Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Lineamientos sobre informes

3 de diciembre de 2024

Facundo Sorasio
108882

Camilo Ignacio Campos Durán
109368

1. Introducción

Este trabajo práctico evalúa el desarrollo y análisis de algoritmos aplicados a juegos entre dos hermanos, abordando distintos tipos de algoritmos. En la **primera parte**, se presenta un juego de selección de monedas, donde Sophia aplica un algoritmo ***Greedy*** para ganar. En la **segunda parte**, Sophia usa ***Progrmación Dinámica*** para aumentar sus probabilidades de ganar, mientras Mateo utiliza estrategias Greedy. Finalmente, en la **tercera parte**, se analiza el problema de "La Batalla Naval", aplicando algoritmo, tales como *Backtracking, Greedy y/o Aproximación*.Se incluyen mediciones de tiempos y análisis de complejidad.

2. Juego Monedas

En esta sección se trabaja sobre una lista ***monedas*** con diferentes valores dentro de la misma, y dependiendo de qué algoritmo se llega a implementar, veremos sus mediciones en tiempo de ejecución. Los algoritmos con los que trabajaremos son ***Greedy*** y ***Programación Dinámica***

Se muestra un ejemplo de una lista ***monedas***

```
monedas = [1, 4, 5, 21, 100, 90, 3]
```

2.1. Algoritmo Greedy

El algoritmo Greedy (o voraz) es una técnica computacional que sigue un enfoque iterativo para resolver problemas de optimización. En cada paso, selecciona la opción que maximiza (o minimiza) un criterio local definido, sin considerar el impacto de esta decisión en iteraciones futuras. Esta característica hace que el algoritmo sea eficiente en términos de implementación y complejidad temporal, pero puede llevar a resultados subóptimos en problemas donde las decisiones locales no garantizan una solución globalmente óptima.

- A continuación se muestra el código de solución Greedy:

```
def sophia_elige(monedas):

    if primer_moneda_mayor_valor(monedas):

        return monedas[0], monedas[1:]
    else:

        return monedas[-1], monedas[:-1]
```

Como se puede observar en el código, nuestra regla es tomar la moneda de al mayor valor entre la primera y última de la lista, en cada iteración. Permitiendo hacer una suma de los valores más altos a la hora de elegir dicha moneda

Por siguiente, se mostrará un gráfico donde la cantidad de monedas dentro de la lista, ira en aumento. Esto con el fin de evaluar el tiempo que se toma en resolver el problema planteado.

2.1.1. Medición

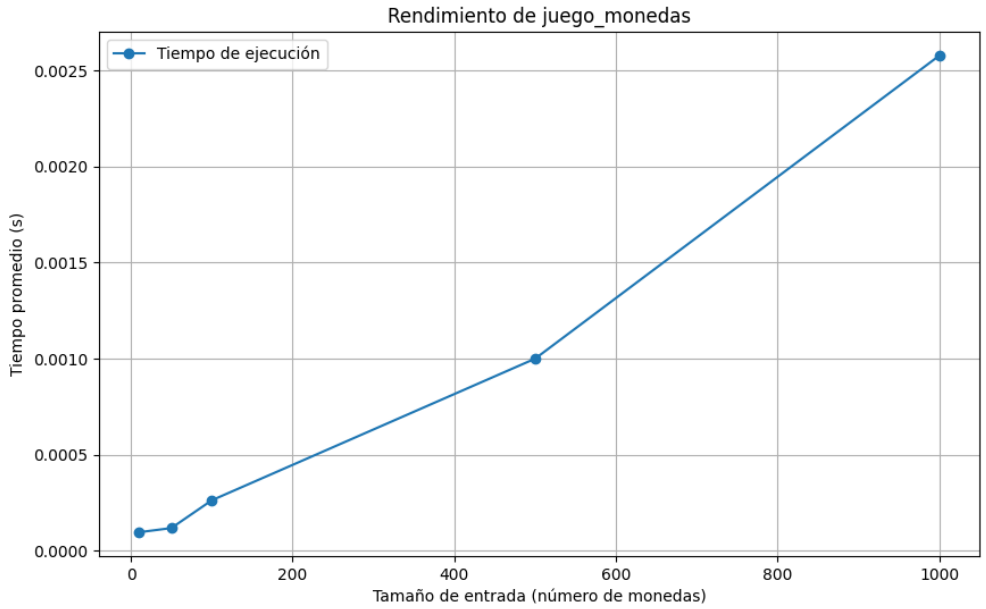

Figura 1: Tiempo con Algoritmo de Greedy

En este caso, la ecuación de recurrencia es:

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

Su complejidad temporal es de $\mathcal{O}(n)$.

