






Programación dinámica

Algoritmos y estructuras de datos II

José Antonio Hernández López¹

¹Departamento de Informática y Sistemas
Universidad de Murcia

24 de marzo de 2025

- 1 Programación dinámica: ¿qué, cómo, cuándo?
- 2 Recorrido turístico por Manhattan 
- 3 Bellman-Ford en un DAG 
- 4 Mochila 0/1 
- 5 Cambio de monedas 
- 6 Conclusiones 

Definición

Es una **poderosa** técnica de diseño de algoritmos que resuelven problemas de optimización. Hay dos formas equivalentes de definirla:

- Recursión + memorización (PD con memorización)
- Subproblemas pequeños + combinación (PD ascendente)



Término n -ésimo de la sucesión Fibonacci

La sucesión de Fibonacci tiene la siguiente ecuación de recurrencia:

$$f_n = \begin{cases} 1 & n \leq 2 \\ f_{n-1} + f_{n-2} & n > 2 \end{cases}$$

Queremos devolver el término n -ésimo.

Término n -ésimo de la sucesión Fibonacci

La sucesión de Fibonacci tiene la siguiente ecuación de recurrencia:

$$f_n = \begin{cases} 1 & n \leq 2 \\ f_{n-1} + f_{n-2} & n > 2 \end{cases}$$

Queremos devolver el término n -ésimo.

El algoritmo recursivo es:

```
1 def fib(n):
2     if n <= 2:
3         return 1
4     else:
5         return fib(n-1) + fib(n-2)
```

Término n -ésimo de la sucesión Fibonacci

La sucesión de Fibonacci tiene la siguiente ecuación de recurrencia:

$$f_n = \begin{cases} 1 & n \leq 2 \\ f_{n-1} + f_{n-2} & n > 2 \end{cases}$$

Queremos devolver el término n -ésimo.

El algoritmo recursivo es:

```
1 def fib(n):
2     if n <= 2:
3         return 1
4     else:
5         return fib(n-1) + fib(n-2)
```

Este algoritmo es $t(n) = t(n-1) + t(n-2) + C \in \Theta(\Phi^n)$. Es decir es un algoritmo pésimo.

Programación dinámica con memorización

Vamos a añadir una caché memo (usando diccionario o array):

Vamos a añadir una caché memo (usando diccionario o array):

```
1 def fib_memo(n, memo):
2     if n in memo: # asumimos que esto es constante
3         return memo[n]
4     if n <= 2:
5         f = 1
6     else:
7         f = fib_memo(n-1, memo) + fib_memo(n-2, memo)
8     memo[n] = f # asumimos que esto es constante
9     return f
```

¿Qué orden tiene este algoritmo?

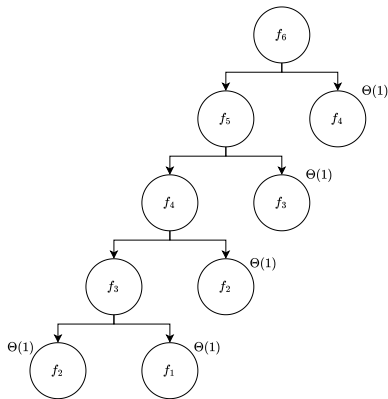
Vamos a añadir una caché memo (usando diccionario o array):

```
1 def fib_memo(n, memo):
2     if n in memo: # asumimos que esto es constante
3         return memo[n]
4     if n <= 2:
5         f = 1
6     else:
7         f = fib_memo(n-1, memo) + fib_memo(n-2, memo)
8     memo[n] = f # asumimos que esto es constante
9     return f
```

¿Qué orden tiene este algoritmo?

El orden es $\Theta(n)$.

Programación dinámica con memorización



En general, $t(n) = t(n-1) + C \in \Theta(n)$. Esto es porque el resultado de $\text{fib}(n-2, \text{memo})$ ya se ha ejecutado y guardado en algún punto de $\text{fib}(n-1, \text{memo})$ y pasa a ser constante.

Programación dinámica con memorización

- `fib_memo(k, memo)` solo ejecuta llamadas recursivas la primera vez que es llamado para todo k (que no sean casos base).
- Las llamadas *memorizadas* (las que no recurren) son gratis, es decir, son $\Theta(1)$.
- El trabajo no recursivo en cada llamada es constante $\Theta(1)$ (condicionales y una suma).
- Así pues, tenemos $n - 2$ llamadas que hacen $\Theta(1)$ y dos llamadas a los casos base que hacen $\Theta(1)$.
- En general tenemos n llamadas/subproblemas que se resuelven en $\Theta(1)$. Por tanto, `fib_memo(n, memo)` es $\Theta(n)$.

Programación dinámica con memorización

La base de la programación dinámica con memorización es:

Recordar y reutilizar subproblemas

Una vez resuelto un subproblema, apuntamos la solución (en nuestra caché memo) y la reutilizamos después.

Programación dinámica con memorización

La base de la programación dinámica con memorización es:

Recordar y reutilizar subproblemas

Una vez resuelto un subproblema, apuntamos la solución (en nuestra caché memo) y la reutilizamos después.

Además el tiempo del algoritmo viene dado por:

$\# \text{subproblemas diferentes} \times \text{tiempo trabajo no recursivo/subproblema}$

Programación dinámica con memorización

La base de la programación dinámica con memorización es:

Recordar y reutilizar subproblemas

Una vez resuelto un subproblema, apuntamos la solución (en nuestra caché `memo`) y la reutilizamos después.

Además el tiempo del algoritmo viene dado por:

$\# \text{subproblemas diferentes} \times \text{tiempo trabajo no recursivo/subproblema}$

En `fibo_memo(n, memo)` tenemos n subproblemas y en cada subproblema gastamos $\Theta(1)$ en trabajo no recursivo (es una suma). Así pues $t(n) \in \Theta(n)$.

Programación dinámica con memorización

La base de la programación dinámica con memorización es:

Recordar y reutilizar subproblemas

Una vez resuelto un subproblema, apuntamos la solución (en nuestra caché memo) y la reutilizamos después.

Además el tiempo del algoritmo viene dado por:

$\# \text{subproblemas diferentes} \times \text{tiempo trabajo no recursivo/subproblema}$

En `fibo_memo(n, memo)` tenemos n subproblemas y en cada subproblema gastamos $\Theta(1)$ en trabajo no recursivo (es una suma). Así pues $t(n) \in \Theta(n)$.

¡ESTO LO PODEMOS HACER CON CUALQUIER RECURSIÓN!

y podemos llegar a pasar de un algoritmo exponencial a uno polinomial...

