Solución con Programación Dinámica

Integrantes

- Christian Echeverría 221441
- Gustavo Cruz 22779
- Josué Say 22801
- Mathew Cordero 22982
- Pedro Guzmán 22111

Definición de Programación Dinámica

La **programación dinámica** es una técnica de diseño de algoritmos que busca resolver problemas complejos dividiéndolos en subproblemas más pequeños y simples, evitando el recálculo de subproblemas superpuestos y aprovechando la reutilización de resultados. Para ello, se almacena la solución de cada subproblema en una estructura de datos (por ejemplo, una tabla o un diccionario), lo que permite mejorar la eficiencia del algoritmo en términos de tiempo y espacio.

Pasos de Soltys para Programación Dinámica

Según Soltys (2012), los pasos para desarrollar una solución con programación dinámica son los siguientes:

1. **Definir una clase de subproblemas**:
    - Identificar los subproblemas más pequeños en los que se puede descomponer el problema original.
    - Verificar que dichos subproblemas se solapan, es decir, se repiten dentro de la solución global.
2. **Proveer una recurrencia que resuelva los problemas en términos de subproblemas**:
    - Establecer la relación matemática o fórmula de recurrencia que describa cómo la solución de cada subproblema contribuye a la solución del problema completo.
    - Esta recurrencia debe aprovechar la estructura óptima del problema (la solución óptima global puede construirse a partir de soluciones óptimas de subproblemas).
3. **Proveer un algoritmo que compute la recurrencia**:
    - Diseñar un procedimiento, generalmente iterativo (tabulación) o recursivo con memoización, que calcule todas las soluciones de los subproblemas y las combine para obtener la solución del problema original.

- Asegurarse de no recalcular subproblemas, evitando así la redundancia en los cómputos.

Ejemplo: Problema de la Mochila 0/1

Para ilustrar estos pasos, consideremos el **Problema de la Mochila 0/1.** Dado un conjunto de n objetos, cada uno con un peso w_i y un valor v_i, y una capacidad máxima de la mochila W, se busca la combinación de objetos que maximice el valor total sin exceder el peso máximo permitido.

1. **Definición de subproblemas**:
   - Sea $dp[i][j]$ la solución óptima (valor máximo) al considerar los primeros i objetos con una capacidad de mochila de j.
   - Es decir, $dp[i][j]$ representa la mejor ganancia (valor) posible usando objetos entre 1 y i, sin exceder el peso j.
2. **Recurrencia**:
   - Si no tomamos el objeto i, la solución se mantiene igual a la de $i-1$ con capacidad j:

$$dp[i][j] = dp[i-1][j]$$

   - Si tomamos el objeto i, sumamos el valor de dicho objeto v_i y nos quedamos con la capacidad restante $j - w_i$:

$$dp[i][j] = dp[i-1][j-w_i] + v_i$$

   - Por lo tanto, la recurrencia se define como:

$$dp[i][j] = \max(dp[i-1][j],\ dp[i-1][j-w_i] + v_i)$$

   - Con la restricción de que $j \geq w_i$ (para poder incluir el objeto i).
   - Casos base:

$$dp[0][j] = 0 \quad \text{para todo } j, \quad dp[i][0] = 0 \quad \text{para todo } i.$$

3. **Algoritmo** (Tabulación):
   - A continuación se presenta una implementación en Python:

“‘python def knapsack_01(weights, values, W): """ weights: lista con los pesos de los objetos values: lista con los valores de los objetos W: capacidad máxima de la mochila Retorna: valor máximo que se puede obtener sin exceder W """ n =

```
len(weights) # Creamos una tabla (n+1) x (W+1) para almacenar los resultados
dp = [[0] * (W + 1) for _ in range(n + 1)]

for i in range(1, n + 1):
    for j in range(1, W + 1):
        # Caso 1: no tomar el objeto i-ésimo
        dp[i][j] = dp[i-1][j]

        # Caso 2: tomar el objeto i-ésimo (si cabe en la mochila)
        if weights[i-1] <= j:
            dp[i][j] = max(dp[i][j], dp[i-1][j - weights[i-1]] + values[i-1])

# La respuesta está en dp[n][W]
return dp[n][W]
```

Ejemplo de uso

if **name** == “**main**”: pesos = [2, 3, 4, 5] valores = [3, 4, 5, 6] capacidad = 5 print(“Valor máximo:”, knapsack_01(pesos, valores, capacidad))