Estructuras de Datos No Lineales: Programación Dinámica

Juan Vasquez, Santiago Rodriguez, Miguel Celis

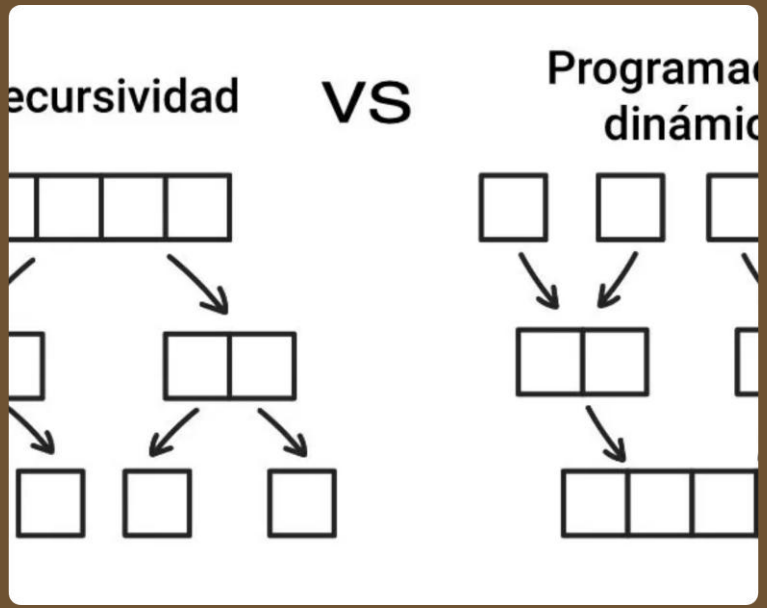
Introducción a la Programación Dinámica

La **programación dinámica** es una técnica algorítmica que resuelve problemas complejos dividiéndolos en subproblemas más simples y almacenando sus soluciones para evitar recalcularlas.

Fue desarrollada por Richard Bellman en la década de 1950 y se ha convertido en una herramienta fundamental en la optimización y resolución de problemas.

Se basa en dos conceptos clave:

- **Subestructura óptima:**
  La solución óptima contiene soluciones óptimas de subproblemas
- **Subproblemas superpuestos:**
  El mismo subproblema se resuelve múltiples veces



Programación Dinámica: Descripción y Definición

> **Definición formal:** La programación dinámica es un método para resolver problemas complejos descomponiéndolos en subproblemas más simples, resolviendo cada subproblema una sola vez y almacenando sus soluciones.

Se basa en el **principio de optimalidad de Bellman** : una solución óptima a un problema contiene soluciones óptimas a sus subproblemas.

El proceso de resolución sigue estos pasos:

- Definir la estructura de la solución óptima
  Formular recursivamente el valor de la solución óptima
- Calcular el valor de la solución óptima de manera ascendente

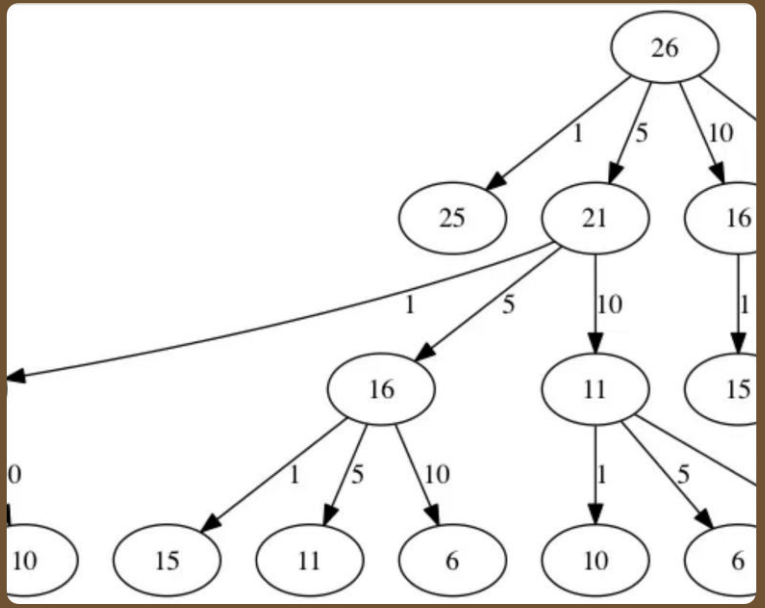
Programación Dinámica: Características

Subestructura óptima: La solución óptima del problema contiene dentro de sí las soluciones óptimas de los subproblemas.

