.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico Anual
Lineamientos sobre informes

3 de diciembre de 2024

Castaño Mateo
110114
Rodrigo Pesa Leandro
110076

Oviedo Ignacio Sebastián
109821
Trezeguet Santiago Bautista
110343
Suxo Juan
108280

Primera parte: Introducción y primeros años

1. Algoritmo de victoria versión greedy

En esta sección del informe, se analizará un algoritmo greedy donde Sofía siempre resulta ganadora del juego.

1.1. Algoritmo

El funcionamiento del algoritmo es el siguiente: en cada turno, cuando Sofía debe elegir entre los valores disponibles en los extremos (derecha e izquierda), selecciona el valor más alto. Por otro lado, durante el turno de Mateo, Sofía toma la decisión por él y elige el valor más bajo.

A continuación se muestra el código de solución greedy al problema

```
def juego_greedy(vec):

    v_sofia = [] * (len(vec)//2)
    v_mateo = [] * (len(vec)//2)

    for i in range(0, len(vec)):
        if (i%2==0 or i==0):
            resultado, vec = turno_sofia(vec)
            v_sofia.append(resultado)
        else:
            resultado, vec = turno_mateo(vec)
            v_mateo.append(resultado)
    return sum(v_sofia)
```

Notese que, en cada iteración, se aplica una regla sencilla: elegir el valor mayor o menor, dependiendo de quién tenga el turno. Esta decisión se basa en seleccionar la mejor opción posible en el momento específico de la elección. Las funciones $turnosofia$ y $turnomateo$ son las encargadas de tomar esta decisión, evaluando el vector y comparando los valores en las posiciones inicial y final.

1.2. Complejidad

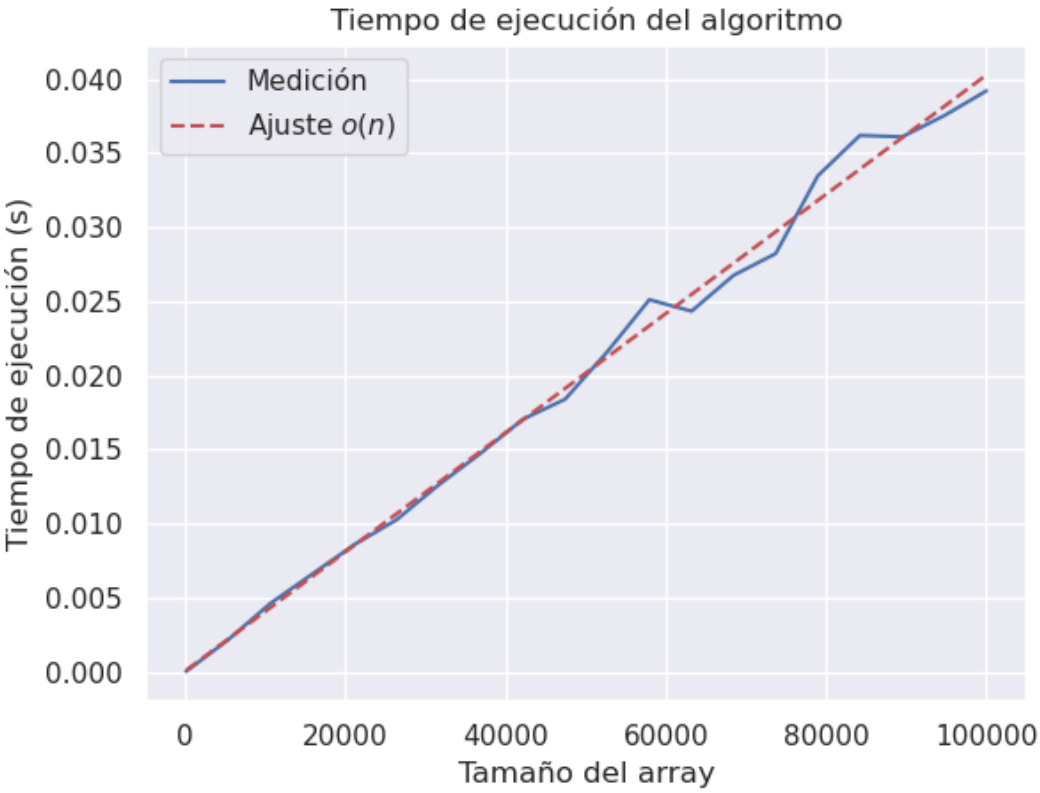
Para determinar la complejidad planteamos la ecuación de recurrencia. Notese que lo que está fuera del bucle es $O(1)$. El bucle recorre desde 0 hasta $n-1$, siendo n la longitud del vector. Dentro del bucle se realizan comparaciones cuya complejidad es $O(1)$. Por lo tanto, podemos afirmar que la ecuación de recurrencia es la siguiente:

$$T(n) = T(n-1) + n^0.$$

Por el teorema maestro, podemos concluir que la complejidad es $O(n)$. Esto se ve fácilmente ya que el bucle se repite $n-1$ veces y dentro del bucle la complejidad de las comparaciones es $O(1)$.

1.3. Mediciones

Para comprobar que la complejidad es la correcta, se probó utilizando la técnica de cuadrados mínimos, ajustando la función a una lineal. En la siguiente imagen podemos ver ese ajuste:



Como podemos ver, las mediciones tomadas tienden a ser una recta, por lo que podemos asumir que la complejidad teórica del algoritmo es correcta.

1.4. Optimalidad

El algoritmo garantiza una solución óptima para cualquier conjunto de valores. Para ilustrarlo, consideremos un ejemplo trivial con tres valores: $[9, 10, 2]$. En este caso, en el primer paso elegiríamos el valor más grande entre los dos extremos, es decir, el 9. Luego, seleccionaríamos el valor más pequeño de los restantes, que sería el 2. Finalmente, elegiríamos el valor 10, obteniendo un total de 19 puntos para nosotros y 2 puntos para nuestro hermano.

Es importante notar que, independientemente del orden de los valores, el resultado será siempre el mismo cuando el array es de tres.

Analicemos ahora un caso con cuatro valores: $[1, 9, 10, 2]$. En el primer paso, elegiríamos el 2 (el mayor de los extremos). Luego, tomaríamos el 1 (el menor de los valores restantes). A continuación, seleccionaríamos el 10 y finalmente el 9. Nuestro puntaje final sería 12, mientras que el de nuestro hermano sería 10. Como se observa, seguimos ganando.

Sin embargo, en este caso, aunque el orden del arreglo no afecta al ganador, sí influye en los resultados finales. Por ejemplo, si el arreglo fuera $[9, 1, 10, 2]$, nuestro puntaje sería 19 y el de nuestro hermano 3.

Podemos concluir que no siempre obtendremos la puntuación más alta posible. Sin embargo, para demostrar que siempre ganaremos, consideremos el siguiente arreglo: $[x_1, p, y_1]$, donde p representa un subarreglo de $n - m$ elementos, siendo n la longitud total del arreglo y m la suma de las monedas ya elegidas y las que están por ser seleccionadas.

Supongamos que x_1 es menor que y_1. En ese caso, tras nuestra primera elección, obtendremos un puntaje de y_1, y el arreglo quedará como $[x_1, p, y_2]$. Posteriormente, si x_1 es menor que y_2,

escogeremos x_1 para nuestro hermano. En este punto, nuestro puntaje total será y_1, mientras que el de nuestro hermano será x_1.

En la siguiente iteración, elegiremos el valor más alto de los extremos disponibles, lo que aumentará nuestro puntaje, asegurando que será mayor al de nuestro hermano, ya que cualquier número que seleccionemos será mayor que x_1.

Si continuamos aplicando esta estrategia, llegaremos a dos escenarios posibles:

1. Nuestro turno ocurre cuando quedan tres monedas.

2. Nuestro turno ocurre cuando quedan cuatro monedas.

En ambos casos, como hemos demostrado previamente, siempre obtendremos la victoria.

Por lo tanto, podemos concluir que este enfoque greedy garantiza la victoria, aunque no necesariamente la puntuación más alta posible.

1.5. Conclusion

Podemos concluir que el algoritmo planteado resuelve nuestro problema de forma óptima, es decir, siempre garantiza nuestra victoria. Sin embargo, dentro de esa optimalidad, los resultados finales pueden variar dependiendo del orden de las monedas en el arreglo.