2.2. Algoritmo Programación Dinámica

La **programación dinámica (PD)** es una técnica algorítmica para resolver problemas dividiéndolos en subproblemas más pequeños y reutilizando sus soluciones. Se basa en la **subestructura óptima**, donde una solución global se construye a partir de soluciones óptimas locales, y la **superposición de subproblemas**, almacenando resultados intermedios para evitar cálculos redundantes. Este enfoque utiliza estructuras tabulares para mejorar la eficiencia, reduciendo complejidad en problemas de optimización y toma de decisiones.

A continuación se muestra el código implementado:

- Función `llenar_tabla_dp`

```
def llenar_tabla_dp(monedas, suma_acumulada):
    n = len(monedas)
    dp = [[0] * n for _ in range(n)]

    for longitud in range(1, n + 1):
        for i in range(n - longitud + 1):
            j = i + longitud - 1
            suma_intervalo = suma_acumulada[j + 1] - suma_acumulada[i]
            elegir_primera = monedas[i] + (suma_intervalo - dp[i + 1][j] if i
    + 1 <= j else 0)
            elegir_ultima = monedas[j] + (suma_intervalo - dp[i][j - 1] if i
    <= j - 1 else 0)
            dp[i][j] = max(elegir_primera, elegir_ultima)
    return dp
```

Para cada intervalo de monedas $[i, j]$, evalúa dos opciones: elegir la primera o la última moneda del intervalo, y almacena el valor óptimo en la tabla `dp`. Esta tabla se llena iterativamente, considerando todos los intervalos posibles, y se devuelve con las soluciones óptimas para cada subproblema.

- Función `determinar_decision`

```
def determinar_decision(monedas, suma_acumulada, dp):
    i, j = 0, len(monedas) - 1
    while i <= j:
        suma_intervalo = suma_acumulada[j + 1] - suma_acumulada[i]
        if i + 1 <= j and dp[i][j] == monedas[i] + (suma_intervalo - dp[i +
    1][j]):
            moneda = monedas[i]
            i += 1
        else:
            moneda = monedas[j]
            j -= 1
        break
    monedas_actualizadas = monedas[i:j + 1]
    return moneda, monedas_actualizadas
```

El algoritmo recorre el intervalo de monedas $[i, j]$ y evalúa si la mejor opción es elegir la primera o la última moneda del intervalo, usando la información almacenada en `dp`. El proceso se repite hasta que se determina la moneda a elegir y se actualiza el conjunto de monedas disponibles para el jugador.

