


Programación Dinamica

2da Parte

TTPS - 30/Septiembre

Programación Dinamica

Problemas

- RSQ (RANGE SUM QUERY)
- LIS (LONGEST INCREASING SUBSEQUENCE)
- LCS (LONGEST COMMON SUBSEQUENCE)
- ED (EDIT DISTANCE)
- **KNAPSACK** 

Knapsack Problem

Knapsack

Problema

Se tiene una mochila que puede **almacenar elementos**, con un **peso total máximo de K** . Dados **N elementos**, donde **cada uno tiene un peso W y un valor C** . ¿Cuál es la cantidad óptima de elementos que puede cargar en la mochila, de forma tal de **maximizar el valor total de los elementos**?



Knapsack

Ejemplos

- Capacidad de la mochila $\rightarrow K = 9$
- Cantidad de objetos $\rightarrow N = 3$
- Peso y valor de los objetos $\rightarrow E[(W, C)] = \{(4, 3), (4, 2), (5, 4)\}$
- Solución \rightarrow Se seleccionan elementos 0 y 2, resultando en un peso total de $4 + 5 = 9$ (máximo) y un valor de $3 + 4 = 7$

Knapsack

Ejemplos

- Capacidad de la mochila $\rightarrow K = 7$
- Cantidad de objetos $\rightarrow N = 4$
- Peso y valor de los objetos $\rightarrow E[(W, C)] = \{(1, 1), (3, 4), (4, 5), (5, 7)\}$
- Solución \rightarrow Se seleccionan elementos 1 y 2, resultando en un peso total de $4 + 3 = 7$ (máximo) y un valor de $4 + 5 = 9$

Knapsack

Problema

- Debemos verificar que disponemos de las siguientes dos propiedades para resolver utilizando Programación Dinámica:
 - Optimal Substructure
 - Overlapping Subproblems

