2.2.4 El problema de la mochila entera

Considere una mochila capaz de almacenar un peso máximo *M*, y *n* elementos con pesos $\{p_1, p_2, p_3, \dots, p_n\}$ y los beneficios $\{b_1, b_2, b_3, \dots, b_n\}$. Tanto *M* como los p_i serán enteros, lo que permitirá utilizarlos como índices en una tabla. Se trata de encontrar qué combinación de elementos, representada mediante la tupla $x = (x_1, x_2, x_3, \dots, x_n)\ con\ x_i \in \{0,1\}$, que maximiza los beneficios empleando la ecuación:

$$f(x) = \sum_{i=0}^{n-1} b_i x_i \qquad (1)$$

Sin sobrepasar el peso máximo *M*:

$$\sum_{i=0}^{n-1} b_i x_i < M \qquad (2)$$

Teniendo una versión recursiva y partiendo de un prefijo *x* de longitud *k* y un peso disponible *r*, devolverá una solución óptima junto con el máximo valor alcanzable de la función objetivo (1). Para ello, a partir del prefijo *x*, se exploran las dos opciones posibles:

1. Añadir el objeto *k* (comprobando la condición (2)) o
2. No añadirlo.

Versión recursiva discreta

Se planteó que los pesos p_i son enteros, así que el algoritmo será como este:

```
// Solucion y maximo beneficio alcanzable
// con los objetos 0,..,i-1, teniendo un peso m
// disponible en la mochila
def insertarObjetosEnMochila (i, M, p, b):
    // base de la recurrencia: 0 objetos
    if i ==0:
        return [], 0
    // opcion 1: el objeto i -1 NO se introduce
    solNO, maxBNO = mochilaDiscretaRec (i-1, M, p, b)
    // opcion 2: el objeto i -1 SI se introduce
    if p[i-1] < M:
        solSI, maxBSI = mochilaDiscretaRec(i-1, M-p[i-1], p, b)
        if b[i-1] + maxBSI > maxBNO:
            return [1] + solSI, b[i-1] + maxBSI
    return [0] + solNO, maxBNO
```

Optimalidad del algoritmo

Sea $x = (x_1, x_2, x_3, \dots, x_n)$ la solución óptima del problema (1, *n*, *M*) con $x_i \in \{0,1\}\ \forall_i$ y la subsecuencia $x' = (x_i, \dots, x_j)$ de *x* es también óptima para el subproblema (1, *j*, *M-K*), donde K es el peso aportado a la solución óptima por los objetos 1, 2, ….., *i*-1 y *j*+1,….,*n*:

$$K = \sum_{k=1}^{i-1} x_k p_k + \sum_{k=j+1}^{n} x_k p_k \qquad (3)$$

Si x' no es solución óptima para el subproblema (1, *j*, *M*-*K*) entonces existirá otra subsecuencia diferente $y' = (y_i, \dots, y_j)$ tal que

$$\sum_{k=1}^{j} y_k b_k > \sum_{k=1}^{i} x_k p_k \qquad (4)$$

Con la restricción

$$\sum_{k=1}^{j} y_k p_k \leq M - K$$

Si en esta ecuación se sustituye el valor de K por la ecuación 3 queda lo siguiente

$$\sum_{k=1}^{j} y_k p_k + \sum_{k=1}^{i-1} x_k p_k + \sum_{k=j+1}^{n} x_k p_k \leq M$$

Y de la ecuación 4

$$\sum_{k=1}^{i-1} x_k b_k + \sum_{k=i}^{j} y_k b_k + \sum_{k=j+1}^{n} x_k b_k > \sum_{k=1}^{n} x_k b_k$$

Esto implica que la secuencia $y = (x_1, \dots, x_{i-1}, y_i, \dots, y_j, x_{j+1}, \dots, x_n)$ es la solución óptima al problema (1, n, M). Así que, cualquier subsecuencia x' de la solución óptima x debe ser asimismo solución óptima del subproblema asociado (principio de optimalidad).

Versión de programación dinámica

El beneficio acumulado de la mejor solución para cada caso (*i*, *m*) se almacena en una matriz *t*, de tamaño (*n*+1) x (*m*+1).

```
// Devuelve el maximo beneficio alcanzable con los objetos
// de pesos p y beneficios b, teniendo un peso M disponible
def insertarObjetosEnMochilaPD1 (p, b, M):
    n = len (p)
    t = [[0 for m in range (M+1) ] for i in range (n+1)]
    for i in range (1, n+1):
        for m in range (1, M+1):
            // si se puede introducir el objeto i y el beneficio
            // es mayor que no haciendolo, se introduce
            if p[i-1]<=m and (b[i-1] + t[i-1][m-p[i-1]])>t[i-1][ m]:
                t[i][m] = b[i-1] + t[i-1][m-p[i-1]]
            // en caso contrario, no se introduce
            else:
                t[i][m] = t[i-1][ m]
    return t[n][M]
```