Con respecto a la complejidad, podemos decir que se comprobó tanto teóricamente como empíricamente que es $O(n)$.

Segunda parte: Mateo empieza a Jugar

2. Algoritmo de victoria versión programación dinámica

En esta sección del informe, se analizará un algoritmo de programación dinámica donde Sofía siempre resulta ganadora del juego con el mayor puntaje posible, dado que Mateo siempre elija el valor mas alto de los extremos.

2.1. Algoritmo

Definiendo:

dp[i][j]: el valor máximo que Sophia puede obtener si el rango de monedas considerado está entre las posiciones i y j. (inclusive).

Caso base:

Si $i = j$ (solo hay una moneda disponible):

$$dp[i][j] = \text{monedas}[i]$$

Casos generales:

Cuando hay más de una moneda, Sophia tiene dos opciones:

1. **Elegir la moneda al inicio (i):** La elección de Mateo (siguiendo una estrategia Greedy) quedará entre monedas$[i+1]$ y monedas$[j]$:

- Si el mayor es monedas$[i+1]$, Mateo elegirá esta, y el rango restante quedaría en $dp[i+2][j]$ para Sophia.
- Si el mayor es monedas$[j]$, Mateo elegirá esta, y el rango restante quedaría en $dp[i+1][j-1]$ para Sophia.

De esta forma, el valor acumulado al elegir i será:

$$\text{pick}_i = \text{monedas}[i] + \begin{cases} dp[i+2][j], & \text{si monedas}[i+1] \geq \text{monedas}[j] \\ dp[i+1][j-1], & \text{si monedas}[i+1] \leq \text{monedas}[j] \end{cases}$$

2. **Elegir la moneda al final (j):** La elección de Mateo (siguiendo una estrategia Greedy) quedará entre monedas$[i]$ y monedas$[j-1]$:

- Si el mayor es monedas$[i]$, Mateo elegirá esta, y el rango restante quedaría en $dp[i+1][j-1]$ para Sophia.
- Si el mayor es monedas$[j-1]$, Mateo elegirá esta, y el rango restante quedaría en $dp[i][j-2]$ para Sophia.

De esta forma, el valor acumulado al elegir j será:

$$\text{pick}_j = \text{monedas}[j] + \begin{cases} dp[i+1][j-1], & \text{si monedas}[i] \geq \text{monedas}[j-1] \\ dp[i][j-2], & \text{si monedas}[i] \leq \text{monedas}[j-1] \end{cases}$$

Maximización: Sophia seleccionará la opción que maximice su valor:

$$dp[i][j] = \text{máx}(\text{pick}_i, \text{pick}_j)$$

A continuación el algoritmo que implementa esta lógica:

```python
def juego_programacion_dinamica(monedas):
    n = len(monedas)
    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:
                dp[i][j] = monedas[i]
            else:
                # Si Sophia elige la moneda en la posici n i
                if monedas[i + 1] >= monedas[j]:
                    pick_i = monedas[i] + dp[i + 2][j] if i + 2 <= j else monedas[i]
                else:
                    pick_i = monedas[i] + dp[i + 1][j - 1] if i + 1 <= j - 1 else monedas[i]

                # Si Sophia elige la moneda en la posici n j
                if monedas[i] >= monedas[j - 1]:
                    pick_j = monedas[j] + dp[i + 1][j - 1] if i + 1 <= j - 1 else monedas[j]
                else:
                    pick_j = monedas[j] + dp[i][j - 2] if i <= j - 2 else monedas[j]

                dp[i][j] = max(pick_i, pick_j)
    return dp[0][n - 1]
```

La complejidad temporal del algoritmo es $O(n^2)$, ya que se llena una tabla de tamaño $n \times n$, donde cada celda se calcula en tiempo constante $O(1)$. El proceso considera n posibles tamaños de subrangos, y para cada tamaño se analizan hasta n posiciones iniciales. En consecuencia, la combinación de estos factores conduce a un costo computacional cuadrático respecto al tamaño de entrada.

