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.

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

Este problema se puede resolver utilizando **programación dinámica** con un enfoque similar al problema de la mochila, La idea es contruir una tabla `dp[i][j]`, donde:

- `dp[i][j]` es `True` si es posible obtener una suma `j` utilizando los primeros `i` elementos del conjunto A.
- `dp[i][j]` es `False` en caso contrario.

1) Casos base
   1) `dp[0][0]=True:` Es posible obtener una suma de 0 utilizando un subconjunto vacío.
   2) `dp[i][0]=True` **para todo** `i:` También es posible obtener 0 sin incluir ningún elemento.
   3) `dp[0][j]=False` **para todo** `j` $>$ `0:` No es posible obtener una suma mayor que 0 con un subconjunto vacío.
2) Relación de recurrencia

   Para cada `i` y `j`, el estado de `dp[i][j]` depende de:
   1) No incluir el elemento actual $\texttt{a}_\texttt{i}$: Entonces `dp[i][j]=dp[i-1][j]`.
   2) Incluir el elemento actual $\texttt{a}_\texttt{i}$ (si $\texttt{j} \geq \texttt{a}_\texttt{i}$): Entonces `dp[i][j]=dp[i-1][j-`$\texttt{a}_\texttt{i}$`]`.

   Por lo tanto:

   $$\texttt{dp[i][j]=dp[i-1][j]} \ \textbf{OR} \ \texttt{dp[i-1][j-}\texttt{a}_\texttt{i}\texttt{]} \quad (\text{si } \texttt{j} \geq \texttt{a}_\texttt{i}).$$
3) Complejidad temporal: $O(n \cdot K)$, donde n es el número de elementos en A y K es la suma objetivo.

Implementación

```python
def subset_sum(A, K):
    n = len(A)
    # Crear tabla DP
    dp = [[False for _ in range(K + 1)] for _ in range(n + 1)]

    # Caso base
    for i in range(n + 1):
        dp[i][0] = True # Sumar 0 siempre es posible

    # Llenar la tabla DP
    for i in range(1, n + 1):
        for j in range(K + 1):
            dp[i][j] = dp[i-1][j] # No incluir el elemento actual
            if j >= A[i-1]: # Incluir el elemento actual si cabe
                dp[i][j] = dp[i][j] or dp[i-1][j - A[i-1]]

    # La solución está en dp[n][K]
    return dp, dp[n][K]

# Ejemplo
A = [2, 3, 5, 2]
K = 7
dp, exists = subset_sum(A, K)
print(f"¡Existe una solución? {exists}")
```

```
## ¿Existe una solución? True
```

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

Para obtener el subconjunto que produce la suma K, podemos rastrear la tabla `dp` desde `dp[n][K]` hacia atrás:

1) Comenzamos desde `dp[n][K]` y verificamos si el elemento `A[i]` está incluido:
   - Si `dp[i][K]`$\neq$`dp[i-1][K]`, significa que el elemento `A[i-1]` está incluido.
   - Restamos `A[i-1]` de K y continuamos con `dp[i-1][K - A[i-1]]`.
2) Si `dp[i][K]=dp[i-1][K]`, el elemento no está incluido.
3) Detenemos cuando $K = 0$.

Implementación

```python
def find_subset(dp, A, K):
    n = len(A)
    subset = []
    i, j = n, K

    while i > 0 and j > 0:
        # Si el elemento está incluido
        if dp[i][j] != dp[i-1][j]:
            subset.append(A[i-1])
            j -= A[i-1]
        i -= 1
    return subset

# Obtener el subconjunto
subset = find_subset(dp, A, K)
print(f"Subconjunto encontrado: {subset}")
```

```
## Subconjunto encontrado: [5, 2]
```

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

Para $A = \{2, 3, 5, 2\}$ y $K = 7$, construimos la tabla `dp`:

i/j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	T	F	F	F	F	F	F	F
1(2)	T	F	T	F	F	F	F	F
2(3)	T	F	T	T	F	T	F	F
3(5)	T	F	T	T	F	T	T	T
4(2)	T	F	T	T	T	T	T	T

- **Solución:** `dp[4][7]=True`, por lo que existe al menos un subconjunto.

