Programación dinámica

Algoritmos y estructuras de datos II

José Antonio Hernández López[1]

[1]Departamento de Informática y Sistemas
Universidad de Murcia

16 de marzo de 2025

Índice

Programación dinámica

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)

Programación dinámica con memorización

Término $n-$ésimo de la sucesión Fibonacci

La succesión de Fibonacci tiene la siguente 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 $n-$éismo.

Programación dinámica con memorización

Término $n-$ésimo de la sucesión Fibonacci

La sucesión de Fibonacci tiene la siguente 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 $n-$éismo.

El algoritmo recursivo es:

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

Programación dinámica con memorización

Término $n-$ésimo de la sucesión Fibonacci

La sucesión de Fibonacci tiene la siguente 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 $n-$éismo.

El algoritmo recursivo es:

```python
def fib(n):
    if n <= 2:
        return 1
    else:
        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):

Programación dinámica con memorización

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

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

¿Qué orden tiene este algoritmo?

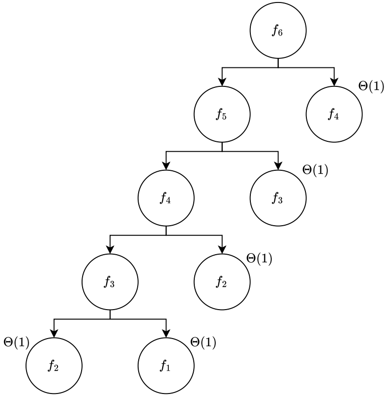
Programación dinámica con memorización

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

```
def fib_memo(n, memo):
    if n in memo: # asumimos que esto es constante
        return memo[n]
    if n <= 2:
        f = 1
    else:
        f = fib_memo(n-1,memo) + fib_memo(n-2, memo)
    memo[n] = f  # asumimos que esto es constante
    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 `fib(n-2, memo)` ya se ha ejecutado y guardado en algún punto de `fib(n-1, 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 son $\Theta(1)$ y dos casos base $\Theta(1)$.
- 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*):

```
def fib_ascendente(n):
    arr = [0] * (n + 1)
    for k in range(1, n + 1):
        if k <= 2:
            f = 1
        else:
            f = arr[k-1] + arr[k-2]
        arr[k] = f
    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*):

```
def fib_ascendente(n):
    arr = [0] * (n + 1)
    for k in range(1, n + 1):
        if k <= 2:
            f = 1
        else:
            f = arr[k-1] + arr[k-2]
        arr[k] = f
    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.

Programación dinámica ascendente

La base de la PD ascendente es:

Resolución de subproblemas pequeños

Resolvemos los subproblemas plequeños, 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.**

Programación dinámica ascendente

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)

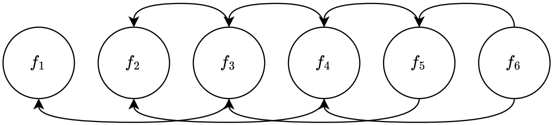
Estudiar las dimensiones de la tabla

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

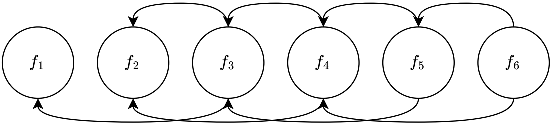
La ecuación de recurrencia depende de un parámetro. Así pues, la tabla para estos problemas suele ser unidimensional. Como hay n subproblemas diferentes, la tabla tendrá dimensión n y cada i contendrá la solución $i-$ésima (término de fibonacci $i-$éismo).

Estudiar el DAG de los subproblemas

Estudiar el DAG de los subproblemas

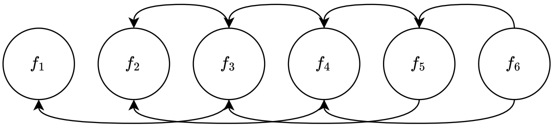


Estudiar el DAG de los subproblemas



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, . . . , f_n.