2.2. Seguimiento

Para validar el funcionamiento del algoritmo, se realizó un seguimiento paso a paso utilizando el conjunto de datos siguiente:

$$\text{monedas} = [8, 15, 7, 9, 20, 1, 6]$$

Casos base:

$$dp[0][0] = 8$$
$$dp[1][1] = 15$$
$$dp[2][2] = 7$$
$$dp[3][3] = 9$$
$$dp[4][4] = 20$$
$$dp[5][5] = 1$$
$$dp[6][6] = 6$$

Tabla Inicial:

$$\begin{bmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 20 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Sub-secuencias de longitud 2:

$$dp[0][1] = \text{máx}(\text{pick\_i} = 8, \text{pick\_j} = 15) = 15$$
$$dp[1][2] = \text{máx}(\text{pick\_i} = 15, \text{pick\_j} = 7) = 15$$
$$dp[2][3] = \text{máx}(\text{pick\_i} = 7, \text{pick\_j} = 9) = 9$$
$$dp[3][4] = \text{máx}(\text{pick\_i} = 9, \text{pick\_j} = 20) = 20$$
$$dp[4][5] = \text{máx}(\text{pick\_i} = 20, \text{pick\_j} = 1) = 20$$
$$dp[5][6] = \text{máx}(\text{pick\_i} = 1, \text{pick\_j} = 6) = 6$$

Sub-secuencias de longitud 3:

$$dp[0][2] = \text{máx}(\text{pick\_i} = 8 + 7, \text{pick\_j} = 7 + 8) = 15$$
$$dp[1][3] = \text{máx}(\text{pick\_i} = 15 + 7, \text{pick\_j} = 9 + 7) = 22$$
$$dp[2][4] = \text{máx}(\text{pick\_i} = 7 + 9, \text{pick\_j} = 20 + 7) = 27$$
$$dp[3][5] = \text{máx}(\text{pick\_i} = 9 + 1, \text{pick\_j} = 1 + 9) = 10$$
$$dp[4][6] = \text{máx}(\text{pick\_i} = 20 + 1, \text{pick\_j} = 6 + 1) = 21$$

Sub-secuencias de longitud 4:

$$dp[0][3] = \text{máx}(\text{pick\_i} = 8 + 9, \text{pick\_j} = 9 + 15) = 24$$
$$dp[1][4] = \text{máx}(\text{pick\_i} = 15 + 9, \text{pick\_j} = 20 + 9) = 29$$
$$dp[2][5] = \text{máx}(\text{pick\_i} = 7 + 20, \text{pick\_j} = 1 + 9) = 27$$
$$dp[3][6] = \text{máx}(\text{pick\_i} = 9 + 6, \text{pick\_j} = 6 + 20) = 26$$

Sub-secuencias de longitud 5:

$$dp[0][4] = \text{máx}(\text{pick\_i} = 8 + 22, \text{pick\_j} = 20 + 15) = 35$$
$$dp[1][5] = \text{máx}(\text{pick\_i} = 15 + 10, \text{pick\_j} = 1 + 22) = 25$$
$$dp[2][6] = \text{máx}(\text{pick\_i} = 7 + 21, \text{pick\_j} = 6 + 10) = 28$$

Sub-secuencias de longitud 6:

$$dp[0][5] = \text{máx}(\text{pick\_i} = 8 + 27, \text{pick\_j} = 1 + 24) = 35$$
$$dp[1][6] = \text{máx}(\text{pick\_i} = 15 + 26, \text{pick\_j} = 6 + 27) = 41$$

Sub-secuencia de longitud 7 (completa):