Obtener subconjunto:

Siguiendo la tabla:

- `A[3]=2` está incluido ($K = 7 \to K = 5$).
- `A[2]=5` está incluido ($K = 5 \to K = 0$).

Subconjunto: $\{5, 2\}$.

Comprobación de unicidad:

Para verificar si hay múltiples soluciones, buscamos otras formas de descomponer $K = 7$:

1) Si `dp[i][K - A[i-1]]=True` para diferentes `i`, hay más soluciones.
2) En este caso, otra solución es $\{3, 2, 2\}$.

Conclusión: 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, $\texttt{p} = (\texttt{p}_1, \texttt{p}_2, \ldots, \texttt{p}_\texttt{n})$ pesos de los objetos, $\texttt{b} = (\texttt{b}_1, \texttt{b}_2, \ldots, \texttt{b}_\texttt{n})$ beneficios de los objetos.

Definición del subproblema

Sea $\texttt{dp[i]}[\texttt{c}_1][\texttt{c}_2]$ es el beneficio máximo que se puede obtener utilizando los primeros `i` objetos, con capacidad disponible $\texttt{c}_1$ en la mochila 1 y $\texttt{c}_2$ en la mochila 2.

Casos base:

1) $\texttt{dp}[0][\texttt{c}_1][\texttt{c}_2] = 0$: Si no consideramos ningún objeto ($\texttt{i} = 0$), el beneficio es 0, sin importar las capacidades de $\texttt{c}_1$ y $\texttt{c}_2$.
2) $\texttt{cp}[\texttt{i}][0][\texttt{c}_2] = \texttt{dp}[\texttt{i}][\texttt{c}_1][0] = 0$: Si alguna mochila tiene capacidad 0, no se puede incluir objetos en ella.

Relación de recurrencia:

Para cada objeto `i` con peso `p[i]` y beneficio `b[i]`, tenemos tres opciones:

1) No incluir el objeto `i` en ninguna mochila:

$$\texttt{dp}[\texttt{i}][\texttt{c}_1][\texttt{c}_2] = \texttt{dp}[\texttt{i} - 1][\texttt{c}_1][\texttt{c}_2].$$

2) Incluir el objeto `i` en la mochila 1 (si cabe):

$$\texttt{dp}[\texttt{i}][\texttt{c}_1][\texttt{c}_2] = \max(\texttt{dp}[\texttt{i}][\texttt{c}_1][\texttt{c}_2], \texttt{dp}[\texttt{i} - 1][\texttt{c}_1 - \texttt{p}[\texttt{i}]][\texttt{c}_2] + \texttt{b}[\texttt{i}]) \quad \text{si } \texttt{c}_1 \geq \texttt{p}[\texttt{i}].$$

3) Incluir el objeto `i` en la mochila 2 (si cabe):

$$\texttt{dp}[\texttt{i}][\texttt{c}_1][\texttt{c}_2] = \max(\texttt{dp}[\texttt{i}][\texttt{c}_1][\texttt{c}_2], \texttt{dp}[\texttt{i} - 1][\texttt{c}_1][\texttt{c}_2 - \texttt{p}[\texttt{i}]] + \texttt{b}[\texttt{i}]) \quad \text{si } \texttt{c}_2 \geq \texttt{p}[\texttt{i}].$$

En resumen:

$$\texttt{dp}[\texttt{i}][\texttt{c}_1][\texttt{c}_2] = \max(\texttt{dp}[\texttt{i} - 1][\texttt{c}_1][\texttt{c}_2], \texttt{dp}[\texttt{i} - 1][\texttt{c}_1 - \texttt{p}[\texttt{i}]][\texttt{c}_2] + \texttt{b}[\texttt{i}], \texttt{dp}[\texttt{i} - 1][\texttt{c}_1][\texttt{c}_2 - \texttt{p}[\texttt{i}]] + \texttt{b}[\texttt{i}]).$$

Algoritmo

1) Inicialización:
   - Crear una tabla $\texttt{dp}[\texttt{i}][\texttt{c}_1][\texttt{c}_2]$ de dimensiones $(n + 1) \times (M_1 + 1) \times (M_2 + 1)$.