Programación dinámica ascendente

Si deshacemos la recursión de `fib_memo(n, memo)` y lo plasmamos en un programa iterativo, tenemos lo siguiente (en vez de un diccionario, usamos un array como *caché*):

```
1 def fib_ascendente(n):
2     arr = [0] * (n + 1)
3     for k in range(1, n + 1):
4         if k <= 2:
5             f = 1
6         else:
7             f = arr[k-1] + arr[k-2]
8         arr[k] = f
9     return arr[n]
```


Programación dinámica ascendente

Si deshacemos la recursión de `fib_memo(n, memo)` y lo plasmamos en un programa iterativo, tenemos lo siguiente (en vez de un diccionario, usamos un array como *caché*):

```
1 def fib_ascendente(n):
2     arr = [0] * (n + 1)
3     for k in range(1, n + 1):
4         if k <= 2:
5             f = 1
6         else:
7             f = arr[k-1] + arr[k-2]
8         arr[k] = f
9     return arr[n]
```

Es decir, estamos resolviendo primero los subproblemas pequeños y los estamos guardando en un array. Luego esos resultados se reutilizan para subproblemas más grandes. Al final, devolvemos `arr[n]` que es el elemento que nos interesa.

La base de la PD ascendente es:

Resolución de subproblemas pequeños

Resolvemos los subproblemas pequeños, los apuntamos en un array y vamos resolviendo subproblemas más grandes combinando los pequeños.

- Este método es equivalente a la PD con memorización.
- En la práctica, este método es más eficiente porque no hay recursiones.
- Este método suele usarse con una tabla como *caché*.
- El tiempo de ejecución es lo que se tarda en rellenar la tabla.
- **Este método es el que se va a ser priorizado en la asignatura.**

Teniendo una ecuación de recurrencia, los pasos de la PD ascendente son los siguientes:

- 1 Estudiar las dimensiones de la tabla.
- 2 Estudiar el DAG de dependencia los subproblemas.
- 3 Seleccionar un orden topológico con sentido para resolver los subproblemas.
- 4 Rellenar la tabla siguiendo ese orden topológico y devolver el problema más grande (normalmente en el que estamos interesados en resolver)

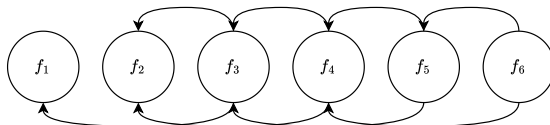
Estudiar las dimensiones de la tabla

$$f_n = \begin{cases} 1 & n \leq 2 \\ f_{n-1} + f_{n-2} & n > 2 \end{cases}$$

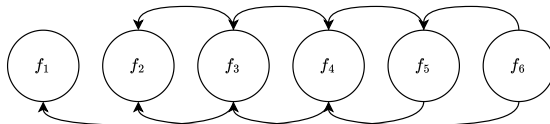
La ecuación de recurrencia depende de un parámetro. Así pues, la tabla para estos problemas suele ser unidimensional. Como hay n subproblemas diferentes, la tabla tendrá dimensión n y cada i contendrá la solución i -ésima (término de fibonacci i -ésimo).

Estudiar el DAG de los subproblemas

Estudiar el DAG de los subproblemas

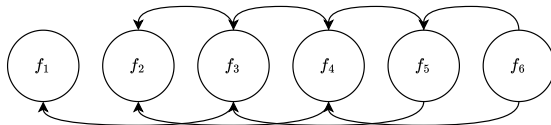


Estudiar el DAG de los subproblemas



Seleccionar un orden topológico con sentido

Estudiar el DAG de los subproblemas



Seleccionar un orden topológico con sentido

Resolver los subproblemas en este orden: $f_1, f_2, f_3, f_4, \dots, f_n$.

Rellenar la tabla siguiendo el orden seleccionado

```
1 def fib_ascendente(n):
2     arr = [0] * (n + 1)
3     for k in range(1, n + 1):
4         # casos base
5         if k <= 2:
6             f = 1
7         else:
8             # esto puedo hacerlo porque he escogido un orden válido
9             # y los dos subproblemas están resueltos
10            f = arr[k-1] + arr[k-2]
11            # guardamos el resultado
12            arr[k] = f
13    # devolvemos el resultado al problema grande
14    return arr[n]
```

El orden de este algoritmo es $\Theta(n)$ ya que rellenamos una tabla de orden tamaño n .

¿Cómo aplicar la programación dinámica?

Dado un problema, hay que aplicar los siguientes pasos:

- 1 Sacar la ecuación de recurrencia (esto es lo más difícil).
- 2 Escoger la técnica de PD a usar y aplicarla. Si la técnica escogida es PD con memorización, entonces es directo. Si la técnica escogida es PD ascendente:
 - 1 Estudiar las dimensiones de la tabla.
 - 2 Estudiar el DAG de los subproblemas.
 - 3 Seleccionar un orden topológico con sentido.
 - 4 Rellenar la tabla siguiendo ese orden topológico.

Finalmente, una vez diseñado el algoritmo, lo ejecutamos para el problema más grande.

- 3 Reconstruir la solución a partir de los valores de la tabla o del diccionario (dependiendo del método escogido). Esto es porque la solución es usualmente el valor de máx o mín no arg máx o arg mín .

¿Cuándo funciona la programación dinámica?

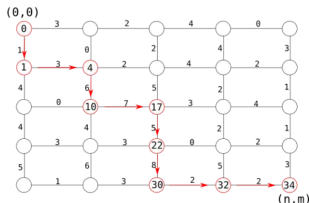
Un problema es candidato para ser resuelto con PD si cumple las siguientes propiedades:

- Subestructura óptima: La solución óptima del problema grande se puede obtener mediante las soluciones óptimas del problema pequeño. Si el problema cumple esta propiedad, el algoritmo devolverá la solución óptima.
- Intersección de subproblemas: Diferentes subproblemas van a ser reutilizados en varios problemas más grandes.
- Subproblemas polinomiales: El número de problemas ha de ser de orden polinomial.

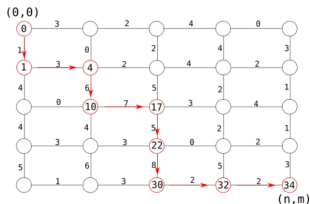


Descripción del problema

- Tenemos un grafo $G = (V, E)$ en forma de *grid*. Cada arista tiene un peso asignado.
- Estamos en una posición inicial $(0, 0)$ y queremos llegar a la posición (n, n) .
- Solo podemos movernos hacia abajo y hacia la derecha siempre que se pueda.
- **Objetivo:** Encontrar el camino más largo.



Recorrido turístico por Manhattan



También vamos a asumir lo siguiente:

- Tablero cuadrado de tamaño n . Hay $n \times n$ nodos.
- Se numera del 0 hasta el $n - 1$.

El **primer paso** es encontrar la ecuación de recurrencia y esto es lo *complicado*. En PD, lo que devuelve la ecuación de recurrencia es lo que queremos maximizar o minimizar.

