor_solucion, menor_demanda_incumplida

    ubicar_barco(tablero, dem_fil, dem_col, barco_actual, barcos[barco_actual],
    pos_actual, orientacion_actual, solucion_parcial)

    proximo_barco, proxima_posicion, proxima_orientacion = proxima_ubicacion(
    tablero, dem_fil, n, dem_col, m, barcos, k, barco_actual+1, (0,0), HORIZONTAL,
    solucion_parcial)

    sol_ubicando = batalla_naval_bt(tablero, dem_fil, n, dem_col, m, barcos, k,
    proximo_barco, proxima_posicion, proxima_orientacion, solucion_parcial,
    mejor_solucion, menor_demanda_incumplida)
```

```
23
24      quitar_barco(tablero, dem_fil, dem_col, barco_actual, barcos[barco_actual],
        pos_actual, orientacion_actual, solucion_parcial)
25
26      posible_proxima_posicion = proxima_posicion_valida_desde(tablero, dem_fil, n,
        dem_col, m, pos_actual)
27
28      proximo_barco, proxima_posicion, proxima_orientacion = proxima_ubicacion(
        tablero, dem_fil, n, dem_col, m, barcos, k, barco_actual,
        posible_proxima_posicion, orientacion_actual, solucion_parcial)
29
30      sol_sin_ubicar = batalla_naval_bt(tablero, dem_fil, n, dem_col, m, barcos, k,
        proximo_barco, proxima_posicion, proxima_orientacion, solucion_parcial,
        mejor_solucion, menor_demanda_incumplida)
31
32      return sol_ubicando if sol_ubicando[DEMANDA_INCUMPLIDA_SOLUCION] <
        sol_sin_ubicar[DEMANDA_INCUMPLIDA_SOLUCION] else sol_sin_ubicar
33
34  def batalla_naval(dem_fil, dem_col, barcos):
35
36      n = len(dem_fil)
37      m = len(dem_col)
38      k = len(barcos)
39
40      tablero = [[LIBRE for _ in range(m)] for _ in range(n)]
41      barcos.sort(reverse=True)
42
43      solucion_parcial = [None]*k
44
45      barco_inicial, pos_inicial, orientacion_inicial = proxima_ubicacion(tablero,
        dem_fil, n, dem_col, m, barcos, k, 0, (0,0), HORIZONTAL, solucion_parcial)
46
47      return batalla_naval_bt(tablero, dem_fil, n, dem_col, m, barcos, k,
        barco_inicial, pos_inicial, orientacion_inicial, solucion_parcial, [], [sum(
        dem_fil) + sum(dem_col)])
```

Se genera un tablero para ir ubicando y quitando barcos a medida que se recorren las diferentes posibles distribuciones de los barcos. Los barcos se ordenan de mayor a menor tamaño para optimizar el algoritmo ubicando primero los más grandes. Se tiene una función proxima_ubicación que, dado un tablero con barcos ubicados y las demandas por filas y columnas resultantes de esas ubicacions , la lista de barcos y un barco y una posición a partir de la cual ubicarlo, devuelve cuál es el índice del próximo barco que puede ubicarse, en la posición (inicial) que puede ubicarse y con qué orientación. Si el barco no puede ubicarse a partir de la posición dada, se intenta en las siguientes posiciones del tablero (recorriendo las columnas y bajando en las filas) y, si ya no entra en el tablero, se sigue por el siguiente barco.

En la función de backtracking se busca la mejor solución posible ubicando el barco y la mejor sin ubicarlo en la posición que se recibió, sino ubicándolo en la próxima posible (si ya no se puede ubicar se pasa al siguiente barco y se recorren los escenarios en que no se ubicó ese barco en ningún lugar del tablero).

2.2. Analisis de Complejidad

Mas allá de los chequeos por las demandas de filas y la validez de las posiciones en las que se intenta ubicar los barcos, por cada barco se están considerando dos orientaciones por cada posición en el tablero. Es decir que, en el peor caso posible (que por las podas que se hacen es prácticamente imposible que ocurra), se probarán todas las combinaciones posibles que resultan de ubicar a cada barco en cada posición del tablero, con cada orientación posible. Es decir que la complejidad es de $O((2.n.m)^k)$. Como era de esperarse, el algoritmo es exponencial sobre la cantidad de barcos.

2.3. Mediciones

Se corrieron los ejemplos de prueba de la cátedra y todos pasan en menos de un segundo, a excepción del último que tarda apróximadamente una hora ((30_25_25), que tiene muchos barcos y al parecer las podas del algoritmo no son suficientes para ese caso). No llegué a hacer mediciones como correspondería, variando los valores de n, m y k y comprobando la complejidad temporal. Pido perdón al respecto.

3. Algoritmo de aproximación

John Jellicoe (almirante de la Royal Navy durante la batalla de Jutlandia) nos propone el siguiente algoritmo de aproximación: Ir a fila/columna de mayor demanda, y ubicar el barco de mayor longitud en dicha fila/columna en algún lugar válido. Si el barco de mayor longitud es más largo que dicha demanda, simplemente saltearlo y seguir con el siguiente. Volver a aplicar hasta que no queden más barcos o no haya más demandas a cumplir.

3.1. Código

