.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico
Juegos de Hermanos

Primer entrega: 2 de diciembre de 2024
Reentrega: 4 de febrero de 2025

Franco Rodriguez
108799

Joaquin Ordoñez
108865

Amparo Zanardi
108360

Juan Patricio Amboage
106892

Santiago Antón
107542

Índice

1. Introducción y primeros años **4**
1.1. Análisis del problema . . . 4
1.2. Demostración de optimalidad . . . 4
1.2.1. Demostración por contradicción . . . 4
1.3. Solución Propuesta . . . 5
1.3.1. Implementacion . . . 5
1.3.2. Análisis de complejidad . . . 5
1.4. Análisis de variabilidad . . . 5
1.5. Ejemplos de ejecución . . . 5
1.5.1. Ejemplo 1 . . . 5
1.5.2. Ejemplo 2 . . . 7
1.5.3. Ejemplo 3 . . . 8
1.6. Mediciones . . . 9
1.7. Conclusiones . . . 10

2. Mateo empieza a Jugar **11**
2.1. Análisis del problema y elaboración de la ecuación de recurrencia ¿Encuentra siempre la solución óptima? . . . 11
2.2. Algoritmo y Complejidad . . . 14
2.3. Mediciones Empíricas de Complejidad . . . 15
2.4. Conclusión . . . 16

3. Cambios **17**
3.1. "$BatallaNaval$" $\in NP$. . . . 17
3.1.1. Certificador . . . 17
3.1.2. Complejidad . . . 18
3.2. "$BatallaNaval$" $\in NP - C$. . . . 19
3.2.1. 3-Partition . . . 19
3.2.2. Entradas y Salidas . . . 20
3.2.3. $3 - Partition \leq BatallaNaval$ . . . 20
3.2.4. Complejidad . . . 20
3.2.5. Ejemplo . . . 21
3.3. Resolución de $BatallaNaval$ por Backtracking . . . 22
3.3.1. Implementación . . . 22
3.3.2. Sets de datos . . . 26
3.3.3. Mediciones . . . 29
3.4. Resolución de $BatallaNaval$ por Aproximación . . . 31
3.4.1. ¿Es buena aproximación? . . . 31
3.4.2. Implementación . . . 31
3.4.3. Análisis de complejidad . . . 32

3.4.4. Cálculo de la cota del algoritmo. 33
3.5. Resolución de *BatallaNaval* por Greedy 34
3.5.1. Implementación 34
3.5.2. Análisis de complejidad 36
3.5.3. Mediciones 37
3.5.4. Conclusiones 39
3.6. Concluciones 39

4. Observaciones **40**

5. Correcciones **41**
5.1. Introducción y primeros años 41
5.1.1. Demostración de optimalidad 41
5.1.2. Ajuste de medición 42
5.2. Mateo empieza a Jugar 43
5.2.1. Ecuación de recurrencia 43
5.2.2. Demostración de optimalidad 43

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. Ya desde muy chicos, ella jugaba mucho con su hermano.
Pasaron los años, y fueron cambiando los juegos. Cuando Mateo cumplió 4 años, el padre de ambos le explicó un juego a Sophia: Se dispone una fila de n monedas, de diferentes valores. En cada turno, un jugador debe elegir alguna moneda. Pero no puede elegir cualquiera: solo puede elegir o bien la primera de la fila, o bien la última. Al elegirla, la remueve de la fila, y le toca luego al otro jugador, quien debe elegir otra moneda siguiendo la misma regla. Siguen agarrando monedas hasta que no quede ninguna. Quien gane será quien tenga el mayor valor acumulado (por sumatoria).
El problema es que Mateo es aún pequeño para entender cómo funciona esto, por lo que Sophia debe elegir las monedas por él. Digamos, Mateo está "jugando". Aquí surge otro problema: Sophia es muy competitiva. Será buena hermana, pero no se va a dejar ganar (consideremos que tiene 7 nada más). Todo lo contrario. En la primaria aprendió algunas cosas sobre algoritmos greedy, y lo va a aplicar.

1.1. Análisis del problema

El problema consiste en maximizar el valor acumulado de Sophia, y minimizar el de Mateo.
Por lo tanto, Sophia debe tomar la moneda de mayor valor en su turno, y la de menor valor en el turno de Mateo; de esta manera siempre ganará.
El algoritmo itera n veces para n monedas, y divide los turnos entre pares e impares, ya que como Sophia siempre comienza le corresponden los turnos pares. Sophia toma la moneda más grande entre la primera y la última de la fila y la agrega a su colección. Para el turno de Mateo, Sophia hace lo mismo pero con la moneda más chica y la colección de Mateo. El algoritmo es greedy porque en cada turno toma la mejor decisión local posible: la moneda de mayor valor disponible para Sophia y la de menor valor para Mateo. Esto asegura que cada turno contribuya de manera óptima al objetivo global.

1.2. Demostración de optimalidad

1.2.1. Demostración por contradicción

¿existe otra estrategia que pueda ser mejor para Sophia? Teniendo en cuenta que solo puede tomar la primera o la última moneda, y con el algoritmo propuesto actualmente toma la de mayor valor, la única otra estrategia posible es que tome la moneda de menor valor. Esto no es óptimo ya que no maximiza su ganancia ni minimiza la de Mateo, por lo tanto, no existe otra estrategia greedy que pueda ser mejor.

1.3. Solución Propuesta

1.3.1. Implementacion

```
1
2  def monedas(arr, arr_soph, arr_mat):
3      izq = 0
4      der = len(arr) - 1
5      res = ""
6      for i in range(len(arr)):
7          prim = arr[izq]
8          ult = arr[der]
9          if i % 2 == 0:
10             if prim > ult:
11                 res += "Primera moneda para Sophia; "
12                 arr_soph.append(prim)
13                 izq += 1
14             else:
15                 res += " ltima  moneda para Sophia; "
16                 arr_soph.append(ult)
17                 der -= 1
18         else:
19             if arr[izq] < arr[der]:
20                 res += "Primera moneda para Mateo; "
21                 arr_mat.append(prim)
22                 izq += 1
23             else:
24                 res += " ltima  moneda para Mateo; "
25                 arr_mat.append(ult)
26                 der -= 1
27     return res[:-2]
```

1.3.2. Análisis de complejidad

El algoritmo itera n veces, siendo n la cantidad de monedas, ya que por cada moneda hay un turno.
En cada iteración se toma una decisión en $\mathcal{O}(1)$ entre elegir la primera o última moneda de la fila, por lo tanto, su complejidad es $\mathcal{O}(n)$.

1.4. Análisis de variabilidad

La variabilidad de los valores de las monedas no influye en los tiempos del algoritmo, ya que únicamente se comparan las dos, para ver cuál de las dos es la de mayor o menor valor.
El algoritmo sigue siendo óptimo para cualquier variabilidad de los valores de las monedas, ya que no importa si la diferencia entre los valores son muy grandes o pequeños, siempre se tomará la decisión que maximice la ganancia de Sophia en cada turno. Está diseñado para garantizar la mejor opción en cada turno para Sophia y la peor para Mateo, sin importar la distribución de los valores.

1.5. Ejemplos de ejecución

1.5.1. Ejemplo 1

Recibiendo el arreglo Monedas = [5, 9, 1, 2, 4, 8, 6, 3, 7]

El algoritmo resuelve:

1. Última moneda para Sophia:
   - Monedas = [5, 9, 1, 2, 4, 8, 6, 3]
   - Sophia = [7]
   - Mateo = []
2. Última moneda para Mateo;
   - Monedas = [5, 9, 1, 2, 4, 8, 6]
   - Sophia = [7]
   - Mateo = [3]
3. Última moneda para Sophia;
   - Monedas = [5, 9, 1, 2, 4, 8]
   - Sophia = [7 , 6]
   - Mateo = [3]
4. Primera moneda para Mate:
   - Monedas = [9, 1, 2, 4, 8]
   - Sophia = [7, 6]
   - Mateo = [3, 5]
5. Primera moneda para Sophia:
   - Monedas = [1, 2, 4, 8]
   - Sophia = [7, 6, 9]
   - Mateo = [3, 5]
6. Primera moneda para Mateo:
   - Monedas = [2, 4, 8]
   - Sophia = [7, 6, 9]
   - Mateo = [3, 5, 1]
7. Última moneda para Sophia:
   - Monedas = [2, 4]
   - Sophia = [7, 6, 9, 8]
   - Mateo = [3, 5, 1]
8. Primera moneda para Mateo:
   - Monedas = [4]
   - Sophia = [7, 6, 9, 8]
   - Mateo = [3, 5, 1, 2]
9. Última moneda para Sophia:
   - Monedas = []
   - Sophia = [7, 6, 9, 8, 4]
   - Mateo = [3, 5, 1, 2]

Sophia se quedó con las monedas = [7, 6, 9, 8, 4], suma: 34
Mateo se quedó con las monedas = [3, 5, 1, 2], suma: 11

1.5.2. Ejemplo 2

Recibiendo el arreglo Monedas = [1, 100, 2, 99, 3, 98, 4, 97]
El algoritmo resuelve:

1. Última moneda para Sophia:
   - Monedas = [1, 100, 2, 99, 3, 98, 4]
   - Sophia = [97]
   - Mateo = []
2. Primera moneda para Mateo:
   - Monedas = [100, 2, 99, 3, 98, 4]
   - Sophia = [97]
   - Mateo = [1]
3. Primera moneda para Sophia:
   - Monedas = [2, 99, 3, 98, 4]
   - Sophia = [97, 100]
   - Mateo = [1]
4. Primera moneda para Mateo:
   - Monedas = [99, 3, 98, 4]
   - Sophia = [97, 100]
   - Mateo = [1, 2]
5. Primera moneda para Sophia:
   - Monedas = [3, 98, 4]
   - Sophia = [97, 100, 99]
   - Mateo = [1, 2]
6. Primera moneda para Mateo:
   - Monedas = [98, 4]
   - Sophia = [97, 100, 99]
   - Mateo = [1, 2, 3]
7. Primera moneda para Sophia:
   - Monedas = [4]
   - Sophia = [97, 100, 99, 98]
   - Mateo = [1, 2, 3]
8. Última moneda para Mateo:
   - Monedas = []
   - Sophia = [97, 100, 99, 98]
   - Mateo = [1, 2, 3, 4]

Sophia se quedó con las monedas = [97, 100, 99, 98], suma: 394
Mateo se quedó con las monedas = [1, 2, 3, 4], suma: 10

1.5.3. Ejemplo 3

Recibiendo el arreglo Monedas = [9, 8, 7, 6, 5, 4, 3, 2, 1]
El algoritmo resuelve:

1. Primera moneda para Sophia:
   - Monedas = [8, 7, 6, 5, 4, 3, 2, 1]
   - Sophia = [9]
   - Mateo = []
2. Última moneda para Mateo:
   - Monedas = [8, 7, 6, 5, 4, 3, 2]
   - Sophia = [9]
   - Mateo = [1]
3. Primera moneda para Sophia:
   - Monedas = [7, 6, 5, 4, 3, 2]
   - Sophia = [9, 8]
   - Mateo = [1]
4. Última moneda para Mateo:
   - Monedas = [7, 6, 5, 4, 3]
   - Sophia = [9, 8]
   - Mateo = [1, 2]
5. Primera moneda para Sophia:
   - Monedas = [6, 5, 4, 3]
   - Sophia = [9, 8, 7]
   - Mateo = [1, 2]
6. Última moneda para Mateo:
   - Monedas = [6, 5, 4]
   - Sophia = [9, 8, 7]
   - Mateo = [1, 2, 3]
7. Primera moneda para Sophia:
   - Monedas = [5, 4]
   - Sophia = [9, 8, 7, 6]
   - Mateo = [1, 2, 3]
8. Última moneda para Mateo:
   - Monedas = [5]
   - Sophia = [9, 8, 7, 6]
   - Mateo = [1, 2, 3, 4]
9. Última moneda para Sophia:
   - Monedas = []
   - Sophia = [9, 8, 7, 6, 5]

- Mateo = [1, 2, 3, 4]

Sophia se quedó con las monedas = [9, 8, 7, 6, 5], suma: 35
Mateo se quedó con las monedas = [1, 2, 3, 4], suma: 10

Los casos de prueba particulares del curso se encuentran en el archivo "tests.py".