- Inicializar $\texttt{dp}[0][\texttt{c}_1][\texttt{c}_2] = 0$ para todas las combinaciones de $\texttt{c}_1$ y $\texttt{c}_2$.

2) Llenado de la tabla:

- Iterrar sobre los objetos ($\texttt{i}$ de 1 a n).
- Para cada combinación de capacidades $\texttt{c}_1$ y $\texttt{c}_2$, calcular el valor de $\texttt{dp}[\texttt{i}][\texttt{c}_1][\texttt{c}_2]$ usando la relación de recurrencia.

3) Resultado:

- El beneficio máximo está en $\texttt{dp}[\texttt{n}][\texttt{M}_1][\texttt{M}_2]$.

4) Reconstrucción de la solución:

- Para determinar qué objetos se seleccionaron y en qué mochila, rastrear las decisiones tomadas al llenar la tabla $\texttt{dp}$.

Implementación en Python

```python
def two_knapsacks(n, M1, M2, p, b):
    # Crear tabla DP
    dp = [[[0 for _ in range(M2 + 1)] for _ in range(M1 + 1)] for _ in range(n + 1)]

    # LLenar la tabla DP
    for i in range(1, n + 1):
        for c1 in range(M1 + 1):
            for c2 in range(M2 + 1):
                # No incluir el objeto
                dp[i][c1][c2] = dp[i-1][c1][c2]
                # Incluir en la mochila 1 si cabe
                if c1 >= p[i-1]:
                    dp[i][c1][c2] = max(dp[i][c1][c2], dp[i-1][c1 - p[i-1]][c2] + b[i-1])
                # Incluir en la mochila 2 si cabe
                if c2 >= p[i-1]:
                    dp[i][c1][c2] = max(dp[i][c1][c2], dp[i-1][c1][c2 - p[i-1]] + b[i-1])

    return dp, dp[n][M1][M2]

# Reconstrucción de la solución
def find_items(dp, n, M1, M2, p, b):
    items_in_knapsack1 = []
    items_in_knapsack2 = []
    c1, c2 = M1, M2

    for i in range(n, 0, -1):
        if dp[i][c1][c2] != dp[i-1][c1][c2]:
            if c1 >= p[i-1] and dp[i][c1][c2] == dp[i-1][c1 - p[i-1]][c2] + b[i-1]:
                items_in_knapsack1.append(i-1)
                c1 -= p[i-1]
            elif c2 >= p[i-1] and dp[i][c1][c2] == dp[i-1][c1][c2 - p[i-1]] + b[i-1]:
                items_in_knapsack2.append(i-1)
                c2 -= p[i-1]

    return items_in_knapsack1, items_in_knapsack2

# Ejemplo
n = 4
```

```
M1 = 5
M2 = 6
p = [2, 3, 4, 1]
b = [3, 4, 5, 6]

dp, max_benefit = two_knapsacks(n, M1, M2, p, b)
items_in_knapsack1, items_in_knapsack2 = find_items(dp, n, M1, M2, p, b)

print(f"Beneficio máximo: {max_benefit}")
print(f"Objetos en la mochila 1: {items_in_knapsack1}")
print(f"Objetos en la mochila 2: {items_in_knapsack2}")
```

```
## Beneficio máximo: 18
## Objetos en la mochila 1: [3, 2]
## Objetos en la mochila 2: [1, 0]
```

Complejidad temporal: $O(n \cdot M_1 \cdot M_2)$, ya que iteramos sobre n, M_1 y M_2.

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.

El problema se puede resolver utilizando una variante del algoritmo de **Floyd-Warshall** para calcular el tiempo mínimo necesario para viajar entre dos ciudades teniendo en cuenta tiempos de escala adicionales.

1) Representación del problema

- Supongamos que tenemos `n` ciudades, y una matriz `T[i,j]` que almacena el tiempo de vuelo directo entre las ciudades `i` y `j`. Si no hay vuelo directo entre dos ciudades, representaremos esto como infinito (∞).
- Además, la matriz `E[i,j,k]` indica el tiempo de escala adicional necesario si tomamos el vuelo de `i` a `j` y luego de `j` a`k`.