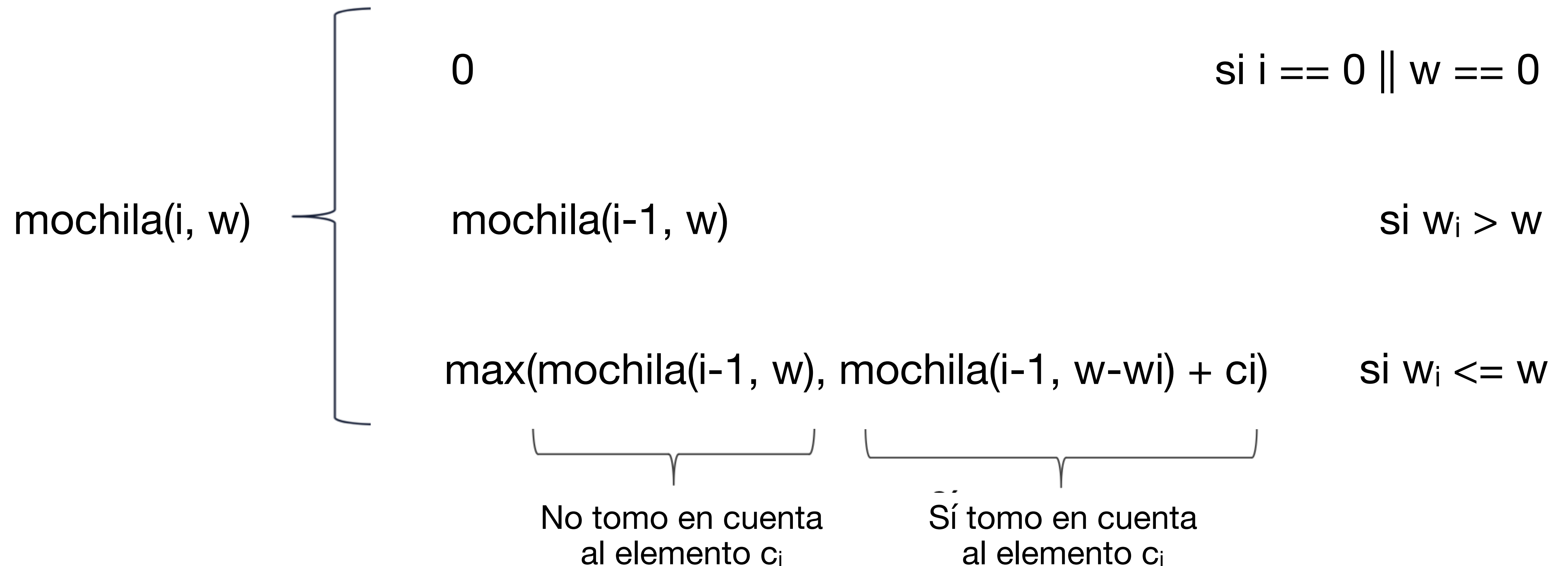
Optimal Substructure

Knapsack

- Tenemos que encontrar una función recursiva que resuelva el problema.
- El principio básico es que, para cada objeto, puedes tomar dos decisiones:
 - No incluir el objeto en la mochila.
 - Incluir el objeto en la mochila (si el peso del objeto es menor o igual a la capacidad disponible).
- La función recursiva deberá evaluar ambas posibilidades y devolverá el valor máximo.

