.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(TB024) Curso Buchwald - Genender

Trabajo Práctico: Juegos de Hermanos

02 de diciembre de 2024

Integrantes:

Kevin Emir Carbajal Robles
108304

Ramiro Mineldin
101010

Nicolas Chen
105907

Franco Corn
101010

Fabio Sicca
101010

Primera parte

1. Introducción

Los hermanos Sophia y Mateo comienzan un juego creado por su padre, donde quien gane será quien tenga el mayor valor acumulado (por sumatoria) de monedas. En este juego, los jugadores pueden tomar monedas de una fila únicamente desde el inicio o el final de la fila. Ambos hermanos se turnan para seleccionar una moneda en cada ronda.

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

Quien gane será quien tenga el mayor valor acumulado (por sumatoria)

2. Análisis del problema

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

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

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

2.1. Estrategia de Sophia

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

$$S(i,j) = \text{máx}\,(v_i, v_j)$$

2.2. Estrategia de Mateo

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

$$M(i,j) = \text{mín}(v_i, v_j)$$

2.3. Notación

Donde:

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

3. Algoritmo Greedy

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

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

3.1. Implementacion

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

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

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

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

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

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

    return resultado
```

3.2. Complejidad

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

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

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

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

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

3.3. ¿Es óptimo?

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

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

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

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

Caso base (primer turno): En el primer turno, S elige entre las dos monedas de los extremos. Como S siempre toma la moneda de mayor valor en su turno, elige esa misma. Como M tomará una moneda en el siguiente turno, sera la otra moneda de los extremos (que es menor en comparación con la que tomó S). Por lo tanto, la proposición se cumple en este turno inicial.

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

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

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

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

4. Mediciones de tiempos y validación de complejidad

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

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

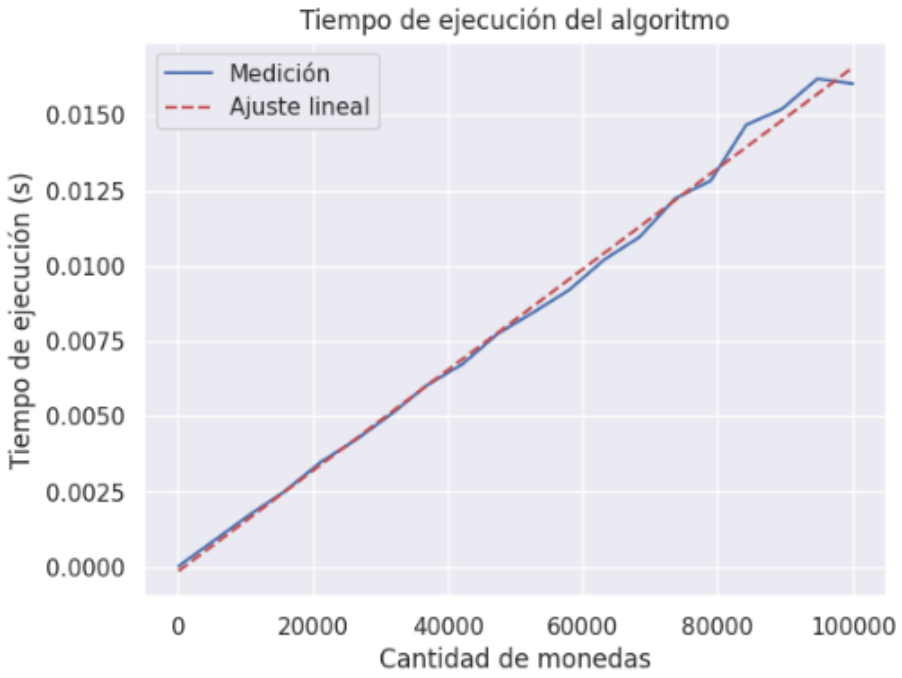
Archivo	Ganancia de Sophia	Ganancia de Mateo	Tiempo (s)
20.txt	7165	3474	$9{,}059906 \times 10^{-6}$
25.txt	9046	3577	$1{,}168251 \times 10^{-5}$
50.txt	17750	7651	$1{,}645088 \times 10^{-5}$
100.txt	35009	14641	$2{,}741814 \times 10^{-5}$
1000.txt	357814	148747	$1{,}900196 \times 10^{-4}$
10000.txt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

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

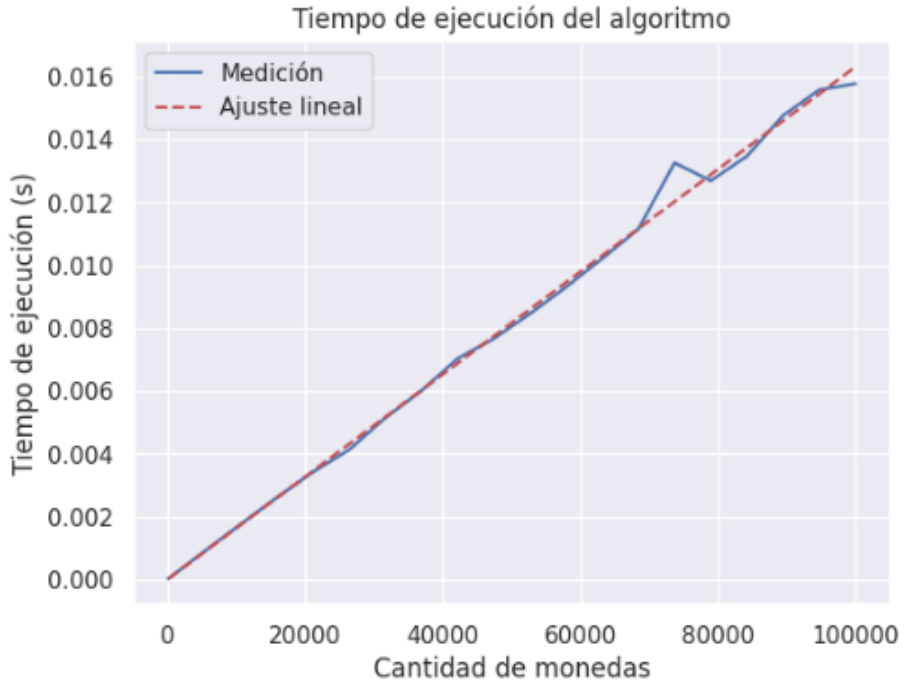
4.1. Variabilidad de la cantidad y valor de las monedas

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

4.1.1. Conjunto 1: Valores entre 0 y 1000



4.1.2. Conjunto 2: Valores entre 100000000 y 200000000



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

5. Conclusiones

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

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

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

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

Segunda parte

Tercera parte

6. Introducción

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

El objetivo es determinar si es posible colocar los barcos (no necesariamente todos) de acuerdo con las restricciones de ubicación y satisfacer, en lo posible, las demandas de cada fila y columna.

7. Análisis del Problema

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

7.1. Restricciones de demanda

Cada fila y columna tiene una demanda de cuántos casilleros deben ser ocupados. Las longitudes de los barcos deben distribuirse de manera que la suma de los casilleros ocupados en cada fila y columna coincida con la demanda dada o, en su defecto, se acerque lo más posible a ella.

En otras palabras, se debe minimizar la demanda incumplida de cada fila y columna. Es posible que algunas filas o columnas no estén ocupadas, no porque no haya demanda, sino debido a las restricciones de ubicación de los barcos.

7.2. Restricciones de ubicación

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

Además, es posible que no se ubiquen todos los barcos, ya sea por las restricciones de demanda o por la ubicación de otros barcos.

8. Backtracking

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

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

8.1. Implementación

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

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

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

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

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

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

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

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

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

