

PROBLEMA DE LA MOCHILA ENTERA

Solución utilizando programación dinámica para la versión entera del problema de la mochila. Los objetos no se pueden fraccionar

O el objeto se mete entero en la mochila, o si no NO se mete.

ALBERTO VERDEJO



UNIVERSIDAD
COMPLUTENSE
MADRID

Problema de la mochila (versión entera)

- ▶ Hay n objetos, cada uno con un peso (entero) $p_i > 0$ y un valor (real) $v_i > 0$.
- ▶ La mochila soporta un peso total (entero) máximo $M > 0$.
- ▶ Y el problema consiste en maximizar

No podemos superar el peso máximo mayor que 0 que es entero.

$$\sum_{i=1}^n x_i v_i$$

con la restricción

$$\sum_{i=1}^n x_i p_i \leq M,$$

El problema en el cual si que se pueden partir se puede resolver con un algoritmo voraz ordenando los objetos de mayor a menor densidad de valor -> v/p

donde $x_i \in \{0, 1\}$ indica si hemos cogido (1) o no (0) el objeto i .

No se pueden partir.

Problema de la mochila (versión entera)

- ▶ En las soluciones no importa el orden en el que los objetos son introducidos en la mochila.
- ▶ Las soluciones dependen de los objetos que tengamos disponibles para introducir en la mochila y del peso que soporte esta.
- ▶ Definimos la siguiente función:

O cogemos el objeto, o lo descartamos. Cada vez tenemos menos objetos disponibles para completar la mochila y cada vez la mochila soporta un peso máximo menor.

$mochila(i, j)$ = máximo valor que podemos poner en una mochila de peso máximo j considerando los objetos del 1 al i

- ▶ Se cumple el principio de optimalidad de Bellman.

Si decidimos meter un objeto en la mochila, el resto de la mochila tendrá que rellenarse de forma óptima con el resto de los objetos. Y si decidimos descartar el objeto, la mochila también tendrá que rellenarse de forma óptima.

Definición recursiva

- ▶ Casos recursivos:

$$mochila(i, j) = \begin{cases} mochila(i-1, j) \\ \max(mochila(i-1, j), mochila(i-1, j-p_i) + v_i) \end{cases}$$

con $1 \leq i \leq n$ y $1 \leq j \leq M$

Tenemos que rellenarlo con el resto de los objetos.

Si el peso del objeto supera al peso actual de la mochila

si $p_i > j$
si $p_i \leq j$

No introducimos nuestro objeto y probamos con los demás

Si lo introducimos.

Nos quedamos con la que devuelva un valor mayor.