2.2.1. Medición

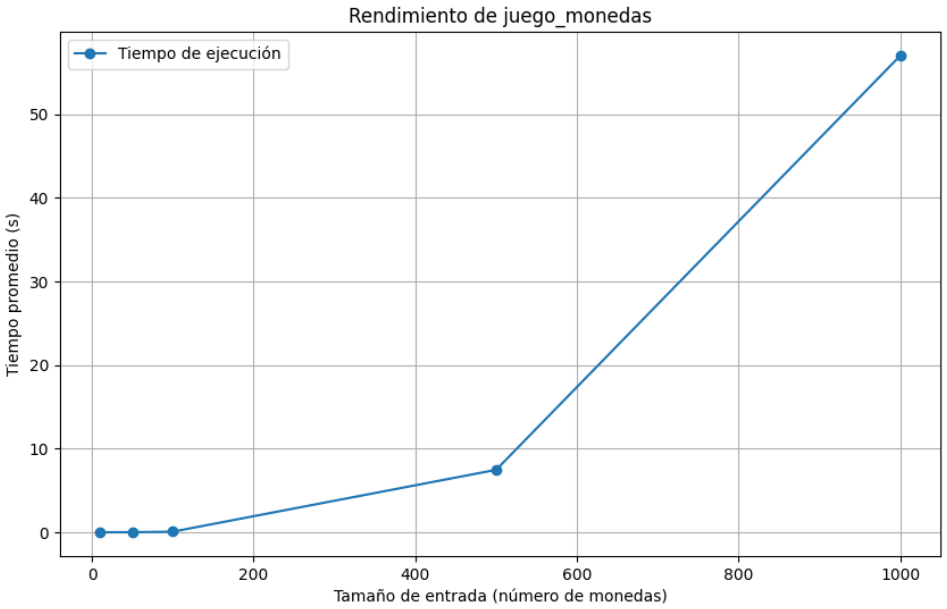

Figura 2: Tiempo con Algoritmo de Programación Dinámica

En este caso, la ecuación de recurrencia es:

$$\mathcal{T}(n) = \mathcal{O}(n^2)\cdot$$

Su complejidad temporal es de $\mathcal{O}(n^2)$.

3. Batalla Naval

El presente trabajo aborda el desarrollo de un sistema para la colocación automática de barcos en un tablero de Batalla Naval, el cual tenemos un tablero de NxM casilleros, y K barcos. Cada barco tiene Bi de largo. El usuario debe ingresar dichos valores. Luego selecciona una estrategia para la colocación desde un menú interactivo. Las opciones disponibles son ***Greedy*** y ***Backstracking***, ambas diseñadas para respetar las restricciones del juego, como la separación entre barcos y los límites del tablero.

dicho programa, presenta de 3 archivos .py para su debida ejecucion, los cuales son:

`Barcos.py, Tablero.py y main.py`

Dentro de nuestro archivo **main.py** es donde se ejecuta nuestro programa casi en su totalidad. Luego de que el usuario ingrese los respectivos valores, se ingresará a un menú interactivo donde debe decidir qué estrategia utilizar o simplemente salir del programa.

```
Que estraegia desea utilizar para colocar los barcos?
1. Greedy
2. Backtracking
3. Salir
Ingrese su opcion:
```

Si el usuario escoge el enfoque `Greedy`, este colocará los barcos de forma ordenada, de mayor a menor longitud, con la intención de ocupar la mayor cantidad de celdas posibles. En el caso de que no se pueda colocar un barco, simplemente lo saltará y continuará con el siguiente.

```python
def colocar_barco_greedy(tablero, largo):
    for fila in range(len(tablero)):
        for columna in range(len(tablero[0])):
            for direccion in ["H", "V"]:

                if es_valida(tablero, fila, columna, largo, direccion):
                    colocar_barco(tablero, fila, columna, largo, direccion)
                    return True
    return False
```

```
Tablero:
1 1 1 1 0 |   4
0 0 0 0 0 |   0
1 1 1 0 1 |   4
0 0 0 0 1 |   1
1 0 0 0 0 |   1
- - - - -
3 2 2 1 2

¡Todos los barcos fueron colocados exitosamente!
```

Figura 3: Se ingresaron 1 Porta aviones (2 largo), 1 Submarino (3 largo), 1 Destructores (1 largo), 1 Lanchas (4 largo) en un tablero (5x5).

En este caso, la ecuación de recurrencia es:

$$\mathcal{T}(n) = \mathcal{O}(K \cdot W)$$

Su complejidad temporal es de $\mathcal{O}(K \cdot W)$.

Mientras que, si el usuario seleccionase la estrategia de `Backstracking`, el proceso de colocación de los barcos se realizará de manera diferente. En este enfoque, el algoritmo intentará colocar los barcos de manera ordenada, pero si en algún momento no puede colocar un barco (por ejemplo, si no hay espacio disponible para colocar un barco en la orientación seleccionada), el algoritmo retrocederá a la colocación anterior y probará una alternativa, asegurando que los barcos sean colocados de forma correcta en el tablero. Es una técnica de prueba y error, donde se exploran todas las posibilidades de colocación de los barcos, y si una opción falla, se prueba con la siguiente.

```python
def colocar_barcos_backtracking(tablero, barcos, indice=0):

    if indice == len(barcos):
        return True

    tipo, largo = barcos[indice]

    for fila in range(len(tablero)):
        for columna in range(len(tablero[0])):
            for direccion in ["H", "V"]:
                if es_valida(tablero, fila, columna, largo, direccion):
                    colocar_barco(tablero, fila, columna, largo, direccion)
                    if colocar_barcos_backtracking(tablero, barcos, indice + 1):
                        return True
                    retirar_barco(tablero, fila, columna, largo, direccion)
    return False
```

```
Tablero:
1 1 0 1 0 |  3
0 0 0 1 0 |  1
1 0 0 1 0 |  2
0 0 0 0 0 |  0
1 1 1 1 0 |  4
- - - - -
3 2 1 4 0

¡Todos los barcos fueron colocados exitosamente!
```

