Reporte: Laboratorio de Consolidación - Programación Dinámica

Semana 2: Estructuras de Datos Avanzadas

Introducción

Este laboratorio consolida la transición de **recursión ingenua** a **Programación Dinámica (DP)** mediante ejercicios prácticos incrementales. El objetivo es desarrollar la habilidad de identificar cuándo aplicar DP y cómo transformar soluciones recursivas en soluciones eficientes.

Ejercicio 1: Detección de Patrones

Objetivo

Identificar qué funciones recursivas tienen subproblemas repetidos y se beneficiarían de DP.

Análisis de Funciones

Función A: Tribonacci

```
def misterio_A(n):
    if n <= 0: return 0
    if n == 1: return 1
    if n == 2: return 1
    return misterio_A(n-1) + misterio_A(n-2) + misterio_A(n-3)
```

Resultado: ☑ TIENE subproblemas repetidos

- Para n=4, `misterio_A(2)` se calcula 2 veces
- `misterio_A(1)` se calcula 4 veces
- **Conclusión:** Se beneficiaría enormemente de DP

Función B: Suma Acumulativa

```
def misterio_B(n):
    if n <= 1: return n
    return misterio_B(n-1) + n
```

Resultado: ❌ NO tiene subproblemas repetidos

- Es una cadena lineal, no un árbol

- Cada valor se calcula exactamente una vez
- **Conclusión:** Ya es O(n), DP no ayudaría

Función C: Factorial

```
def misterio_C(n):
    if n <= 1: return 1
    return n * misterio_C(n-1)
```

Resultado: ❌ NO tiene subproblemas repetidos

- Cadena lineal de llamadas
- Cada factorial se calcula una sola vez
- **Conclusión:** Ya es óptimo en O(n)

Criterio de Identificación

Una función se beneficia de DP si:

1. Tiene **múltiples llamadas recursivas** en cada nivel
2. Los **mismos valores** se calculan más de una vez
3. El árbol de llamadas tiene **ramas que se solapan**

Ejercicio 2: Transformación Guiada

Problema

Contar de cuántas formas se puede formar el número `n` usando pasos de 1 o 2.

Paso 1: Versión Ingenua

```
def formas_ingenuo(n):
    if n <= 0: return 1
    if n == 1: return 1
    return formas_ingenuo(n-1) + formas_ingenuo(n-2)
```

Complejidad: O(2^n) - Exponencial

Paso 2: Identificación de Repeticiones

Para `formas(4)`:

- `formas(2)` se calcula **2 veces**
- `formas(1)` se calcula **3 veces**
- `formas(0)` se calcula **2 veces**

Paso 3: Memoización (Top-Down)

```python
def formas_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n <= 0: return 1
    if n == 1: return 1
    if n in memo:
        return memo[n]
    resultado = formas_memo(n-1, memo) + formas_memo(n-2, memo)
    memo[n] = resultado
    return resultado
```

Complejidad: O(n) - Lineal

Paso 4: Tabulación (Bottom-Up)

```python
def formas_tabla(n):
    if n <= 1: return 1
    dp = [0] * (n + 1)
    dp[0] = 1
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Complejidad: O(n) - Lineal

Comparación de Rendimiento (n=30)

Método	Tiempo (ms)
Ingenua	~300
Memoización	~0.05
Tabulación	~0.02

Mejora: ~15,000x más rápido con DP

Ejercicio 3: Problema del Cambio de Monedas

Enunciado

Monedas disponibles: `[1, 3, 4]` centavos
Objetivo: Encontrar el número **mínimo** de monedas para formar `n` centavos.

Casos Manuales

n	Combinación óptima	Número de monedas
1	1	1
2	1+1	2
3	3	1
4	4	1
5	4+1	2
6	3+3	2

Relación de Recurrencia

```
min_monedas(n) = 1 + min(
    min_monedas(n-1),  # si uso moneda de 1
    min_monedas(n-3),  # si uso moneda de 3
    min_monedas(n-4)   # si uso moneda de 4
)
```

El `+1` representa usar **una moneda**, más el óptimo para el resto.

Implementación DP

```python
def min_monedas(n, monedas=[1,3,4]):
    if n == 0: return 0
    if n < 0: return float('inf')

    dp = [float('inf')] * (n + 1)
    dp[0] = 0

    for i in range(1, n + 1):
        for moneda in monedas:
            if i >= moneda:
                dp[i] = min(dp[i], 1 + dp[i - moneda])

    return dp[n] if dp[n] != float('inf') else -1
```

Complejidad

- **Temporal:** O(n × m), donde m = número de denominaciones
- **Espacial:** O(n)

Ejercicio 4: Debugging DP

Errores Comunes Identificados

Error 1: Tamaño de Array Insuficiente