Subproblemas superpuestos: El mismo subproblema se resuelve múltiples veces al resolver el problema original de forma recursiva.

Memorización (memoization): Almacenamiento de resultados de subproblemas para evitar recalcularlos, mejorando significativamente la eficiencia.

Enfoque ascendente: Resolución de subproblemas más pequeños primero y construcción progresiva de soluciones para problemas más grandes.



Programación Dinámica: Usos y Aplicaciones

La programación dinámica se aplica en numerosos campos para resolver problemas de optimización:

Teoría de grafos

Algoritmos de ruta más corta como Floyd-Warshall y Bellman-Ford

Bioinformática

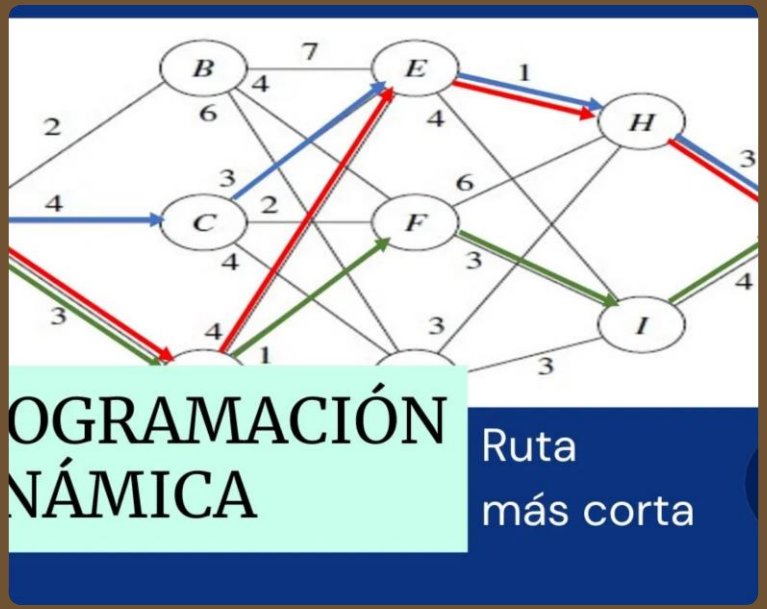
Alineamiento de secuencias de ADN y proteínas

Procesamiento de texto

Corrección ortográfica y distancia entre cadenas

Optimización de recursos

Problema de la mochila y planificación de tareas



Problemas clásicos resueltos con programación dinámica:

- **Subsecuencia común más larga**
- **Distancia de edición**
- **Problema de la mochila**