```
1 # Devuelve las posiciones del barco solo si las mismas no esten afuera del
      tablero
2 def colocar_barco(direccion, pos_fil, pos_col, filas, columnas,
      longitud_barco):
3     (fila_act, columna_act) = (pos_fil, pos_col)
4     posiciones_ocupadas = set()
5     se_pudo = False
6
7     # Caso borde, posicion invalida (no tendria que ocurrir... pero por las
      dudas)
8     if fila_act >= len(filas) or columna_act >= len(columnas):
9         return posiciones_ocupadas, se_pudo
```

```
10
11      # Trato de colocarlo horizontal o verticalmente
12      if direccion == 'horizontal':
13          # Me fijo si el largo excede las filas
14          if columna_act + (longitud_barco-1) < len(columnas):
15              for m in range(columna_act, columna_act + longitud_barco):
16                  posiciones_ocupadas.add((fila_act, m))
17              se_pudo = True
18      else:
19          # Idem arriba
20          if fila_act + (longitud_barco-1) < len(filas):
21              for n in range(fila_act, fila_act + longitud_barco):
22                  posiciones_ocupadas.add((n, columna_act))
23              se_pudo = True
24
25      return posiciones_ocupadas, se_pudo
26
27  # Devuelve si se pudo ubicar un barco
28  def tratar_de_ubicar_barco(puestos, filas, columnas, barcos, barco_act,
        direccion, pos_fil, pos_col, mejores_puestos):
29      posiciones, se_pudo = colocar_barco(direccion, pos_fil, pos_col, filas,
        columnas, barcos[barco_act][1])
30      if se_pudo:
31          barco = Barco(barcos[barco_act][0], posiciones, direccion)
32          # Verifico que no se superponga con los demas, o que sea adyacente, o
         que no exceda demanda
33          if estan_superpuesto(puestos, barco) or estan_adyacentes(puestos,
        barco) or barco.excede_demanda(filas, columnas, puestos):
34              return False
35          puestos.add(barco)
36          ubicaciones_barco(puestos, filas, columnas, barcos, barco_act+1,
        mejores_puestos)
37          puestos.remove(barco)
38          return True
39      else:
40          return False
41
42  # Ubica a los barcos tratando de minimizar la demanda incumplida, obteniendo
        asi un optimo
43  def ubicaciones_barco(puestos, filas, columnas, barcos, barco_act,
        mejores_puestos):
44      # La mejor solucion es usar todos los barcos, ya no sigo buscando
45      if len(mejores_puestos) == len(barcos):
46          return
47
48      demanda_actual = demanda(puestos, filas, columnas)
49      demanda_mejor = demanda(mejores_puestos, filas, columnas)
50      demanda_faltante = demanda_barcos_faltantes(barcos, barco_act)
51
52      # Si con los barcos que tengo puestos (y los que me faltan colocar)
53      # no llego a cubrir la maxima demanda que obtuve, no sigo
54      if demanda_actual + demanda_faltante <= demanda_mejor:
55          return
56
57      # Llegue a una mejor solucion, es decir, tengo mayor demanda cumplida que
         antes
58      if demanda_actual > demanda_mejor:
59          mejores_puestos[:] = puestos
60
61      # Me fijo que no me exceda por las dudas
62      if barco_act >= len(barcos):
63          return
64
65      se_pudo_colocar = False
66      for nro_fila, demanda_fila in ((filas)).items():
67          # Salteo las filas con demanda 0 o las que tengan su demanda al
        maximo
68          if demanda_fila == 0 or demanda_por_fila(nro_fila, puestos) ==
        demanda_fila:
69              continue
```