Figura 4: Se ingresaron 1 Porta aviones (2 largo), 1 Submarino (3 largo), 1 Destructores (1 largo), 1 Lanchas (4 largo) en un tablero (5x5).

En este caso, la ecuación de recurrencia es:

$$\mathcal{T}(n) = \mathcal{O}(K \cdot W \cdot 2 \cdot n) \cdot \mathcal{T}(n-1)$$

Su complejidad temporal es de $\mathcal{O}(K \cdot W \cdot n)$.

4. ¿Problema pertenece a NP?

La complejidad NP (*Nondeterministic Polynomial Time*) se refiere a la clase de problemas de decisión para los cuales, si se proporciona un certificado (solución candidata), este puede ser verificado en tiempo polinómico con respecto al tamaño de la entrada.

En el caso del Problema de la Batalla Naval, se cumple que:

1. Existe un **certificado** que contiene:
   - La posición inicial (i, j) de cada barco en el tablero.
   - La orientación de cada barco (*horizontal* o *vertical*).
   - La longitud de los barcos.
2. La verificación de este certificado puede realizarse en tiempo polinómico:
   - Comprobando que los barcos no se solapan, lo cual requiere recorrer las celdas ocupadas.
   - Verificando que los barcos están dentro de los límites del tablero.
   - Corroborando que el número y longitud de los barcos coinciden con los datos de entrada.
   - Asegurando que las restricciones de separación (por ejemplo, no adyacencia) se cumplen.

   Este proceso puede implementarse en $\mathcal{O}(N \cdot M \cdot n)$, donde N y M son las dimensiones del tablero y n la cantidad de barcos.

Dado que la verificación de una solución propuesta (es decir, la colocación de los barcos en el tablero) se puede realizar en tiempo polinómico respecto al tamaño del tablero y las características de los barcos, podemos afirmar que el Problema de la Batalla Naval **pertenece** a la clase **NP**. Esto se debe a que existe un algoritmo no determinista que, dado un certificado (una posible solución), puede verificar su validez en tiempo polinómico.

5. Batalla Naval, es un NP-Completo

Para demostrar que el Problema de la Batalla Naval es NP-Completo.

- **El problema debe pertenecer a NP.**

  Esto ya fue demostrado hace un momento.

- **Realizar una reducción desde un problema NP-Completo conocido.**

  Para demostrar que el Problema de la Batalla Naval es NP-Completo, debemos reducir un problema NP-Completo conocido, como *Bin-Packing*, al Problema de la Batalla Naval. Esta reducción debe realizarse en tiempo polinómico y debe demostrar que si podemos resolver el Problema de la Batalla Naval.

5.1. Reducción de Bin-Packing a Batalla Naval

El problema de *Bin-Packing* consiste en determinar si un conjunto de objetos de diferentes tamaños puede ser empaquetado en un número mínimo de contenedores, de tal manera que la suma de los tamaños de los objetos en cada contenedor no exceda la capacidad del mismo. Este problema es conocido por ser NP-Completo.

Dado un conjunto de objetos con tamaños $S_1, S_2, \ldots, S_k$ y una capacidad de contenedor C, podemos construir una instancia del Problema de la Batalla Naval donde:

- Cada objeto S_i se representa como un barco con longitud S_i.
- La capacidad del contenedor C se representa como el tamaño del tablero $N \times M$.

El objetivo es determinar si es posible colocar estos barcos en el tablero sin violar las restricciones del problema de la Batalla Naval (es decir, que los barcos no se solapen y que todos se ajusten dentro del tablero acorde a las restricciones). Esta colocación es equivalente a decidir si los objetos pueden ser empaquetados en los contenedores sin exceder la capacidad, es decir, si el problema de *Bin-Packing* tiene una solución.