Rellenar la tabla siguiendo el orden seleccionado

```
def fib_ascendente(n):
    arr = [0] * (n + 1)
    for k in range(1, n + 1):
        # casos base
        if k <= 2:
            f = 1
        else:
            # esto puedo hacerlo porque he escogido un orden válido
            # y los dos subproblemas están resueltos
            f = arr[k-1] + arr[k-2]
        # guardamos el resultado
        arr[k] = f
    # devolvemos el resultado al problema grande
    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 ténica 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. Recomponer 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 $\max$ o $\min$ no $\arg\max$ o $\arg\min$.

¿Cuándo funciona la programación dinámica?

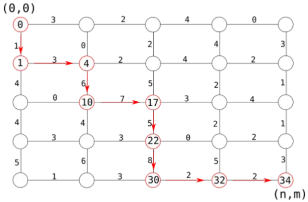
Un problema es candidato para ser resuelto con PD si cumple las siguientes propiedades:

- 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.

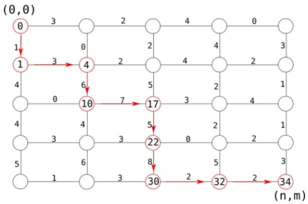
Recorrido turístico por Manhattan 🏛️

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 🏛️



Vamos a asumir que el grafo está codificado de la siguiente forma:

```
# moverse abajo de (i,j)
# down[0][0] es el peso de la arista que va desde (0,0) a (1,0)
down = [
    [1, 0, 2, 4, 3],
    [4, 6, 5, 2, 1],
    [4, 4, 5, 2, 1],
    [5, 6, 8, 5, 3]
]

# moverse derecha desde (i,j)
# right[0][0] es el peso de la arista que va desde (0,0) a (0,1)
right = [
    [3, 2, 4, 0],
    [3, 2, 4, 2],
    [0, 7, 3, 4],
    [3, 3, 0, 2],
    [1, 3, 2, 2]
]
```

También vamos a asumir lo siguiente:

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

Recorrido turístico por Manhattan 🏛️

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.

Recorrido turístico por Manhattan 🏛️

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*...

Recorrido turístico por Manhattan 🏛️

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*...

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

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

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

Recorrido turístico por Manhattan 🏛️

El segundo paso es escoger el método PD y aplicarlo. Supongamos que escogemos PD con memorización.
Primero, escribimos la recursión ingénua:

Recorrido turístico por Manhattan 🏛️

El segundo paso es escoger el método PD y aplicarlo. Supongamos que escogemos PD con memorización.
Primero, escribimos la recursión ingénua:

```python
def manhattan_recursivo(i, j):
    if i == j == 0:
        m = 0
    else:
        if i > 0:
            e_arriba = down[i - 1][j]
            m1 = manhattan_recursivo(i - 1, j) + e_arriba
        else:
            m1 = float("-inf")

        if j > 0:
            e_izquierda = right[i][j - 1]
            m2 = manhattan_recursivo(i, j - 1) + e_izquierda
        else:
            m2 = float("-inf")

        m = max(m1, m2)
    return m
```

La complejidad de esto es

Recorrido turístico por Manhattan

El segundo paso es escoger el método PD y aplicarlo. Supongamos que escogemos PD con memorización.
Primero, escribimos la recursión ingénua:

```python
def manhattan_recursivo(i, j):
    if i == j == 0:
        m = 0
    else:
        if i > 0:
            e_arriba = down[i - 1][j]
            m1 = manhattan_recursivo(i - 1, j) + e_arriba
        else:
            m1 = float("-inf")

        if j > 0:
            e_izquierda = right[i][j - 1]
            m2 = manhattan_recursivo(i, j - 1) + e_izquierda
        else:
            m2 = float("-inf")

        m = max(m1, m2)
    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:

```python
def manhattan_memo(i, j, memo):
    if (i, j) in memo:
        return memo[(i,j)]
    if i == j == 0:
        m = 0
    else:

        if i > 0:
            e_arriba = down[i - 1][j]
            m1 = manhattan_memo(i - 1, j, memo) + e_arriba
        else:
            m1 = float("-inf")

        if j > 0:
            e_izquierda = right[i][j - 1]
            m2 = manhattan_memo(i, j - 1, memo) + e_izquierda
        else:
            m2 = float("-inf")

        m = max(m1, m2)
    memo[(i,j)] = m
    return m
```

¿Orden?

Recorrido turístico por Manhattan 🏛️

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 $\arg\max$.