El **primer paso** es encontrar la ecuación de recurrencia y esto es lo *complicado*. En PD, lo que devuelve la ecuación de recurrencia es lo que queremos maximizar o minimizar. Definimos $S(i, j)$ como el coste máximo de llegar a la casilla i, j desde $0, 0$. Para definir la ecuación de recurrencia usad la siguiente *idea: si no sabes la respuesta, adivina probando todas las combinaciones...*

El **primer paso** es encontrar la ecuación de recurrencia y esto es lo *complicado*. En PD, lo que devuelve la ecuación de recurrencia es lo que queremos maximizar o minimizar. Definimos $S(i, j)$ como el coste máximo de llegar a la casilla i, j desde $0, 0$. Para definir la ecuación de recurrencia usad la siguiente *idea: si no sabes la respuesta, adivina probando todas las combinaciones...*

$$S(i, j) = \begin{cases} 0 & i = j = 0 \\ S(i - 1, j) + e_{\text{arriba}} & j = 0 \\ S(i, j - 1) + e_{\text{izquierda}} & i = 0 \\ \max(S(i - 1, j) + e_{\text{arriba}}, S(i, j - 1) + e_{\text{izquierda}}) & \text{otro caso} \end{cases}$$

$$e_{\text{arriba}} = \text{down}[i - 1][j]$$

$$e_{\text{izquierda}} = \text{right}[i][j - 1]$$

El segundo paso es escoger el método PD y aplicarlo. Supongamos que escogemos PD con memorización.

Primero, escribimos la recursión ingénua:

El segundo paso es escoger el método PD y aplicarlo. Supongamos que escogemos PD con memorización.

Primero, escribimos la recursión ingénua:

```
1 def manhattan_recursivo(i, j):
2     if i == j == 0:
3         m = 0
4     else:
5         if i > 0:
6             e_arriba = down[i - 1][j]
7             m1 = manhattan_recursivo(i - 1, j) + e_arriba
8         else:
9             m1 = float("-inf")
10
11         if j > 0:
12             e_izquierda = right[i][j - 1]
13             m2 = manhattan_recursivo(i, j - 1) + e_izquierda
14         else:
15             m2 = float("-inf")
16
17         m = max(m1, m2)
18     return m
```

La complejidad de esto es

El segundo paso es escoger el método PD y aplicarlo. Supongamos que escogemos PD con memorización.

Primero, escribimos la recursión ingénua:

```
1 def manhattan_recursivo(i, j):
2     if i == j == 0:
3         m = 0
4     else:
5         if i > 0:
6             e_arriba = down[i - 1][j]
7             m1 = manhattan_recursivo(i - 1, j) + e_arriba
8         else:
9             m1 = float("-inf")
10
11         if j > 0:
12             e_izquierda = right[i][j - 1]
13             m2 = manhattan_recursivo(i, j - 1) + e_izquierda
14         else:
15             m2 = float("-inf")
16
17         m = max(m1, m2)
18     return m
```

La complejidad de esto es $\approx \Theta(2^{i+j})$... Como tenemos $i = j = n$ (resolvemos el subproblema grande $S(n-1, n-1)$ de tamaño n asumiendo $n \times n$) tenemos $\approx \Theta(4^n)$...

Si añadimos memorización quedaría así:

```
1 def manhattan_memo(i, j, memo):
2     if (i, j) in memo:
3         return memo[(i,j)]
4     if i == j == 0:
5         m = 0
6     else:
7
8         if i > 0:
9             e_arriba = down[i - 1][j]
10            m1 = manhattan_memo(i - 1, j, memo) + e_arriba
11        else:
12            m1 = float("-inf")
13
14        if j > 0:
15            e_izquierda = right[i][j - 1]
16            m2 = manhattan_memo(i, j - 1, memo) + e_izquierda
17        else:
18            m2 = float("-inf")
19
20        m = max(m1, m2)
21        memo[(i,j)] = m
22    return m
```

¿Orden?

Si aplicamos la fórmula

$$\# \text{subproblemas diferentes} \times \text{tiempo/subproblema}$$

tenemos que

$$\# \text{subproblemas diferentes} = n^2$$

$$\text{tiempo/subproblema} = \Theta(1)$$

por tanto el orden es $\Theta(n^2)$.

Si ejecutamos `manhattan_memo(n-1, n-1, memo)` obtenemos la distancia máxima recorrida pero no tenemos el camino... ¿cómo reconstruimos el camino?

Solución: usando el diccionario `memo` y viendo, en cada paso, cual es el `arg máx.`

Tras la ejecución del método, `memo` tiene la siguiente forma:

```
1 {(0, 0): 0,  
2  (0, 1): 3,  
3  ...  
4  (4, 2): 30,  
5  (4, 3): 32,  
6  (4, 4): 34}
```

`memo` representa el siguiente tablero:

```
1 [[0, 3, 5, 9, 9],  
2  [1, 4, 7, 13, 15],  
3  [5, 10, 17, 20, 24],  
4  [9, 14, 22, 22, 25],  
5  [14, 20, 30, 32, 34]]
```

- 1 Estamos en la posición $(4, 4)$ y sabemos que el camino máximo tiene beneficio 34. Tenemos que ver de donde viene ese 34.
- 2 A la casilla $(4, 4)$ solo se ha podido acceder desde arriba $(3, 4)$ o desde la izquierda $(4, 3)$.
- 3 $34 = \max(\text{izq} = 32 + 2, \text{arriba} = 25 + 3)$, así pues el `arg máx` es `izq`. De modo que el camino óptimo viene de $(4, 3)$.
- 4 Pasamos a $(4, 3)$ y repetimos lo mismo hasta llegar al $(0, 0)$.



```
1 def reconstruir_memo(memo, n, m):
2     pos_actual = (n, m)
3     S = []
4     while pos_actual != (0,0):
5         S.append(pos_actual)
6         i, j = pos_actual
7         valor = memo[(i,j)]
8
9         if i > 0:
10             e_arriba = down[i - 1][j]
11             m1 = memo[(i-1,j)] + e_arriba
12         else:
13             m1 = float("-inf")
14
15         if j > 0:
16             e_izquierda = right[i][j - 1]
17             m2 = memo[(i,j-1)] + e_izquierda
18         else:
19             m2 = float("-inf")
20
21         if valor == m1:
22             pos_actual = (i- 1, j)
23         else:
24             pos_actual = (i , j - 1)
25     return S
```

Ahora pasamos al otro método de DP, el ascendente que consta de los siguientes pasos:

- 1 Estudiar las dimensiones de la tabla.
- 2 Estudiar el DAG de los subproblemas.
- 3 Seleccionar un orden topológico con sentido.
- 4 Rellenar la tabla siguiendo ese orden topológico.