Por lo que, gracias a que, podemos reducir una instancia del problema NP-Completo de *Bin-Packing* a una instancia del Problema de la Batalla Naval en tiempo polinómico, podemos concluir que el Problema de la Batalla Naval es, por si mismo, **NP-Completo**.

6. Mediciones

Las mediciones realizadas muestran gráficos que ilustran el rendimiento del algoritmo en diferentes pruebas de volumen. Estas pruebas varían el tamaño del tablero, la cantidad de barcos, entre otras cosas.

6.1. Tablero Fijo

Aquí se presentan los resultados con un tablero de tamaño fijo. En las pruebas se varió la cantidad de barcos colocados y se midió el tiempo de ejecución.

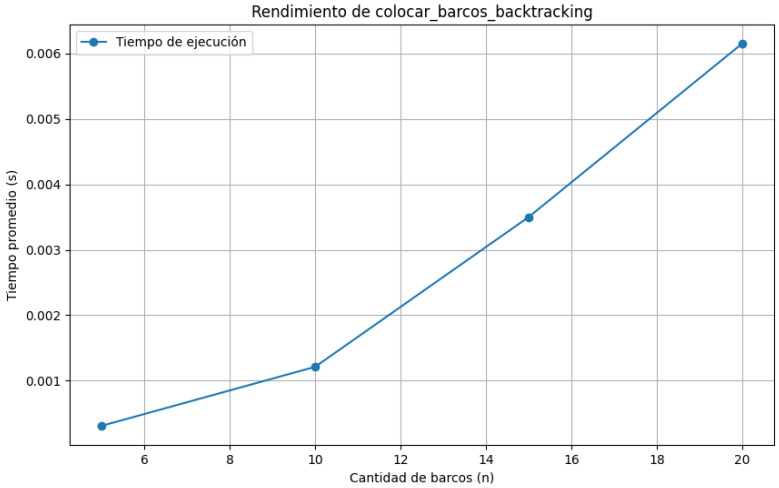

Figura 5: Rendimiento del algoritmo aumentando la cantidad de barcos.

El gráfico muestra cómo el tiempo de ejecución aumenta a medida que incrementa la cantidad de barcos (k), mientras que el tamaño del tablero (N y M) permanece constante. A mayor cantidad de barcos, el algoritmo debe realizar más intentos recursivos para encontrar una solución válida. Esto incrementa la complejidad, ya que el número de combinaciones posibles aumenta. Aunque N y M se mantienen fijos, el rendimiento se ve afectado principalmente por el número de barcos, lo que provoca un aumento significativo en el tiempo de ejecución.

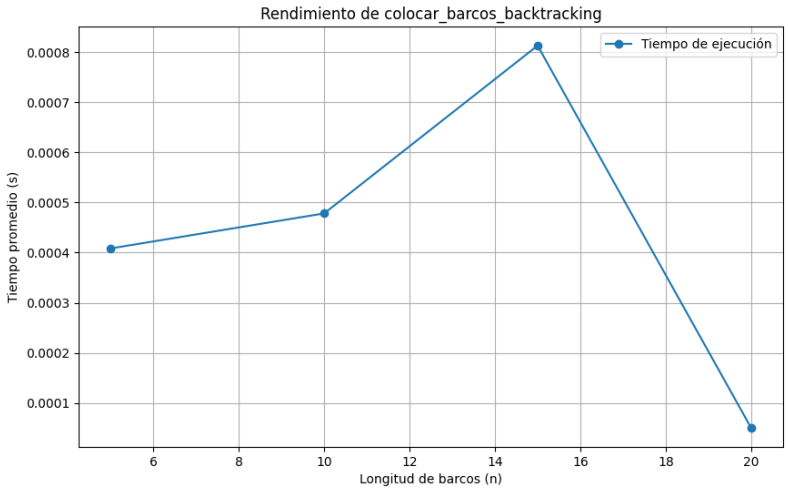

Figura 6: Rendimiento del algoritmo aumentando la longitud de los barcos.