2) Idea general

1) Usa programación dinámica para calcular la tabla `dp[k][i][j]`, que representa el tiempo mínimo necesario para viajar de `i` a `j` utilizando como máximo `k` ciudades intermedias.

2) Define:

- Si no se usa ninguna ciudad intermedia `(k=0)`, el tiempo es simplemente tiempo de vuelo directo `T[i][j]`.
- Para `k` $>$ `0`, el tiempo mínimo se calcular como:

$$\texttt{dp[k][i][j]} = \min(\texttt{dp[k-1][i][j]}, \texttt{dp[k-1][i][m]} + \texttt{E[i,m,j]} + \texttt{dp[k-1][m][j]})$$

donde `m` es una ciudad intermedia.

3) El tiempo óptimo para ir de `A` a `B` se encuentra en `dp[n][A][B]`, donde `n` es el número total de ciudades.

3) Algoritmo paso a paso

1) Inicializa `dp[0][i][j]` con los valores de `T[i][j]`. Si no hay vuelo directo, establece `dp[0][i][j]` $= \infty$.

2) Para cada `k`, calcula `dp[k][i][j]` utilizando la fórmula recursiva.

3) Almacena en una tabla auxiliar `next[i][j]` el vuelo intermedio que minimiza el tiempo. Esto permite reconstruir el conjunto de vuelos necesarios.

4) Obtención del conjunto de vuelos

1) A partir de la tabla `next`, se puede reconstruir el conjunto de vuelos necesaros de la siguiente manera:

- Si `next[A][B]=C`, significa que la ruta óptima de `A` a `B` pasa por `C`.
- Recursivamente, descompín `A→B` en `A→C` y `C→B`.

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	-

Dado:

$$\mathtt{T} = \begin{bmatrix} - & 2 & 1 & 3 \\ 2 & - & - & 1 \\ 3 & 4 & 8 & - \end{bmatrix}, \quad \mathtt{E[i,j,k]} = 1.$$

Paso 1: Inicialización

- La matriz inicial (`dp[0][i][j]`) es:

$$\mathtt{dp[0]} = \begin{bmatrix} \infty & 2 & 1 & 3 \\ 2 & \infty & \infty & 1 \\ 3 & 4 & 8 & \infty \end{bmatrix}.$$

Paso 2: Iteración para `k=1`

- Para cada par de ciudades `i` y `j`, actualizamos:

$$\mathtt{dp[1][i][j]} = \min(\mathtt{dp[0][i][j]}, \mathtt{dp[0][i][m]} + \mathtt{E[i,m,j]} + \mathtt{dp[0][m][j]}),$$

donde `m` es la ciudad intermedia.

Por ejemplo:

1) Para `A` $\rightarrow$ `D`:

$$\mathtt{dp[1][A][D]} = \min(\mathtt{dp[0][A][D]}, \mathtt{dp[0][A][B]} + \mathtt{E[A,B,D]} + \mathtt{dp[0][B][D]}).$$

Sustituyendo:

$$\mathtt{dp[1][A][D]} = \min(3, 2 + 1 + 1) = \min(3, 4) = 3.$$

- Repite este cálculo para todos los pares `(i,j)`.

c) Resultado final

La matriz `dp[n]` después de iterar por todas las ciudades intermedias será:

$$\mathtt{dp} = \begin{bmatrix} \infty & 2 & 1 & 3 \\ 2 & \infty & 3 & 1 \\ 3 & 4 & 8 & 4 \end{bmatrix}.$$