$$dp[0][6] = \text{máx}(\text{pick\_i} = 8 + 28, \text{pick\_j} = 6 + 25) = 36$$

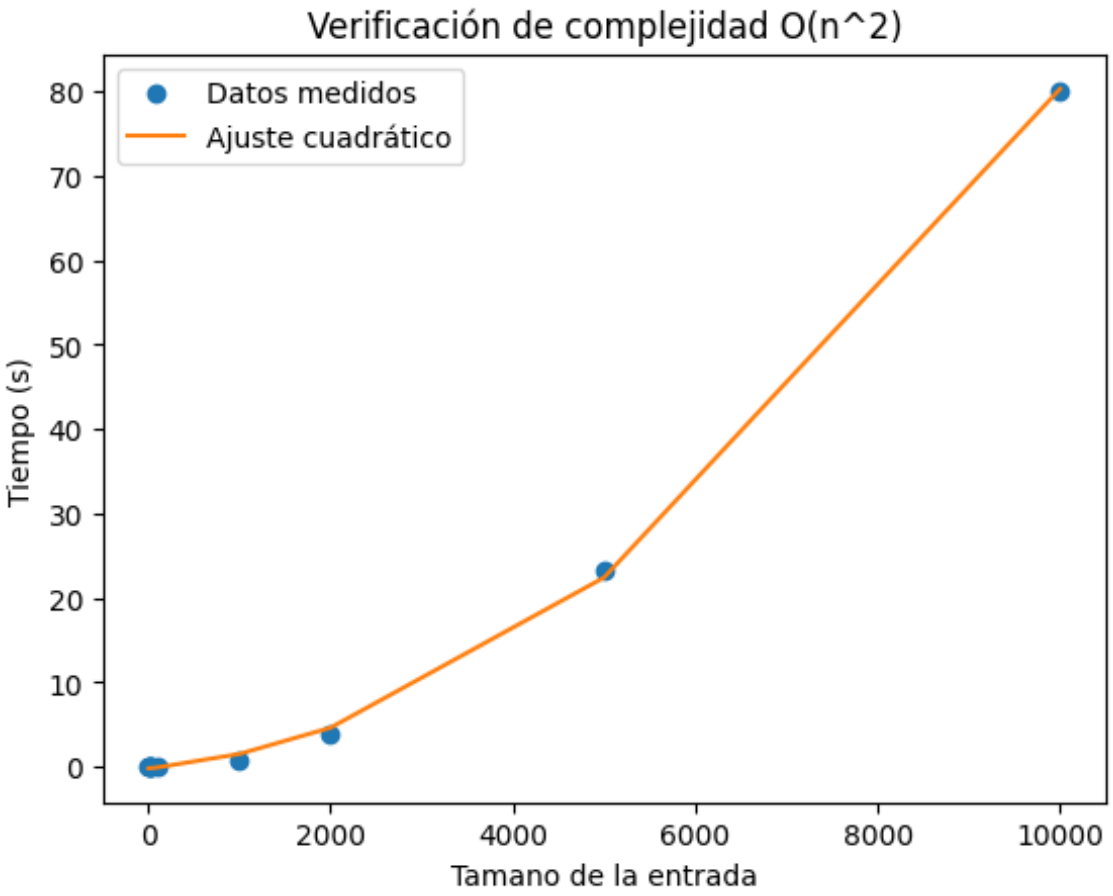
Tabla Final:

$$\begin{bmatrix} 8 & 15 & 15 & 24 & 35 & 35 & 36 \\ 0 & 15 & 15 & 22 & 29 & 25 & 41 \\ 0 & 0 & 7 & 9 & 27 & 27 & 28 \\ 0 & 0 & 0 & 9 & 20 & 10 & 26 \\ 0 & 0 & 0 & 0 & 20 & 20 & 21 \\ 0 & 0 & 0 & 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Acumulación total: 36

2.3. Mediciones

Para comprobar que la compejidad teórica es la correcta se realizo un ajuste cuadrático, utilizando la técnica de cuadrados mínimos. Dicho ajuste podemos verlo en la siguiente imagen:



Como podemos comprobar visualmente, las mediciones tienden a tener forma cuadrática, por lo que podemos concluir que las medidas tomadas corresponden con la complejidad teórica.

2.4. Conclusión

El algoritmo propuesto tiene complejidad $O(n^2)$, donde n es el número de monedas. Esta complejidad es independiente de los valores de las monedas, ya que la cantidad de operaciones realizadas depende únicamente del número de monedas, no de sus valores específicos. El algoritmo

llena una matriz *dp* de tamaño $n \times n$, y cada subproblema se resuelve en $O(1)$, por lo que el número total de operaciones es proporcional a n^2.

Cabe aclarar que, aunque los valores de las monedas afectan las decisiones que toma el algoritmo al seleccionar la moneda más beneficiosa en cada paso, esto no cambia el número de comparaciones y asignaciones realizadas. El algoritmo sigue un patrón de llenado de la matriz que es constante, independientemente de si las monedas tienen valores uniformes o muy dispares.