Recorrido turístico por Manhattan 🏛️

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

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

`memo` representa el siguiente tablero:

```
[[0, 3, 5, 9, 9],
[1, 4, 7, 13, 15],
[5, 10, 17, 20, 24],
[9, 14, 22, 22, 25],
[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. $34 = \text{máx}(\text{izq} = 32 + 2, \text{arriba} = 25 + 3)$, 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 $(0, 0)$.

Recorrido turístico por Manhattan 🏛️

```python
def reconstruir_memo(memo, n, m):
    pos_actual = (n, m)
    S = []
    while pos_actual != (0,0):
        S.append(pos_actual)
        i, j = pos_actual
        valor = memo[(i,j)]

        if i > 0:
            e_arriba = down[i - 1][j]
            m1 = memo[(i-1,j)] + e_arriba
        else:
            m1 = float("-inf")

        if j > 0:
            e_izquierda = right[i][j - 1]
            m2 = memo[(i,j-1)] + e_izquierda
        else:
            m2 = float("-inf")

        if valor == m1:
            pos_actual = (i- 1, j)
        else:
            pos_actual = (i , j - 1)
    return S
```

Recorrido turístico por Manhattan 🏛️

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 $S(i,j)$ $i,j = 0,\ldots,n-1$ 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.

Recorrido turístico por Manhattan 🏛️

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:

$$
\begin{tikzcd}
S_{0,0} & S_{0,1} \arrow[l] & S_{0,2} \arrow[l] \\
S_{1,0} \arrow[u] & S_{1,1} \arrow[l] \arrow[u] & S_{1,2} \arrow[l] \arrow[u] \\
S_{2,0} \arrow[u] & S_{2,1} \arrow[l] \arrow[u] & S_{2,2} \arrow[l] \arrow[u]
\end{tikzcd}
$$

Así pues, nos valen dos órdenes topológicos para rellenar tabla...
Rellenar la tabla siguiendo ese orden topológico.

Recorrido turístico por Manhattan 🏛️

Estudiar las dimensiones de la tabla.
En total tenemos $S(i,j)$ $i,j = 0,\ldots,n-1$ 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:

$$\begin{tikzcd}
S_{0,0} & S_{0,1} \arrow[l] & S_{0,2} \arrow[l] \\
S_{1,0} \arrow[u] & S_{1,1} \arrow[l] \arrow[u] & S_{1,2} \arrow[l] \arrow[u] \\
S_{2,0} \arrow[u] & S_{2,1} \arrow[l] \arrow[u] & S_{2,2} \arrow[l] \arrow[u]
\end{tikzcd}$$

Así pues, nos valen dos órdenes topológicos para rellenar tabla...
Rellenar la tabla siguiendo ese orden topológico.
El orden: para cada $i = 0,\ldots,n-1$ y para cada $j = 0,\ldots,n-1$ rellenar $S(i,j)$.

Recorrido turístico por Manhattan 🏛️

```python
def manhattan_ascendente(n, m):
    array = [[0]*(n+1) for i in range(m+1)]
    for i in range(0, n + 1):
        for j in range(0, m + 1):
            if i == 0 == j:
                array[i][j] = 0
            else:
                if i > 0:
                    e_arriba = down[i - 1][j]
                    m1 = array[i-1][j] + e_arriba
                else:
                    m1 = float("-inf")

                if j > 0:
                    e_izquierda = right[i][j - 1]
                    m2 = array[i][j-1] + e_izquierda
                else:
                    m2 = float("-inf")
                array[i][j] = max(m1, m2)
    return array[n][m]
```

Recorrido turístico por Manhattan 🏛️

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

```python
def manhattan_ascendente_solución(n, m):
    ... # rellenamos la tabla

    pos_actual = (n, m)
    S = []
    while pos_actual != (0,0):
        S.append(pos_actual)
        i, j = pos_actual
        valor = array[i][j]

        if i > 0:
            e_arriba = down[i - 1][j]
            m1 = array[i-1][j] + e_arriba
        else:
            m1 = float("-inf")

        if j > 0:
            e_izquierda = right[i][j - 1]
            m2 = array[i][j-1] + e_izquierda
        else:
            m2 = float("-inf")

        if valor == m1:
            pos_actual = (i- 1, j)
        else:
            pos_actual = (i , j - 1)

