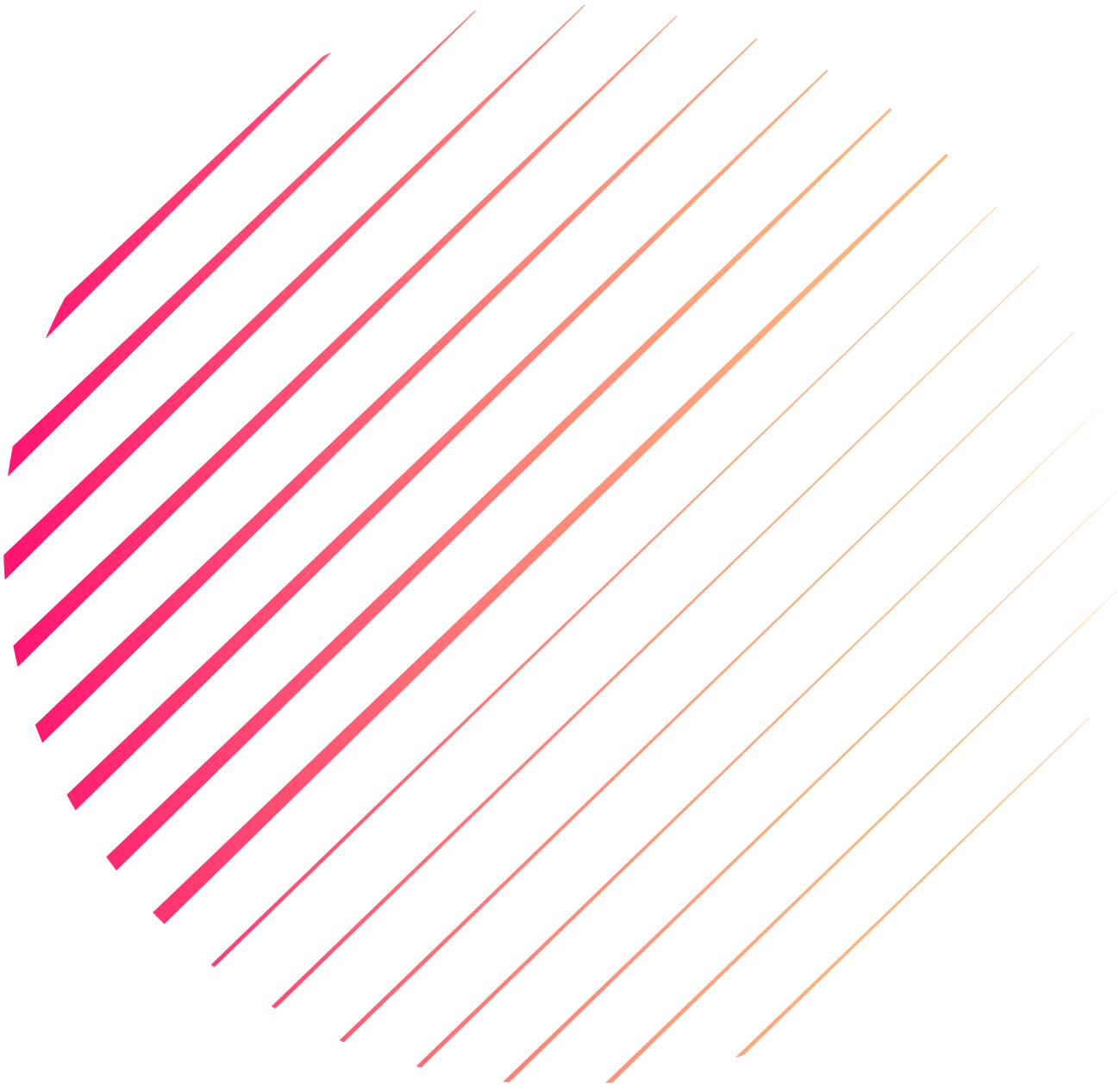
POGRAMACIÓN DINÁMICA



KATERYNA PASTERNAK
IVÁN ANDRÉS YAGUAL MÉNDEZ

PROBLEMA 4

1. INTRODUCCIÓN.

Dado un paño de pared, de M metros de longitud, y N estanterías cuyas medidas son de altura a y ancho b, se debe determinar la manera de llenar la pared utilizando el mayor número de estanterías posibles. Cada estantería puede colocarse de manera horizontal o vertical.

El objetivo es encontrar la combinación óptima de orientaciones de las estanterías de tal modo que maximice su uso, sin exceder la longitud de la pared.

Resolver este problema requiere explorar distintas combinaciones de estanterías y orientaciones, lo que resulta ineficiente si se hace mediante búsqueda exhaustiva. Por ello, se hará uso de programación dinámica, que permite descomponer el problema en subproblemas más pequeños y resolverlo de manera eficiente.