```
# ❌ INCORRECTO
dp = [0] * n  # Si necesitas dp[n], esto falla

# ☑ CORRECTO
dp = [0] * (n + 1)  # Ahora dp[n] es válido
```

Error 2: Rango del Bucle Incorrecto

```
# ❌ INCORRECTO
for i in range(2, n):  # No incluye n

# ☑ CORRECTO
for i in range(2, n + 1):  # Incluye n
```

Error 3: No Manejar Casos Especiales

```
# ☑ CORRECTO
if n <= 0: return 0
if n == 1: return 1
# Ahora el resto del código es seguro
```

Checklist de Validación

- ☐ ¿Definí correctamente qué significa `dp[i]`?
- ☐ ¿Inicialicé TODOS los casos base?
- ☐ ¿El tamaño del array es suficiente?
- ☐ ¿El bucle llena la tabla en el orden correcto?
- ☐ ¿La recurrencia usa solo valores ya calculados?
- ☐ ¿Manejé casos especiales (n=0, n=1)?
- ☐ ¿Probé con casos pequeños?

Ejercicio 5: Mini-Proyecto - Salto de Ranas

Problema

Una rana puede saltar 1, 2 o 3 casillas. ¿De cuántas formas puede llegar a la casilla `n`?

Recurrencia

```
f(n) = f(n-1) + f(n-2) + f(n-3)
```

Implementaciones

1. Memoización

```
def salto_ranas_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n < 0: return 0
    if n == 0: return 1
    if n == 1: return 1
    if n == 2: return 2
    if n in memo:
        return memo[n]
    resultado = (salto_ranas_memo(n-1, memo) +
                 salto_ranas_memo(n-2, memo) +
                 salto_ranas_memo(n-3, memo))
    memo[n] = resultado
    return resultado
```

2. Tabulación

```
def salto_ranas_tabla(n):
    if n < 0: return 0
    if n <= 2: return [1, 1, 2][n]

    dp = [0] * (n + 1)
    dp[0], dp[1], dp[2] = 1, 1, 2

    for i in range(3, n + 1):
        dp[i] = dp[i-1] + dp[i-2] + dp[i-3]

    return dp[n]
```

3. Optimizado en Espacio

```
def salto_ranas_optimizado(n):
    if n < 0: return 0
    if n <= 2: return [1, 1, 2][n]

    a, b, c = 1, 1, 2  # últimos 3 valores

    for i in range(3, n + 1):
        nuevo = a + b + c
        a, b, c = b, c, nuevo

    return c
```

Comparación de Rendimiento (n=100)

Método	Tiempo (ms)	Espacio
Memoización	~0.15	O(n)
Tabulación	~0.08	O(n)
Optimizado	~0.06	O(1)

Resultados para Casos Pequeños

n	Formas
0	1
1	1
2	2
3	4
4	7
5	13
6	24

Conclusiones Generales

¿Cuál enfoque preferir?

Memoización (Top-Down)

Ventajas:

- Fácil de implementar desde la versión recursiva
- Solo calcula subproblemas necesarios
- Más intuitivo para principiantes

Desventajas:

- Usa más memoria (diccionario + pila de recursión)
- Overhead de llamadas recursivas
- Riesgo de stack overflow para n muy grandes

Tabulación (Bottom-Up)

Ventajas:

- Más rápida (sin overhead de recursión)

- Fácil de optimizar el espacio
- No hay riesgo de stack overflow

Desventajas:

- Requiere pensar en el orden de llenado
- Calcula todos los subproblemas (incluso innecesarios)
- Menos intuitivo al principio

Mi Recomendación

Para APRENDER: Memoización

- Es más natural pensar recursivamente
- Fácil de derivar de la definición del problema

Para PRODUCCIÓN: Tabulación u Optimizado

- Mejor rendimiento
- Más predecible
- Optimizado si solo necesitas el resultado final

Autoevaluación

- [x] Puedo identificar cuándo un problema recursivo tiene subproblemas repetidos
- [x] Entiendo la diferencia entre top-down (memo) y bottom-up (tabla)
- [x] Puedo transformar código recursivo a DP paso a paso
- [x] Sé cómo inicializar correctamente los casos base en DP
- [x] Puedo debuggear errores comunes en implementaciones DP

Referencias y Recursos Adicionales

- **Complejidad Temporal:** Todas las soluciones DP presentadas son O(n)
- **Complejidad Espacial:** Varía de O(1) a O(n) según la optimización
- **Mejora vs Recursión Ingenua:** Hasta 15,000x más rápido para n=30

Fecha de Completación: Diciembre 2025
Curso: Estructuras de Datos Avanzadas
Semana: 2 - Programación Dinámica