Estudiar las dimensiones de la tabla.

Estudiar las dimensiones de la tabla.

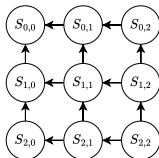
En total tenemos $S(i, j)$ $i, j = 0, \dots, n - 1$ problemas. Es decir, un total de n^2 problemas parametrizados por (i, j) . Así pues, tendremos que rellenar una tabla de $n \times n$ donde la celda (i, j) tendrá la solución de $S(i, j)$.

Estudiar el DAG de los subproblemas y orden topológico.

Estudiar las dimensiones de la tabla.

En total tenemos $S(i, j)$ $i, j = 0, \dots, n-1$ problemas. Es decir, un total de n^2 problemas parametrizados por (i, j) . Así pues, tendremos que rellenar una tabla de $n \times n$ donde la celda (i, j) tendrá la solución de $S(i, j)$.

Estudiar el DAG de los subproblemas y orden topológico. El DAG de la tabla o de los subproblemas tiene la siguiente forma:



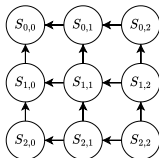
Así pues, nos valen dos órdenes topológicos para rellenar tabla...

Rellenar la tabla siguiendo ese orden topológico.

Estudiar las dimensiones de la tabla.

En total tenemos $S(i, j)$ $i, j = 0, \dots, n - 1$ problemas. Es decir, un total de n^2 problemas parametrizados por (i, j) . Así pues, tendremos que rellenar una tabla de $n \times n$ donde la celda (i, j) tendrá la solución de $S(i, j)$.

Estudiar el DAG de los subproblemas y orden topológico. El DAG de la tabla o de los subproblemas tiene la siguiente forma:



Así pues, nos valen dos órdenes topológicos para rellenar tabla...

Rellenar la tabla siguiendo ese orden topológico.

El orden: para cada $i = 0, \dots, n - 1$ y para cada $j = 0, \dots, n - 1$ rellenar $S(i, j)$.



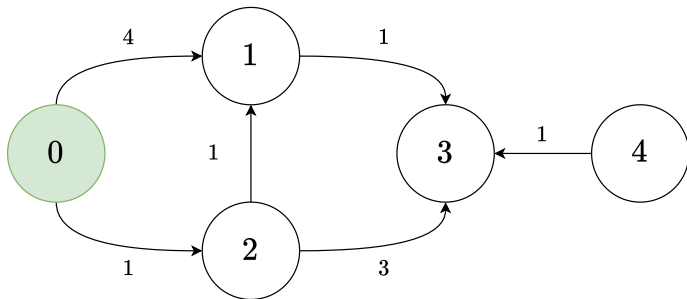
```
1 def manhattan_ascendente(n, m):
2     array = [[0]*(n+1) for i in range(m+1)]
3     for i in range(0, n + 1):
4         for j in range(0, m + 1):
5             if i == 0 == j:
6                 array[i][j] = 0
7             else:
8                 if i > 0:
9                     e_arriba = down[i - 1][j]
10                    m1 = array[i-1][j] + e_arriba
11                else:
12                    m1 = float("-inf")
13
14                if j > 0:
15                    e_izquierda = right[i][j - 1]
16                    m2 = array[i][j-1] + e_izquierda
17                else:
18                    m2 = float("-inf")
19                array[i][j] = max(m1, m2)
20     return array[n][m]
```

Para reconstruir la solución, la idea la misma a DP con memorización:

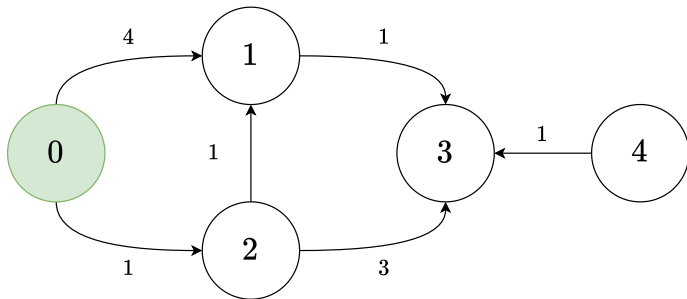
```
1 def manhattan_ascendente_solución(n, m):
2     ... # rellenamos la tabla
3
4     pos_actual = (n, m)
5     S = []
6     while pos_actual != (0,0):
7         S.append(pos_actual)
8         i, j = pos_actual
9         valor = array[i][j]
10
11         if i > 0:
12             e_arriba = down[i - 1][j]
13             m1 = array[i-1][j] + e_arriba
14         else:
15             m1 = float("-inf")
16
17         if j > 0:
18             e_izquierda = right[i][j - 1]
19             m2 = array[i][j-1] + e_izquierda
20         else:
21             m2 = float("-inf")
22
23         if valor == m1:
24             pos_actual = (i- 1, j)
25         else:
26             pos_actual = (i , j - 1)
27
28     return array[n][m], S
```


Descripción del problema

- Tenemos un DAG (grafo acíclico dirigido) $G = (V, E)$ con pesos positivos en las aristas, un nodo s origen y un nodo destino v .
- **Objetivo:** Calcular el camino mínimo de s a todos los nodos.



Bellman-Ford en un DAG 🚗



Codificación del grafo (listas de aristas incidentes con sus pesos):

```
1 G = {  
2   0: [],  
3   1: [(0, 4), (2, 1)],  
4   2: [(0, 1)],  
5   3: [(1, 1), (2, 3), (4, 1)],  
6   4: []  
7 }
```

Ecuación recurrente similar a Manhattan. Definimos $\delta(s, v)$ como el peso del camino mínimo de s a v . La idea es similar a Manhattan: estoy en v , ¿de dónde puede venir el camino mínimo?

Ecuación recurrente similar a Manhattan. Definimos $\delta(s, v)$ como el peso del camino mínimo de s a v . La idea es similar a Manhattan: estoy en v , ¿de dónde puede venir el camino mínimo?

$$\delta(s, v) = \min_{(u,v) \in E} \{\delta(s, u) + w(u, v)\}$$

Ecuación recurrente similar a Manhattan. Definimos $\delta(s, v)$ como el peso del camino mínimo de s a v . La idea es similar a Manhattan: estoy en v , ¿de dónde puede venir el camino mínimo?

$$\delta(s, v) = \min_{(u, v) \in E} \{\delta(s, u) + w(u, v)\}$$

Los casos base son:

Ecuación recurrente similar a Manhattan. Definimos $\delta(s, v)$ como el peso del camino mínimo de s a v . La idea es similar a Manhattan: estoy en v , ¿de dónde puede venir el camino mínimo?

$$\delta(s, v) = \min_{(u,v) \in E} \{\delta(s, u) + w(u, v)\}$$

Los casos base son:

- Si $s = v$ (igual que en Manhattan), entonces $\delta(s, v) = 0$

Ecuación recurrente similar a Manhattan. Definimos $\delta(s, v)$ como el peso del camino mínimo de s a v . La idea es similar a Manhattan: estoy en v , ¿de dónde puede venir el camino mínimo?

$$\delta(s, v) = \min_{(u, v) \in E} \{\delta(s, u) + w(u, v)\}$$

Los casos base son:

- Si $s = v$ (igual que en Manhattan), entonces $\delta(s, v) = 0$
- Si $v \neq s$ y v no tiene aristas incidentes, entonces $\delta(s, v) = \infty$

Ecuación recurrente similar a Manhattan. Definimos $\delta(s, v)$ como el peso del camino mínimo de s a v . La idea es similar a Manhattan: estoy en v , ¿de dónde puede venir el camino mínimo?

$$\delta(s, v) = \min_{(u, v) \in E} \{\delta(s, u) + w(u, v)\}$$

Los casos base son:

- Si $s = v$ (igual que en Manhattan), entonces $\delta(s, v) = 0$
- Si $v \neq s$ y v no tiene aristas incidentes, entonces $\delta(s, v) = \infty$

Recursivo:

```
1 def bf_recursivo_aux(G, s, v):
2     if s == v:
3         return 0
4     elif len(G[v]) == 0:
5         return float("inf")
6     else:
7         m = float("inf")
8         for u, w in G[v]:
9             m = min(m, bf_recursivo_aux(G, s, u) + w)
10        return m
11
12 def bf_recursivo(G, s):
13     resultado = [-1] * len(G)
14     for v in G:
15         resultado[v] = bf_recursivo_aux(G, s, v)
16     return resultado
17
18 bf_recursivo(G, 0)
19 # [0, 2, 1, 3, inf]
```

Esto tiene orden exponencial...

PD con memorización:

```
1 def bf_recur_sivo_aux_memo(G, s, v, memo):
2     if v in memo:
3         return memo[v]
4     if s == v:
5         m = 0
6     elif len(G[v]) == 0:
7         m = float("inf")
8     else:
9         m = float("inf")
10        for u, w in G[v]:
11            m = min(m, bf_recur_sivo_aux(G, s, u) + w)
12    memo[v] = m
13    return m
14
15 def bf_recur_sivo_memo(G, s):
16     memo = {}
17     for v in G:
18         bf_recur_sivo_aux_memo(G, s, v, memo)
19     return memo
20
21 bf_recur_sivo_memo(G, 0)
22 # {0: 0, 1: 2, 2: 1, 3: 3, 4: inf}
```

Bellman-Ford en un DAG

¿Qué orden tiene el algoritmo con memorización?

Bellman-Ford en un DAG

¿Qué orden tiene el algoritmo con memorización?

En este caso no podemos aplicar la fórmula:

$$\# \text{subproblemas diferentes} \times \text{tiempo/subproblema}$$

Ya que el tiempo/subproblema no es constante y depende del nodo v ...

Bellman-Ford en un DAG

¿Qué orden tiene el algoritmo con memorización?

En este caso no podemos aplicar la fórmula:

#subproblemas diferentes \times tiempo/subproblema

Ya que el tiempo/subproblema no es constante y depende del nodo v ...

$$\delta(s, v) = \min_{(u,v) \in E} \{\delta(s, u) + w(u, v)\}$$

La operaciones no recursivas usadas para resolver ese subproblema son:

Bellman-Ford en un DAG

¿Qué orden tiene el algoritmo con memorización?

En este caso no podemos aplicar la fórmula:

$$\# \text{subproblemas diferentes} \times \text{tiempo/subproblema}$$

Ya que el tiempo/subproblema no es constante y depende del nodo v ...

$$\delta(s, v) = \min_{(u,v) \in E} \{ \delta(s, u) + w(u, v) \}$$

La operaciones no recursivas usadas para resolver ese subproblema son:

$$C \cdot \text{in_degree}(v) + C'$$

Como el número de subproblemas diferentes es igual al número de nodos, podemos calcular el tiempo:

Bellman-Ford en un DAG

¿Qué orden tiene el algoritmo con memorización?

En este caso no podemos aplicar la fórmula:

#subproblemas diferentes \times tiempo/subproblema

Ya que el tiempo/subproblema no es constante y depende del nodo v ...

$$\delta(s, v) = \min_{(u, v) \in E} \{ \delta(s, u) + w(u, v) \}$$

La operaciones no recursivas usadas para resolver ese subproblema son:

$$C \cdot \text{in_degree}(v) + C'$$

Como el número de subproblemas diferentes es igual al número de nodos, podemos calcular el tiempo:

$$\sum_{v \in V} [C \cdot \text{in_degree}(v) + C'] = C \sum_{v \in V} \text{in_degree}(v) + C'|V| = C|E| + C'|V|$$

Bellman-Ford en un DAG

Para reconstruir la solución aplicamos la misma idea que en el recorrido por Manhattan. La caché memo tiene la siguiente forma:

```
1 {0: 0,  
2 1: 2,  
3 2: 1,  
4 3: 3,  
5 4: inf}
```


Para reconstruir la solución aplicamos la misma idea que en el recorrido por Manhattan. La caché memo tiene la siguiente forma:

```
1 {0: 0,  
2 1: 2,  
3 2: 1,  
4 3: 3,  
5 4: inf}
```

Supongamos que queremos reconstruir el camino desde el nodo 3 cuyo peso del camino mínimo es 3:

- 1 Estamos en el nodo 3 y sabemos que el camino viene desde uno de sus vecinos: 2 o 1.

Bellman-Ford en un DAG

Para reconstruir la solución aplicamos la misma idea que en el recorrido por Manhattan. La caché memo tiene la siguiente forma:

```
1 {0: 0,  
2 1: 2,  
3 2: 1,  
4 3: 3,  
5 4: inf}
```

Supongamos que queremos reconstruir el camino desde el nodo 3 cuyo peso del camino mínimo es 3:

- 1 Estamos en el nodo 3 y sabemos que el camino viene desde uno de sus vecinos: 2 o 1.
- 2 $3 = \min(\text{desde } 1 = 2 + 1, \text{desde } 2 = 1 + 3)$. Por tanto, venimos desde 1.

Para reconstruir la solución aplicamos la misma idea que en el recorrido por Manhattan. La caché memo tiene la siguiente forma:

```
1 {0: 0,  
2 1: 2,  
3 2: 1,  
4 3: 3,  
5 4: inf}
```

Supongamos que queremos reconstruir el camino desde el nodo 3 cuyo peso del camino mínimo es 3:

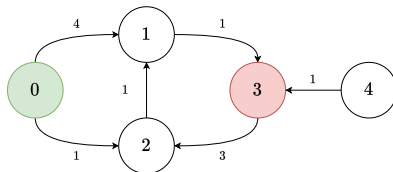
- 1 Estamos en el nodo 3 y sabemos que el camino viene desde uno de sus vecinos: 2 o 1.
- 2 $3 = \min(\text{desde } 1 = 2 + 1, \text{ desde } 2 = 1 + 3)$. Por tanto, venimos desde 1.
- 3 Así pues pasamos al nodo 1 y repetimos el mismo procedimiento hasta llegar al origen.

