Análisis y Diseño de Algoritmos
Ejercicios Tema 5

Francisco Javier Mercader Martínez

1) En el problema de la mochila (igual que en el problema del cambio de monedas) puede existir en general más de un solución óptima para unas entradas determinadas. ¿Cómo se puede comprobar si una solución óptima es única o no, suponiendo que hemos resuelto el problema utilizando programación dinámica? Dar un algoritmo para que, a paritr de las tablas resultantes del problema de la mochila, muestre todas las soluciones óptimas.

Para determinar si una solución óptima al problema de la mochila es única y mostrar toas las soluciones óptimas, se puede utilziar el siguiente engoque basado en tablas generadas por el algoritmo de programación dinámica.

Algoritmo

1) Resuelve el problema de la mochila usando programación dinámica:
   - Construye la tabla `dp[i][w]`, donde `dp[i][w]` es el valor máximo que se puede obtener utilizando los primeros `i` elementos y un peso máximo `w`.
2) Encuentra el valor óptimo:
   - El valor óptimo se encuentra en `dp[n][W]`, donde n es el número de elementos y W es la capacidad máxima de la mochila.
3) Rastrea todas la soluciones posibles:
   - A partir de `dp[n][W]`, rastrea todas las combinaciones de elementos que generan el valor `dp[n][W]`.
   - Para cada elemento `i`, verifica si está incluido en la solución utilizando la condición:

     `dp[i][w]=dp[i-1][w]` o `dp[i][w]=dp[i-1][w-peso[i]] + valor[i]`.
   - Si el elemento puede estar incluido, explora ambas posibilidades (incluirlo y no incluirlo).
4) Construye todas las combinaciones:
   - Usa una búsqueda recursiva o iterativa para explorar todas las combinaciones posibles.
   - Si llegas a un peso `w=0` o `i=0`, almacena la combinación como una solución válida.
5) Verifica unicidad:
   - Si encuentras más de una solución válida, la solución no es única. Si solo encuentras una solución, es única.

Implementación en psudocódigo:

```
def find_all_solutions(dp, weights, values, n, W):
    solutions = []

    def bactrack(i, w, current_solution):
        if i == 0 or w == 0:
            solutions.append(current_solution[:])
            return

        # Caso donde el elemento no está incluido
```

```
        if dp[i][w] == dp[i - 1][w]:
            bactrack(i-1, w, current_solution)

        # Caso donde el elemento está incluido
        if w >= weights[i-1] and dp[i][w] == dp[i - 1][w - weights[i-1]] + values[i-1]:
            current_solution.append(i-1) # Incluye el íindice del elemento
            bactrack(i-1, w - weights[i-1], current_solution)
            current_solution.pop() # Deshace la inclusión para explorar otras ramas

    bactrack(n, W, [])
    return solutions

# Verificación de unicidad:
solutions = find_all_solutions(dp, weights, values, n, W)
if len(solutions) == 1:
    print(f"La solución óptima es única {solutions[0]}")
else:
    print(f"Existen múltiples soluciones óptima: {solutions}")
```

Explicación

1) Tabla `dp`:
   - Esta tabla se generó al resolver el problema de la mochila mediante programación dinámica.
2) Búsqueda de todas las soluciones:
   - La función `backtrack` recorre la tabla `dp` desde `dp[n][W]` hacia atrás, explorando todas las combinaciones posibles de inclusión y exclusión de elementos que generan el valor óptimo.
3) Verificación de unicidad:
   - La lista `solutions` contendrá todas las combinaciones óptimas. Si su longitud es mayor a 1, significa que hay múltiples soluciones.

Complejidad:

- La complejidad temporal adicional para encontrar todas las soluciones es proporcional al número de combinaciones óptimas, que puede ser exponencial en el peor caso.
- Sin embargo, este proceso es manejable para entradas pequeñas o moderadas.

2) El número de combinaciones de `m` objetos entre un conjunto de `n`, denotado por $\binom{n}{m}$, para $n \leq 1$ y $0 \leq m \leq n$, se puede definir recurisvamente por:

$$\binom{n}{m} = 1 \qquad \text{Si } m = 0 \text{ ó } m = n$$
$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} \qquad \text{Si } 0 < m < n$$

Conociendo el resultado puede er calculado también con la fórmula: $\dfrac{n!}{m! \cdot (n-m)!}$

a) Dar una función recursiva para calcular $\binom{n}{m}$, usando la primera de las definiciones. ¿Cuál será el orden de complejidad de este algoritmo? Sugerencia: la respuesta es inmediata.