1.6. Mediciones

Realizamos mediciones sobre los tiempos de resolución con base en los casos de prueba propuestos por la cátedra:

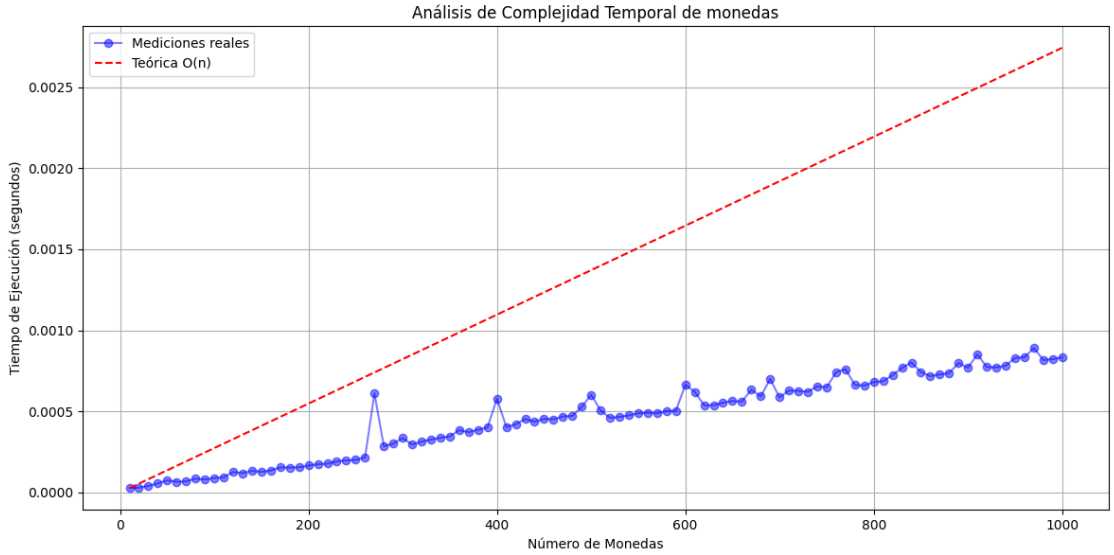

Figura 1: tiempos de resolución

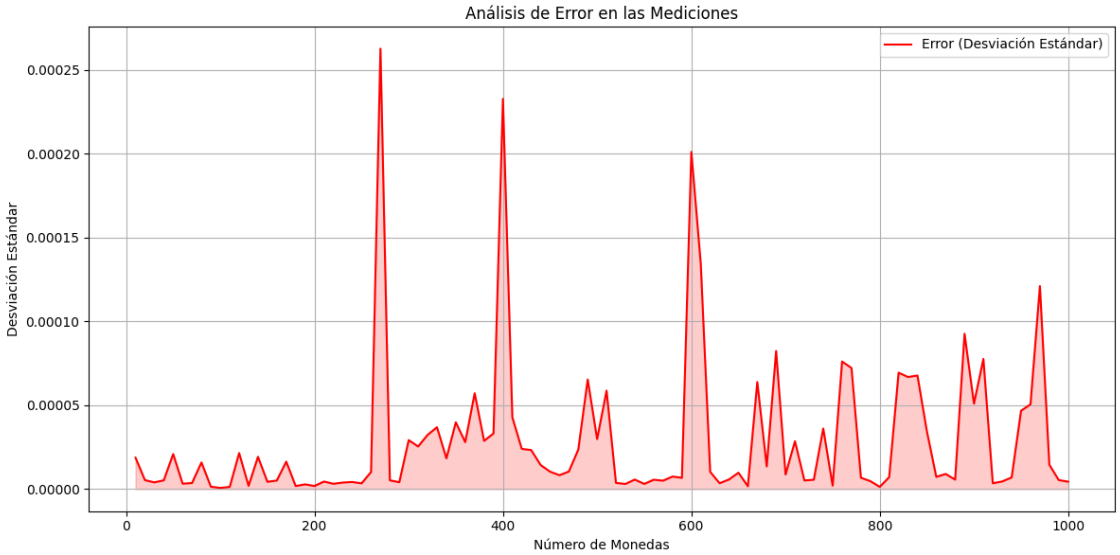

Figura 2: Error de las mediciones

1.7. Conclusiones

Finalmente, queda verificado que el algoritmo cumple con lo pedido en el enunciado mediante el análisis de los resultados y gráficos. En los gráficos se ve que justamente el tiempo de ejecución del algoritmo respecto de la cantidad de monedas sigue una trayectoria lineal, y que el error absoluto es muy pequeño, disminuyendo a medida que crece la cantidad de monedas.

2. Mateo empieza a Jugar

Continuamos con la situación planteada en la primera sección, solo que ahora han transcurrido algunos años. Mateo ya tiene siete, edad suficiente para convertirse en un oponente respetable. Para Sophia no será posible continuar sometiéndolo a sus engaños, ya que el joven Mateo ingresó en el campo de los algoritmos greedy, comenzándolos a aplicar y torciendo un poco las cosas para su hermana. Y esto a ella no le gusta nada. Afortunadamente, tampoco se quedó de brazos cruzados y comenzó a aprender sobre programación dinámica, por lo que va a contraatacar con esta nueva técnica para asegurarse ganar siempre que pueda.

2.1. Análisis del problema y elaboración de la ecuación de recurrencia ¿Encuentra siempre la solución óptima?

La primera pregunta al iniciar con el planteo de la ecuación de recurrencia es: ¿Existe alguna forma de resolver un juego de n monedas a partir de juegos más pequeños? La respuesta, lógicamente, tiene que ser afirmativa, ya que de lo contrario Sophia tendría que volver a jugar con Greedy o bien buscarse un método alternativo.

Comenzar con los casos bases siempre fue la mejor opción, así que rápidamente se pueden realizar dos deducciones:

- Si el juego tiene una sola moneda, la mejor opción es la misma moneda.
- Si el juego cuenta con dos monedas, elegir la de mayor valor.

Teniendo ya un punto de partida, surge otro cuestionamiento: ¿es posible construir una estructura memoriosa que registre estos dos casos base y nos permita avanzar con un juego de tres monedas en adelante?

Sea, por ejemplo, la serie de monedas $[1, 10, 5]$, la estructura debería guardar los resultados del siguiente conjunto de juegos:

- Tres juegos de una moneda: [1], [10], [5].
- Dos juegos de dos monedas: [1, 10], [10, 5].
- Un juego de tres monedas: [1, 10, 5] (el que queremos saber)

Observar que a medida que se construye la solución partiendo desde juegos más pequeños, la cantidad de juegos por paso disminuye en uno, por lo cual, una matriz cuadrada triangular podría adecuarse al contexto.

$$\begin{bmatrix} x_1 & y_1 & z_1 \\ 0 & x_2 & y_2 \\ 0 & 0 & x_3 \end{bmatrix}$$

x: resultados de los juegos singulares y: resultados de los juegos de dos monedas z: resultado del juego completo de tres monedas

De esta manera, ya es posible pensar con claridad en ecuaciones de recurrencia, y en un algoritmo consecuente. Sea i el número de fila y j el número de columna, para los casos donde $i = j$, es decir, la diagonal de variables x_i, se completa con los resultados de todos los juegos de una moneda.

```
    if i == j:
        matriz[i][j] = monedas[i]
```

$$\begin{bmatrix} 1 & y_1 & z_1 \\ 0 & 10 & y_2 \\ 0 & 0 & 5 \end{bmatrix}$$

Para y_1 e y_2 tenemos, respectivamente, los juegos de monedas [1, 10] y [10, 5], donde fácilmente podemos resolver eligiendo el máximo de cada uno. Particularmente, notar que cuando tenemos juegos de dos monedas, se cumple la relación $i = j - 1$. En el caso de la matriz, y_1 e y_2 se posicionan en $matriz[0][1]$ y $matriz[1][2]$. Entonces, ya tenemos una segunda ecuación para incorporar.

```
    if i == j-1:
        matriz[i][j] = max(monedas[i],monedas[j])
```

O de manera similar:

```
    if i == j-1:
        matriz[i][j] = max(matriz[i][i],matriz[j][j])
```

$$\begin{bmatrix} 1 & 10 & z_2 \\ 0 & 10 & 10 \\ 0 & 0 & 5 \end{bmatrix}$$

Y por último, el juego de tres monedas, que ya no es tan lineal. ¿Qué variantes pueden ocurrir en esta situación?

- *Alternativa 1*: Si Sophia elige la moneda de valor 1, su siguiente elección será la peor entre 10 y 5, ya que cuenta con que Mateo se apropiará de la mejor opción.
- *Alternativa 2*: Si elige la de valor 5, entonces su segunda moneda será la peor entre 1 y 10.

A Sophia claramente le servirá aquella que le deje más puntos. Por lo tanto, su juego de monedas óptimo vendrá dado por:

```
    max(alternativa 1, alternativa 2)
```

A su vez, podría definirse a una *alternativa i* como:

```
    alternativa_i = monedas[i] + Mejor juego post-elecci n de Mateo
```

Entonces, si Sophia elige la moneda de valor 5, ¿Cuál es el mejor juego post-elección de Mateo? Siendo que a Mateo le queda el juego de dos monedas [1, 10], el mejor juego post-elección de Mateo será el resultado del juego singular [1].

```
    Mejor juego post-elecci n de Mateo = Mejor resultado[ Juego original - {moneda
     elegida por Sophia} - {moneda elegida por Mateo} ]
```

Y dicho juego de una moneda ya lo resolvimos anteriormente, solo tenemos que acceder al lugar de la matriz indicado. Por consiguiente, para este caso de tres monedas:

- *Alternativa 1*: 1 + matriz[2][2]
- *Alternativa 2*: 5 + matriz[0][0]

Ambas alternativas dan el mismo resultado: 6. Sophia está condenada a perder está partida (Mateo hará 10 puntos). Para su disfortunio, la moneda más elevada está situada en el medio, al cuál nunca tendrá acceso. Vemos que, a pesar de que Sophia encontró la solución óptima, no existe algoritmo posible que vulnere el orden de turnos. Solo le queda perder dejando su mayor esfuerzo en batalla.

$$\begin{bmatrix} 1 & 10 & 6 \\ 0 & 10 & 10 \\ 0 & 0 & 5 \end{bmatrix}$$

Pero entonces, ¿Para qué sirve la diagonal y_i si solo tuvimos en cuenta a los juegos de una moneda? En efecto, en este caso no hicieron falta, ya el juego de dos monedas fue elección para Mateo. Sophia tuvo que elegir, en su primer turno, con un juego de tres monedas, que requirió el servicio del juego de una moneda (su segundo turno).

¿Y si ampliamos el set a cuatro monedas? Por ejemplo: $[1, 10, 5, 7]$

Para los juegos de una y dos monedas, la matriz nos queda:

$$\begin{bmatrix} 1 & 10 & 0 & 0 \\ 0 & 10 & 10 & 0 \\ 0 & 0 & 5 & 7 \\ 0 & 0 & 0 & 7 \end{bmatrix}$$

En la tercera iteración tenemos dos juegos de tres monedas para resolver, $[1, 10, 5]$ y $[10, 5, 7]$. Para el primero, ya sabemos que el valor óptimo es 6. Para el segundo, la cuenta sería:

- $Alternativa_1 = 10 + \text{mín}(dp[3][3], dp[2][2]) = 10 + \text{mín}(7, 5) = 15.$
- $Alternativa_2 = 7 + \text{mín}(dp[2][2], dp[1][1]) = 7 + \text{mín}(5, 10) = 12.$

$dp[1][3] = \text{máx}(15, 12) = 15.$

$$\begin{bmatrix} 1 & 10 & 6 & 0 \\ 0 & 10 & 10 & 15 \\ 0 & 0 & 5 & 7 \\ 0 & 0 & 0 & 7 \end{bmatrix}$$

Para la cuarta iteración, siendo i=primera moneda, j=ultima moneda, Sophia puede:

- $Alternativa_1$: Elegir la moneda de 1 (i), siguiendo con Mateo seleccionando el máximo entre $i + 1$ y j (10, 7), que será 10, y luego Sophia reutilizando el resultado obtenido del mejor juego entre [5,7], y ahora sí utilizamos la diagonal y_i.
- $Alternativa_2$: Elegir la moneda de 7 (j), siguiendo con Mateo seleccionando el máximo entre i y $j - 1$ (1,5), que será 5, y luego Sophia reutilizando el resultado obtenido del mejor juego entre [1, 10].