En conclusión, puede decirse que la variabilidad en los valores de las monedas no impacta significativamente los tiempos de ejecución del algoritmo. La eficiencia temporal del algoritmo se mantiene con complejidad $O(n^2)$, ya que, aunque los valores influencian las decisiones estratégicas, no alteran el número de operaciones necesarias para resolver el problema.

Tercera parte: Diversión NP-Completa

3. Nuevo juego: Batalla Naval Individual

En esta sección del informe, se analizará un algoritmo de backtrancking para resolver un problema NP-Completo.

3.1. Problema

En primer lugar, se procederá a demostrar que el problema pertenece a la clase NP, describiendo el proceso para verificar una solución en tiempo polinómico. Para este propósito, se propone el siguiente algoritmo:

```
def es_batalla_navala(rec_fil, rec_col, matriz):
    cont = 0
    i=0
    j=0
    rec_col_finales = [0]*len(rec_col)

    for fila in matriz:
        for celda in fila:
            if celda == 1:
                rec_col_finales[j] += 1
                cont += 1
            j += 1

        if cont <= rec_fil[i]:
            return False

        i += 1
        cont=0
        j=0

    for i in range(len(rec_col)):
        if rec_col[i] < rec_col_finales[i]:
            return False

    return True
```

La función recibe como entrada las restricciones correspondientes a las filas y las columnas, así como una matriz que representa una posible solución, en la cual cada casilla con un valor de 1 indica la presencia de un barco.

El algoritmo realiza un recorrido completo de la matriz una única vez, donde cada iteración del bucle implica operaciones con complejidad $O(1)$. En consecuencia, la primera parte de la función tiene una complejidad de $O(n \times m)$.

Posteriormente, se itera sobre un vector auxiliar de longitud n, efectuando operaciones con complejidad $O(1)$ en cada iteración. Esto implica que esta etapa tiene una complejidad de $O(n)$.

Por lo tanto, al analizar ambas etapas, se concluye que la complejidad total del algoritmo puede acotarse como $O(n \times m)$. Dado que este algoritmo permite verificar una solución en tiempo polinómico, se puede determinar que el problema pertenece a la clase NP.

3.2. Reducción

Para comprobar que el problema de *batalla naval* es un problema NP-Completo, se eligió el problema *bin packing unario* para reducirlo a *batalla naval*. Cabe destacar que podemos probar que *bin packing unario* es un problema NP-Completo. Esto se puede ver trivialmente si reducimos el problema *b-partition*, donde el conjunto de números que queremos probar si suman el valor X en B subconjuntos lo podemos transformar a *bin packing unario* simplemente siendo B la cantidad de bins y X el valor C o cantidad.

Para reducir el problema de *bin packing unario* al problema de la *batalla naval*, primero debemos definir qué es cada elemento. El conjunto de n números en unario representará los barcos. La matriz se construirá de la siguiente manera: la cantidad de columnas será igual a la cantidad de contenedores o bins, y la cantidad de filas será la sumatoria de los números del conjunto más $n-1$, siendo n la cantidad de números dentro del conjunto. Las restricciones de las columnas serán todas iguales al valor de la capacidad de los bins, y las restricciones de las filas representarán, en unario, los valores de los números del conjunto separados por 0.

Para entenderlo mejor, presentaremos un ejemplo: supongamos que tenemos el conjunto de números $\{11111, 11, 1111, 1\}$ y queremos ver si es posible que puedan dividirse en 11 bins de capacidad 111111 utilizando el problema de la batalla naval. Entonces transformamos los datos: los barcos serán $\{5, 2, 4, 1\}$, la matriz será de 2×15, las restricciones de las columnas serán 6 y las restricciones de las filas siguen el siguiente patrón: las primeras 5 filas, siendo 5 el primer valor en el conjunto, tienen como restricción 1, luego la siguiente tiene como restricción 0. Las próximas 2 filas tendrán 1 como restricción y terminarán con un 0, y continuaremos hasta completar con todos los números del conjunto, finalizando en este caso con el 1. En el siguiente gráfico podemos ver como quedaría la matriz:

1		
1		
1		
1		
1		
0		
1		
1		
0		
1		
1		
1		
1		
0		
1		
-	6	6

Cuadro 1: Matriz transformada del Bin Packing al Batalla Naval

Ahora podemos ingresar nuestro nuevo set de datos al algoritmo que resuelva el problema de la batalla naval, lo que daría por resultado lo siguiente:

1	1	0
1	1	0
1	1	0
1	1	0
1	1	0
0	0	0
1	0	1
1	0	1
0	0	0
1	0	1
1	0	1
1	0	1
1	0	1
0	0	0
1	1	0
-	6	6

Cuadro 2: Matriz que representa la solución del problema de Batalla Naval

Podemos ver que en cada columna se almacenan, en unario, los valores que deberian pertenecer a cada uno de los bins.

Ahora que comprendemos la lógica de la reducción, procedamos a analizar si es posible transformar los datos del problema de bin packing en tiempo polinomial.

Primero, la transformación del conjunto de números al conjunto de barcos puede realizarse en tiempo constante, bajo la suposición de que ambos conjuntos están representados con enteros en lugar de en notación unaria. En el caso de notación unaria, la transformación implicaría una mayor complejidad, específicamente $O(n)$, donde n representa la cantidad de unos en la representación unaria.

A continuación, la definición de las restricciones para las columnas es directamente proporcional a la capacidad de los bins, lo cual puede realizarse en tiempo polinómico.

De manera análoga, la definición de las restricciones para las filas, aunque implique una lógica adicional, también es factible en tiempo polinómico. Cabe destacar que la complejidad asociada a este proceso sería $O(n)$, siendo n la cantidad de unos en la representación unaria.

Podemos concluir, entonces, que la reducción es valida, y el problema de la batalla naval se encuentra en NP-Completo.

3.3. ALGORITMO

3.3.1. Complejidad del Algoritmo

El número total de soluciones posibles crece exponencialmente con el tamaño del espacio de búsqueda. Dado que cada posición en la cuadrícula puede estar ocupada por un barco o no, la cantidad de configuraciones posibles es de orden $O(2^n)$, donde n es el número de celdas en la cuadrícula.

Al ser un problema NP-completo, la solución óptima no puede ser encontrada en un tiempo polinomial en el peor de los casos. La complejidad del algoritmo sin optimización es exponencial, lo que hace que la resolución del problema en instancias grandes sea imposible debido al crecimiento desmesurado del espacio de soluciones.

3.3.2. Optimización mediante Podas

Para hacer frente a esta complejidad, el algoritmo utiliza una serie de técnicas de poda para reducir el espacio de búsqueda.

Poda basada en la cantidad de barcos restantes

Una de las podas más efectivas es la que ocurre cuando no quedan más barcos para colocar. Si el número de barcos restantes es cero, significa que el algoritmo ya ha completado la colocación y no necesita continuar explorando esa rama del árbol de búsqueda. Este tipo de poda permite cortar ramas de manera temprana, lo que reduce el tiempo de cómputo.

Poda por demanda máxima posible

Otra poda importante se basa en la demanda máxima posible de la configuración actual. Si la demanda máxima de la solución parcial es menor o igual a la demanda cumplida de la mejor solución encontrada hasta el momento, se puede descartar esa rama. Este tipo de poda es esencial para evitar explorar soluciones que no tienen el potencial de mejorar la mejor solución encontrada.

Poda por demanda cumplida y barcos restantes

Cuando la suma de la demanda cumplida hasta el momento y los barcos restantes es menor o igual a la demanda cumplida de la mejor solución, la rama también se poda. Esto ocurre porque incluso si los barcos restantes se colocan de manera óptima, no será posible mejorar la solución actual.

Poda cuando se alcanza la solución óptima

Si la mejor solución ya ha alcanzado la demanda total, es decir, la suma de las restricciones de filas y columnas se ha cumplido, el algoritmo ya ha encontrado una solución óptima. En este caso, la rama se poda, ya que no hay necesidad de continuar buscando.

3.4. Conclusion

El algoritmo propuesto presenta una complejidad exponencial. Las técnicas de poda implementadas en la solución actual han sido seleccionadas para optimizar el rendimiento dentro de las posibilidades disponibles. Se considera que existen otras podas adicionales que podrían contribuir a mejorar aún más la eficiencia del algoritmo.