Peso de j menos el peso del objeto introducido

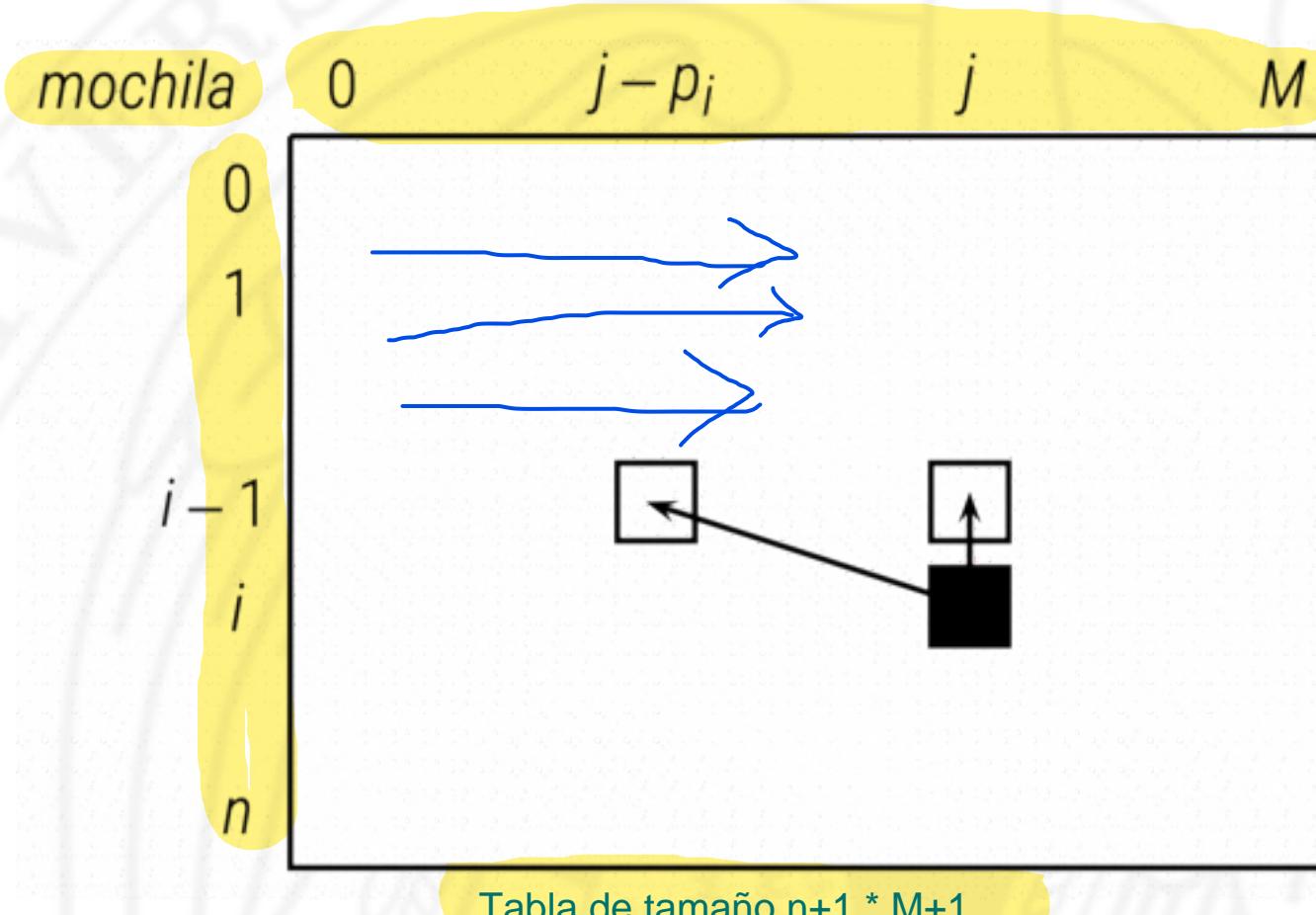
- ▶ Casos básicos:

$$\begin{aligned} mochila(0, j) &= 0 & 0 \leq j \leq M \\ mochila(i, 0) &= 0 & 0 \leq i \leq n \end{aligned}$$

Si no tenemos objetos disponibles, el valor de lo que hay en la mochila será 0.
Si no nos queda peso en la mochila disponible, No podemos rellenarla con ningún objeto, por tanto el valor será 0.