2. ECUACIÓN DE RECURRENCIA.

Para modelar este problema, definimos una función de programación dinámica. La ecuación de recurrencia es la siguiente:

1. Caso base:
    - Si la longitud de la pared es 0, no es posible colocar ninguna estantería.
    - Si no quedan estanterías disponibles, tampoco es posible colocar ninguna.
2. Caso imposible:
    - Si la longitud de la pared es negativa, el estado es inválido, y se define el resultado como -∞. Esto asegura que este estado no sea elegido como parte de la solución óptima.
3. Caso recursivo:
    - Si hay suficiente espacio para considerar al menos una de las orientaciones de la estantería actual, el problema se reduce a elegir entre tres opciones:
        1. **No usar la estantería actual:**
           Es decir, resolver el problema ignorando la estantería.
        2. **Usar la estantería en orientación horizontal:**
           Se suma porque se ha colocado una estantería, y el problema se reduce a llenar los metros restantes con las estanterías restantes.
        3. **Usar estantería en orientación vertical:**
           De manera análoga, se suma y se reduce el espacio restante.

- La solución óptima se obtiene tomando el máximo de estas tres opciones.

Teniendo en cuenta los diferentes casos, el problema se resume en la siguiente ecuación:

$$\text{relleno\_pared}(M,N) = \begin{cases} 0 & \text{si } M = 0 \text{ o } N = 0, \\ -\infty & \text{si } M < 0, \\ \max \begin{pmatrix} \text{relleno\_pared}(M, N-1), \\ 1 + \text{relleno\_pared}(M - b_i, N-1), \\ 1 + \text{relleno\_pared}(M - a_i, N-1) \end{pmatrix} & \text{si } M \geq \min(a_i, b_i). \end{cases}$$

M: Metros de longitud de la pared.
N: Nº de estanterías.
a: Alto de la estantería.
b: Ancho de la estantería.

NOTA: A pesar de la que la ecuación contiene un caso imposible, cuando $M < 0$, este caso no ha sido implementado explícitamente en el código, ya que el algoritmo de programación dinámica maneja de forma implícita los casos donde la longitud de la pared es negativa.Esto se logra mediante la inicialización de la tabla de programación dinámica donde se asume que si el valor de j(longitud de la pared restante) es negativo, no se realiza ninguna acción en las iteraciones. Como resultado, la solución nunca permitirá un estado de longitud negativa y se evitará elegir ese camino como la solución óptima.

3. ECUACIÓN DE RECURRENCIA.

Cabe destacar que en el código se utiliza una notación diferente para algunas de las variables que se mencionan en la formulación matemática. A continuación se detallan las equivalencias entre la notación matemática y la utilizada en el código, con el fin de clarificar la correspondencia entre ambas:

- N(número de estanterías) se representa como ***n_shelves*** en el código.
- M(longitud de la pared) se representa como ***wall_shelves*** en el código.
- a (alto de la estantería) se representa como ***hi*** en el código.
- b(ancho de la estantería) se representa como ***wi*** en el código.

Estas modificaciones de nombres se realizan para mejorar la legibilidad del código y para seguir convenciones estándar de nomenclatura en programación, sin alterar la estructura o lógica del algoritmo original. Las equivalencias anteriores ayudan a comprender cómo se mapean las variables del modelo matemático a la implementación en el código.

4. CONSTRUCCIÓN DE TABLA (ENFOQUE ASCENDENTE).

Para resolver este problema mediante programación dinámica, se construye una tabla bidimensional dp, donde cada celda $dp[i][j]$ representa el número máximo de estanterías que se pueden colocar en una pared de longitud j utilizando las primeras i estanterías disponibles. La construcción de esta tabla se realiza con un enfoque ascendente, resolviendo subproblemas pequeños y utilizando estas soluciones para construir la respuesta final.

CONSTRUCCIÓN DE LA TABLA.

1. **Inicialización de la tabla:**
   La tabla dp se inicializa con valores de cero. Esto refleja, que si no hay estanterías disponibles o la longitud de la pared es cero, no se pueden colocar estanterías.
   - $dp[0, j] = 0$ para todo j: Sin estanterías disponibles , el máximo es 0.
   - $dp[i, 0] = 0$ para todo i: Pared de longitud 0, no cabe ninguna estantería.