En el gráfico se puede observar que, al alcanzar una longitud de 15, el tiempo de ejecución disminuye abruptamente. Esto ocurre porque la longitud de los barcos supera el tamaño del tablero, lo que hace que el algoritmo saltee la colocación de esos barcos. Al no tener que evaluar más combinaciones para barcos de mayor tamaño, el rendimiento mejora, lo que se refleja en la caída en el tiempo de ejecución.

6.2. Barco Fijo

Aquí se presentan los resultados con barcos fijos. En las pruebas se varió el tamaño del tablero.

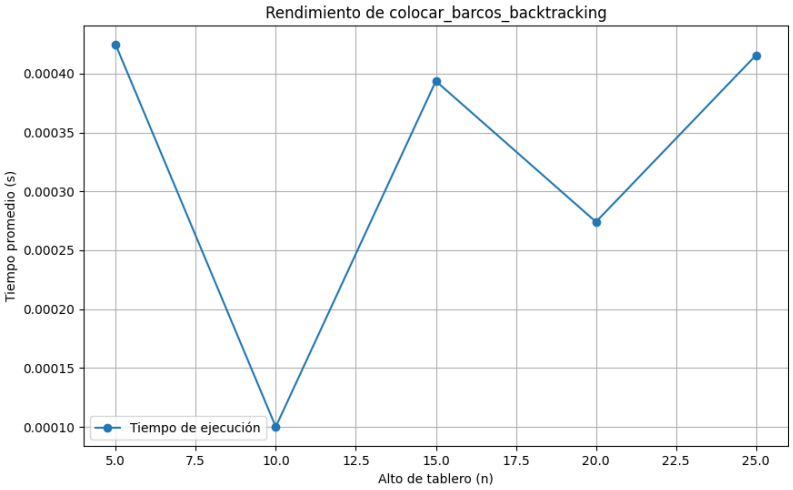

Figura 7: Rendimiento del algoritmo aumentando el alto del tablero.

El gráfico presenta un comportamiento en forma de "zig-zag"debido a que, al aumentar el alto del tablero, el algoritmo enfrenta variaciones en la cantidad de combinaciones posibles para colocar los barcos. En algunas iteraciones, el aumento en el tamaño permite una colocación más eficiente, mientras que en otras, el algoritmo debe probar más combinaciones, lo que incrementa temporalmente el tiempo de ejecución.

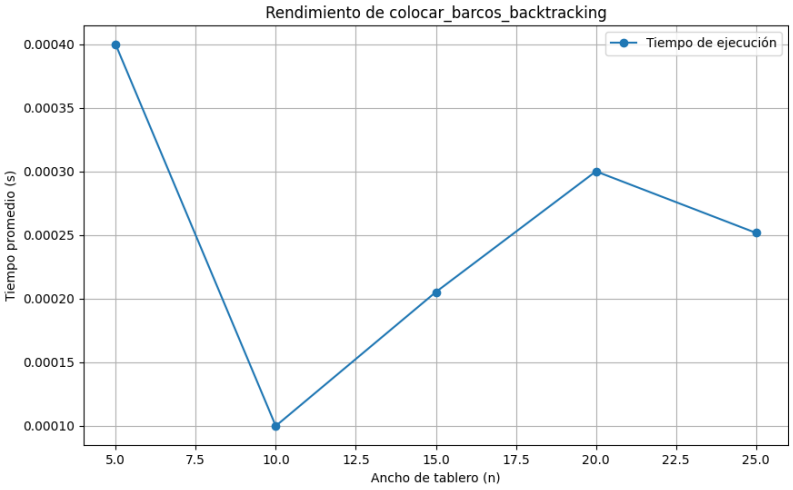

Figura 8: Rendimiento del algoritmo aumentando el ancho del tablero.

El gráfico muestra que al inicio, con un tablero de ancho pequeño, la complejidad es alta debido a las limitaciones en las combinaciones posibles para colocar los barcos. Al alcanzar un tamaño de tablero que se acerca a un cuadrado perfecto, el algoritmo encuentra un equilibrio entre el espacio disponible y las combinaciones, lo que mejora su eficiencia. Sin embargo, a medida que el tablero vuelve a tener un tamaño rectangular más grande, la complejidad aumenta nuevamente, ya que

el número de posibles ubicaciones para los barcos crece considerablemente.

