Memoria Práctica 3: ADA

1. Descripción del problema

El problema de la mochila, comúnmente abreviado por KP (del inglés Knapsack problem) es un problema de optimización combinatoria, es decir, que busca la mejor solución entre un conjunto finito de posibles soluciones a un problema. Modela una situación análoga al llenar una mochila, incapaz de soportar más de un peso determinado, con todo o parte de un conjunto de n ítems, cada uno con un peso y valor específicos. Cada uno de estos ítems, estará disponible un número qi de veces. Los objetos colocados en la mochila deben maximizar el valor total sin exceder el peso máximo.

Supongamos que tenemos n distintos tipos de ítems, que van del 1 al n. De cada tipo de ítem se tienen qi ítems disponibles, donde qi es un entero positivo que cumple $1 \leq qi < \infty$. Cada tipo de ítem i tiene un beneficio asociado dado por vi y un peso (o volumen) wi. Usualmente se asume que el beneficio y el peso no son negativos. Por otro lado se tiene una mochila, donde se pueden introducir los ítems, que soporta un peso máximo (o volumen máximo) W > 0. El problema consiste en meter en la mochila ítems de tal forma que se maximice el valor de los ítems que contiene y siempre que no se supere el peso (o volumen) máximo que puede soportar la misma. La solución al problema vendrá dado por la secuencia de variables x1, x2, ..., xn donde el valor entero de xi indica cuantas copias se meterán en la mochila del tipo de ítem i. Para el ejemplo de la figura 1, W=15, si suponemos que hay dos unidades del ítem 1, una sola unidad del ítem 2 y dos unidades del ítem 3, es decir (q1, q2, q3) = (2,1,2); con un vector de pesos (w1, w2, w3) = (9,6,5); y un vector de valores asociados de (v1, v2, v3) = (38,40,24), la solución óptima consiste en meter en la mochila una unidad del ítem 1 y otra unidad del ítem 2; es decir el resultado puede expresarse como (x1, x2, x3) = (1,1,0).

Sin embargo, si modificamos algunos de estos valores la solución cambia. Por ejemplo, si hay dos ítems disponibles de cada tipo: (q1, q2, q3) = (2,2,2), manteniendo los demás parámetros, la solución sería (x1, x2, x3) = (0,2,0). Si además de haber dos ítems de cada tipo, el tamaño de la mochila crece W=16, el resultado ahora sería ahora (x1, x2, x3) = (0,1,2).

2. Código implementado

-Fuerza bruta:

```
public class MochilaFB extends Mochila {


	public SolucionMochila resolver(ProblemaMochila pm) {
		int size = pm.size();
		int [] solParcial = new int [pm.size()];
		int [] sol = new int [pm.size()];
		int solPeso = 0;
		int solValor = 0;

		boolean combinaciones = true;
```

```
            while(combinaciones) {
                  boolean siguiente= true;
                  for (int i=0; i<size && siguiente; i++) {
                        int cantidad = solParcial[i]+1;

                        if(cantidad >
pm.getItems().get(i).unidades) {
                              cantidad =0;
                              combinaciones = (i!=size-1);
                        }else {
                              siguiente=false;
                        }
                        solParcial[i]=cantidad;
                  }
                  int valor = pm.sumaValores(solParcial);
                  int peso = pm.sumaPesos(solParcial);

                  if(valor>solValor && peso<=pm.pesoMaximo) {
                        sol= Arrays.copyOf(solParcial,
solParcial.length);
                        solPeso= peso;
                        solValor= valor;
                  }
            }
            return new SolucionMochila(sol, solPeso, solValor);
      }

}
```

-Programación dinámica:

```
import java.util.ArrayList;

public class MochilaPD extends Mochila {

      public SolucionMochila resolver(ProblemaMochila pm) {
            SolucionMochila sm=null;
            int filas = pm.size()+1;
            int col = pm.getPesoMaximo()+1;
            int [][] F = new int [filas][col];
            for (int i=0; i<filas; i++) {
                  F[i][0]=0; //Casos base
            }
            for (int j=0; j<col; j++) {
                  F[0][j]=0; //Casos base
            }
            for (int i=1; i<filas; i++) {
                  for(int j=0; j<col; j++) {
                        int peso = pm.getPeso(i-1);
                        if(j-peso<0) {
```

```
                              F[i][j]=F[i-1][j];
                        }else {
                              F[i][j]= Math.max(F[i-1][j], F[i-
1][j-peso]+ pm.getValor(i-1));
                        }
                  }
            }
            int res = F[filas-1][col-1];
            int valores = res;
            int w= pm.getPesoMaximo();
            ArrayList<Integer> itemsSol = new ArrayList<>();
            ArrayList<Item> items = pm.getItems();

            for(int i=filas-1; i>0 && res>0; i--) {
                  if(res!=F[i-1][w]) {
                        itemsSol.add(items.get(i-1).index);
                        res-=items.get(i-1).valor;
                        w-=items.get(i-1).peso;
                  }
            }

            ArrayList<Integer> v= new ArrayList<>();
            for (int i=0; i<items.size(); i++) {
                  v.add(0);
            }

            for(Integer val: itemsSol) {
                  v.set(val, 1);
            }

            sm= new SolucionMochila(v, pm.getPesoMaximo()-w,
valores);
            return sm;
      }
}
```