Bellman-Ford en un DAG

```
1 def reconstruir_memo(G, memo, s, t):
2     if memo[t] == float("inf"):
3         return "No hay camino"
4
5     nodo_actual = t
6     camino = [nodo_actual]
7     while nodo_actual != s:
8         valor = memo[nodo_actual]
9         siguiente = -1
10        for incidente, peso in G[nodo_actual]:
11            if valor == peso + memo[incidente]:
12                siguiente = incidente
13                break
14        camino.append(siguiente)
15        nodo_actual = siguiente
16    return camino[::-1]
```

Bellman-Ford en un DAG 🚗

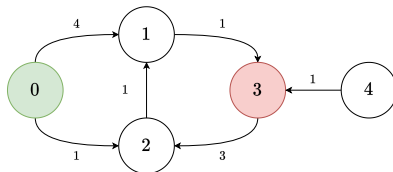
¿Por qué pedimos que no haya ciclos en el grafo?



$$\delta(s, v) = \min_{(u,v) \in E} \{\delta(s, u) + w(u, v)\}$$

Bellman-Ford en un DAG 🚗

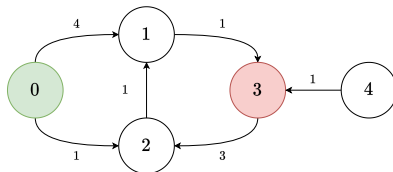
¿Por qué pedimos que no haya ciclos en el grafo?



$$\delta(s, v) = \min_{(u,v) \in E} \{\delta(s, u) + w(u, v)\}$$

Vamos a calcular $\delta(0, 3)$:

¿Por qué pedimos que no haya ciclos en el grafo?

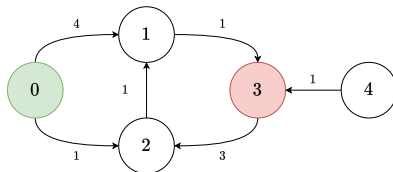


$$\delta(s, v) = \min_{(u,v) \in E} \{\delta(s, u) + w(u, v)\}$$

Vamos a calcular $\delta(0, 3)$:

- Para calcular $\delta(0, 3)$ se necesita $\delta(0, 1)$ y $\delta(0, 2)$.

¿Por qué pedimos que no haya ciclos en el grafo?

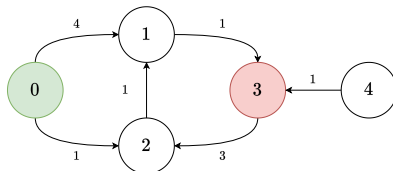


$$\delta(s, v) = \min_{(u,v) \in E} \{\delta(s, u) + w(u, v)\}$$

Vamos a calcular $\delta(0, 3)$:

- Para calcular $\delta(0, 3)$ se necesita $\delta(0, 1)$ y $\delta(0, 4)$.
- Para calcular $\delta(0, 1)$ necesitamos $\delta(0, 2)$ y $\delta(0, 0)$.

¿Por qué pedimos que no haya ciclos en el grafo?

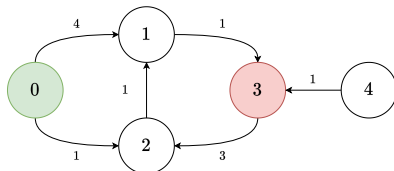


$$\delta(s, v) = \min_{(u,v) \in E} \{\delta(s, u) + w(u, v)\}$$

Vamos a calcular $\delta(0, 3)$:

- Para calcular $\delta(0, 3)$ se necesita $\delta(0, 1)$ y $\delta(0, 2)$.
- Para calcular $\delta(0, 1)$ necesitamos $\delta(0, 2)$ y $\delta(0, 0)$.
- Para calcular $\delta(0, 2)$ necesitamos $\delta(0, 0)$ y $\delta(0, 3)$.

¿Por qué pedimos que no haya ciclos en el grafo?



$$\delta(s, v) = \min_{(u,v) \in E} \{\delta(s, u) + w(u, v)\}$$

Vamos a calcular $\delta(0, 3)$:

- Para calcular $\delta(0, 3)$ se necesita $\delta(0, 1)$ y $\delta(0, 4)$.
- Para calcular $\delta(0, 1)$ necesitamos $\delta(0, 2)$ y $\delta(0, 0)$.
- Para calcular $\delta(0, 2)$ necesitamos $\delta(0, 0)$ y $\delta(0, 3)$. Pero $\delta(0, 3)$ es el primer problema!!! Esto da lugar a un bucle infinito!

Descripción del problema

- Tenemos $O = \{1, \dots, n\}$, n objetos con pesos $p_i > 0$ beneficios $b_i > 0$.
- En la mochila tenemos que meter objetos, con un peso máximo de M . **No** se pueden fraccionar los objetos. Un objeto o se mete o no se mete.
- **Objetivo:** Llenar la mochila maximizando el beneficio sin superar la capacidad máxima.

Vamos a asumir que los pesos p_i y M son enteros.

Sea $P(j, m)$ el beneficio máximo de considerar una mochila con los objetos 1 hasta j disponibles con la capacidad disponible m . Asumimos que los objetos están numerados $1, \dots, n$.

Sea $P(j, m)$ el beneficio máximo de considerar una mochila con los objetos 1 hasta j disponibles con la capacidad disponible m . Asumimos que los objetos están numerados $1, \dots, n$.

El primer paso es la ecuación de recurrencia... Recordad, no sé cual es la solución, probemos todas las combinaciones.

Sea $P(j, m)$ el beneficio máximo de considerar una mochila con los objetos 1 hasta j disponibles con la capacidad disponible m . Asumimos que los objetos están numerados $1, \dots, n$.

El primer paso es la ecuación de recurrencia... Recordad, no sé cual es la solución, probemos todas las combinaciones.

$$P(j, m) = \max\{b_j + P(j-1, m - p_j), P(j-1, m)\}$$

Los casos base son:

Sea $P(j, m)$ el beneficio máximo de considerar una mochila con los objetos 1 hasta j disponibles con la capacidad disponible m . Asumimos que los objetos están numerados $1, \dots, n$.