- ▶ Llamada inicial: $mochila(n, M)$

Tabla

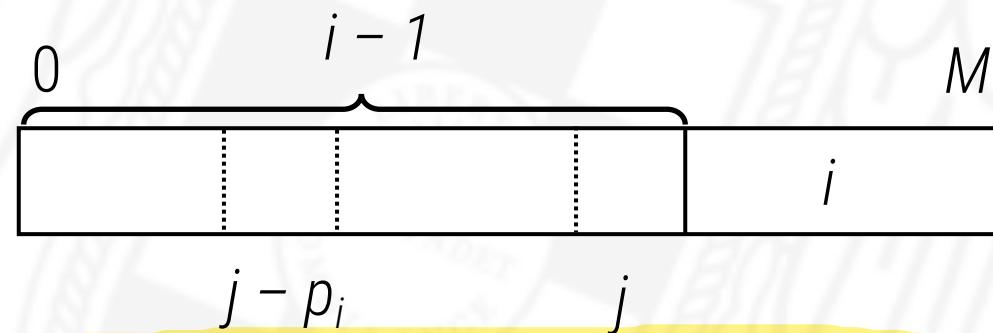
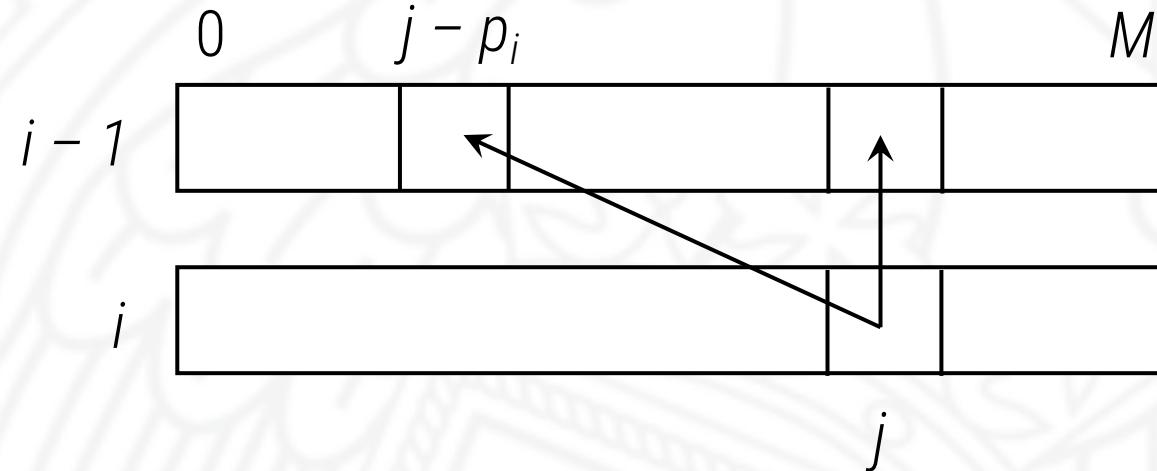


Casos básicos corresponden con la primera fila y la primera columna.

Recorremos la tabla de arriba a abajo y de izquierda a derecha.

Mejorar espacio adicional

- ▶ ¿Podemos reducir la memoria necesaria?



Podemos reducir la matriz a un vector de $M+1$ posiciones.

Reconstrucción de la solución óptima

$$mochila(i,j) = \max(\underbrace{mochila(i-1,j)}, \underbrace{mochila(i-1,j-p_i) + v_i})$$