-Algoritmos voraces:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
public class MochilaAV extends Mochila {

      static class ValorItem implements Comparable<ValorItem>{
            int peso, valor, indice, unidades;
            double densidad;

            public ValorItem(int peso, int valor, int indice, int
unidades) {
                  this.peso=peso;
```

```
                this.valor=valor;
                this.indice=indice;
                this.unidades=unidades;
                densidad= (double) this.valor/ (double)
this.peso;
          }
          public int compareTo(ValorItem item2) {
                if(densidad==item2.densidad &&
indice==item2.indice) {
                     return 0;
                }else if(densidad==item2.densidad  && indice>
item2.indice) {
                     return 1;
                }else if(densidad<item2.densidad) {
                     return 1;
                }else {
                     return -1;
                }
          }
     }
     public SolucionMochila resolver(ProblemaMochila pm) {
          int [] pesos= pm.getPesos();
          int [] valores = pm.getValores();
          int [] unidades = pm.getUnidades();
          int w= pm.getPesoMaximo();
          int n= pesos.length;

          ValorItem [] vali = new ValorItem[n];
          int [] sol = new int [n];
          for(int i=0; i<n; i++) {
                sol[i]=0;
          }
          for(int i=0; i<n; i++) {
                vali[i]= new ValorItem(pesos[i], valores[i], i,
unidades[i]);
          }
          Arrays.sort(vali);
          int pesoTotal= 0;
          int valorTotal=0;

          int i=0;
          int pesoMaximo= w;

          while(i<n && pesoTotal<pesoMaximo) {
                if(vali[i].peso<=pesoMaximo-pesoTotal &&
vali[i].unidades>=1) {
                     sol[vali[i].indice]++;
                     pesoTotal+=vali[i].peso;
                     valorTotal+=vali[i].valor;
```

```
                    vali[i].unidades--;
               }else {
                    i++;
               }
          }
          ArrayList<Integer> solFinal =  new ArrayList<>();
          solFinal = ArrayUtils.toArray(sol);
          SolucionMochila sm = new SolucionMochila(solFinal,
pesoTotal,  valorTotal);
          return sm;
     }
}
```

3. Decisiones de diseño

-Fuerza Bruta:

Para resolver este problema mediante un enfoque de fuerza bruta creamos una variable *size* que almacena el número de ítems de nuestro problema y dos arrays de enteros que van a almacenar la solución parcial y la final. Además, también tomamos dos variables solPeso y solValor de enteros que almacenarán en cada momento el peso y el valor de la solución actual. A continuación creamos un bucle que iterará sobre todas las posibles combinaciones, estrá controlado por un boolean combinaciones que comprueba que queden combinaciones. En caso de que el valor de la solución parcial actual sea mayor que la última que habíamos acumulado y que su peso sea menor al máximo se actualiza la solución con la nueva mejor solución. El bucle continua iterando hasta comprobar todas las combinaciones posibles y quedarnos con la mejor. Por último se devuelve una SolucionMochila con la solución obtenida.

-Programación Dinámica:

Enfocando el problema mediante programación dinámica creamos una tabla, con un número de filas igual a el número de ítems más uno y como columnas el peso máximo más de uno. A continuación, establecemos los casos base (primera fila y columna) como 0. Una vez hecho esto la tabla se va rellenando de arriba abajo y de izquierda a derecha con el caso general dependiendo de si j<wi. Una vez hemos rellenado la tabla , la solución se encuentra en la celda [filas-1][col-1].

Por último construimos la solución con el formato que nos dan, en el que cada posición de un array representa el ítem y el contenido de este la cantidad utilizada.

-Algoritmos Ávidos:

Para implementar el algoritmo ávido necesitamos crear una clase ValorItem que nos permita ordenar todos los ítems de mejor a peor para poder tomar los mejores ítems (los que tienen un mayor valor en función de su densidad) posteriormente. Una vez implementada esta clase ordenamos los elementos en función de su densidad gracias al método sort, y los vamos añadiendo a la solución siempre y cuando quepan. Finalmente

pasamos el array a un ArrayList y devolvemos una MochilaSolucion con los datos obtenidos.

4. Complejidad de los algoritmos

-Fuerza Bruta: la complejidad espacial es E(2n) ya que utiliza dos arrays para almacenar en primer lugar la solución parcial y finalmente la solución definitiva. En cuanto a la complejidad temporal, se ejecuta un bucle while correspondiente a las posibles combinaciones y dentro de este otro bucle for que en el peor de los casos se ejecuta un número de veces que coincide con los ítems. A su vez, dentro del bucel while se realiza una comparación y dentro del bucle for otra comparación, por tanto, la complejidad es O(combinaciones x size + combinaciones).

-Programación dinámica: la complejidad espacial del algoritmo depende de las dimensiones de la tabla que utilizamos, así como del número de ítems que utilizamos en la solución, por tanto es E(pm.size()+1 x pm.PesoMaximo()+1 + items.size()). Por otro lado la complejidad temporal es O(2*filas + col + filas x col + items.size()) ya que se ejecutan dos bucles for para los casos base, otros dos anidados para los casos generales y por último otros tres para construir la solución.

-Algoritmos voraces: la complejidad espacial es E(pm.getPesos() + pm.getValores() + pm.getUnidades() + pesos.length()) correspondiente a los arrays utilizados en el peor de los casos. En cuanto a la complejidad temporal es O(3n) correspondiente a los tres bucles for que se ejecutan en el peor de los casos.