La alternativa 1 ofrece un resultado de 8, mientras que la alternativa 2 suma un total de 17. Esta vez, Sophia no fue perjudicada por el orden y le dio una paliza táctica a su hermano.

$$\begin{bmatrix} 1 & 10 & 6 & 17 \\ 0 & 10 & 10 & 15 \\ 0 & 0 & 5 & 7 \\ 0 & 0 & 0 & 7 \end{bmatrix}$$

La ecuación de recurrencia resultante será:

```
    if coins[i+1]>coins[j]:
            alternativa_1 = monedas[i] + matriz[i+2][j]
    else:
            alternativa_1 = monedas[i] + matriz[i+1][j-1]

    if coins[j-1]>coins[i]:
            alternativa_2 = monedas[j] + matriz[i][j-2]
    else:
            alternativa_2 = monedas[j] + matriz[i+1][j-1]
```

Haciendo la revisión manual:

1. Sophia elige la moneda 7.

2. Mateo elige la moneda 5.
3. Sophia elige la moneda 10.
4. Mateo elige la moneda 1.

Podemos ver que a medida que vamos incluyendo más monedas al juego, la matriz va a ir creciendo y las decisiones se van tomando sobre juegos anteriores cada vez más grandes. Para 5 monedas, la última iteración surgirá de decidir entre tres juegos de tres monedas. Este sistema de decisiones recursivas es lo que permite a Sophia asegurarse la mayor puntuación posible, aunque como vimos en el primer ejemplo, no siempre obteniendo garantías de una victoria.

Incluso pueden presentarse casos donde, gracias a las virtudes de este algoritmo, Sophia estratégicamente elige una moneda de menor valor, para así evitar que Mateo se quede con una moneda alta en el siguiente turno. Por ejemplo, en el juego $[3, 10, 5, 1]$, elegir 1 (sacrificando a la moneda de 3) y luego 10 gana la partida (11 puntos). Si, en cambio, Sophia siguiera el enfoque Greedy de Mateo y pensara localmente, elegiría 3, Mateo luego se llevaría el 10, Sophia concluiría con el 5 y Mateo con el 1, perdiendo 8 a 11.

2.2. Algoritmo y Complejidad

El siguiente código muestra la implementación del algoritmo en Python:

```python
def maxCoins(coins):
    n = len(coins)
    dp = [[0] * n for _ in range(n)]

    for length in range(1 , n +1):
        for i in range(n - length + 1):
            j = i + length - 1
            if i == j : # Solo una moneda
                dp [ i ][ j ] = coins [ i ]
            elif i  == j-1:
                dp[i][j] = max(coins[i],coins[j])
            else :
                if coins[i+1]>coins[j]:
                    pick_i = coins[i] + dp[i+2][j]
                else:
                    pick_i = coins[i] + dp[i+1][j-1]
                if coins[j-1]>coins[i]:
                    pick_j = coins[j] + dp[i][j-2]
                else:
                    pick_j = coins[j] + dp[i+1][j-1]
                dp[i][j] = max(pick_i, pick_j)
    return  dp[0][n-1]
```

La complejidad del algoritmo es $O(n^2)$ debido al doble bucle para llenar la matriz `dp`. Dado que la elección de monedas es constante, la variabilidad de sus valores no afecta el tiempo de ejecución.

2.3. Mediciones Empíricas de Complejidad

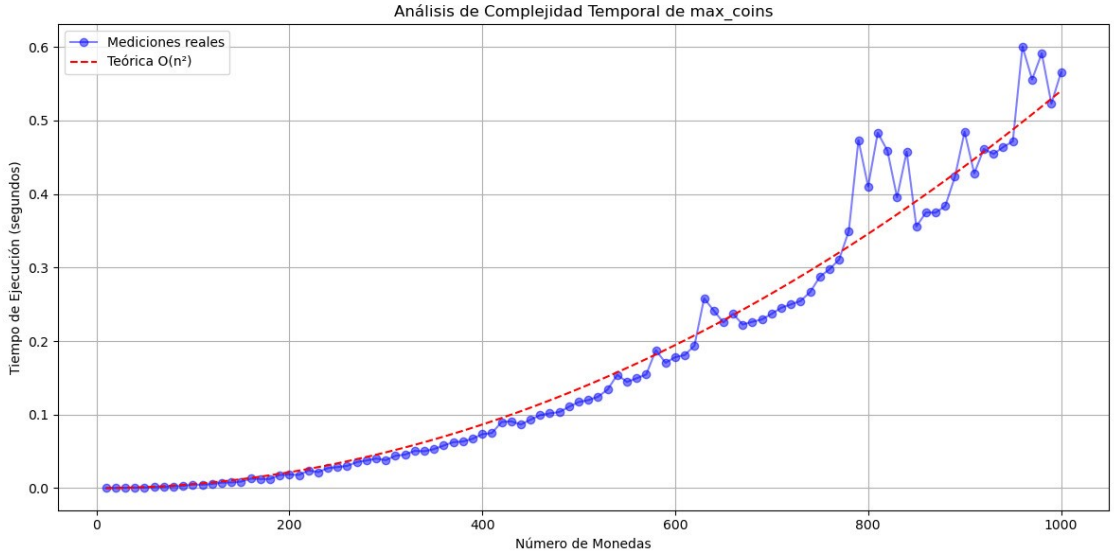

Figura 3: Complejidad de 0 a 1000 elementos

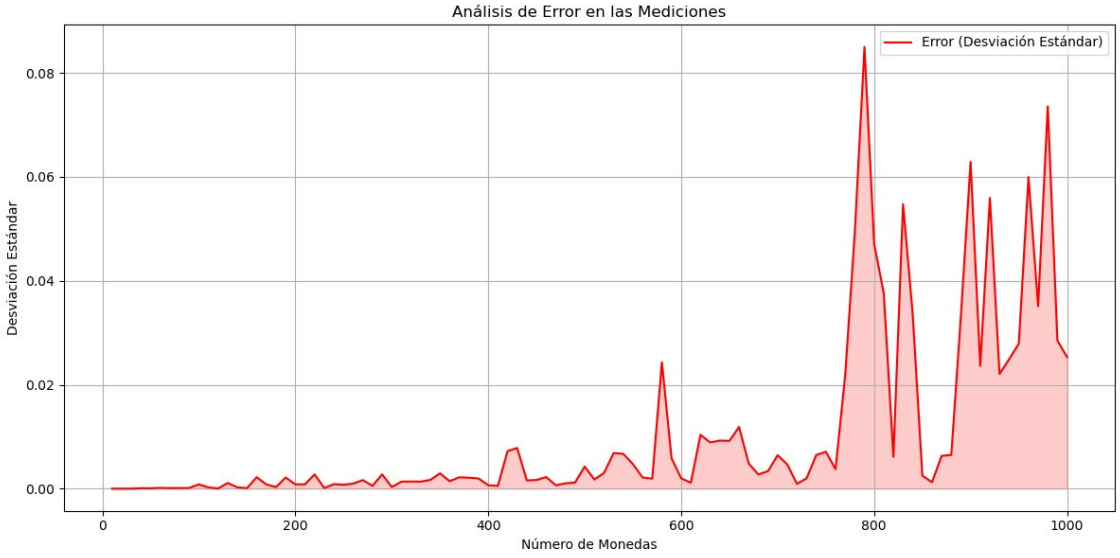

Figura 4: Error de las mediciones

Tras realizar ejecuciones del algoritmo con sets de prueba incrementales (10, 100 y 1000 monedas) pudimos verificar que, en efecto, la complejidad temporal se curva en forma de parábola, lo cual condice con el orden $O(n^2)$.
Para hacer las pruebas del algoritmo se utilizaron distintos sets de "monedas". Estos sets se hicieron correr por una función llamada test que corría el algoritmo anteriormente visto y validaba que la devolución del algoritmo de programación dinámica fuera la esperada.
A continuación se muestra el código que se usó para las pruebas.

```
1 from parte_dos import maxCoins
```

```
2
3  def test(coins, expectedAnswer):
4      got = maxCoins(coins)
5      if got != expectedAnswer:
6
7          print(f"\u2717 Coins: {coins}, expected: {expectedAnswer}, got: {got}")
8          return
9
10     print(f"\u2713 Coins: {coins}, expected: {expectedAnswer}")
11
12     return
```

Esta función $test(coins, expectedAnswer)$ toma el set de monedas a probar y el valor máximo acumulado esperado por Sophia. Luego corre el algoritmo de programación dinámica con el set de monedas y compara si el resultado obtenido es igual al esperado.

Algunos de los sets y sus resultados esperados, utilizados para testear el código, fueron

```
1  # set, resultado esperado
2  [1, 10, 5], 6
3  [3, 7, 2, 8, 5], 10
4  [1, 2, 3, 4, 5, 6], 12
5  [1, 2, 3, 4, 5, 6, 7, 8], 20
6  [20, 30, 2, 10], 40
7  [1, 2, 3, 4, 5, 6, 7], 16
8  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 30
9  [10, 9, 8, 7, 6, 5, 4, 3, 2, 1], 30
10 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 5
11 [2, 4, 6, 8, 10, 12, 14, 16, 18, 20], 60
12 [1, 2, 4, 8, 16, 32, 64, 128, 256, 512], 682
13 [10, 20, 30, 40, 50, 60, 70, 80, 90, 100], 300
```

Por otro lado, se testeó el algoritmo con las pruebas proporcionadas por la cátedra, para los cuales el algoritmo se probo exitoso.

2.4. Conclusión

Hemos visto que un algoritmo de programación dinámica ofrece una variante de juego con mayor profundidad táctica, en comparación al rendimiento lineal y predecible del algoritmo Greedy implementado en la primera parte. Sin embargo, el juego sigue estando sujeto a quien tenga la suerte de empezar, por lo que Sophia nunca tendrá el control total de las partidas.

3. Cambios

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 cuánto notaron que siempre ganaba quién 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 (uno de los fundadores de Ediciones de Mente) 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 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. 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 directamente). Debemos ubicar todos los barcos de tal manera que se cumplan todos los requisitos.

3.1. "$BatallaNaval$" $\in NP$.

Para demostrar que "$BatallaNaval$" $\in NP$ debemos probar que una solución de "$BatallaNaval$" puede certificarse en tiempo polinomial, para ello propondremos un algoritmo que efectivamente lo haga.

3.1.1. Certificador

El programa certificador recibe:

- *tablero* , una matriz de $n \times m$ casilleros. Cada casillero tendrá un valor 0 si está vacío o 1 si está ocupado por un barco;
- *barcos* , una lista de k barcos, donde el barco en la posición i tiene b_i de largo;
- *restricciones_filas* , una lista de restricciones para las filas, donde la restricción en la posición j corresponde a la cantidad de casilleros a ser ocupados en la fila j;
- *restricciones_columnas*, una lista de restricciones para las columnas, donde la restricción en la posición h corresponde a la cantidad de casilleros a ser ocupados en la columna h.

Y verifica que se cumplan las restricciones correspondientes de la siguiente manera:

```
def certificador_batalla_naval (tablero, barcos, restricciones_filas,
    restricciones_columnas):

    n = len(tablero)
    m = len(tablero[0])

    # 1 - Verificamos que las posiciones ocupadas por fila sean las
    correspondientes.

    for i in range(n):
        if ( sum(tablero[i]) != restricciones_filas[i] ):
            return False

    # 2 - Verificamos que las posiciones ocupadas por columna sean las
    correspondientes.

    for j in range(m):
        if ( sum(tablero[i][j] for i in range(n)) != restricciones_columnas[j] ):
            return False

```

```
    # Identificamos los barcos.

    visitados = [[False] * m for _ in range(n)]
    barcos_identificados = []

    def dfs(x, y, barco_actual):
        if ( not ( (0 <= x < n) and (0 <= y < m)) or visitados[x][y] or (tablero[x
    ][y] == 0) ):
            return

        visitados[x][y] = True
        barco_actual.append((x, y))

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            dfs(x + dx, y + dy, barco_actual)

    for i in range(n):
        for j in range(m):
            if ( (tablero[i][j] != 0) and not visitados[i][j] ):
                barco_actual = []
                dfs(i, j, barco_actual)
                barcos_identificados.append(barco_actual)


    # Identificamos los tama os de los barcos encontrados

    tama os_encontrados = []

    for barco in barcos_identificados:

        #3 - Verificamos que el ancho del barco es de un casillero

        filas = {x for x, y in barco}
        columnas = {y for x, y in barco}

        if ( (len(filas) > 1) and (len(columnas) > 1) ):
            return False

        tama os_encontrados.append(len(barco))


        #4 - Verificamos que no haya adyacencia con otros barcos
        for x, y in barco:
            for dx, dy in [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1),
    (1, 0), (1, 1)]:

                adyacente_x = x + dx
                adyacente_y = y + dy

                if ( (0 <= adyacente_x < n) and (0 <= adyacente_y < m) and (tablero
    [adyacente_x][adyacente_y] == 1) ):
                    if (adyacente_x, adyacente_y) not in barco:
                        return False


    # 5 - Verificamos que los barcos identificados coincidan con los barcos dados

    if sorted(tama os_encontrados) != sorted(barcos):
        return False

    return True
```

3.1.2. Complejidad

Ahora analizamos la complejidad del certificador propuesto:

- Verificación de restricciones de filas: $\mathcal{O}(n * m)$, ya que se procesan las n filas, cada una con m elementos.