La función recursiva se basa directametne en la definición dada:

1) Definición recursiva:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0\, m = n \\ \binom{n-1}{m} + \binom{n-1}{m-1} & \text{si } 0 < m < n \end{cases}$$

2) Implementación en Python:

```
def binom_rec(n, m):
    if m == 0 or m == n:
        return 1
    return binom_rec(n-1, m) + binom_rec(n-1, m-1)
```

3) Complejidad:

La complejidad de este algoritmo es **exponencial**, $O(2^n)$, porque calcula repetidamente los mismos valores para diferentes combinaciones de n y m. Por ejemplo, calcular $\binom{n-1}{m}$, se vuelve a calcular los subproblemas de $\binom{n-2}{m}$ y $\binom{n-2}{m-1}$, causando redundancia.

b) Diseñar un algoritmo de programación dinámica para calcular $\binom{n}{m}$. Nota: la tabla construida por el algoritmo es conocida como "el triángulo de Pasca". ¿Cuál será el tiempo de ejecución en este caso?

Para evitar cálculos repetidos, podemos usar un enfoque de **programación dinámica** que construye una tabla que almacena los valores de $\binom{n}{m}$. Esta tabla es conocida como el "Triángulo de Pascal".

1) Idea básica:

- Construimos: una tabla `dp` de dimensiones $(n+1) \times (m+1)$.
- Inicializamos las condiciones base:
  - `dp[i][0]=1` para todo `i`, porque $\binom{\texttt{i}}{0} = 1$.
  - `dp[i][i]=1` para todo `i`, porque $\binom{\texttt{i}}{\texttt{i}} = 1$.
- Rellemanos la tabla usando la fórmula recursiva:

$$\texttt{dp[i][j]=dp[i-1][j]+dp[i-1][j-1]}.$$

c) Implementación en Python

```
def binom_dp(n, m):
    # Crear la tabla dp
    dp = [[0 for _ in range(m + 1)] for _ in range(n + 1)]

    # Condiciones base
    for i in range(n + 1):
        dp[i][0] = 1  # Caso m = 0
        if i <= m:
            dp[i][i] = 1  # Caso m = n

    # Llenar la tabla
    for i in range(1, n + 1):
        for j in range(1, min(i, m + 1)):
            dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1]

    return dp[n][m]
```

d) Complejidad:

- **Tiempo:** $O(n \cdot m)$. Cada celda en la tabla se calcula una vez y depende de dos valores previos.

e) Ejemplo de uso:

```
n, m = 5, 2
print(f"Resultado con recursión: {binom_rec(n, m)}")
print(f"Resultado con DP: {binom_dp(n, m)}")

## Resultado con recursión: 10
## Resultado con DP: 10
```

Comparación de enfoques:

Enfoque	**Complejidad temporal**	**Ventajas/Desventajas**
Recursivo	$O(2^n)$	Simple de implementar pero extremadamente ineficiente.
Programación dinámica	$O(n \cdot m)$	Eficiente, especialmente para valores grandes de n y m.

3) Considerar el problema de la mochila 0/1. En este ejercicio estamos interesados en calcular el número de formas distintas de meter o no los objetos en la mochila, pero respetando la capcaidad máxima de la mochila. Por ejemplo, si todos los `n` objetos cupieran en la mochila, existirían 2^n formas posibles. Pero en general, si no caben todos, habrán muchas menos. Resolver mediante programación dinámica el problema de calcular el número de formas distintas de completar total o parcialmente la mochila. Datos del problema: `n` objetos, `M` capacidad de la mochila, $\texttt{p} = (\texttt{p1}, \texttt{p2}, \ldots, \texttt{p}_\texttt{n})$ pesos de los objetos.

Definición del subproblema

Sea `dp[i][w]` el número de formas de llenar la mochila con un subconjunto de los primeros `i` objetos, de manera que el peso total sea `w`.