Multiplicación de cadenas de matrices

Subsecuencia Común más Larga: Descripción y Definición

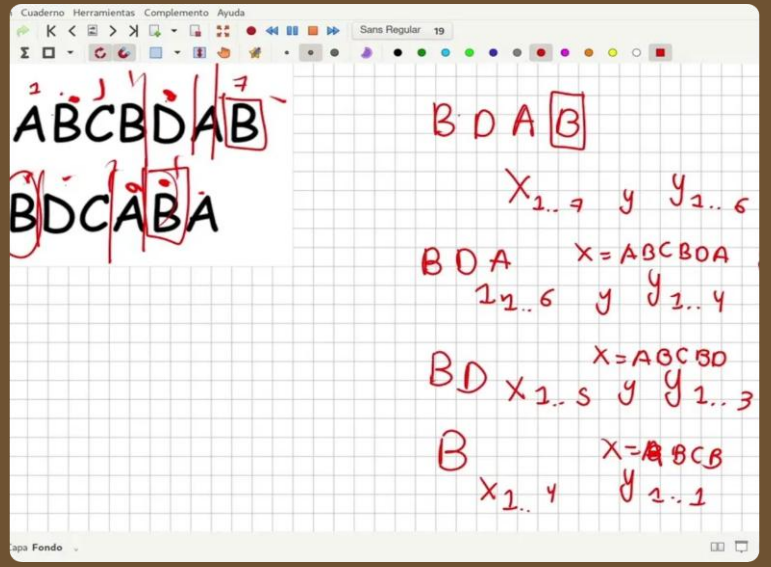
La **Subsecuencia Común más Larga** (LCS, por sus siglas en inglés) es un problema que busca encontrar la secuencia más larga que es subsecuencia de dos o más secuencias dadas.

Una **subsecuencia** es una secuencia que puede derivarse de otra secuencia eliminando algunos elementos sin cambiar el orden de los elementos restantes.

El problema LCS tiene **subestructura óptima** y **subproblemas superpuestos**, lo que lo hace ideal para aplicar programación dinámica.

Ejemplo:

Secuencia 1: **ABCBDAB**
Secuencia 2: **BDCABA**
LCS: **BCBA** (longitud 4)




```python
def LCS(args1, args2):
  n = len(args1)
  m = len(args2)

  matrix = [[0] * (m + 1) for i in range(n + 1)]

  for i in range(1,n + 1):
    for j in range(1,m + 1):
      if args1[i - 1] == args2[j - 1]:
        matrix[i][j] = matrix[i - 1][j - 1] + 1
      else:
        matrix[i][j] = max(matrix[i - 1][j], matrix[i][j - 1])

  return matrix[n][m]

LCS("ABCDGH", "AEDFHR")
```

Subsecuencia Común más Larga: Características y Aplicaciones

Características

- Complejidad temporal: O(m×n) donde m y n son las longitudes de las secuencias
- Complejidad espacial: O(m×n) para almacenar la tabla de programación dinámica

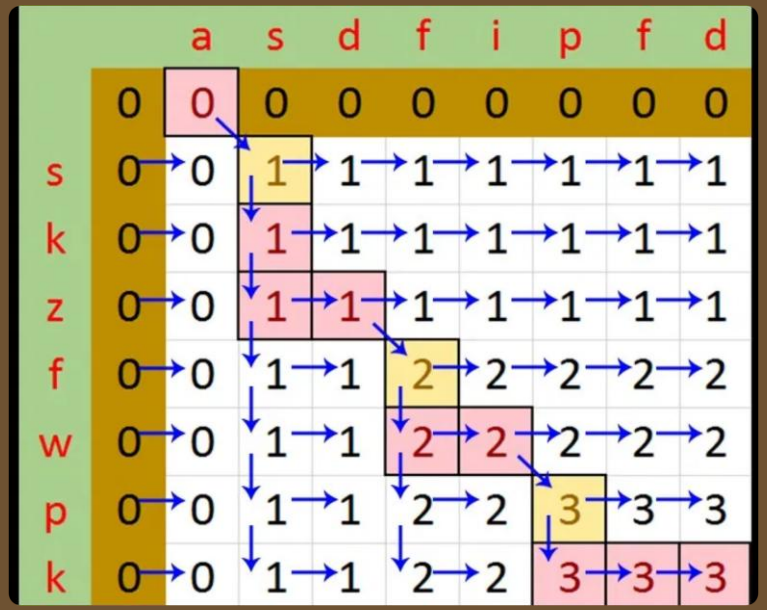
Aplicaciones

Bioinformática

Comparación de secuencias de ADN, ARN y proteínas

Control de versiones

Detección de cambios entre versiones de archivos



Distancia de Edición: Descripción y Definición

La **Distancia de Edición** (o Distancia de Levenshtein) es una medida de la diferencia entre dos cadenas de caracteres. Se define como el número mínimo de operaciones elementales necesarias para transformar una cadena en otra.