Ejemplo

$$p = [2, 5, 3, 6, 1]$$
$$b = [28, 33, 5, 12, 20]$$

La tabla 2.2.4.1 muestra el desarrollo del algoritmo

Tabla 2.2.4.1 Ejecución del algoritmo

i \ m	0	1	2	3	4	5	6	7	8	9	10
0	0	0	**0**	0	0	0	0	0	0	0	0
1	0	0	28	28	**28**	28	28	28	28	28	28
2	0	0	28	28	28	33	33	61	61	**61**	61
3	0	0	28	28	28	33	33	61	61	**61**	66
4	0	0	28	28	28	33	33	61	61	**61**	66
5	0	20	28	48	48	48	53	61	81	81	**81**

Ahora es turno del código del algoritmo en Java, se ha programado una clase Mochila que es la que implementa el algoritmo y una clase ObjetoDeMochila que es lo que queda almacenado en una lista ligada en la clase Mochila.

Código 2.2.4.1 Implementación del algoritmo de la mochila discreta

```
/**
 * @author sdelaot
 */
public class ObjetoDeMochila {
    private int peso;
    private int beneficio;
    public ObjetoDeMochila(int peso, int beneficio) {
        this.peso = peso;
        this.beneficio = beneficio;
    }
    public int getPeso() {
        return peso;
    }
    public int getBeneficio() {
        return beneficio;
    }
    @Override
    public String toString() {
        return "ObjetoDeMochila{" + "peso=" + peso + ", beneficio=" + beneficio + '}';
    }
}
import java.util.LinkedList;
/**
 * @author sdelaot
 */
public class Mochila {
    private LinkedList<ObjetoDeMochila> objetos;
```

```
public Mochila() {
   objetos = new LinkedList<>();
}
/**
 * Solucion y maximo beneficio alcanzable con los objetos 0,..,i-1,
 * teniendo un peso m disponible en la mochila (estructurado), genera un
 * ObjetoDeMochila y se introduce el que si me brinda mejor beneficio
 * (recursiva)
 *
 * @param i numero total de objetos
 * @param M peso maximo de la mochila
 * @param p arreglo de pesos de los objetos
 * @param b arreglo de beneficios de cada objeto
 * @return devuelve el peso maximo de que no se introduce el objeto
 */
public int insertarObjetosEnMochila( int i, int M, int [] p, int [] b ) {
   int maxBNO;
   int maxBSI;
   // base de la recurrencia: 0 objetos
   if( i==0 ) {
      return 0;
      }
   // opcion 1: el objeto i -1 NO se introduce
   maxBNO = insertarObjetosEnMochila(i-1, M, p, b);
   // opcion 2: el objeto i -1 SI se introduce
   if( p[i-1]<M ) {
      maxBSI = insertarObjetosEnMochila(i-1, M-p[i-1], p, b);
      if( b[i-1]+maxBSI>maxBNO ){
         addObjeto(new ObjetoDeMochila(p[i-1],b[i-1]));
         return b[i-1] + maxBSI;
         }
      }
   return maxBNO;
}
/**
 * Solucion y maximo beneficio alcanzable con los objetos 0,..,i-1,
 * teniendo un peso m disponible en la mochila (orientado a objetos),
 * toma el ObjetoDeMochila y se introduce el que si me brinda mejor
 * beneficio (recursiva)
 *
 * @param i numero total de objetos
 * @param M peso maximo que soporta la mochila
 * @param objetos los posibles objetos que se pueden transportar en la
 * mochila
 * @return devuelve el peso maximo de que no se introduce el objeto
 */
public int insertarObjetosEnMochila( int i, int M,
      ObjetoDeMochila [] objetos ) {
   int maxBNO;
   int maxBSI;
```

```
        if( i==0 ) {
            return 0;
            }
        maxBNO = insertarObjetosEnMochila(i-1, M, objetos);
        if( objetos[i-1].getPeso()<M ) {
            maxBSI = insertarObjetosEnMochila(i-1, M-objetos[i-1].getPeso(),
                    objetos);
            if( objetos[i-1].getBeneficio()+maxBSI>maxBNO ){
                addObjeto(objetos[i-1]);
                return objetos[i-1].getBeneficio() + maxBSI;
                }
            }
        return maxBNO;
    }
    /**
     * Devuelve los objetos contenidos en la mochila
     *
     * @return devuelve la lista de objetos dentro de la mochila
     */
    public LinkedList<ObjetoDeMochila> getObjetos() {
        return objetos;
    }
}
/**
 * @author sdelaot
 */
public class ProbadorDeMochila {
    public static void main(String[] args) {
        int [] p = {2,5,3,6,1};
        int [] b = {28,33,5,12,20};
        ObjetoDeMochila [] objetos = {
            new ObjetoDeMochila(p[0],b[0]),
            new ObjetoDeMochila(p[1],b[1]),
            new ObjetoDeMochila(p[2],b[2]),
            new ObjetoDeMochila(p[3],b[3]),
            new ObjetoDeMochila(p[4],b[4]),
            };
        int M = 10;
        int i = p.length;
        Mochila mochila = new Mochila();
        System.out.println( mochila.insertarObjetosEnMochila(i, M, p, b) );
        System.out.println(mochila.getObjetos());
        Mochila mochilaObj = new Mochila();
        System.out.println(
                mochilaObj.insertarObjetosEnMochila(i, M, objetos) );
        System.out.println(mochilaObj.getObjetos());
    }
}
```