Casos base:

1) `dp[0][w]=1` para todo `w`$\geq$ `0`: hay exactamente una forma de no seleccionar ningún objeto (la mochila vacía).
2) `dp[i][0]=1` para todo `i`: hay exactamente una forma de obtener un peso de 0 (no seleccionar ningún objeto).

Relación de recurrencia:

Para cada objeto `i` con el peso `p[i]`:

- Si no seleccionados el objeto `i`, las formas posibles son `dp[i-1][w]` (todas las formas de llenar la mochila con los `i-1` objetos y el mismo peso `w`).
- Si seleccionamos el objeto `i`, las formas posibles son `dp[i-1][w-p[i]]` (todas las formas de llenar la mochila con `i-1` objetos y un peso `w-p[i]`).

Por lo tanto:

`dp[i][w]=dp[i-1][w]+dp[i-1][w-p[i]]` si `w` $\geq$ `p[i]`.

Algoritmo

1) **Inicializa la tabla** `dp` con dimensiones $(n+1) \times (M+1)$, donde n es el número de objetos y M es la capacidad de la mochila.
2) Usa la relación de recurrencia para llenar la tabla.
3) La suma de todos los valores en la última fila (`dp[n][w]` para `w=0` hasta `w=W`) será el número total de formas válidas.

Implementación en Python

```python
def count_ways_to_fill_knapsack(n, M, p):
    # Crear la tabla DP
    dp = [[0 for _ in range(M+ 1 )] for _ in range(n + 1)]

    # Inicial el caso base
    for i in range(n + 1):
        dp[i][0] = 1    # Una forma de llenar peso 0 (mochila vacía)

    # LLenar la tabla DP
    for i in range(1, n + 1):
        for w in range(M + 1):
            dp[i][w] = dp[i - 1][w] # No tomar el objeto i
            if w >= p[i - 1]: # Si cabe el objeto i
                dp[i][w] += dp[i - 1][w - p[i - 1]]

    # Sumar todas las formas válidas
    return sum(dp[n])

# Datos de ejemplo
n = 3 # Número de objetos
M = 4 # Capacidad de la mochila
p = [1, 2, 3] # Pesos de los objetos

print(f"Número de formas de llenar la mochila: {count_ways_to_fill_knapsack(n, M, p)}")
```

```
## Número de formas de llenar la mochila: 6
```

Ejemplo de ejecución

Supongamos:

- $n = 3$
- $M = 4$
- $p = [1, 2, 3]$

Paso 1: Inicialización

La tabla `dp` se inicializa con:

`dp[0][w]=1` para todo `w`.

Paso 2: Llenado de la tabla

- Para `i=1 (p[1]=1)`:

  `dp[1][w]=dp[0][w] + dp[0][w-1]` si `w` $\geq$ 1.

- Para `i=2 (p[2]=2)`:

  `dp[2][w]=dp[1][w] + dp[1][w-2]` si `w` $\geq$ 2.

- Para `i=3 (p[3]=3)`:

  `dp[3][w]=dp[2][w] + dp[2][w-3]` si `w` $\geq$ 3.

Resultado final

La suma de `dp[3][w]` para `w=0` a `w=4` da el número total de formas válida.

4) Una variante del problema de la mochila es la siguiente. Tenemos un conjunto de enteros (positivos) $A = \{a_1, a_2, \ldots, a_n\}$ y un entero K. El objetico es encontrar si existe algún subconjunto de A cuya suma sea exactamente K.

1) Desarrollar un algoritmo para resolver este problema, utilizando programación dinámica. ¿Cuál es el orden de complejidad de este algoritmo?

2) Mostrar cómo se puede obtener el conjunto de objetos resultantes (en caso de existir solución) a partir de las tablas utilizadas por el algoritmo.

3) Aplicar el algoritmo sobre el siguiente ejemplo $A = \{2, 3, 5, 2\}$, $K = 7$. ¿Cómo se puede comprobar que la solución no es única?