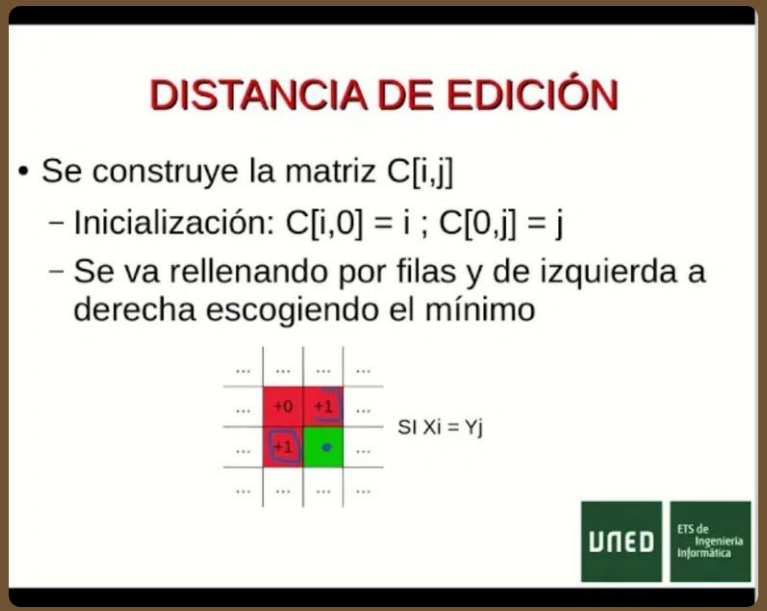
Este problema es un ejemplo clásico de aplicación de programación dinámica, donde se construye una matriz para almacenar resultados de subproblemas.

Operaciones permitidas:

- ➕ Inserción de un carácter
- ➖ Eliminación de un carácter
- 🔁 Sustitución de un carácter

Ejemplo:

Cadena 1: CASA



Distancia de Edición: Características y Aplicaciones

Características

- Complejidad temporal: O(m×n) donde m y n son las longitudes de las cadenas
- Complejidad espacial: O(m×n) para la matriz de programación dinámica
- Métrica: Cumple las propiedades de no negatividad, identidad, simetría y desigualdad triangular

Aplicaciones

Correctores ortográficos

Sugerencia de palabras correctas basadas en la similitud

Bioinformática

Comparación de secuencias genéticas

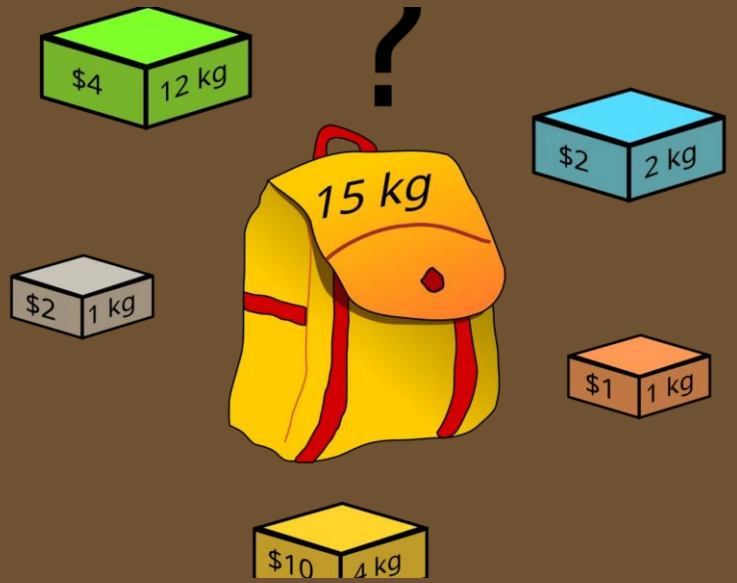
distancia de la edicion.py

C: > Users > juanp > AppData > Roaming > .minecraft > versions > NEOFORGE > distancia de la edicion.py > ...

```python
def levenshtein(A, B):
    m, n = len(A), len(B)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if A[i-1] == B[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(
                    dp[i-1][j],    # eliminar
                    dp[i][j-1],    # insertar
                    dp[i-1][j-1]   # sustituir
                )

    return dp[m][n]
```

Problema de la Mochila: Descripción y Definición

El **Problema de la Mochila** (Knapsack Problem) es un problema de optimización combinatoria donde se busca seleccionar objetos con peso y valor para maximizar el valor total sin exceder una capacidad máxima.