El primer paso es la ecuación de recurrencia... Recordad, no sé cual es la solución, probemos todas las combinaciones.

$$P(j, m) = \max\{b_j + P(j-1, m - p_j), P(j-1, m)\}$$

Los casos base son:

- Si tenemos capacidad negativa $m < 0$, eso quiere decir que hemos metido un objeto que no cabe: el beneficio es $-\infty$ para que en el máx no compute.

Sea $P(j, m)$ el beneficio máximo de considerar una mochila con los objetos 1 hasta j disponibles con la capacidad disponible m . Asumimos que los objetos están numerados $1, \dots, n$.

El primer paso es la ecuación de recurrencia... Recordad, no sé cual es la solución, probemos todas las combinaciones.

$$P(j, m) = \max\{b_j + P(j-1, m - p_j), P(j-1, m)\}$$

Los casos base son:

- Si tenemos capacidad negativa $m < 0$, eso quiere decir que hemos metido un objeto que no cabe: el beneficio es $-\infty$ para que en el máx no compute.
- Si nos hemos quedado sin objetos $j = 0$, entonces el beneficio es 0.

Sea $P(j, m)$ el beneficio máximo de considerar una mochila con los objetos 1 hasta j disponibles con la capacidad disponible m . Asumimos que los objetos están numerados $1, \dots, n$.

El primer paso es la ecuación de recurrencia... Recordad, no sé cual es la solución, probemos todas las combinaciones.

$$P(j, m) = \max\{b_j + P(j-1, m - p_j), P(j-1, m)\}$$

Los casos base son:

- Si tenemos capacidad negativa $m < 0$, eso quiere decir que hemos metido un objeto que no cabe: el beneficio es $-\infty$ para que en el máx no compute.
- Si nos hemos quedado sin objetos $j = 0$, entonces el beneficio es 0.
- Si nos hemos quedado sin capacidad $m = 0$, entonces el beneficio es 0.

Empecemos por memorización:

Empecemos por memorización:

```
1 def mochila_memo(P, M, j, B, memo):
2     if (j, M) in memo:
3         return memo[(j, M)]
4     if M < 0:
5         m = -float("inf")
6     elif j == 0:
7         m = 0
8     elif M == 0:
9         m = 0
10    else:
11        m1 = B[j-1] + mochila_memo(P, M - P[j-1], j - 1, B, memo)
12        m2 = mochila_memo(P, M, j - 1, B, memo)
13        m = max(m1, m2)
14    memo[(j, M)] = m
15    return m
```

Se puede probar que el número de subproblemas es $O(nM)$ (la prueba de esto queda fuera del alcance de la asignatura). Por otro lado, el tiempo no recursivo usado para resolver cada subproblemas es constante. Así pues el orden es $O(nM)$.

Ahora pasamos al método DP ascendente.
Estudiar las dimensiones de la tabla.

Ahora pasamos al método DP ascendente.

Estudiar las dimensiones de la tabla. Definimos una tabla para almacenar los resultados a los subproblemas. La tabla tiene dos dimensiones j y m . Y como queremos resolver $P(n, M)$, la tabla tendrá dimensiones $(n + 1)(M + 1)$ (ya que incluimos los 0s).

Ahora pasamos al método DP ascendente.

Estudiar las dimensiones de la tabla. Definimos una tabla para almacenar los resultados a los subproblemas. La tabla tiene dos dimensiones j y m . Y como queremos resolver $P(n, M)$, la tabla tendrá dimensiones $(n + 1)(M + 1)$ (ya que incluimos los 0s).

Estudiar el DAG de los subproblemas y orden topológico Supongamos $n = 3$, $M = 6$, $p = (2, 3, 4)$ y $b = (1, 2, 5)$. La tabla tiene esta forma:

		m						
		0	1	2	3	4	5	6
j	0	$P(0, 0)$	$P(0, 1)$	$P(0, 2)$	$P(0, 3)$	$P(0, 4)$	$P(0, 5)$	$P(0, 6)$
	1	$P(1, 0)$	$P(1, 1)$	$P(1, 2)$	$P(1, 3)$	$P(1, 4)$	$P(1, 5)$	$P(1, 6)$
	2	$P(2, 0)$	$P(2, 1)$	$P(2, 2)$	$P(2, 3)$	$P(2, 4)$	$P(2, 5)$	$P(2, 6)$
	3	$P(3, 0)$	$P(3, 1)$	$P(3, 2)$	$P(3, 3)$	$P(3, 4)$	$P(3, 5)$	$P(3, 6)$

Rellenar la tabla siguiendo ese orden topológico.

El orden: para cada $j = 0, \dots, n$ y para cada $m = 0, \dots, M$ rellenar $P(j, m)$.

```
1 def mochila_ascendente(P, M, n, B):
2     array = [[0]*(M+1) for i in range(n+1)]
3     for x in range(1, n + 1):
4         for y in range(1, M + 1):
5             if y - P[x - 1] >= 0:
6                 m1 = B[x-1] + array[x - 1][y - P[x - 1]]
7                 m2 = array[x - 1][y]
8             else:
9                 m1 = -float("inf")
10                m2 = array[x - 1][y]
11                array[x][y] = max(m1, m2)
12     return array[n][M]
```

Ahora hay que reconstruir la solución. Vamos a hacerlo para DP ascendente.

```
1 def mochila_ascendente_reconstruir(P, M, n, B):
2     ... # construimos la tabla
3
4     x_actual = n
5     y_actual = M
6     S = []
7     while x_actual != 0:
8         if y_actual - P[x_actual - 1] >= 0:
9             m1 = B[x_actual-1] + array[x_actual - 1][y_actual - P[x_actual - 1]]
10            m2 = array[x_actual - 1][y_actual]
11        else:
12            m1 = -float("inf")
13            m2 = array[x_actual - 1][y_actual]
14        if m1 > m2:
15            S.append(x_actual - 1)
16            y_actual -= P[x_actual - 1]
17            x_actual = x_actual - 1
18    return array[n][M], S
```


¿Qué complejidad tiene el algoritmo ascendente?

¿Qué complejidad tiene el algoritmo ascendente?

El algoritmo ascendente consiste en rellenar una tabla de tamaño $(M + 1) \times (n + 1)$. Así pues, es $O(Mn)$.

¿Qué complejidad tiene el algoritmo ascendente?

El algoritmo ascendente consiste en rellenar una tabla de tamaño $(M + 1) \times (n + 1)$. Así pues, es $O(Mn)$.

Pero el problema es NP-completo, ¿qué está pasando?

¿Qué complejidad tiene el algoritmo ascendente?

El algoritmo ascendente consiste en rellenar una tabla de tamaño $(M + 1) \times (n + 1)$. Así pues, es $O(Mn)$.

Pero el problema es NP-completo, ¿qué está pasando?

$O(Mn)$ parece polinomial, pero no lo es; es **pseudo-polinomial**. El algoritmo es lineal en el valor de M pero exponencial en la longitud de M .