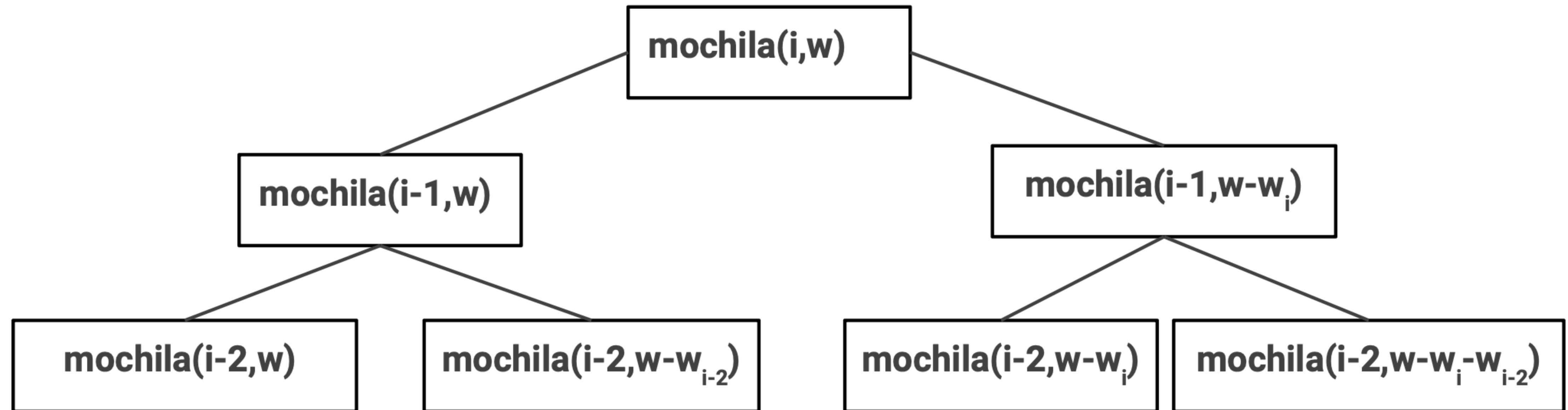
Optimal Substructure

Knapsack - Función Recursiva



Overlapping Subproblems

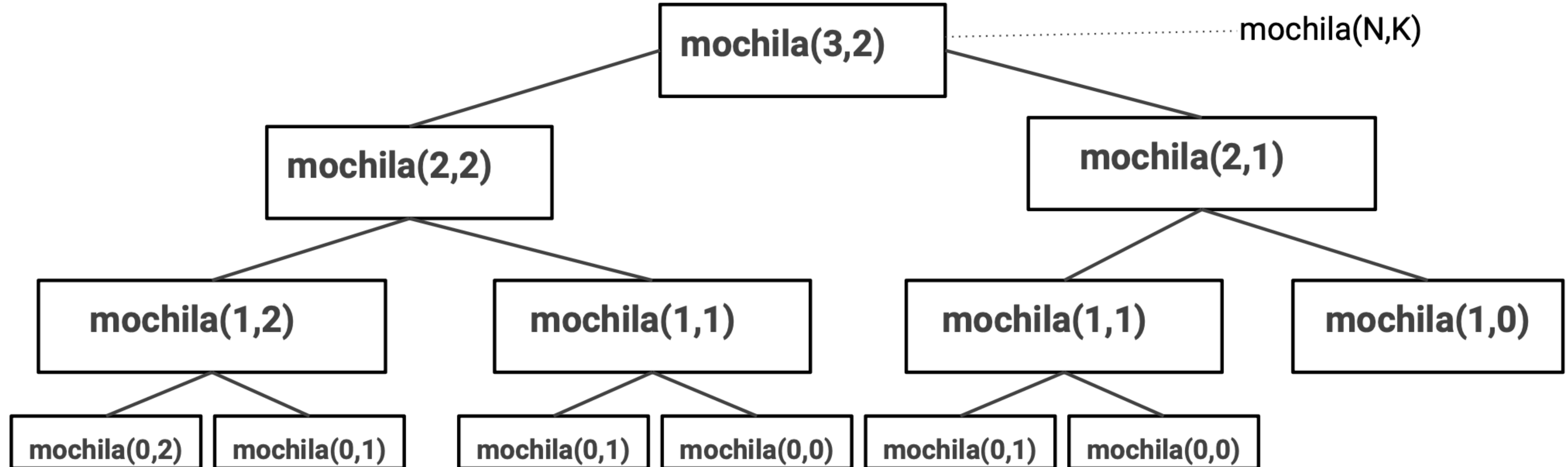
Knapsack



Overlapping Subproblems

Knapsack

Ejemplo: $K = 2$; $N = 3$; $E[(W, C)] = \{ (1, 10), (1, 20), (1, 30) \}$



Que estructura podemos utilizar para implementar PD?

Solución Knapsack

Estructura

- Utilizaré una matriz para representar los objetos a poner en la mochila:

	Peso	Valor
0	4	3
1	4	2
2	5	4

Estructura

- Por otro lado, utilizaremos una matriz para memorizar las soluciones de los subproblemas ya calculados

Tendremos una
matriz $N+1 \times K+1$

Las filas
representan los
objetos y las
columnas las
capacidades

[illegible]

Solución Knapsack

Proceso

- Por cada objeto y para cada capacidad hay dos opciones:
 - Si el peso del objeto actual es mayor que la capacidad actual, no se puede incluir el objeto y por tanto el valor es igual a no haberlo incluido
 - Si el peso del objeto actual es menor o igual que la capacidad actual obtengo el máximo entre: no incluir el objeto e incluirlo

	0	1	2	3
S/E	0	0	0	0
E-0	0			
E-1	0			
E-2	0			

Solución Knapsack

Proceso

- En este caso el primer objeto tiene peso igual a 4 y valor igual a 3.
- Si lo incluyo me paso la capacidad, por lo que esta se mantiene igual
- Tomo el mismo valor que tenia con anteriormente $DP[i-1][k]$

	0	1	2	3
S/E	0	0	0	0
E-0	0	0		
E-1	0			
E-2	0			

Solución Knapsack

Proceso

- Continuando con el proceso, cuando tengo capacidad para incluirlo:
- Maximo entre $DP[i-1][k]$; y $DP[i-1][k - elem[i-1][0]] + elem[i-1][1]$

	0	1	2	3	4
S/E	0	0	0	0	0
E-0	0	0	0	0	3
E-1	0				
E-2	0				

Solución Knapsack

Proceso

- Y así deberíamos continuar para cada uno de los elementos y para cada una de las capacidades.

	0	1	2	3	4	5	6	7	8	9
S/E	0	0	0	0	0	0	0	0	0	0
E-0	0	0	0	0	3	3	3	3	3	3
E-1	0	0	0	0	3	3	3	3	5	5
E-2	0	0	0	0	3	4	4	4	5	7

Analicemos estos casos particulares

Solución Knapsack

Bottom-up

```
int N, K //Elementos y capacidad
int[N][2] elem; //Peso y valor de elementos
int[N+1][K+1] DP; // Matriz de memorización

int mochila () {

    // Inicialización
    for (int i=0; i<=N; i++) DP[i][0] = 0;
    for (int j=0; j<=K; j++) DP[0][j] = 0;

    for (int i=1; i<=N; i++) {
        for (int j=1; j<=K; j++) {
            if (elem[i-1][0] > j) // En caso que no entre en la mochila
                DP[i][j] = DP[i-1][j];
            else // En caso de que si tengamos espacio
                DP[i][j] = max(
                    DP[i-1][j],
                    DP[i-1][j-elem[i-1][0]] + elem[i-1][1]
                );
        }
    }
    return DP[N][K];
}
```

Solución Knapsack

Top-Down

```
int N, K //Elementos y capacidad
int[N][2] elem; //Peso y valor de elementos
int[N][K] DP; // Matriz de memorización

int mochila (int i, int w) {

    if (DP[i][w] != -1) return DP[i][w];

    if (i == 0 || w == 0) return 0;

    int ans = mochila(i-1, w);

    if (elem[i][0] <= w)
        ans = max(
            ans,
            mochila(i-1, w-elem[i][0]) + elem[i][1]
        );

    return DP[i][w] = ans;
}
```