- Verificación de restricciones de columnas: $\mathcal{O}(n*m)$, ya que se procesan las m columnas, cada una con n elementos.
- Identificar barcos:
  - Inicializar matriz de visitados es $\mathcal{O}(n*m)$, ya que se recorre todo el tablero;
  - El doble bucle es $\mathcal{O}(n*m)$, ya que cada celda del tablero es procesada una vez.
  - DFS es $\mathcal{O}(a)$, siendo a número total de celdas ocupadas por barcos ($a \leq n*m.$)
- Verificar forma y adyacencia: $\mathcal{O}(a)$, siendo a número total de celdas ocupadas por barcos ($a \leq n*m.$), porque revisamos todas las celdas ocupadas.
- Verificar que los barcos existen: $\mathcal{O}(k*log(k))$, ya que esa es la complejidad de ordenar los arreglos y compararlos.

Finalmente, la complejidad del certificador es $\mathcal{O}(n*m)$, ya que $\mathcal{O}(n*m)$ es cota superior tanto de $\mathcal{O}(a)$ como de $\mathcal{O}(k*log(k))$.

Confirmamos así que "$BatallaNaval$" $\in NP$ ya que se pueden certificar sus soluciones en tiempo polinomial.

3.2. "$BatallaNaval$" $\in NP-C$.

Para demostrar que "$BatallaNaval$" $\in NP-C$ hay que demostrar que es NP y NP-Hard. En la sección anterior hemos demostrado la primera condición y la segunda la demostraremos realizando la reducción polinomial de un problema NP-Completo a una instancia de Batalla Naval .

3.2.1. 3-Partition

Hemos decidido realizar la reducción $3-Partition \leq BatallaNaval$ ya que 3-Partition en su versión unaria es un problema NP-Completo.
El problema 3-Partition consiste en decidir si dado un conjunto S de $n = 3*m$ elementos, puede ser dividido en m subconjuntos tal que la suma de sus elementos sea la misma para todos.
Michael R. Garey y David S. Johnson, en la publicación "Complexity results for multiprocessor scheduling under resource constraints", fueron los primeros en probar que el problema de la 3-partición es NP-completo.
Nosotros realizaremos una demostración a través de la reducción $2-Partition \leq 3-Partition$, problema trabajado en clase. Ambos problemas reciben un conjunto y devuelven sub-sets con la misma suma, la diferencia es que en $2-Partion$ se deben generar dos sub-sets, mientras que en $3-partition$ lo que importa son la cantidad de elementos en el sub-set (deben ser tres) y no la cantidad de sub-sets.

Consideremos que la entrada de $2-Partition$ es un conjunto S de $n = 2*m$ elementos.
Se propone la siguiente transformación para la entrada:

- Se agregan a S los elementos ficticios necesarios para que la cantidad de elementos sea $n = 3*m$ (usamos ceros para no afectar la suma) y para que, al mismo tiempo, $n \div 3$ sea par.

La transformación es polinomial ya que la transformación de la entrada es $\mathcal{O}(1)$. Si hay solución para $2-Partition$ hay solución para $3-Partition$ ya que agregamos los elementos ficticios necesarios para que eso sea posible. Y si hay solución para $3-Partition$ hay solución para $2-Partition$ ya que al asegurarnos de que $n \div 3$ sea par en la transformación de la entrada se pueden combinar la mitad de los sub-sets de la solución de $3-Partition$ en un set S_1 y la otra mitad en otro set S_2 que tendrán la misma suma obteniendo una solución para $2-Partition$. Por lo tanto, demostramos que $3-Partition \in NP-C$

3.2.2. Entradas y Salidas

Para $3-Partition$ tenemos:

- Entrada: E, una lista de n elementos.
- Salida: S, una lista de m subconjuntos de 3 elementos tal que la suma de los mismos es igual para todos los subconjuntos.

Para $BatallaNaval$ tenemos:

- Entradas:
  - T , una matriz de $x \times w$ casilleros;
  - B , una lista de k barcos, donde el barco en la posición i tiene b_i de largo;
  - F , una lista de restricciones para las filas, donde la restricción en la posición j corresponde a la cantidad de casilleros a ser ocupados en la fila j;
  - C, una lista de restricciones para las columnas, donde la restricción en la posición h corresponde a la cantidad de casilleros a ser ocupados en la columna h.
- Salida:
  - Z , una matriz de $x \times w$ casilleros. Cada casillero tendrá un valor 0 si está vacío o 1 si está ocupado por un barco, cumpliendo con las restricciones impuestas;

Para la reducción se considerarán los problemas como problemas de decisión, es decir, ambos problemas deberán devolver True o False dependiendo si hay solución para el problema o no. Sin embargo, consideramos importante mencionar las salidas de los problemas de optimización para entender mejor por qué si hay solución para uno de los problemas hay solución para el otro.

3.2.3. $3-Partition \leq BatallaNaval$

Proponemos la siguiente transformación para la entrada:

- $x = ((n/3) * 2) - 1$. Una fila para cada subconjunto a crear más las necesarias para asegurar que se cumpla la condición de no adyacencia de $BatallaNaval$.
- $w = [\sum_{i=0}^{n} e_i] + n - 1$. Una columna para cada posición a ocupar por cada elemento de E más las necesarias para asegurar que se cumpla la condición de no adyacencia de $BatallaNaval$.
- Creamos una matriz T de dimensiones xxw y la inicializamos.
- Creamos la lista F con x elementos y la inicializamos intercalando $[\sum_{i=0}^{n} e_i]/(n/3)$ en una posición y 0 en la siguiente.
- Creamos la lista C con w elementos y la inicializamos cargando un 1 en las posiciones correspondientes a e_i, un 0 en la siguiente posición a modo de separación y luego un 1 en las posiciones correspondientes a e_{i+1}. Así sucesivamente hasta cargar las posiciones correspondientes a los n elementos.
- Creamos la lista B y la inicializamos con los elementos de E.

3.2.4. Complejidad

Analizamos la complejidad de la transformación propuesta:

- Calcular x tiene una complejidad $\mathcal{O}(1)$.

- Calcular w tiene una complejidad de $\mathcal{O}(n)$, ya que hacemos una sumatoria de todos los elementos.
- Inicializar la matriz T tiene una complejidad $\mathcal{O}(x * w)$, ya que recorremos toda la matriz.
- Crear la lista F tiene una complejidad $\mathcal{O}(n)$, ya que hacemos una sumatoria de todos los elementos.
- Crear la lista C tiene una complejidad $\mathcal{O}(n)$, ya que cargamos todos los elementos en la lista.
- Crear la lista B tiene una complejidad $\mathcal{O}(n)$, ya que cargamos todos los elementos en la lista.

Considerando que x y w son proporcionales a n, la complejidad de $\mathcal{O}(x * w)$ es equivalente a $\mathcal{O}(n^2)$. Siendo esta la cota superior, la complejidad de la transformación es polinomial: $\mathcal{O}(n^2)$.

3.2.5. Ejemplo

Proponemos un ejemplo ilustrativo:
Dado $E = [11, 11, 11, 11, 1, 1, 1, 1, 111, 111, 111, 111]$

Realizamos la transformación:

- $x = ((12/3) * 2) - 1 = 7$
- $w = 24 + 12 - 1 = 35$
- Creamos una matriz T de dimensiones 7x35.
- Creamos la lista $F = [6, 0, 6, 0, 6, 0, 6]$
- Creamos la lista $C = [1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1]$
- Creamos la lista B y la inicializamos con los elementos de E.

Al resolver $BatallaNaval$ obtendremos la siguiente matriz:

$$\begin{pmatrix}
1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0
\end{pmatrix}$$

Recorriendo las filas podríamos obtener los sub-sets: $S = [[11, 1, 111], [11, 1, 111], [11, 1, 111], [11, 1, 111]]$
Siendo S la solución para $3 - Partition$.

Como se puede observar si hay una solución para $BatallaNaval$ entonces habrá una solución para $3 - partition$, ya que:

- Las filas de separación evitarán que un elemento esté en más de un sub-set, ya que se podrán ubicar únicamente de manera horizontal, quedando contenidos en una sola fila.
- Al tener la misma restricción para el resto de las filas nos aseguramos que la suma de los elementos en los sub-sets será igual para todos ellos.
- Las restricciones de las columnas aseguran que cada elemento se pueda ubicar una vez.

Y si hay solución para $3 - Partition$ habrá solución para $BatallaNaval$ ya que la transformación propuesta asegura que se puedan ubicar todos los elementos de forma no adyacente y cumpliendo todas las demandas solo si es que hay solución para $3 - Partition$.

Por lo tanto, queda demostrado que "$BatallaNaval$" $\in NP - C$ ya que se puede realizar la reducción $3 - Partition \leq BatallaNaval$ en tiempo polinomial.