5) En el problema de la mochila 0/1 disponemos de dos mochilas, con capacidades `M1` y `M2`. El objetivo es maxmizar la suma de beneficios de los objetos transportados en ambas mochilas, respetando las capacidades de cada una. Resolver el problema mediante programación dinámica, definiendo la ecuación recurrente, las tablas usadas y el algoritmo para rellenarlas.
Datos del problema: `n` objetos, `M1` capacidad de la mochila 1, `M2` capacidad de la mochila 2, $\mathtt{p} = (\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n)$ pesos de los objetos, $\mathtt{b} = (\mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n)$ beneficios de los objetos.

6) Una agencia de turismo realiza planificaciones de viajes aéreos. Para ir de una ciudad `A` a `B` puede ser necesario coger varios vuelos distintos. El tiempo de un vuelo directo desde `I` hasta `J` será `T[i,j]`. Hay que tener en cuenta que si cogemos un vuelo (de `A` a `B`) y después otro (de `B` a `C`) será necesario esperar un tiempo de "escala" adicional en el aeropuesto (almacenado en `E[A, B, C]`).

a) Diseñar una solución para resolver este problema utilizando programación dinámica. Explicar cómo, a partir de las tablas, se puede obtener el conjunto de vuelos necesarios para hacer un viaje concreto.

b) Mostrar la ejecución del algoritmo sobre la siguiente matriz `T`, suponiendo que todos los `E[A, B , C]` tienen valor 1. ¿Cuál es el orden de complejidad del algoritmo?

T[i,j]	A	B	C	D
A	-	2	1	3
B	2	2	-	1
D	3	4	8	-

7) Supongamos una serie de `n` trabajos denominados `a, b, c, ...` y una tabla `B[1..n, 1..n]`, en la que cada posición `B[i,j]` almacena el beneficio de ejecutar el trabajo `i` y a continuación el trabajo `j`. Se quiere encontrar la sucesión de `m` trabajos que dé un beneficio óptimo. No hay límite en el número de veces que se puede ejecutar un trabajo concreto.

a) Idear un algoritmo por programación dinámica que resuelva el problema. Para ello, definir un subproblema (que permita realiza la combinación de problemas pequeños para resolver problemas grandes), especifica la ecuación de recurrencia para el mismo (con sus cosas base) y después describe las tablas necesarias y cómo son rellenadas.

b) Ejecutar el algoritmo sobre la siguiente tabla, suponiendo que `m=5`.

B[i,j]	a	b	c
A	2	2	5
B	4	1	3
C	3	2	2

c) Estimar el tiempo de ejecución del algoritmo. El tiempo estimado ¿es un orden exacto o una cota superior del peor caso?

8) Resolver el siguiente problema con programación dinámica. Tenemos un conjunto de n objetos, cada uno con un peso $p = (p_1, p_2, ..., p_n)$. El objetivo es repartir los objetos entre dos montones diferentes, de manera que queden lo más equilibrados posible en peso. Esto es, se debe minimizar la diferencia entre los pesos totales de ambos montones. Aplciar sobre el ejemplo con $n=4$ y $p=(2,1,3,4)$.

9) Nos vamos de comprar al mercado. Tenemos K euros en el bolsillo y una lista de m productos que podemos comprar. Cada producto tiene un precio p_i (que será simpre un número entero), y una utilidad, u_i. De cada producto podemos comprar como máximo 3 unidades. Además, tenemos una oferta según según la cual la segunda unidad nos cuesta 1 euro menos, y la tercera 2 euros menos. Queremos elegir los productos a comprar, maximizando la utilidad de los productos comprados. Resolver el problema por programación dinámica, indicando la ecuación recurrene, con sus casos base, las tablas, el algoritmo para rellenarlas y la forma de componer la solución a partir de las tablas.

10) Resolver el siguiente problema usando programación dinámica. Dada una secuencia de enteros positivos $(a_1, a_2, a_3, \ldots, a_n)$, encontrar la subsecuencia creciente más larga de elementos no necesariamente consecutivos. Es decir, encontrar una subseuencia $(a_{i1}, a_{i2}, \ldots, a_{ik})$, con $(a_{i1} < a_{i2} < \cdots < a_{ik})$ y $(1 \leq i1 < i2 < \cdots < ik \leq n)$. Por ejemplo, para la siguiente secuencia la solución sería longitud 6 (formada por los números señalados en negrita): 3, **1**, 3, **2**, **3**, 8, **4**, 7, **5**, 4, **6**.