Implementación en Python

```python
import numpy as np


def flight_schedule(T, E):
    """
    Resuelve el problema de planificación de vuelos con programación dinámica.

    :param T: Matriz de tiempos directos entre ciudades (np.array).
    :param E: Tiempo de escala adicional entre ciudades (np.array).
    :return: Matriz dp con tiempos mínimos y matriz next para reconstruir las rutas.
    """
    n = len(T)
    # Inicializar las matrices de DP y next
    dp = np.full((n, n), float('inf'))  # dp[i][j] almacena el tiempo mínimo de i a j
    next_city = [[-1 for _ in range(n)] for _ in range(n)]  # Para reconstruir rutas

    # Condiciones iniciales: tiempos directos de T
    for i in range(n):
        for j in range(n):
            if i != j:  # No hay bucles en sí mismos
                dp[i][j] = T[i][j] if T[i][j] != -1 else float('inf')
                if T[i][j] != -1:  # Si hay un vuelo directo
                    next_city[i][j] = j  # La siguiente ciudad en la ruta es directamente
                        j

    # Aplicar la fórmula de DP considerando ciudades intermedias
    for k in range(n):  # Consideramos cada ciudad como intermedia
        for i in range(n):
            for j in range(n):
                # Actualizar el tiempo mínimo si pasar por k mejora la ruta
                if dp[i][k] + E[i][k][j] + dp[k][j] < dp[i][j]:
                    dp[i][j] = dp[i][k] + E[i][k][j] + dp[k][j]
                    next_city[i][j] = next_city[i][k]  # Actualizamos la ciudad siguiente

    return dp, next_city


def reconstruct_path(next_city, start, end):
    """
    Reconstruye la ruta óptima de start a end usando la matriz next_city.

    :param next_city: Matriz para reconstruir rutas (lista de listas).
    :param start: Ciudad de inicio.
    :param end: Ciudad de destino.
    :return: Lista con la secuencia de ciudades de la ruta óptima.
```

```
    """
    n = len(next_city)  # Número de ciudades
    if start < 0 or start >= n or end < 0 or end >= n:  # Validación de índices
        raise ValueError(f"Indices fuera de rango: start={start}, end={end}")

    if next_city[start][end] == -1:  # Si no hay ruta disponible
        return []
    path = [start]  # Inicializamos la ruta con la ciudad de inicio
    while start != end:  # Mientras no lleguemos al destino
        start = next_city[start][end]  # Avanzamos a la siguiente ciudad
        if start == -1:  # Verificación adicional para evitar errores
            return []
        path.append(start)  # Añadimos la ciudad actual al camino
    return path


# Ejemplo de uso:
T = np.array([
    [-1, 2, 1, 3],  # Tiempos de vuelo desde A
    [2, -1, -1, 1],  # Tiempos de vuelo desde B
    [3, 4, 8, -1]  # Tiempos de vuelo desde D
])

# Matriz de tiempos de escala (suponemos escala de 1 para todas las combinaciones)
n = len(T)
E = np.full((n, n, n), 1)  # Escalas constantes de 1 para simplificar

# Resolver el problema
dp, next_city = flight_schedule(T, E)

# Mostrar resultados
print("Matriz de tiempos mínimos (dp):")
print(dp)
print("\nMatriz next_city:")
print(next_city)

# Reconstruir ruta óptima de A (0) a D (2)
start, end = 0, 2 # Corregido el índice de destino para evitar errores
route = reconstruct_path(next_city, start, end)
print(f"\nRuta óptima de {start} a {end}: {route}")
```

```
## Matriz de tiempos mínimos (dp):
## [[5. 2. 1.]
## [2. 5. 4.]
## [3. 4. 5.]]
##
## Matriz next_city:
## [[1, 1, 2], [0, 0, 0], [0, 1, 0]
##
## Ruta óptima de 0 a 2: [0, 2]]]]]
```

Complejidad temporal: $O(n^3)$, esto se debe a que para cada par de ciudades `(i,j)`, iteramos sobre todas las ciudades intermedias `m`.

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.

1) Definición del subproblema:

   Sea `dp[k][j]` el beneficio máximo que se pued eobtener ejeutando `k` trabajos, donde el último trabajo es `j`. El objetivo es calcular:

   $$\max_{j} \texttt{dp}[\texttt{m}][\texttt{j}]$$

   donde `m` es el número total de trabajos en la secuencia.