3.3. Resolución de *BatallaNaval* por Backtracking

3.3.1. Implementación

Para la implementación de la resolución por Backtracking hacemos uso de la siguiente librería *utils.py*:

```
'''
    Pre: 'demandas' es un array con los valores de las demandas que
         tiene una fila/columna.

    Pos: Devuelve un array con cada indice i de 'demandas' tal que
         se cumple que demandas[i] >= barco_size.
'''
def obtener_candidatos_por_demanda(demandas, barco_size):
    candidatos = []
    for i, demanda in enumerate(demandas):
        if demanda >= barco_size:
            candidatos.append(i)
    return candidatos

'''
    Pre: 'columnas_disponibles' es una lista con indices de columnas
         del tablero cuya demanda es >= k.
         'f' es el indice de una fila en el tablero.

    Pos: Devuelve una lista de grupos de k puntos, todos los puntos
         que esten en un mismo grupo van a estar en la misma fila f.
         (fila f fija, columna c_i variable para cada punto de un grupo).

         Cada punto de un mismo grupo es adyacente con el anterior
         (i.e. a (f, c_i - 1)) y con el siguiente (f, c_i + 1).

         (Cada grupo de puntos representa un lugar en el tablero
          en donde podemos poner un barco de tama o k)
'''
def posiciones_para_barco_por_fila(columnas_disponibles, f, k, todas = True):
    if len(columnas_disponibles) < k:
        return []

    casilleros = []
    for c in columnas_disponibles:
        casilleros.append((f, c))

    posiciones = []
    for i in range(len(casilleros) - k + 1):
        son_consecutivos = True
        consecutivos = []
        for j in range(k):
            actual = casilleros[i + j]
            consecutivos.append(actual)
            if j >= k - 1:
                continue

            siguiente = casilleros[i + j + 1]
            if actual[1] + 1 != siguiente[1]:
                son_consecutivos = False

        if son_consecutivos:
            posiciones.append(consecutivos.copy())
```

```
54
55              if not todas:
56                  # Devolvemos una sola posicion.
57                  return posiciones
58          son_consecutivos = False
59
60      return posiciones
61
62  '''
63      Pre: 'filas_disponibles' es una lista con indices de filas del
64           tablero cuya demanda es >= k.
65           'c' es el indice de una columna en el tablero.
66
67      Pos: Devuelve una lista de grupos de k puntos, todos los puntos
68           que esten en un mismo grupo van a estar en la misma columna c.
69           (columna c fija, fila f_i variable).
70
71           Cada punto de un mismo grupo es adyacente con el anterior
72           (i.e. a (f_i - 1, c)) y con el siguiente (f_i + 1, c).
73
74           (Cada grupo de puntos representa un lugar en el tablero
75            en donde podemos poner un barco de tama o k)
76  '''
77  def posiciones_para_barco_por_columna(filas_disponibles, c, k, todas = True):
78      if len(filas_disponibles) < k:
79          return []
80
81      casilleros = []
82      for f in filas_disponibles:
83          casilleros.append((f, c))
84
85      posiciones = []
86      for i in range(len(casilleros) - k + 1):
87          son_consecutivos = True
88          consecutivos = []
89          for j in range(k):
90              actual = casilleros[i + j]
91              consecutivos.append(actual)
92              if j >= k - 1:
93                  continue
94
95              siguiente = casilleros[i + j + 1]
96              if actual[0] + 1 != siguiente[0]:
97                  son_consecutivos = False
98
99          if son_consecutivos:
100             posiciones.append(consecutivos.copy())
101
102             if not todas:
103                 # Devolvemos una sola posicion.
104                 return posiciones
105         son_consecutivos = False
106
107     return posiciones
108
109 def guardar_nuevos_candidatos(demanda, candidatos, nuevos_candidatos):
110     if len(nuevos_candidatos) == 0:
111         return
112
113     if not demanda in candidatos:
114         candidatos[demanda] = []
115
116     for lista in nuevos_candidatos:
117         candidatos[demanda].append(lista)
118
119 '''
120     Pos: Devuelve una lista de tuplas.
121
122          Cada tupla contiene informacion de la forma (demanda,
    posiciones_candidatas),
```

```
123         en donde 'demanda' es un entero relacionada con linea del tablero (fila o
124         columna) y 'posiciones_candidatas' es una lista de listas con grupos de
    puntos
125         (x_i, y_i) que podria ocupar el barco si lo ubicamos en el tablero.
126 
127         Los puntos en cada grupo de puntos son posiciones validas, es decir, no
    exceden
128         a la demanda de ninguna fila ni columna, ni son adyacentes con ningun
    casillero
129         que otro barco este ocupando en el tablero.
130 '''
131 def obtener_candidatos(tablero, demanda_filas, demanda_columnas, barco_size, todos
    = True):
132     candidatos_fila = obtener_candidatos_por_demanda(demanda_filas, barco_size)
133     posiciones_candidatas = {}
134 
135     for f in candidatos_fila:
136         columnas_disponibles = []
137         for c in range(len(demanda_columnas)):
138             if demanda_columnas[c] == 0 or tablero[f][c] != None:
139                 continue
140             elif not hay_adyacencias(f, c, tablero):
141                 columnas_disponibles.append(c)
142         demanda = demanda_filas[f]
143         nuevos_candidatos = posiciones_para_barco_por_fila(columnas_disponibles, f,
     barco_size, todos)
144         guardar_nuevos_candidatos(demanda, posiciones_candidatas, nuevos_candidatos
    )
145         if not todos and len(nuevos_candidatos) > 0:
146             return list(posiciones_candidatas.items())
147 
148     # Para que no se dupliquen los candidatos si un barco ocupa 1 casillero.
149     if (barco_size == 1):
150         return list(posiciones_candidatas.items())
151     candidatos_columna = obtener_candidatos_por_demanda(demanda_columnas,
    barco_size)
152 
153     for c in candidatos_columna:
154         filas_disponibles = []
155         for f in range(len(demanda_filas)):
156             if demanda_filas[f] == 0 or tablero[f][c] != None:
157                 continue
158             elif not hay_adyacencias(f, c, tablero):
159                 filas_disponibles.append(f)
160         demanda = demanda_columnas[c]
161         nuevos_candidatos = posiciones_para_barco_por_columna(filas_disponibles, c,
     barco_size, todos)
162         guardar_nuevos_candidatos(demanda, posiciones_candidatas, nuevos_candidatos
    )
163         if not todos and len(nuevos_candidatos) > 0:
164             return list(posiciones_candidatas.items())
165 
166     return list(posiciones_candidatas.items())
167 
168 def puedo_poner(tablero, demanda_filas, demanda_columnas, barco_size):
169     alternativas = obtener_candidatos(tablero, demanda_filas, demanda_columnas,
    barco_size, False)
170     if len(alternativas) == 0:
171         return False
172     demanda, alternativa = alternativas[0]
173     return len(alternativa[0]) > 0
174 
175 '''
176     Pos: Basandose en el estado actual del tablero y las demandas, y teniendo en
    cuenta
177          solamente a los barcos de la lista que se encuentran entre los indices
178          [i+1, final_idx] y caben en el tablero, se retorna el valor de la demanda
    total
179          que podria satisfacerse si pudieramos colocarlos a todos ellos.
180 '''
```

```
def demanda_proyectada(tablero, demanda_filas, demanda_columnas, barcos, i):
    demandas = []
    for j in range(i + 1, len(barcos)):
        barco_idx, barco_size = barcos[j]
        if puedo_poner(tablero, demanda_filas, demanda_columnas, barco_size):
            demandas.append(barco_size*2)
    return sum(demandas)

'''
    Pre: barcos es una lista de tuplas de la forma (barco_id, barco_size) ordenada de
         manera descendente seg n el valor "barco_size".

    Pos: Sea barcos[i] el barco actual, devuelve un indice de la lista en el cual haya
         otro barco con un "barco_size" distinto al barco actual.
'''
def obtener_siguiente_distinto(barcos, i):
    actual = barcos[i]
    actual_idx, actual_size = actual
    for j in range(i + 1, len(barcos)):
        siguiente_idx, siguiente_size = barcos[j]
        if siguiente_size != actual_size:
            return j
    return i + 1
```

Y la implementación:

```
from utils import *

def batalla_naval_bt(tablero, barcos, demanda_filas, demanda_columnas,
                     puestos, optimos, i):
    demanda_cumplida = sum(size * 2 for (idx, size) in puestos.keys())
    demanda_optima = sum(size * 2 for (idx, size) in optimos.keys())
    if i == len(barcos):
        if demanda_cumplida > demanda_optima:
            optimos.clear()
            optimos.update({barco: pos.copy() for barco, pos in puestos.items()})
            return True
        return False

    barco = barcos[i]
    barco_idx, barco_size = barco

    alternativas = obtener_candidatos(tablero, demanda_filas, demanda_columnas, barco_size)
    hay_alternativas = len(alternativas) > 0

    if not hay_alternativas:
        return batalla_naval_bt(tablero, barcos, demanda_filas, demanda_columnas, puestos,
                                optimos, i + 1)

    proyectada = demanda_proyectada(tablero, demanda_filas, demanda_columnas, barcos, i - 1)

    if demanda_cumplida + proyectada <= demanda_optima: # No vamos a mejorar.
        return batalla_naval_bt(tablero, barcos, demanda_filas, demanda_columnas, puestos,
                                optimos, i + 1)

    puestos[barco] = []
    demanda_cumplida = sum(size * 2 for (idx, size) in puestos.keys())

    for demanda, alternativa in alternativas:
        for casilleros in alternativa:
            colocar_barco(casilleros, tablero, barco_idx)
            actualizar_demandas(casilleros, demanda_filas, demanda_columnas, False)
            puestos[barco] = casilleros

```

```
            proyectada = demanda_proyectada(tablero, demanda_filas,
    demanda_columnas, barcos, i)
            buena_alternativa = demanda_cumplida + proyectada > demanda_optima

            if buena_alternativa:
                if batalla_naval_bt(tablero, barcos, demanda_filas,
    demanda_columnas,
                                    puestos, optimos, i + 1):
                    demanda_optima = sum(size * 2 for (idx, size) in optimos.keys()
    )

            colocar_barco(casilleros, tablero, None) # Lo sacamos del tablero.
            actualizar_demandas(casilleros, demanda_filas, demanda_columnas, True)

    del puestos[barco]
    k =  obtener_siguiente_distinto(barcos, i)

    return batalla_naval_bt(tablero, barcos, demanda_filas, demanda_columnas,
    puestos,
                            optimos, k)

def batalla_naval(tablero, barcos, demanda_filas, demanda_columnas):
    barcos_ordenados = list(enumerate(barcos))
    barcos_ordenados.sort(key=lambda x: -x[1])
    optimos = {}
    batalla_naval_bt(tablero, barcos_ordenados, demanda_filas, demanda_columnas,
                     {}, optimos, 0)
    return optimos
```

3.3.2. Sets de datos

Generamos sets de datos para corroborar la correctitud de la solución propuesta. Consideramos:

- Demandas por fila $\rightarrow F$
- Demandas por columna $\rightarrow C$
- Barcos $\rightarrow B$

- 9_2_12:
  - $F = [3, 4, 4, 7, 4, 2, 0, 5, 1]$
  - $C = [4, 5, 4, 0, 2, 2, 7, 6]$
  - $B = [2, 8, 4, 4, 5, 5, 7, 4, 3, 5, 7, 8]$
- 9_12_15:
  - $F = [3, 2, 5, 8, 0, 6, 9, 2, 6]$
  - $C = [10, 6, 4, 7, 1, 1, 9, 6, 1, 8, 0, 5]$
  - $B = [1, 8, 3, 10, 1, 9, 3, 2, 12, 10, 9, 8, 5, 11, 3]$
- 8_8_15:
  - $F = [6, 2, 0, 4, 4, 6, 0, 3]$
  - $C = [3, 4, 2, 6, 2, 1, 5, 5]$
  - $B = [1, 1, 2, 1, 8, 4, 5, 4, 5, 7, 4, 8, 8, 7, 6]$
- 8_10_15:
  - $F = [5, 5, 0, 6, 6, 4, 1, 1]$
  - $C = [3, 2, 3, 2, 2, 4, 2, 8, 9, 1]$

- $B = [9, 3, 1, 3, 1, 6, 2, 8, 1, 1, 7, 3, 5, 9, 8]$

- 12_8_12:
  - $F = [3, 2, 3, 7, 2, 0, 3, 3, 3, 6, 4, 0]$
  - $C = [9, 7, 6, 2, 9, 8, 8, 5]$
  - $B = [1, 2, 3, 7, 2, 10, 3, 8, 10, 4, 10, 1]$
- 12_13_14:
  - $F = [1, 9, 1, 7, 2, 10, 4, 1, 10, 2, 1, 4]$
  - $C = [9, 10, 4, 9, 10, 2, 12, 1, 6, 5, 8, 9, 10]$
  - $B = [11, 4, 4, 4, 13, 9, 14, 2, 1, 2, 4, 1, 5, 8]$
- 12_12_12:
  - $F = [4,10, 9, 4, 6, 2, 7, 7, 4, 0, 3, 4]$
  - $C = [6, 8, 2, 1, 2, 7, 6, 6, 2, 7, 7, 5]$
  - $B = [2, 11, 2, 4, 5, 9, 7, 8, 1, 12, 8, 9]$
- 11_8_12:
  - $F = [6, 0, 7, 6, 1, 6, 8, 7, 2, 1, 1, 3]$
  - $C = [1, 2, 4, 3, 5, 0, 5, 0]$
  - $B = [9, 2, 1, 0, 1, 3, 1, 1, 4, 8, 1, 2, 3, 3, 1, 4]$
- 10_8_15:
  - $F = [5, 2, 5, 6, 4, 5, 7, 4]$
  - $C = [8, 4, 7, 2, 1, 2, 2, 7, 1, 0, 0]$
  - $B = [1, 5, 1, 2, 8, 1, 1, 5, 9, 1, 3, 6, 4, 1, 4, 7, 1, 2, 1, 1, 1, 1, 6]$
- 10_11_12:
  - $F = [5, 1, 0, 9, 2, 4, 2, 8, 9, 1, 3, 1, 1]$
  - $C = [6, 8, 0, 8, 5, 3, 8, 4, 1, 0, 0]$
  - $B = [5, 2, 2, 1, 0, 1, 0, 1, 2, 3, 1, 2, 6, 1, 1, 7, 8]$
- 20_20_21:
  - $F = [81463048612436075505]$
  - $C = [8315794570032014410510]$
  - $B = [18711191106393711117510112 6]$
- 20_20_25:
  - $F = [8,5,6,1,6,3,1,0,2,5,4,6,1,7,7,6,4,7,2,1]$
  - $C = [7,10,10,4,5,4,0,1,0,1,3,4,7,4,0,10,10,6,4,2]$
  - $B = [4,7,11,11,6,2,3,6,2,9,5,4,7,4,9,9,10,8,7,11,3,9,11,3,8]$
- 20_23_20:
  - $F = [8,3,2,2,4,5,8,7,0,5,7,4,1,6,2,1,6,0,0,7]$
  - $C = [0,10,3,7,1,4,5,9,9,10,2,10,4,3,2,2,10,9,9,5,3,10,10]$
  - $B = [4,7,4,10,9,10,7,9,2,4,2,11,11,8,3,3,12,6,2,1]$
- 21_21_21:

- F = [7,0,7,6,2,7,7,0,5,2,8,1,2,7,6,8,4,6,6,7,3]
- C = [10,4,2,1,4,0,7,0,8,6,8,9,7,5,2,10,1,7,9,0,9]
- B = [10,3,7,2,9,12,3,11,2,5,3,6,9,11,12,2,11,9,10,5,11]