2. **Relleno de la tabla:**
   A medida que se van evaluando las estanterías y la longitud de la pared decrece, se evalúan los tres posibles casos:
   - **No usar la estantería actual i:** El valor es el mismo que el obtenido sin incluir esta estantería.
   - **Usar la estantería en orientación horizontal:** Si j es mayor o igual al ancho de la estantería (b), podemos incluirla y sumar uno al resultado obtenido.
   - **Usar la estantería en orientación vertical:**Si j es mayor o igual al alto de la estantería (a), podemos incluirla y sumar uno al resultado obtenido.

3. **Transición de estados:**
   La transición de estados se realiza mediante:
   $dp[i][j] = max(dp[i-1],\ 1 + dp[i-1][j - b_i],\ 1 + dp[i-1][j - a_i])$
   donde b_i y a_i son el ancho y el alto de la estantería i.

IMPLEMENTACIÓN DE LA TABLA.

La implementación en Python para la construcción es:

```
dp = np.zeros((n_shelves + 1, wall_length + 1), dtype=int)

for i in range(1, n_shelves + 1):
    wi, hi = shelves[i - 1]  # Ancho y alto de la estantería
    for j in range(1, wall_length + 1):
        # Caso 1: No usar la estantería actual
        dp[i][j] = dp[i - 1][j]
        # Caso 2: Usar la estantería en orientación horizontal
        if j >= wi:
            dp[i][j] = max(dp[i][j], 1 + dp[i - 1][j - wi])
        # Caso 3: Usar la estantería en orientación vertical
        if j >= hi:
            dp[i][j] = max(dp[i][j], 1 + dp[i - 1][j - hi])
```

TABLA DE EJEMPLO.

Para $N = 4$ estanterías de dimensiones $(2, 3),\ (4, 1),\ (3, 2),\ (1, 5)$.

		j					
	Estantería/Pared	**0**	**1**	**2**	**3**	**4**	**5**
	0 (Ninguna)	0	0	0	0	0	0
i	**1 (2, 3)**	0	0	1	1	1	1
	2 (4, 1)	0	0	1	1	1	2
	3 (3, 2)	0	0	1	1	2	2
	4 (1, 5)	0	1	1	1	2	2

5. RECONSTRUCCIÓN DE LA SOLUCIÓN.

Una vez construida la tabla dp, el siguiente paso es rastrear las decisiones que llevaron al resultado óptimo almacenado en $dp[N][M]$. Este proceso identifica qué estanterías fueron utilizadas y en qué orientación.

RECONSTRUCCIÓN.

1. **Punto de inicio:**
   Inicia en $dp[N][M]$, donde N es el total de estanterías y M es la longitud total de la pared.

2. **Rastreo hacia atrás:**
   Para cada estantería i, comparamos el valor actual con el obtenido al ignorar la estantería $dp[i-1][j]$:
   - Si son iguales, la estantería no se utilizó.
   - Si $dp[i][j] = dp[i-1][j-b_i] + 1$, se usó en orientación horizontal (j se reduce en b_i).
   - Si $dp[i][j] = dp[i-1][j-a_i] + 1$, se usó en orientación vertical (j se reduce en a_i).
3. **Resultados:**
   Se genera una lista con la orientación de cada estantería:
   - **0:** Estantería no utilizada.
   - **1:** Estantería en orientación horizontal.
   - **2:** Estantería en orientación vertical.

IMPLEMENTACIÓN.

La implementación en Python para la reconstrucción es:

```
placement = [None] * n_shelves
j = wall_length

for i in range(n_shelves, 0, -1):
    wi, hi = shelves[i - 1]
    if dp[i][j] == dp[i - 1][j]:
        placement[i-1] = 0  # No se usa
    elif j >= wi and dp[i][j] == (dp[i-1][j - wi] + 1):
        placement[i-1] = 1  # Orientación horizontal
        j -= wi
    elif j >= hi and dp[i][j] == (dp[i-1][j - hi] + 1):
        placement[i-1] = 2  # Orientación vertical
        j -= hi
```

6. VALIDACIÓN DEL ALGORITMO CON EJEMPLOS.

Para validar el algoritmo, se realizaron pruebas con ejemplos sencillos que permiten verificar su correcto funcionamiento. A continuación, se presentan los casos evaluados junto con los resultados obtenidos y su justificación.

Caso 1:

Parámetros:

- Longitud de la pared: 5
- Estanterías disponibles: [(2, 3), (4, 1), (3, 2), (1, 5)]

Resultados:

- Orientaciones de las estanterías: [1, 2, 2, 0]
    - Estantería 1: Usada horizontalmente (ocupa 2 unidades de la pared).
    - Estantería 2: Usada verticalmente (ocupa 1 unidad de la pared).
    - Estantería 3: Usada verticalmente (ocupa 2 unidades de la pared).
    - Estantería 4: No usada.
- Máximo número de estanterías usadas: 3.
- Espacio ocupado: 5.