```
def aproximacion_john_jellicoe(n, m, barcos, demandas_filas, demandas_columnas, tablero):

    numero_barco = 0
    solucion = [None]*len(barcos)
    demanda_total = sum(demandas_filas) + sum(demandas_columnas)

    def puede_colocar_barco(fila, columna, longitud, horizontal):
        if horizontal:
            return puedo_ubicar_horizontal(tablero, demandas_filas, n, demandas_columnas, m, longitud, (fila,columna))
        else:
            return puedo_ubicar_vertical(tablero, demandas_filas, n, demandas_columnas, m, longitud, (fila,columna))

    def colocar_barco(tablero, demandas_filas, demandas_columnas, barco, longitud, posicion, solucion, horizontal):
        if horizontal:
            ubicar_barco_horizontal(tablero, demandas_filas, demandas_columnas, barco, longitud, posicion, solucion)
        else:
            ubicar_barco_vertical(tablero, demandas_filas, demandas_columnas, barco, longitud, posicion, solucion)

    while barcos:
        long_barco = barcos.pop(0)

        max_demanda_fila = max(demandas_filas)
        filas_con_max_demanda = [i for i, d in enumerate(demandas_filas) if d == max_demanda_fila]

        max_demanda_columna = max(demandas_columnas)
        columnas_con_max_demanda = [i for i, d in enumerate(demandas_columnas) if d == max_demanda_columna]
        colocado = False

        for fila in filas_con_max_demanda:
            for columna in range(m):
                if not colocado and puede_colocar_barco(fila, columna, long_barco, horizontal=True):
                    colocar_barco(tablero, demandas_filas, demandas_columnas, numero_barco, long_barco, (fila, columna), solucion, horizontal=True)
                    colocado = True

        for columna in columnas_con_max_demanda:
            for fila in range(n):
```

```
37                 if not colocado and puede_colocar_barco(fila, columna, long_barco,
       horizontal=False):
38                     colocar_barco(tablero, demandas_filas, demandas_columnas,
       numero_barco, long_barco, (fila, columna), solucion, horizontal=False)
39                     break
40
41         numero_barco += 1
42
43     demanda_incumplida = sum(demandas_filas) + sum(demandas_columnas)
44
45     return solucion, demanda_incumplida
46
47 def batalla_naval_aprox(dem_fil, dem_col, barcos):
48
49     n = len(dem_fil)
50     m = len(dem_col)
51     tablero = [[LIBRE for _ in range(m)] for _ in range(n)]
52     barcos.sort(reverse=True)
53     solucion = aproximacion_john_jellicoe(n, m, barcos, dem_fil, dem_col, tablero)
54     return solucion
```

3.2. Análisis de complejidad

La generación del tablero tiene complejidad $O(nxm)$.

Se hace un while para recorrer los barcos ($O(k)$) y dentro de este se seleccionan las filas y columnas de mayor demanda, que en el peor caso es seleccionar todas y, por ende, tiene complejidad $O(n+m)$. Por cada fila y columna, se fijará, en el peor caso, para cada posición en esta. En resumidas cuentas, en el peor caso se recorren todas las posiciones intentando ubicar los k barcos y finalmente la complejidad es $O(k.n.m.max(m,n))$ (max(m,n) sale del mayor tamaño posible que puede tener un barco, y se multiplica ya que puedo_ubicar_barco tiene esta complejidad para el peor caso).

3.3. Cota

El peor caso posible con el que nos podemos enfrentar es uno como el del siguiente ejemplo:

	6	1	6	1	6	1	6
4							
4							
4							
7	B	B	B	B	B	B	
4							
4							

Si en el problema se tenían 4 barcos de tamaño 6, el algoritmo de aproximación ubicó un barco en la fila de mayor demanda y ya no podrá ubicar más barcos, cuando en la solución óptima entraban los 4 barcos.

Puede comprobarse fácilmente que para un valor v par, si se tienen v/2 + 1 barcos y un tablero con demandas como el del ejemplo, en donde el algoritmo de aproximación causa que se ubique un sólo barco cuando se podían ubicar todos, se tiene que la solución óptima era $z(I) = (v/2+1)*v*2$ y que la solución aproximada es $A(I) = v*2$ y, por ende $r(A) \geq \frac{1}{(v/2)+1}$.

Puede verse que si v era impar, se tiene $z(I) = ((v - 1/2) + 1) * v * 2$ y entonces el caso de v par es peor.

Si había un barco de valor v+1, entra en la fila con esa demanda y el caso presentado sigue siendo peor (cubre dos menos de demanda). Si había un barco de tamaño mayor a v+1, no entraba y el problema era el mismo. Si había un barco de tamaño menor a v, podía entrar con el algoritmo de aproximación pero no en la solución optima (sigue siendo peor el caso presentado).

Finalmente, se concluye que $r(A) \geq \frac{1}{(v/2)+1}$, en donde v es el tamaño del primer barco que se ubica en el algoritmo de aproximación.

Como ejemplo, se agregó un archivo a la carpeta tp3 - archivos_prueba llamado 10_11_6.txt en donde v = 10. Ejecutando con el algoritmo de aproximación, se cubre una demanda de 20 (10*2). Ejecutando con el de backtracking (óptimo), se cubren 120 de demanda ((5 + 1)*10*2).

Al igual que para el algoritmo de backtrackin, no llegué a hacer mediciones con grandes volúmenes de datos para comprobar lo postulado previamente.

4. Conclusiones

Se demostró que el problema de la Batalla Naval es un problema NP-Completo en su versión de decisión, y luego se implementó un algoritmo de backtracking para resolver la versión de optimización del problema. Como era de esperarse, este algoritmo tiene complejidad exponencial (si lograbamos solucionar óptimamente el problema con complejidad polinomial, no estaríamos escribiendo este informe...). Para evitar largas esperas en la ejecución del algoritmo, se implementó un algoritmo de aproximación propuesto por John Jellicoe que logra llegar a una buen cumplimiento de demanda en tiempo polinomial.

5. Consideraciones

5.1. Cómo ejecutar

Ejecutar juego_batalla_naval.py del mismo modo que para la primera y segunda parte de este trabajo, pero accediendo a la carpeta tp3. Los archivos .txt deben tener el formato provisto por la cátedra en los casos de ejemplo.