- 22_20_22:
  - F = [3,7,4,4,5,4,4,5,0,3,0,4,6,2,1,3,5,1,0,3,8,7]
  - C = [8,5,9,1,10,6,0,2,7,1,0,10,5,6,7,1,0,3,7,5]
  - B = [4,1,3,8,2,3,3,12,6,1,10,4,4,8,10,10,5,6,11,7,10,4]
- 22_22_22:
  - F = [8,0,8,0,5,5,5,8,6,3,0,4,3,1,8,4,2,4,7,5,7,6]
  - C = [8,5,0,6,10,5,4,0,6,7,7,3,2,0,5,0,7,0,2,6,5,3]
  - B = [8,12,1,11,10,7,6,7,8,11,7,9,7,2,7,11,3,1,10,9,6,6]
- 22_24_22:
  - F = [7,1,8,0,8,5,7,7,7,0,2,0,5,2,6,8,2,0,7,7,3,4]
  - C = [7,3,3,8,3,9,3,10,7,3,10,10,8,7,6,2,9,4,1,5,7,2,5,6]
  - B = [2,7,11,6,12,10,5,6,8,6,9,11,4,4,8,12,11,5,3,7,2,6]
- 23_23_20:
  - F = [0,0,1,6,0,5,8,7,5,1,0,0,7,8,2,6,4,3,7,0,7,2,4]
  - C = [5,7,8,9,8,0,7,10,5,3,9,9,9,3,0,9,4,10,2,2,4,4,2]
  - B = [2,12,6,12,11,8,12,12,10,11,3,1,3,6,9,4,2,1,2,5]
- 23_23_23:
  - F = [0,5,5,3,5,0,0,1,1,8,7,8,4,7,1,8,1,8,7,5,7,2,8]
  - C = [9,3,6,1,6,9,0,8,9,2,3,9,6,2,8,10,10,9,8,10,7,3,6]
  - B = [1,11,2,10,1,3,10,2,3,6,2,7,12,3,12,9,7,12,12,12,1,10,7]
- 24_24_24:
  - F = [6,7,0,6,1,0,7,4,1,0,8,2,5,6,0,0,6,0,5,7,5,7,5,2]
  - C = [10,2,6,3,5,9,5,0,9,10,6,8,9,9,9,9,8,6,6,3,10,2,10,2]
  - B = [7,11,11,9,7,11,2,7,12,9,4,10,4,3,6,7,1,6,4,5,12,1,1,5]

3.3.3. Mediciones

Se realizaron mediciones sobre los tiempos de resolución con base en los casos de prueba propuestos por la cátedra:

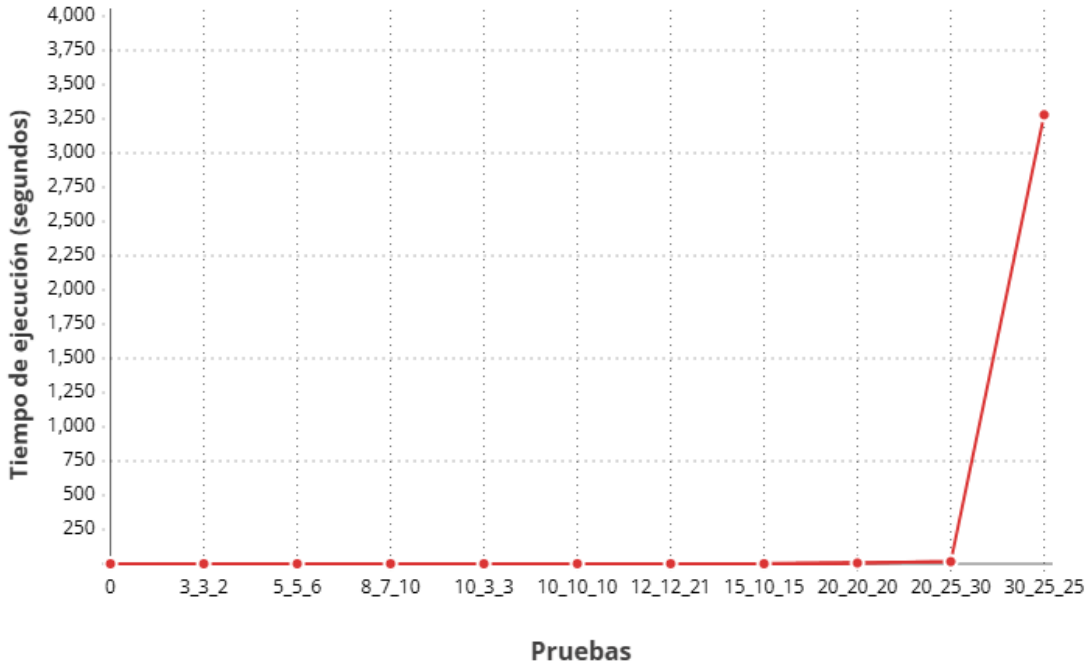

Figura 5: tiempos de resolución de los test de la cátedra

Agregamos otro gráfico sin el test 30_25_25 por motivos ilustrativos:

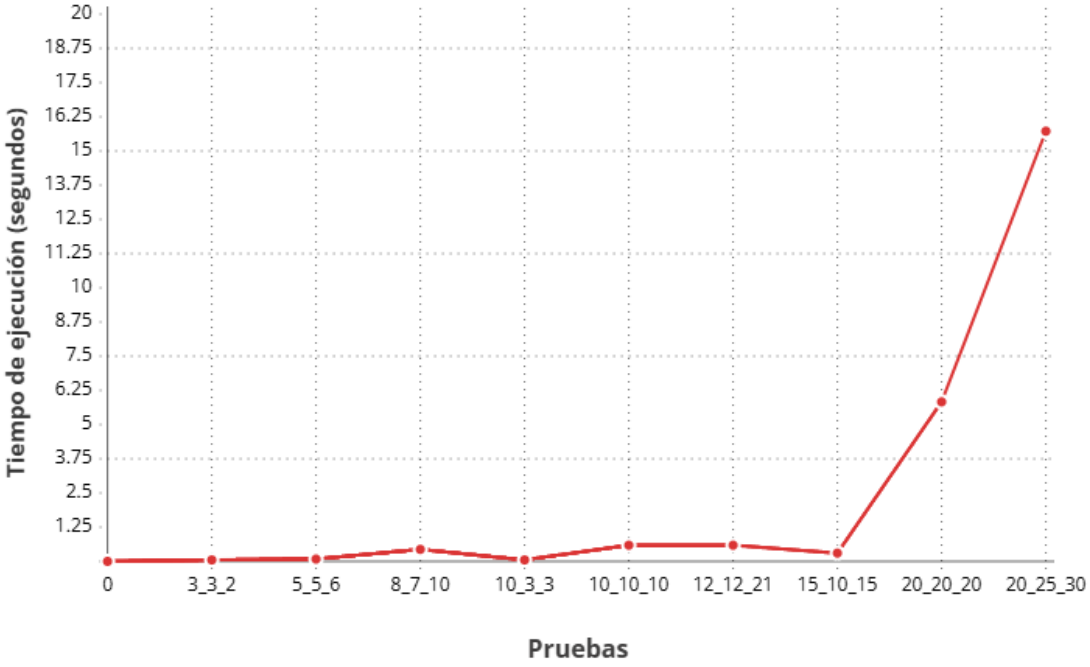

Figura 6: tiempos de resolución sin el test 30_25_25

Y sobre los sets de datos creados en la subsección anterior

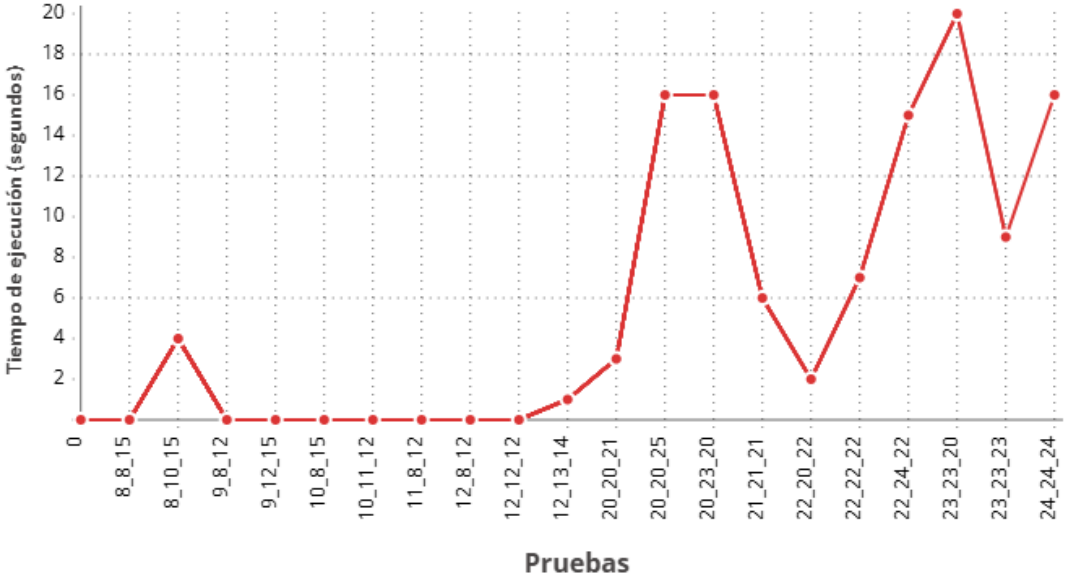

Figura 7: tiempos de resolución sets propios

3.4. Resolución de *BatallaNaval* por Aproximación

En esta sección analizaremos e implementaremos el algoritmo de aproximación propuesto por John Jellicoe:

- Ir a fila/columna de mayor demanda.
- 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.

3.4.1. ¿Es buena aproximación?

Este algoritmo es útil como una aproximación al problema de la batalla naval, busca estratégicamente acercarse lo más posible a la solución óptima, intentando colocar en las filas/columnas de mayor demanda a los barcos de mayor tamaño.

3.4.2. Implementación

La implementación propuesta hace uso de la librería *utils.py*:

```
FILA = 0
COLUMNA = 1

import sys

common_directory_path = '../common/'
sys.path.append(common_directory_path)
from game_logic import *

def ordenar_barcos(barcos):
    referencia_barcos = list(enumerate(barcos))
    referencia_barcos.sort(key=lambda x: -x[1])
    return referencia_barcos

def referenciar_demandas(demanda_filas, demanda_columnas):
    referencias = {}
    for idx, demanda_fila in enumerate(demanda_filas):
        referencias[(idx, FILA)] = demanda_fila
    for idx, demanda_columna in enumerate(demanda_columnas):
        referencias[(idx, COLUMNA)] = demanda_columna
    return referencias

def obtener_alternativa(tablero, demandas, tipo_demanda, demanda_idx, barco_size):
    alternativa = []
    consecutivos = 0
    idx_inicial = 0
    idx_final = None

    for x in range(len(demandas)):
        if demandas[x] == 0:
            idx_inicial = x + 1
            consecutivos = 0
            continue

        if (tipo_demanda == COLUMNA and hay_adyacencias(demanda_idx, x, tablero)) \
        or (tipo_demanda == FILA and hay_adyacencias(x, demanda_idx, tablero)):
            idx_inicial = x + 1
            consecutivos = 0
            continue

        consecutivos += 1
        if consecutivos == barco_size:
```

```
            idx_final = x
            break

    if idx_final is not None:
        for x in range(idx_inicial, idx_final + 1):
            if tipo_demanda == COLUMNA:
                alternativa.append((demanda_idx, x))
            elif tipo_demanda == FILA:
                alternativa.append((x, demanda_idx))

    return alternativa
```

La implementación propuesta:

```
from utils import *

def aproximacion_batalla_naval(tablero, demanda_filas, demanda_columnas, barcos):
    barcos_ordenados = ordenar_barcos(barcos)
    puestos = {}
    demandas = referenciar_demandas(demanda_filas, demanda_columnas)
    while (len(demandas) > 0 and len(puestos) < len(barcos)):
        hubo_inserciones = False
        (idx, tipo_demanda), demanda = max(demandas.items(), key=lambda x: x[1])
        for barco in barcos_ordenados:
            barco_idx, barco_size = barco
            if (barco_size > demanda or barco in puestos):
                continue
            if tipo_demanda == FILA:
                alternativa = obtener_alternativa(tablero, demanda_columnas,
    COLUMNA, idx, barco_size)
            else:
                alternativa = obtener_alternativa(tablero, demanda_filas, FILA, idx
    , barco_size)
            if (len(alternativa) == 0):
                continue
            # -------------------- Colocamos un barco --------------------
            colocar_barco(alternativa, tablero, barco_idx)
            actualizar_demandas(alternativa, demanda_filas, demanda_columnas)
            puestos[barco] = alternativa
            hubo_inserciones = True
            # ------------------- Actualizamos demandas -------------------
            demandas[(idx, tipo_demanda)] -= barco_size
            if tipo_demanda == FILA:
                disminuidos_idx = list(map(lambda x: x[1], alternativa))
                disminuidos = [(x, COLUMNA) for x in disminuidos_idx]
            else:
                disminuidos_idx = list(map(lambda x: x[0], alternativa))
                disminuidos = [(x, FILA) for x in disminuidos_idx]
            for disminuido in disminuidos:
                if disminuido in demandas:
                    demandas[disminuido] -= 1
            break
        if not hubo_inserciones:
            del demandas[(idx, tipo_demanda)]
    return puestos
```

3.4.3. Análisis de complejidad

Para el análisis considerar:

- k, es la cantidad de barcos.
- m, es la cantidad de filas
- n, es la cantidad de columnas del tablero.