    return array[n][m], S
```

Mochila 0/1

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.

Mochila 0/1 🎒

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 $1, \ldots, n$.

Mochila 0/1 🎒

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 $1, \ldots, n$.
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 $1, \ldots, n$.
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 $1, \ldots, n$.
El primer paso es la ecuación de recurrencia... Recordad, no sé cual es la solución, probemos todas las combinaciones.

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

Los casos base son:

- Si nos hemos quedado sin objetos $j = 0$, entonces el beneficio es 0.

Mochila 0/1

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 $1, \ldots, n$.
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 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.

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 $1,\ldots,n$.
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 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.
- 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 $\max$ no compute.

Empecemos por memorización:

Mochila 0/1 🎒

Empecemos por memorización:

```python
def mochila_memo(P, M, j, B, memo):
    if (j, M) in memo:
        return memo[(j, M)]
    if M < 0:
        m = -float("inf")
    elif j == 0:
        m = 0
    elif M == 0:
        m = 0
    else:
        m1 = B[j-1] + mochila_memo(P, M - P[j-1], j - 1, B, memo)
        m2 = mochila_memo(P, M, j - 1, B, memo)
        m = max(m1, m2)
    memo[(j, M)] = m
    return m
```

Mochila 0/1 🎒

Ahora pasamos al método DP ascendente.

Estudiar las dimensiones de la tabla.

Mochila 0/1 🎒

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 nM.

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 nM.

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

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

Mochila 0/1 🎒

Rellenar la tabla siguiendo ese orden topológico.

El orden: para cada $j = 1, \ldots, n$ y para cada $m = 1, \ldots, M$ rellenar $P(j, m)$.

```python
def mochila_ascendente(P, M, n, B):
    array = [[0]*(M+1) for i in range(n+1)]
    for x in range(1, n + 1):
        for y in range(1, M + 1):
            if y - P[x - 1] >= 0:
                m1 = B[x-1] + array[x - 1][y - P[x - 1]]
                m2 = array[x - 1][y]
            else:
                m1 = -float("inf")
                m2 = array[x - 1][y]
            array[x][y] = max(m1, m2)
    return array[n][M]
```

Mochila 0/1

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

```python
def mochila_ascendente_reconstruir(P, M, n, B):
    ... # construimos la tabla

    x_actual = n
    y_actual = M
    S = []
    while x_actual != 0:
        if y_actual - P[x_actual - 1] >= 0:
            m1 = B[x_actual-1] + array[x_actual - 1][y_actual - P[x_actual - 1]]
            m2 = array[x_actual - 1][y_actual]
        else:
            m1 = -float("inf")
            m2 = array[x_actual - 1][y_actual]
        if m1 > m2:
            S.append(x_actual - 1)
            y_actual -= P[x_actual - 1]
        x_actual = x_actual - 1
    return array[n][M], S
```

Mochila 0/1 🎒

¿Qué complejidad tiene el algoritmo ascendente?

¿Qué complejidad tiene el algoritmo ascendente?
El algoritmo ascendente consiste en rellenar una tabla de tamaño $M \times n$.
Así pues, es $O(Mn)$.

Mochila 0/1

¿Qué complejidad tiene el algoritmo ascendente?
El algoritmo ascendente consiste en rellenar una tabla de tamaño $M \times n$.
Así pues, es $O(Mn)$.
Pero el problema es NP-completo, ¿qué está pasando?

Mochila 0/1 🎒

¿Qué complejidad tiene el algoritmo ascendente?
El algoritmo ascendente consiste en rellenar una tabla de tamaño $M \times n$.
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.

Mochila 0/1

¿Qué complejidad tiene el algoritmo ascendente?
El algoritmo ascendente consiste en rellenar una tabla de tamaño $M \times n$.
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.

Mochila 0/1 🎒

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

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

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

Mochila 0/1 🎒

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

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

$t(n) \sim 10 \times 8 = 80$
Si doblo el tamaño n:

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

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

Mochila 0/1

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

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

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

Si doblo el tamaño n:

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

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

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

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

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

Mochila 0/1

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

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

$t(n) \sim 10 \times 8 = 80$
Si doblo el tamaño n:

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

$t(n) \sim 20 \times 8 = 160$, es el doble de operaciones
Si doblo el tamaño de M (no $M = 16$) tenemos que:

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

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