Justificación: El algoritmo seleccionó tres estanterías que se ajustan perfectamente a la longitud de la pared de 5 unidades. Este resultado es óptimo porque no es posible colocar más de tres estanterías dentro de la longitud disponible sin superar el límite.

Caso 2:

Parámetros:

- Longitud de la pared: 8
- Estanterías disponibles: [(4, 5), (3, 7), (8, 2), (7, 2), (2, 2)]

Resultados:

- Orientaciones de las estanterías: [0, 1, 2, 2, 1]
    - Estantería 1: No usada.
    - Estantería 2: Usada horizontalmente (ocupa 3 unidades de la pared).
    - Estantería 3: Usada verticalmente (ocupa 2 unidades de la pared).
    - Estantería 4: Usada verticalmente (ocupa 1 unidad de la pared).
    - Estantería 5: Usada horizontalmente (ocupa 2 unidades de la pared).
- Máximo número de estanterías usadas: 4.
- Espacio ocupado: 8.

Justificación: El algoritmo seleccionó cuatro estanterías que se ajustan perfectamente a la longitud de la pared de 8 unidades. Este resultado es óptimo porque no es posible colocar más de cuatro estanterías dentro de la longitud disponible sin superar el límite.

Caso 3:

Parámetros:

- Longitud de la pared: 8
- Estanterías disponibles: [(5, 5), (7, 7)]

Resultados:

7. Orientaciones de las estanterías: [0, 0]
    a. Estantería 1: No usada.
    b. Estantería 2: No usada.
8. Máximo número de estanterías usadas: 0.
9. Espacio ocupado: 3.

Justificación: : La longitud de la pared (3 unidades) es menor que las dimensiones mínimas de las estanterías disponibles. Por lo tanto, ninguna estantería puede ser colocada, y el resultado es óptimo ya que no existe solución factible.

CONCLUSIONES DE LA VALIDACIÓN

Los resultados obtenidos para los ejemplos evaluados son consistentes con los objetivos del problema. El algoritmo demuestra ser capaz de:

- Maximizar el número de estanterías colocadas dentro de la longitud de la pared.
- Seleccionar orientaciones adecuadas para minimizar el desperdicio de espacio.
- Identificar casos en los que no es posible colocar ninguna estantería.

Estos ejemplos simples validan la correcta implementación y el enfoque del algoritmo.

10. ESTUDIO TEÓRICO.

El algoritmo se compone de dos partes principales:

1. **Rellenar la tabla de programación dinámica**:
   En esta fase, se recorre una tabla bidimensional de tamaño $N \times M$, donde:
    - N es el número de estanterías disponibles.
    - M es la longitud de la pared.

   Por cada celda de la tabla, se realizan un número constante de operaciones para calcular el valor óptimo. Esto implica un tiempo de ejecución en esta fase de $O(N \times M)$.

2. **Reconstruir la solución**:
   En esta fase, se realiza un recorrido hacia atrás desde el final de la tabla para determinar la orientación óptima de las estanterías y si cada estantería se usa o no. Esto implica recorrer N, el número de estanterías, con un tiempo de ejecución de $O(N)$.

Estimación final del orden del algoritmo:

Sumando ambas partes, el tiempo total de ejecución se puede expresar como:

$$O(N \times M) + O(N),$$

entonces, la complejidad final del algoritmo es:

$$O(N \times M)$$

Este resultado es bastante razonable para un problema de programación dinámica de esta naturaleza.

11. CONCLUSIONES.

El algoritmo de programación dinámica desarrollado resultó ser bastante efectivo para este tipo de problema. Fue capaz de encontrar soluciones correctas y óptimas en todos los casos probados, garantizando un uso eficiente de los recursos disponibles (las estanterías y la longitud de la pared).

Un aspecto destacado del enfoque fue la capacidad de generar no solo la solución para la longitud de pared máxima especificada, sino también para cualquier longitud menor, gracias a la matriz obtenida durante el proceso de cálculo. Esto hace que el algoritmo sea versátil y reutilizable para variaciones del mismo problema con mínimas modificaciones.

El recorrido hacia atrás para reconstruir la solución fue sencillo y eficiente. Este paso no solo permitió determinar qué estanterías se usaban, sino también su orientación y si cada una contribuía a la solución final.

Estimación del tiempo invertido por cada miembro del grupo:

- **KATERYNA PASTERNAK:** Desarrollo del algoritmo, pruebas iniciales y corrección de errores: **3 horas.**
- **IVÁN ANDRÉS YAGUAL:** Análisis teórico de la complejidad, validaciones con ejemplos y documentación: **4 horas.**
- **Ambos:** Ajustes finales y revisiones conjuntas: **1 horas.**

Total estimado: 8 horas.