Es un problema **NP-completo**, pero puede resolverse eficientemente mediante programación dinámica cuando los pesos son valores enteros.

Planteamiento formal:

Dados n objetos (cada uno con peso w_i y valor v_i) y una mochila con capacidad W, encontrar un subconjunto de objetos que maximice el valor total sin exceder W.

Variantes principales:

- **0/1 Knapsack:** Cada objeto puede seleccionarse una vez o ninguna
- **Unbounded Knapsack:** Cada objeto puede seleccionarse múltiples veces

Problema de la Mochila: Características y Aplicaciones

Características

- Complejidad temporal: O(nW) donde n es el número de objetos y W la capacidad
- Complejidad espacial: O(nW) para la tabla de programación dinámica

Aplicaciones

Logística y transporte

Optimización de carga en vehículos

Finanzas

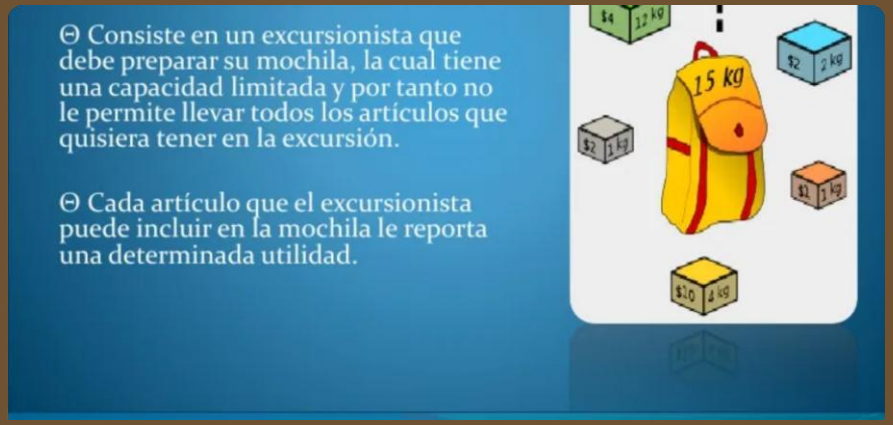
Selección de inversiones con presupuesto limitado

Asignación de recursos

Distribución óptima de recursos limitados

Corte de materiales

Minimización de desperdicio en procesos industriales



Implementación del Problema de la Mochila

Implementación en Python del problema de la mochila 0/1 utilizando programación dinámica:

```
# Función para resolver el problema de la mochila 0/1
def knapsack(W, weights, values):
    n = len(weights)

    # Crear tabla para almacenar resultados
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    # Llenar la tabla
    for i in range(1, n + 1):
        for w in range(1, W + 1):
            if weights[i-1] > w:
                dp[i][w] = dp[i-1][w]
            else:
                dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]],
                               dp[i-1][w])

    return dp[n][W]
```

```
# Solve the Knapsack problem
def Knapsack(W: itemś):
    # Get number of items
    n = léntitems)
    # Initialize te toble
    K = ⁞[8] +(W + 1) fórΓn__• in range(n + 1)]
    # Fíll the table
    for i in range(], n + 1):
        weight, value = items[í −1]
        for w in range(W +1):
          # check ir thé weight˙f item exceeds the capacity
          if weight`> w:
             K[í][w] = K[í −1][w]
          else:  K[í][ẃ] = max(kinclude the item
        # Return the maximum value
    return K[n][w]
```