6.3. Aumento Proporcional

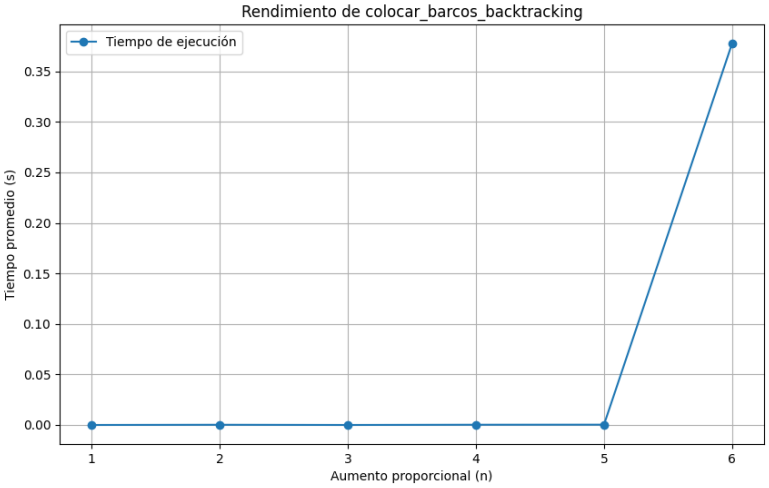

Figura 9: Rendimiento del algoritmo aumentando aumentando propocionalmente todos los valores.

El gráfico muestra un aumento polinomico en el rendimiento del algoritmo al incrementar proporcionalmente todos los valores (tamaño del tablero, cantidad de barcos y longitud de los barcos). Esto se debe a que a medida que crecen las dimensiones y el número de barcos, las posibles combinaciones y la cantidad de operaciones necesarias para colocar los barcos aumentan de manera exponencial. La complejidad del algoritmo refleja esta relación, lo que resulta en un incremento significativo en el tiempo de ejecución.

7. Conclusiones

7.1. Juego de Monedas Greedy

En el caso del algoritmo Greedy para el juego de monedas, hemos comprobado que este enfoque es eficiente en cuanto a tiempo de ejecución, con una complejidad temporal de $\mathcal{O}(n)$. Sin embargo, el algoritmo no siempre garantiza una solución óptima, ya que se basa en decisiones locales, seleccionando las monedas de mayor valor sin considerar el impacto de esta elección en el resultado final. En escenarios donde el problema requiere una visión global, como en el caso de monedas de valores más pequeños al final de la lista, el algoritmo puede no ofrecer la mejor solución. A pesar de ello, el algoritmo Greedy es una opción viable cuando se busca rapidez y simplicidad en la implementación.

7.2. Juego de Monedas con Programación Dinámica

El enfoque de Programación Dinámica (PD) proporciona una mejora significativa en la calidad de la solución, logrando encontrar la distribución óptima de las monedas. Con una complejidad temporal de $\mathcal{O}(n^2)$, este enfoque implica un mayor costo computacional en comparación con el algoritmo Greedy, pero a cambio, garantiza una solución globalmente óptima al considerar todas las posibilidades. La técnica de reutilizar subsoluciones y llenar una tabla de decisiones optimizadas permite a PD manejar de manera efectiva problemas más complejos, donde las decisiones locales pueden no ser adecuadas. Este enfoque es ideal cuando se prioriza obtener una solución óptima a costa de mayor tiempo de procesamiento.

7.3. Batalla Naval

El análisis de la Batalla Naval revela que los algoritmos Greedy y Backtracking presentan características complementarias en cuanto a eficiencia y precisión. El algoritmo Greedy, con una complejidad de $\mathcal{O}(K \cdot W)$, es más rápido, ya que coloca los barcos de manera ordenada y prioriza el uso del espacio del tablero, aunque no garantiza una colocación válida en todos los casos. Por otro lado, el Backtracking, con una complejidad de $\mathcal{O}(K \cdot W \cdot n)$, explora todas las posibilidades, lo que le permite encontrar soluciones válidas, aunque a un costo mayor en términos de tiempo. Dependiendo de los requisitos del problema, ambos enfoques pueden ser adecuados, siendo el Greedy más adecuado para problemas menos complejos y el Backtracking para asegurar soluciones completas y correctas en casos más desafiantes.