no cogemos
el objeto i

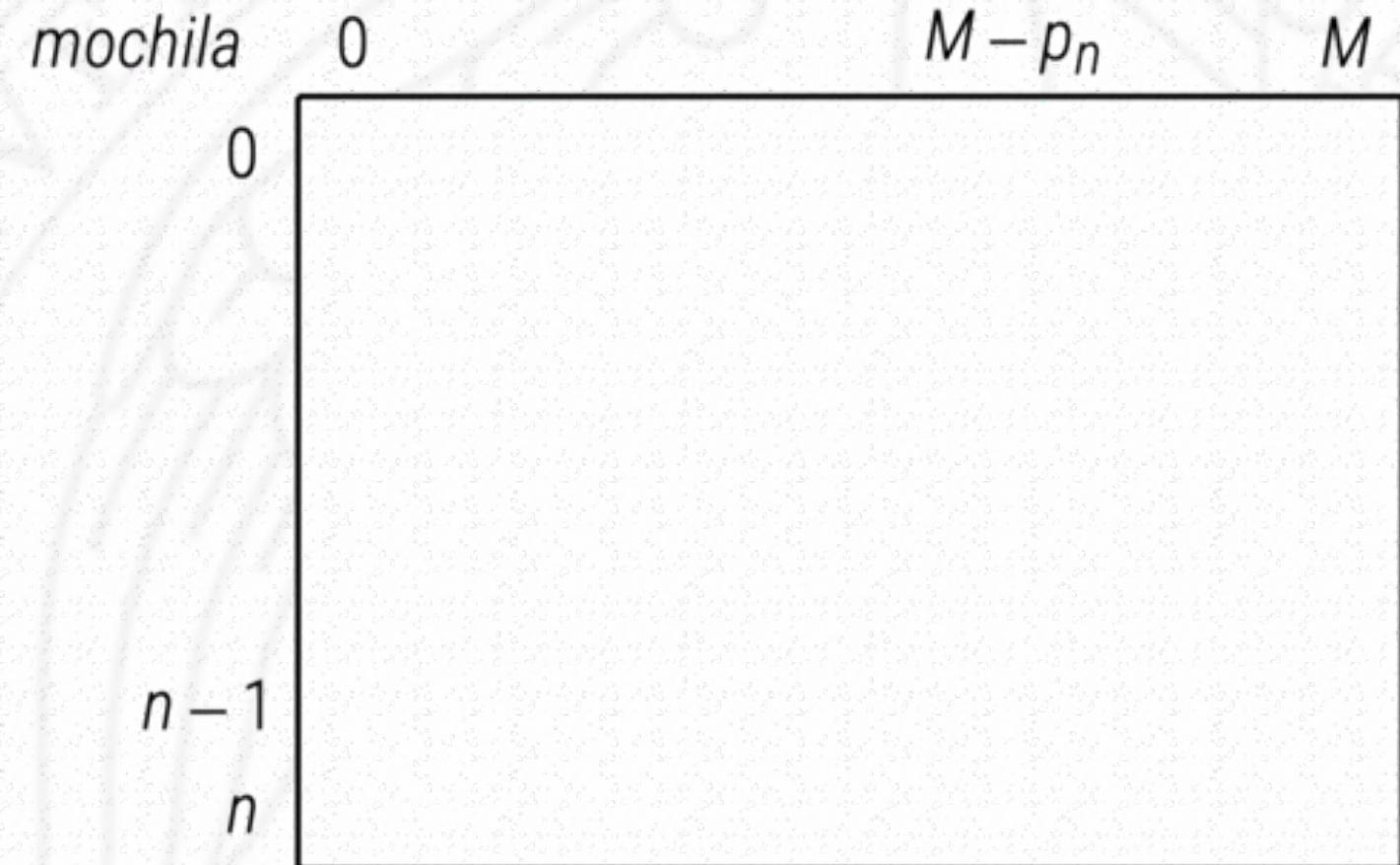
sí cogemos
el objeto i

Si solo tenemos que devolver el valor de la mochila entonces si que podemos reconstruir la matriz y hacer un vector, pero en caso de que tengamos que reconstruir la solución con los objetos que hemos escogido, necesitamos la tabla COMPLETA.

COMO QUEREMOS RECONSTRUIR LA SOLUCIÓN FINAL, NECESITARÍAMOS LA TABLA COMPLETA RELLENA, Y POR TANTO NO PODRÍAMOS MEJORAR EL COSTE EN ESPACIO DE LA FUNCIÓN.

PODRÍAMOS USAR PROGRAMACIÓN DINÁMICA DESCENDENTE Y ASCENDENTE.

Reconstrucción de la solución óptima



Implementación

Utilizamos el enfoque de programación dinámica descendente, con un enfoque recursivo.

```
struct Objeto { int peso; double valor; };

double mochila_rec(vector<Objeto> const& obj, int i, int j,
                    Matriz<double> & mochila) {
    if (mochila[i][j] != -1) // subproblema ya resuelto
        return mochila[i][j];
    if (i == 0 || j == 0) mochila[i][j] = 0;
    else if (obj[i-1].peso > j)
        mochila[i][j] = mochila_rec(obj, i-1, j, mochila);
    else
        mochila[i][j] = max(mochila_rec(obj, i-1, j, mochila),
                            mochila_rec(obj, i-1, j - obj[i-1].peso, mochila)
                            + obj[i-1].valor);
    return mochila[i][j];
}
```

Indicamos con -1 si hemos cogido o no el objeto correspondiente.

v_i

P_i

Implementación

```
double mochila(vector<Objeto> const& objetos, int M, vector<bool> & sol) {  
    int n = objetos.size();  
    Matriz<double> mochila(n+1, M+1, -1);  
    double valor = mochila_rec(objetos, n, M, mochila); O(N*M) en tiempo y espacio.  
    // cálculo de los objetos  
    int i = n, j = M;  
    sol = vector<bool>(n, false);  
    while (i > 0 && j > 0) { Mientras tengamos objetos y mochila  
        if (mochila[i][j] != mochila[i-1][j]) {  
            sol[i-1] = true; j = j - objetos[i-1].peso;  
        }  
        --i;  
    }  
    return valor;  
}
```

Reconstruye la solución óptima.

O(N)