- d, es el tamaño del diccionario de demandas, cuyo valor varía iteración tras iteración. Al inicio del análisis es de tamaño $n + m$, pero su valor irá disminuyendo de 1 en 1 hasta llegar al final del análisis teniendo un tamaño 0.

Complejidad de las operaciones:

- Buscar el máximo en el diccionario demandas: $\mathcal{O}(d)$.
- Para la función *obtener_alternativa*, si se está analizando la demanda que tiene una fila, entonces su complejidad va a ser $\mathcal{O}(n)$. Por otro lado, si se analiza la demanda que tiene una columna, entonces esta función tendrá una complejidad $\mathcal{O}(n)$.

Consideramos importante mencionar antes del análisis que, en el algoritmo propuesto, decidimos almacenar juntas en una misma estructura de tamaño $n + m$ a las demandas de filas y columnas. Para poder diferenciar el tipo de demanda definimos dos constantes, $FILA$ y $COLUMNA$, y se las asignamos a cada demanda según corresponda.
La estructura elegida fue un diccionario. Cada clave es una tupla de dos elementos, siendo el primero el índice de una fila/columna y el segundo una referencia constante a esta, el valor que va a tener asociado es el de su demanda.
El ciclo *while* tiene una complejidad $\mathcal{O}(n + m)$, ya que, en el peor de los casos, si no pudimos poner todos los barcos o pudimos ponerlos a todo pero luego de analizar casi todas las demandas, entonces habremos entrado al ciclo cerca de unas $n + m$ veces.
El análisis de la complejidad total del algoritmo puede explicarse en dos partes:

- Entre todas las veces que entramos en el ciclo *while*, n de ellas realizaran, a lo sumo, k llamados a la función *obtener_alternativa*, la cual en estos n casos tendrá una complejidad $\mathcal{O}(m)$, por lo que la complejidad total de esta primera parte es $\mathcal{O}(n * m * k)$.
- Las m veces restantes (de las $m+n$ totales) se llama a lo sumo k veces a *obtener_alternativa*, que en estos casos tiene una complejidad $\mathcal{O}(n)$, dando así una complejidad total de $\mathcal{O}(m*n*k)$ para esta segunda parte.

Finalmente, la complejidad del algoritmo es $\mathcal{O}(m * n * k)$, ya que es la cota superior.

3.4.4. Cálculo de la cota del algoritmo.

Sean:

- I una instancia del problema $BatallaNaval$.
- $A(I)$ una solución del algoritmo de aproximación.
- $z(I)$ una solución óptima.

Vamos a buscar obtener una cota inferior de forma empírica relacionando ambos algoritmos.

Para el siguiente análisis elegimos arbitrariamente dos instancias particulares:

- $I_1 =$
  - $F = [1, 4, 4, 4, 3, 3, 4, 4]$
  - $C = [6, 5, 3, 0, 6, 3, 3]$
  - $B = [2, 1, 2, 2, 1, 3, 2, 7, 7, 7]$
- $I_2 =$
  - $F = [5, 0, 0, 6, 2, 1, 6, 3, 3, 1, 2, 4, 5, 5, 2, 5, 4, 0, 4, 5]$
  - $C = [0, 5, 5, 0, 6, 2, 2, 6, 2, 1, 3, 1, 2, 3, 1, 4, 5, 2, 1, 6]$

- $B = [5, 5, 6, 5, 1, 3, 1, 5, 1, 1, 1, 1, 2, 3, 1, 1, 6, 7, 7, 4]$

, Que se corresponden a los sets de datos 8_7_10 y 20_20_20 provistos por la cátedra.
Para calcular la cota tomamos como referencia a la instancia I_1 y generamos de manera aleatoria sets de datos de proporciones similares, obteniendo los siguientes resultados.

I	$z(1)$	$A(I)$	$\frac{A(I)}{z(I)}$
8_7_10	26	26	1
8_8_15	38	32	0. 84
8_10_15	54	40	0. 74
9_8_12	38	30	0. 79
9_12_15	44	44	1
10_8_15	52	44	0. 85
10_11_12	66	36	0. 55
11_8_12	26	26	1
12_8_12	46	32	0. 70
12_12_12	62	42	0. 68
12_13_14	88	76	0. 86

Nos quedamos con $r_{(A)} \leq 0{,}5$, que es un valor cercano al obtenido en el peor caso para la relación $\frac{A(I)}{z(I)}$. Podríamos decir entonces que A es una 2 aproximaciones.

Volvimos a tomar mediciones con volúmenes de datos más grandes para verificarlo.
Para ello, como I_2 contiene un poco más del doble de filas, columnas y barcos que I_1, decimos considerarla como un "volumen inmanejable de datos" y medimos otra vez con base en ello.

I	$z(1)$	$A(I)$	$\frac{A(I)}{z(I)}$
20_20_20	140	86	0. 61
20_20_21	84	84	1
20_20_25	132	120	0. 91
20_23_20	106	106	1
21_21_21	114	114	1
22_20_22	138	114	0. 83
22_22_22	86	72	0. 84
22_24_22	158	114	0. 72
23_23_20	86	70	0. 81
23_23_23	110	96	0. 87
24_24_24	140	84	0. 60

3.5. Resolución de *BatallaNaval* por Greedy

Proponemos un algoritmo que resuelve el problema *BatallaNaval* usando la metodología Greedy.

3.5.1. Implementación

Recordemos, una vez más, que la entrada del problema consiste en:

- B , una lista de k barcos, donde el barco en la posición i tiene b_i de largo;
- F , una lista de demanda para las filas, donde el valor en la posición j corresponde a la cantidad de casilleros a ser ocupados en la fila j;

- C, una lista de demanda para las columnas, donde el valor en la posición h corresponde a la cantidad de casilleros a ser ocupados en la columna h.

Creamos la siguiente implementación en Python:

```
1
2  VERTICAL = 'V'
3  HORIZONTAL = "H"
4
5  #Verifica que la pocision dada y sus posiciones adyacentes esten vacias
6  def posicion_vacia (i , j , tablero):
7      for dx, dy in [(0, 0), (-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1),
       (1, 0), (1, 1)]:
8
9          adyacente_x = i + dx
10         adyacente_y = j + dy
11
12         if ( (0 <= adyacente_x < len(tablero)) and (0 <= adyacente_y < len(tablero
       [0])) and (tablero[adyacente_x][adyacente_y] != 0) ):
13             return False
14
15     return True
16
17 # Valida si se puede ubicar un barco en la posicion y direccion recibidas,
       respetando los limites del tablero, cumpliendos las restricciones del mismo y
       evitando adyacencias con otros barcos
18
19 def se_puede_ubicar(i, j, longitud, direccion , tablero, demandas_filas ,
       demandas_columnas):
20
21     if (direccion == HORIZONTAL) :
22
23         # Verificamos que el barco no se sale del tablero
24         if ( (j + longitud) > len(tablero[0]) ):
25             return False
26
27         # Verificamos que los casilleros a ocupar est n libres, no tengan barcos
       adyacentes y cumplan las demandas de las columnas
28         for col in range(j, (j + longitud)):
29             if ((not posicion_vacia(i , col , tablero)) or (demandas_columnas[col]
       < 1)):
30                 return False
31
32     else:
33
34         # Verificamos que el barco no se sale del tablero
35         if ((i + longitud) > len(tablero)):
36             return False
37
38         # Verificamos que los casilleros a ocupar est n libres, no tengan barcos
       adyacentes y cumplan las demandas de las filas
39         for fila in range(i, i + longitud):
40             if ((not posicion_vacia(fila , j , tablero)) or (demandas_filas[fila] <
        1)):
41                 return False
42
43     return True
44
45 # Ubica un barco en el tablero y actualiza las demandas de filas y columnas.
46 def ubicar_barco(i, j, longitud, direccion , tablero , demandas_filas ,
       demandas_columnas):
47     if (direccion == HORIZONTAL):
48         for col in range(j, j + longitud):
49             tablero[i][col] = 1
50             demandas_filas[i] -= 1
51             demandas_columnas[col] -= 1
52     else:
53         for fila in range(i, i + longitud):
54             tablero[fila][j] = 1
55             demandas_filas[fila] -= 1
```

```
            demandas_columnas[j] -= 1


def batalla_naval_greedy (demandas_filas, demandas_columnas, barcos):

    demanda_cumplida = 0
    demanda_total = sum(demandas_filas) + sum(demandas_columnas)
    tablero = [[0]*len(demandas_columnas) for _ in range(len(demandas_filas))]

    # Ordenamos los barcos de mayor a menor longitud
    barcos.sort(reverse=True)


    # Intentamos ubicar cada barco en todos los casilleros del tablero y  en ambos
    sentidos

    for barco in barcos:
        colocado = False

        for fila in range(len(tablero)):
            for col in range(len(tablero[0])):

                if ((demandas_filas[fila] >= barco) and (demandas_columnas[col] >=
    1) and (se_puede_ubicar(fila, col, barco, HORIZONTAL, tablero, demandas_filas ,
     demandas_columnas))):
                    ubicar_barco(fila, col, barco, HORIZONTAL, tablero,
    demandas_filas, demandas_columnas)
                    colocado = True
                    break
                elif ((demandas_filas[fila] >= 1) and (demandas_columnas[col] >=
    barco) and (se_puede_ubicar(fila, col, barco, VERTICAL, tablero, demandas_filas
     , demandas_columnas))):
                    ubicar_barco(fila, col, barco, VERTICAL, tablero,
    demandas_filas, demandas_columnas)
                    colocado = True
                    break

            if colocado:
                break


    demanda_incumplida = sum(demandas_filas) + sum(demandas_columnas)
    demanda_cumplida = demanda_total - demanda_incumplida

    return tablero , demanda_total, demanda_cumplida , demanda_incumplida
```

3.5.2. Análisis de complejidad

Ahora analizamos la complejidad de la solución propuesta:

- La función *posicion_vacia* tiene una complejidad de $\mathcal{O}(1)$, ya que se itera una cantidad de veces constante de 9 veces.
- La función *se_puede_ubicar* tiene una complejidad de $\mathcal{O}(l)$ siendo L la longitud recibida por parámetro, ya que itera sobre esa cantidad de posiciones.
- La función *ubicar_barco* tiene una complejidad de $\mathcal{O}(l)$, por el mismo motivo.
- En la función *batalla_naval_greedy*:
  - Ordenar los barcos tiene una complejidad de $\mathcal{O}(k * log(k)$
  - Se itera sobre los k barcos y las $n * m$ posiciones del tablero. Además, por cada posición se llama a *se_puede_ubicar* y en el peor de los casos también a *ubicar_barco* por lo que resulta una complejidad $\mathcal{O}(k * n * m * l)$

Finalmente, la complejidad total del algoritmo es $\mathcal{O}(k*n*m*l)$, ya que es cota superior respecto a $\mathcal{O}(k*log(k)$.

3.5.3. Mediciones

Realizamos mediciones sobre los tiempos de resolución con base en los casos de prueba propuestos por la cátedra:

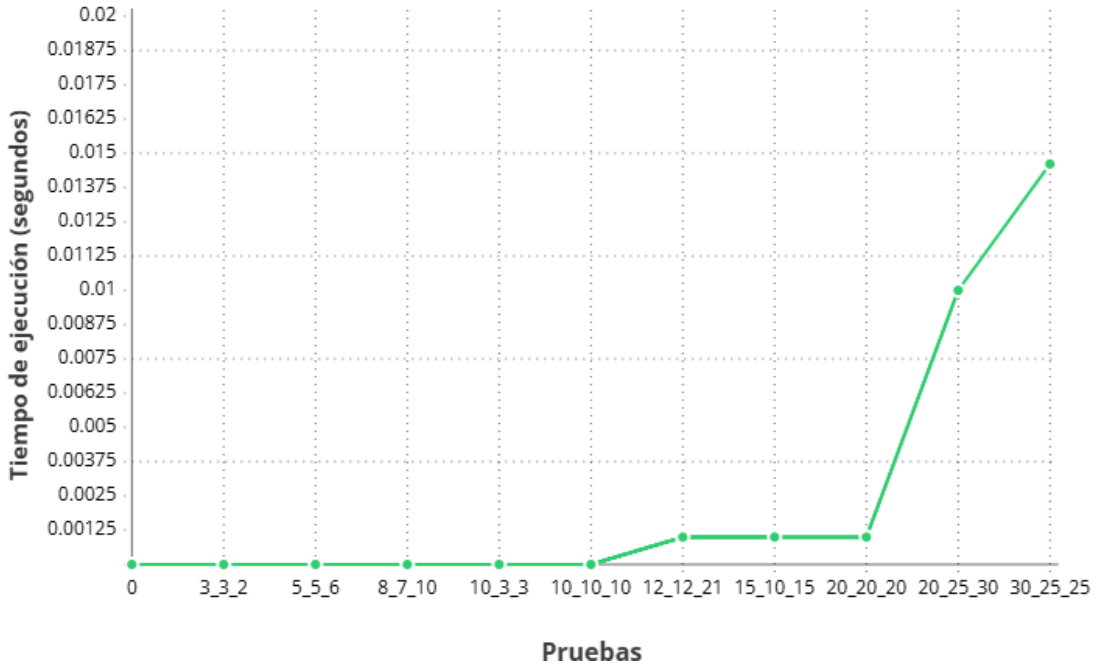

Figura 8: tiempos de resolución

Realizando una comparación con los tiempos obtenidos con la resolución por Backtracking (ver punto 3) tenemos:

Figura 9: Comparación de tiempos Greedy vs. Backtracking

Agregamos otro gráfico sin el test 30_25_25 por motivos ilustrativos:

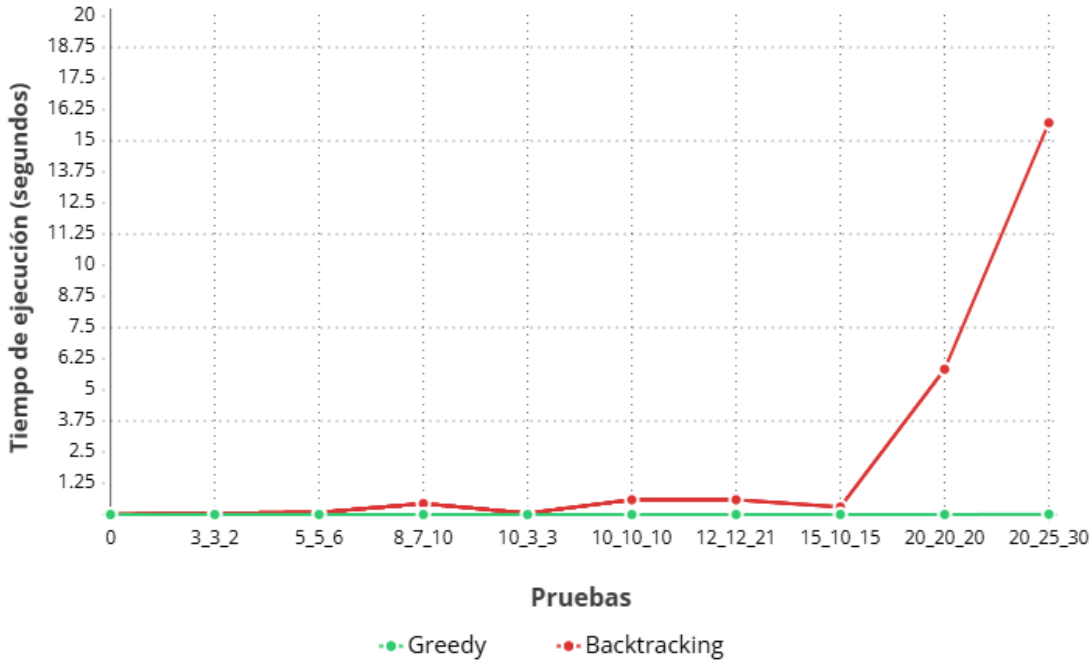

Figura 10: Comparación de tiempos Greedy vs. Backtracking, sin el test 30_25_25

3.5.4. Conclusiones

Consideramos de interés esta resolución ya que, al hacer la comparación, podemos observar las ventajas y desventajas de las distintas metodologías. El algoritmo greedy es más rápido, pero no siempre llega a la solución óptima. Por otro lado, el algoritmo de backtracking es más lento, pero siempre llega a la solución óptima.

3.6. Concluciones

Es muy importante conocer las distintas metodologías de resolución, sus ventajas y sus desventajas para poder elegir la que mejor se adapte a nuestras necesidades. Los algoritmos Greedy y el de aproximación son notablemente más rápidos que el de backtracking, pero este último brinda soluciones óptimas. Usar estos algoritmos menos precisos puede ser útil para establecer rápidamente cotas inferiores en casos de prueba grandes, mientras que el backtracking nos sería más atractivo si lo que buscamos es la solución óptima precisa.

4. Observaciones

Cada sección del trabajo práctico fue implementada por separado y cuenta con su propio README.md con especificaciones para su ejecución.

5. Correcciones

5.1. Introducción y primeros años

5.1.1. Demostración de optimalidad

Para probar la optimalidad del algoritmo hay que probar que si se sigue la regla greedy propuesta (Sophia siempre empieza y, en cada turno, siempre elige la moneda de mayor valor, Mateo siempre elige la de menor valor) entonces Sophia nunca pierde la partida. Para esto realizaremos una demostración por inducción:

Caso base:

Definimos los casos base:

- Si $n = 1$:
  - Sophia toma la única moneda disponible y. por lo tanto, gana.
- Si $n = 2$:
  - Sophia elige la moneda de mayor valor.
  - Mateo elige la moneda restante (por ende la de menor valor).
  - Si las monedas son iguales empatan, si no lo son Sophia gana. Por lo tanto, Sophia nunca pierde.

Hipótesis de Inducción:

Para cualquier cantidad de monedas n, la estrategia greedy garantiza que Sophia obtiene al menos la misma cantidad que Mateo. Es decir, Sophia nunca pierde.

Paso Inductivo:

Debemos demostrar que, si la hipótesis es cierta para n, entonces también lo es para $n + 1$.
Si tenemos $n + 1$ monedas en la fila:

- Sophia juega primero, eligiendo la moneda de mayor valor entre la primera y la última.
- Mateo juega después, eligiendo la moneda de menor valor entre la primera y la última.
- Ahora quedan $n - 1$ monedas, que es un subconjunto del caso de n monedas que asumimos en la hipótesis de inducción, por lo que la misma lógica sigue aplicando: si Sophia ganaba o empataba en n monedas, entonces en $n - 1$ su situación no puede ser peor.

En conclusión, si en cada turno Sophia obtiene al menos la misma cantidad que Mateo, entonces al finalizar el juego tendrá una cantidad mayor o igual que él, por lo que nunca pierde. Por lo tanto, la estrategia greedy elegida es óptima.

5.1.2. Ajuste de medición

Se realizaron los ajustes necesarios en las mediciones:

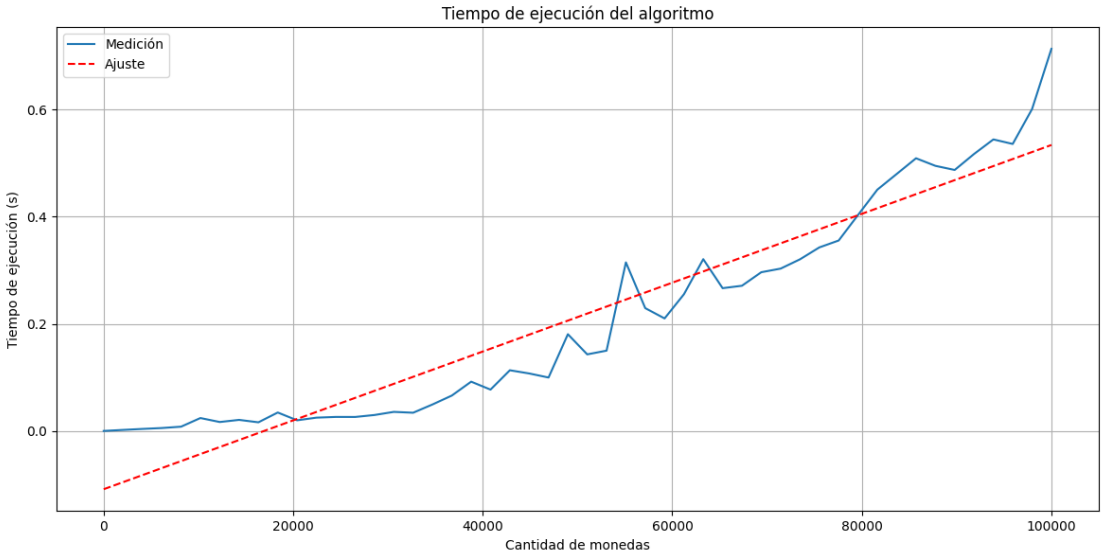

Figura 11: tiempos de resolución

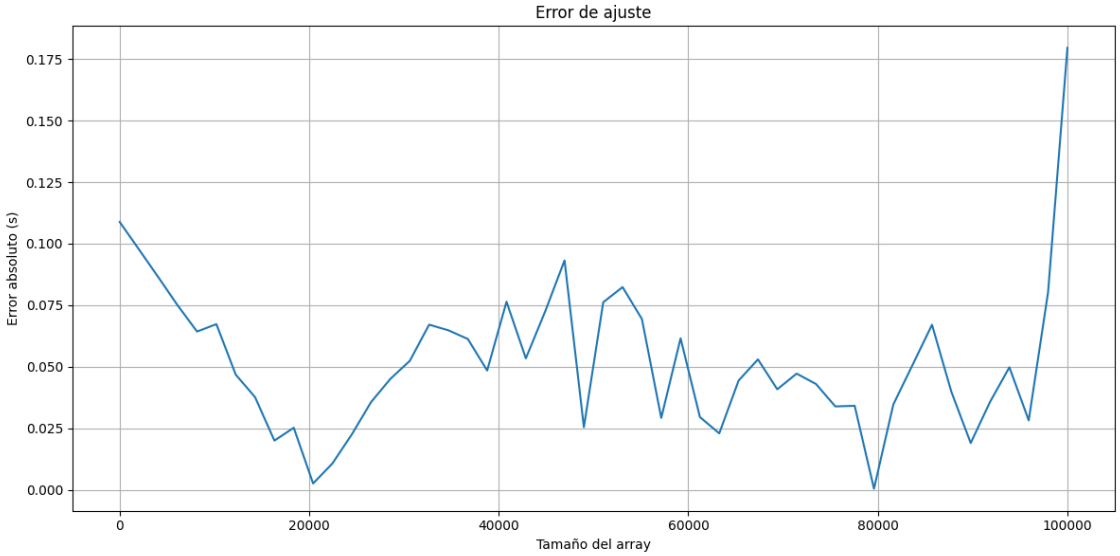

Figura 12: Error de las mediciones

5.2. Mateo empieza a Jugar

5.2.1. Ecuación de recurrencia

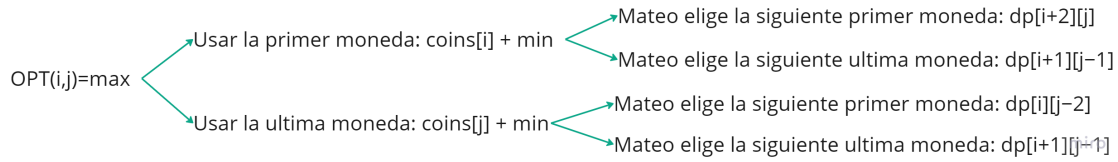

Figura 13: Ecuación de recurrencia

5.2.2. Demostración de optimalidad

Este algoritmo construye la respuesta de manera progresiva, utilizando subproblemas previamente resueltos para tomar decisiones estratégicas en juegos cada vez más grandes. En este caso, y tal como vimos en el desarrollo teórico y práctico de la sección 2.1, se utilizan juegos de monedas más pequeños previamente calculados y guardados en la matriz memoriosa, siendo seleccionados en base a la elección que sabemos que Mateo realizará y sumados al valor actual, de esta forma siempre obtendremos la solución óptima porque estaremos trabajando a partir de las mejores soluciones obtenidas previamente, obtenidas a partir de comparar las distintas variantes que pueden darse en base a elegir la moneda inicial o final.

La ecuación de recurrencia nos permite maximizar el valor acumulado posible para el jugador que comienza, considerando que ambos jugadores juegan de manera óptima (aunque con distintos métodos y alcances). Esto no significa que Sophia siempre gane, ya que su victoria dependerá del valor de las monedas y de las decisiones tomadas estratégicamente en el contexto del juego. Lo importante es que el algoritmo garantiza que Sophia siempre alcance su mejor puntaje posible dado el comportamiento óptimo de Mateo.

Un aspecto clave del algoritmo es que no siempre opta por la moneda de mayor valor disponible en el turno actual, sino que evalúa cuidadosamente cómo cada decisión afectará el resto del juego. Por ejemplo, en el caso del conjunto de monedas [3, 10, 5, 1] , Sophia podría elegir estratégicamente la moneda de menor valor (1), sacrificando la de valor 3, para evitar que Mateo pueda tomar la moneda de mayor valor (10) en su siguiente turno. Esto resulta en una puntuación final de 11 puntos para Sophia (1 y 10), mientras que Mateo obtiene 8 puntos (3 y 5).