```
70
71          for nro_columna, _ in ((columnas)).items():
72              if tratar_de_ubicar_barco(puestos, filas, columnas, barcos,
        barco_act, 'vertical', nro_fila, nro_columna, mejores_puestos):
73                  se_pudo_colocar = True
74              # Para barcos de largo 1, basta con colocarlo horizontal o
        vertical
75              if barcos[barco_act][1] == 1:
76                  continue
77              if tratar_de_ubicar_barco(puestos, filas, columnas, barcos,
        barco_act, 'horizontal', nro_fila, nro_columna, mejores_puestos):
78                  se_pudo_colocar = True
79
80      # Si no pude colocar un barco de cierta longitud, no voy a poder colocar
        el siguiente si tiene la misma longitud
81      if barco_act > 0 and not se_pudo_colocar and barcos[barco_act-1][1] ==
        barcos[barco_act][1]:
82          while barcos[barco_act-1][1] == barcos[barco_act][1]:
83              barco_act += 1
84              if barco_act == len(barcos):
85                  # No hay mas barcos para poner
86                  return
87
88      # No tengo en cuenta este barco (ya sea si lo pude colocar o n o), asi
        que sigo con el siguiente
89      ubicaciones_barco(puestos, filas, columnas, barcos, barco_act+1,
        mejores_puestos)
90
91  def backtracking(filas, columnas, barcos):
92      mejores_puestos = [Barco(-1, set(), True)]
93      # Al final no mejora nada ordenando las filas por mayor demanda
94      dict_filas = {i: demanda for i, demanda in enumerate(filas)}
95      dict_filas_ordenado = dict(sorted(dict_filas.items(), key=lambda item:
        item[0], reverse=False))
96
97      # Al final no mejora nada ordenando las columnas por mayor demanda
98      dict_columnas = {i: demanda for i, demanda in enumerate(columnas)}
99      dict_columnas_ordenado = dict(sorted(dict_columnas.items(), key=lambda
        item: item[0], reverse=False))
100
101     # Ordeno los barcos de mayor a menor longitud
102     tuplas = [(i, largo) for i, largo in enumerate(barcos)]
103     barcos_ordenados = sorted(tuplas, key=lambda x: x[1], reverse=True)
104
105     ubicaciones_barco(set(), dict_filas_ordenado, dict_columnas_ordenado,
        barcos_ordenados, 0, mejores_puestos)
106     return mejores_puestos, demanda(mejores_puestos, filas, columnas)
```

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

8.2. Complejidad

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

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

- **Ordenar los barcos:** $O(b \log b)$
- **Recorrer la matriz:** $O(n \cdot m)$
- **Colocar el barco:** $O(b_i)$
- **Verificar superposición y adyacencia:** $O(b \cdot b_i)$
- **Verificar que no se exceda la demanda:** $O(n + m + b \cdot b_i)$

- **Calcular la demanda actual:** $O(n + m + b \cdot b_i)$
- **Calcular la demanda faltante:** $O(b)$, suponiendo que faltan todos los barcos.
- **Actualizar la mejor ubicación de los barcos:** $O(b)$
- **Evitar colocar barcos de cierta longitud si no fueron colocados previamente:** $O(b)$, suponiendo que todos los barcos tienen la misma longitud.

En resumen, la complejidad total del algoritmo es:

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

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

9. Algoritmo de Aproximacion

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

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

9.1. Análisis de Aproximación

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

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

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

9.2. Implementación

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

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

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

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

```

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

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

```
5                return
6
```

9.3. Cálculo de la aproximación

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

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

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

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

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

10. Mediciones

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

10.1. Conjuntos de Datos Dados por la Cátedra

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

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

Cuadro 1: Tiempos para el algoritmo de backtracking

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

Cuadro 2: Tiempos para el algoritmo de aproximación

10.2. Conjuntos de Datos del Algoritmo de Aproximación

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

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

Cuadro 3: Tiempos para el algoritmo de backtracking

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

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

Cuadro 4: Tiempos para el algoritmo de aproximacion

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

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