Una segunda versión del algoritmo

El procedimiento sólo requiere dos filas de la matriz, la actual y la anterior, puesto que la optimización de la fila *i* sólo depende de la fila i-1.

```
// Devuelve el maximo beneficio alcanzable con los objetos
// de pesos p y beneficios b, teniendo un peso M disponible
def insertarObjetosEnMochilaPD2(p, b, M):
        n= len (p)
        ant = [0 for m in range (M+1)]
        act = [0 for m in range (M+1)]
        for i in range (1, n+1):
                for m in range (1, M+1):
                        // si se puede introducir el objeto i y el beneficio
                        // es mayor que no hacerlo, lo introducimos
                        if p[i-1] <=m and (b[i-1] + ant[m-p[i-1]]) > ant[m]:
                                act[m] = b[i-1] + ant[m-p[i-1]]
                        // en caso contrario, no se introduce
                        else:
                                act[m] = ant[m]
                ant = act[:]
        return act [M]
```

Una tercera versión del algoritmo

Recuperación de la solución, donde una matriz *d* guarda las decisiones tomadas; se retrocede fila a fila desde el elemento d[n][M] hasta la fila 0

```
def insertarObjetosEnMochilaPD3(p, b, M):
        n= len (p)
        ant = [0 for m in range (M +1)]
        act = [0 for m in range (M +1)]
        d = [[0 for m in range (M+1)] for i in range (n+1)]
        for i in range (1, n+1):
                for m in range (1, M+1):
                        if p[i-1] <= m and b[i-1] + ant [m-p[i-1]] > ant [m]:
                                act [m] = b[i-1] + ant [m-p[i-1]]
                                d[i][m] = 1
                        else:
                                act [m] = ant[m]
                                d[i][m] = 0
                        ant = act[:]
                x =[]
                m=M
                for i in range (n , 0, -1):
                        x. insert (0, d[i][m])
                        if d[i][m ]==1:
                                m = m-p[i-1]
        return x, act[M]
```

Una solución más del problema

Recuperación de la solución: los vectores *ant* y *act* almacenan el beneficio máximo alcanzable junto con la lista de decisiones correspondiente. Al terminar, el elemento *act*[*M*] contiene el beneficio máximo alcanzable y la solución.

```
// Devuelve el maximo beneficio alcanzable con los objetos
// de pesos p y beneficios b, teniendo un peso M disponible,
// asi como la solucion que lo produce
def insertarObjetosEnMochilaPD4(p, b, M):
        n = len (p)
        ant = [[0, []] for m in range (M +1)]
        for i in range (1, n+1):
                act = [[0, [0 for j in range (i)]]]
                for m in range (1, M+1):
                        if p[i-1] <= m and b[i-1] + ant [m-p[i-1]][0] > ant [m][0]:
                                act.append ([b[i-1] + ant [m-p[i -1]][0] , ant [m-p[i-1]][1][:] + [1]])
                        else:
                                act.append ([ ant[m][0], ant[m][1][:] + [0]])
                ant = act
        return act[M]
```

TAREA

Como pudieron notar en el código hay dos métodos sobrecargados para hacer el trabajo de maximizar el peso que carga la mochila del primer algoritmo presentado con los objetos que brindan el mayor beneficio, el primero recibe los datos en forma primitiva y almacena los objetos en la lista ligada esta semiestructurado, el segundo recibe los posibles objetos que puede almacenar (ObjetoDeMochila [] objetos) y de aquí se extraen los elegidos (todo orientado a objetos, excepto por M).

La tarea será programar todos los algoritmos (desde **una segunda versión del algoritmo** hasta **una solución más del problema**) con métodos sobrecargados haciéndolo primero con los datos primitivos y después con objetos de tipo ObjetoDeMochila. Algunos tienen un detalle en la solución que devuelven maximizada, hay que arreglar el problema y si no lo puede arreglar, reporten todo solución o no y sus conclusiones.

Entregar reporte (pdf) y código fuente (Java o C++)

NOTA: Un servidor programó todos los algoritmos y funcionan correctamente (no los incluí en el código, esa es su tarea), solo les puedo decir que este seudocódigo se acerca un poco al lenguaje **Python**, esta es una pista por si necesitan ayuda, tengan cuidado como crean los arreglos de más de una dimensión.