2) Ecuación de recurrencia:

   Para calcular `dp[k][j]`, consideramos que el último trabajo de la secuencia es `j` y el penúltimo trabajo es `i`. La relación es:

   $$\texttt{dp}[\texttt{k}][\texttt{j}] = \max_{\texttt{i}}(\texttt{dp}[\texttt{k}-1][\texttt{i}] + \texttt{B}[\texttt{i}][\texttt{j}])$$

   donde:

   - `dp[k-1][i]` es el beneficio para `k-1` trabajos terminando en `i`.
   - `B[i][j]` es el beneficio de ejecutar el trabajo `i` seguido del trabajo `j`.

3) Condiciones base:

   $$\texttt{dp}[1][\texttt{j}] = 0 \quad \forall \texttt{j}$$

   No hay beneficio inicial para `k=1` porque no hay trabajo anterior.

4) Tablas necesarias:

   - Una tabla `dp[k][j]` de dimensiones $m \times n$, donde `k` es el número de trabajos y `j` es el índice del último trabajo.
   - Una tabla auxiliar `prev[k][j]` para reconstruir la secuencia óptima.

5) Proceso de llenado:

   1) Inicializar `dp[1][j]=0`.
   2) Para cada `k` de 2 a `m`, y para cada `j`, calcular:

      $$\texttt{dp}[\texttt{k}][\texttt{j}] = \max_{\texttt{i}}(\texttt{dp}[\texttt{k}-1][\texttt{i}] + \texttt{B}[\texttt{i}][\texttt{j}])$$

      y almacenar en `prev[k][j]` el índice `i` que maximiza el beneficio.

6) Reconstrucción de la solución:

   Comenzando desde el índice `j` que maximizar `dp[m][j]`, usar `prev[k][j]` para retroceder y encontrar la secuencia de trabajos óptima.

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

Tabla B:

$$B = \begin{bmatrix} 2 & 2 & 5 \\ 4 & 1 & 3 \\ 3 & 2 & 2 \end{bmatrix}$$

Parámetros:

- n=3 (número de trabajos).
- m=5 (longitud de la secuencia).

Algoritmo de Python:

```
import numpy as np

def optimal_job_sequence(B, m):
    n = len(B)
    # Inicializar tablas dp y prev
    dp = np.zeros((m + 1, n))
    prev = np.full((m + 1, n), -1)

    # Llenar la tabla dp
    for k in range(2, m + 1): # Desde la segunda iteración
        for j in range(n): # Último trabajo en la secuencia
            max_benefit = -float('inf')
            best_prev = -1
            for i in range(n): # Penúltimo trabajo en la secuencia
                benefit = dp[k-1][i] + B[i][j]
                if benefit > max_benefit:
                    max_benefit = benefit
                    best_prev = i
            dp[k][j] = max_benefit
            prev[k][j] = best_prev

    # Encontrar el máximo beneficio de dp[m]
    max_benefit = -float('inf')
    last_job = -1
    for j in range(n):
        if dp[m][j] > max_benefit:
            max_benefit = dp[m][j]
            last_job = j

    # Reconstruir la secuencia óptima
    sequence = []
    k = m
    while k > 0 and last_job != -1:
        sequence.append(last_job)
        last_job = prev[k][last_job]
```

```
            k -= 1
        sequence.reverse()

        return max_benefit, sequence

# Matriz de beneficios
B = np.array([
    [2, 2, 5], # Beneficios desde a
    [4, 1, 3], # Beneficios desde b
    [3, 2, 2]  # Beneficios desde c
])

# Número de tabajos en la secuencia
m = 5

# Ejecutar el algoritmo
max_benefit, sequence = optimal_job_sequence(B, m)
print(f"Beneficio máximo: {max_benefit}")
print(f"Secuencia óptima: {sequence}")

## Beneficio máximo: 17.0
## Secuencia óptima: [1, 0, 2, 0, 2]
```

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

1) Tiempo de ejecución:

- La fórmula `dp[k][j]` requiere calcular $\max_{i}(\texttt{dp}[k-1][i] + \texttt{B}[i][j])$, lo cual tiene un complejidad de $O(n)$.
- LLenar toda la tabla `dp` implica $O(m \cdot n^2)$

2) Cota:

- El tiempo estimado es una **cota superior del peor caso** porque considera todas las combinaciones posibles de trabajos.

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 compras 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**.