¿Qué complejidad tiene el algoritmo ascendente?

El algoritmo ascendente consiste en rellenar una tabla de tamaño $(M + 1) \times (n + 1)$. Así pues, es $O(Mn)$.

Pero el problema es NP-completo, ¿qué está pasando?

$O(Mn)$ parece polinomial, pero no lo es; es **pseudo-polinomial**. El algoritmo es lineal en el valor de M pero exponencial en la longitud de M . El tamaño de M no es el valor y se mide con los bits que son necesarios para representarlo.

Si asumimos $n = 10$, $M = 8$:

- Objetos $1, \dots, 10$
- Capacidad $M = 1000$, 4 bits

$$t(n) \sim 10 \times 8 = 80$$

Si asumimos $n = 10$, $M = 8$:

- Objetos $1, \dots, 10$
- Capacidad $M = 1000$, 4 bits

$$t(n) \sim 10 \times 8 = 80$$

Si doblo el tamaño n :

- Objetos $1, \dots, 20$
- Capacidad $M = 1000$, 4 bits

$$t(n) \sim 20 \times 8 = 160, \text{ es el doble de operaciones}$$

Si asumimos $n = 10$, $M = 8$:

- Objetos $1, \dots, 10$
- Capacidad $M = 1000$, 4 bits

$$t(n) \sim 10 \times 8 = 80$$

Si doblo el tamaño n :

- Objetos $1, \dots, 20$
- Capacidad $M = 1000$, 4 bits

$$t(n) \sim 20 \times 8 = 160, \text{ es el doble de operaciones}$$

Si doblo el tamaño de M (no $M = 16$) tenemos que:

- Objetos $1, \dots, 10$
- Capacidad $M = 10000000$, 8 bits

$$t(n) \sim 10 \times 128 = 1280, \text{ ha crecido de manera exponencial...}$$

Si asumimos $n = 10$, $M = 8$:

- Objetos $1, \dots, 10$
- Capacidad $M = 1000$, 4 bits

$$t(n) \sim 10 \times 8 = 80$$

Si doblo el tamaño n :

- Objetos $1, \dots, 20$
- Capacidad $M = 1000$, 4 bits

$$t(n) \sim 20 \times 8 = 160, \text{ es el doble de operaciones}$$

Si doblo el tamaño de M (no $M = 16$) tenemos que:

- Objetos $1, \dots, 10$
- Capacidad $M = 10000000$, 8 bits

$t(n) \sim 10 \times 128 = 1280$, ha crecido de manera exponencial... Así pues, realmente, $t(n) \in O(n2^{\text{bits}})$.



Descripción del problema

- Tenemos n monedas (identificadas por $O = \{1, \dots, n\}$) con valores $V = \{v_1, \dots, v_n\}$.
- Nos dan una cantidad C que tenemos que devolver usando ese sistema monetario.
- **Objetivo:** Devolver la cantidad C minimizando el número de monedas.

Ecuación de recurrencia (dos opciones, o cogemos una moneda de valor v_j o no cogemos ninguna de ese valor):

$$P(j, c) = \min\{1 + P(j - 1, c - v_j), P(j - 1, c)\}$$

Los casos base son:

- Si tenemos cantidad negativa $c < 0$, eso quiere decir que hemos metido una moneda demasiado grande: el número de moneadas es ∞ para que en el \min no compute.
- Si nos hemos quedado sin monedas $j = 0$ y la $c > 0$, entonces el número de monedas es ∞ (esto ocurre cuando no hay solución).
- Si ya hemos devuelto toda la cantidad $c = 0$, entonces el beneficio es 0.



PD con memorización:

```
1 def cambio_memo(V, c, j, memo):
2     if (j,c) in memo:
3         return memo[(j, c)]
4     if c < 0 or j < 0:
5         r = float("inf")
6     elif c == 0 and j >= 0:
7         r = 0
8     elif c > 0 and j == 0:
9         r = float("inf")
10    else:
11        r1 = 1 + cambio_memo(V, c - V[j - 1], j, memo)
12        r2 = cambio_memo(V, c, j - 1, memo)
13        r = min(r1, r2)
14    memo[(j, c)] = r
15    return r
16
17 V = [1, 4, 6]
18 C = 8
19 memo = {}
20 cambio_memo(V, C, len(V), memo)
21 # 2
```



PD ascendente:

```
1 def cambio_ascendente(V, C):
2     n = len(V)
3     tabla = [[-1]*(C+1) for i in range(n+1)]
4
5     for j in range(0, n + 1):
6         for c in range(0, C + 1):
7             if c == 0:
8                 tabla[j][c] = 0
9             elif j == 0 and c > 0:
10                 tabla[j][c] = float("inf")
11             else:
12                 if c - V[j - 1] >= 0:
13                     r1 = 1 + tabla[j][c - V[j - 1]]
14                 else:
15                     r1 = float("inf")
16                 r2 = tabla[j-1][c]
17                 r = min(r1, r2)
18                 tabla[j][c] = r
19     return tabla, tabla[n][C]
20
21 V = [1, 4, 6]
22 C = 8
23 tabla, resultado = cambio_ascendente(V, C)
24 resultado
25 # 2
```

El tiempo de este algoritmo es $O(nC)$ y, al igual que el de la mochila, es pseudo-polinomial.



Reconstrucción a partir de la tabla:

```
1 def reconstruir(tabla, V, C):
2     n = len(V)
3     if tabla[n][C] == float("inf"):
4         return "No hay solución"
5     else:
6         S = []
7         j_actual = n
8         c_actual = C
9         while c_actual != 0:
10             if c_actual - V[j_actual - 1] >= 0:
11                 r1 = 1 + tabla[j_actual][c_actual - V[j_actual - 1]]
12             else:
13                 r1 = float("inf")
14             r2 = tabla[j_actual - 1][c_actual]
15             if r1 < r2:
16                 S.append(f"Cojo 1 moneda de {V[j_actual - 1]}")
17                 c_actual -= V[j_actual - 1]
18             else:
19                 j_actual -= 1
20         return S
21
22 V = [1, 4, 6]
23 C = 8
24 reconstruir(tabla, V, C)
25 # ['Cojo 1 moneda de 4', 'Cojo 1 moneda de 4']
```



- Hay dos formas de ver PD: PD con memorización y PD ascendente.
- La idea principal reside en cachear los subproblemas para no repetir cálculos innecesarios.
- En muchos casos, PD transforma recursiones exponenciales en algoritmos polinomiales.
- El cálculo del tiempo de ejecución depende de la técnica PD utilizada.
 - Si PD con memorización, hay que sumar el tiempo no recursivo que se tarda en resolver cada subproblema distinto.
 - Si PD ascendente, hay que calcular el tiempo que se tarda en recorrer y rellenar la tabla.