

A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

11-5-2022

Practica 03

EDA II

Several thin, curved lines in dark blue and light grey originate from the bottom left and sweep upwards and to the right.

Nada Diouane
Hamza El Fallah

Contenido

1.	Objetivo de la practica	2
2.	Implementación	2
3.	Diagrama de clase	2
3.1	Clase ElementBackpack.....	2
a)	Método loadFile	2
b)	Método de backpack o mochila	3
c)	Método objects con matriz A.....	4
d)	Método elementos con array.....	5
e)	Método backpackUnlimitedCapacity	5
f)	Método backpackGreedy	5
3.2	ElementWeightComparator	6
4.	Estudio Teórico.....	7
4.1.	Método de backpack o mochila	7
4.2.	Método backpackUnlimitedCapacity	8
4.3.	Método backpackGreedy	9
5.	Estudio empírico.....	¡Error! Marcador no definido.

1. Objetivo de la practica

- Construir soluciones a un problema utilizando el método algorítmico de programación dinámica (o dynamic programming). Analizar y comparar los algoritmos implementados desde diferentes perspectivas.
- Realizar el análisis de la eficiencia de las soluciones aportadas, y una comparativa tanto desde el punto de vista teórico como práctico.

2. Implementación

En una mochila hay que añadir varios objetos, cada objeto de un tipo indivisible que tiene un peso y un valor.

Hay que especificar la cantidad de objetos que tenemos de cada tipo para tener el valor total máximo de lo que se puede tener en la mochila.

3. Diagrama de clase

- Element
- ElementBackpack
- ElementWeightComparator
- GenerateTestFile
- TestBackpack

3.1 Clase ElementBackpack

a) Método loadFile

En el siguiente método es el responsable de cargar el archivo a partir de la ruta de la base:

- Hace la lectura del primer archivo para conseguir la cantidad, para luego lo asigna al atributo adecuado.
- Hace la lectura del segundo archivo que contiene el peso, hace la creación de un elemento con su nombre y su peso luego establece el valor a 0 para luego añadirlo a la estructura.
- Hace la lectura del tercer archivo, se crea un índice y hace la actualización del valor del elemento que se encuentra en dicha posición

```

45 private void loadFile(String nameFileBase) {
46     File f = new File(ruta + nameFileBase + "_c.txt");
47     String line = "";
48     this.elements = new ArrayList<Element>();
49     Scanner sc = null;
50     int i = 1;
51     try {
52         sc = new Scanner(f);
53         this.capacity = Integer.parseInt(sc.nextLine().trim());
54         sc.close();
55     } catch (FileNotFoundException e1) {
56         e1.printStackTrace();
57     }
58     f = new File(ruta + nameFileBase + "_w.txt");
59     try {
60         sc = new Scanner(f);
61         // Crear objetos sólo con los pesos y agregarlos a la lista
62         while (sc.hasNextLine()) {
63             line = sc.nextLine().trim();
64             Element e = new Element("E0" + i, Double.parseDouble(line), 0);
65             this.elements.add(e);
66             i++;
67         }
68         sc.close();
69     } catch (FileNotFoundException e1) {
70         e1.printStackTrace();
71     }
72     f = new File(ruta + nameFileBase + "_p.txt");
73     try {
74         i = 0;
75         sc = new Scanner(f);
76         // Obtener cada objeto de nuestra lista y agregarle el valor
77         while (sc.hasNextLine()) {
78             line = sc.nextLine().trim();
79             this.elements.get(i).setGanancia(Double.parseDouble(line));
80             i++;
81         }
82         sc.close();
83     } catch (FileNotFoundException e) {
84         e.printStackTrace();
85     }
86 }

```

b) Método de backpack o mochila

Hace la ordenación de menor a mayor peso que se ha fijado en la clase *ElementWeightComparator*.

Hacemos la creación de una matriz, donde $A[i,j]$ nos permite saber el valor máximo de la mochila (*backpack*) utilizando los i primeros elementos y j el peso máximo.

Así, la solución del problema se encuentra en la posición [row, column], cogiendo todos los elementos y el peso máximo.

Hay que recorrer dicha matriz A , asegurando que cada posición se basa en los valores de las posiciones anteriores.

Comprobar que si la posición anterior es óptima y la posterior es óptima también.

Hay que asegurar que la row no se modifica y queda con los ceros iniciales si $i = 0$.

- Si $i=1$:
 - Si $j < p[i]$ no se puede meter el primer elemento que tiene el peso mayor que el peso total j y así la posición va a quedar en 0, de otra manera, la mochila (*backpack*) no va a tener ningún valor.

- Si $j \geq p[i]$ la mochila va a tener el primer elemento que es menor o igual a j con su valor.
- Si $i > 1$:
 - Si $j < p[i]$ el valor del *backpack* va a ser el que hemos obtenido en la fila $i-1$, porque el elemento no va a entrar en el backpack con el peso total j
 - Si $j \geq p[i]$, lo que deduce $A[i,j] = \text{Max}(A[i-1, j], p[i] + A[i-1, j-p[i]])$, en caso de que dos posibilidades caben, hay que tomar la que maximice el valor:
 - El elemento va a tomar el valor de la fila anterior en caso de que el elemento no cabe dentro.
 - Que dicho elemento cabe con lo que le sumaremos $p[i]$ mas el valor de la columna y fila calculados anteriormente que son el resto de peso que falta por meter es decir $p[i] + A[i-1, j-p[i]]$.

```

111
112● public List<Element> backpack() {
113     this.elements.sort(new ElementWeightComparator()); // Ordenar ascendente por peso
114     int row = this.elements.size() + 1;
115     int column = this.capacity + 1;
116     double[][] A = new double[row][column]; // Matriz
117     for (int i = 1; i <= this.elements.size(); i++) {
118         for (int j = 1; j <= this.capacity; j++) {
119             if (this.elements.get(i - 1).getWeight() <= j) {
120                 A[i][j] = Math.max(A[i - 1][j], this.elements.get(i - 1).getGanancia()
121                     + A[i - 1][j - (int) this.elements.get(i - 1).getWeight()]);
122             } else {
123                 A[i][j] = A[i - 1][j];
124             }
125         }
126     }
127     return objects(A); // Recupera los elementos que formarán parte de la mochila
128 }
129

```

c) Método objects con matriz A

La matriz A está diseñada para que el valor más grande del *backpack* esté en el último elemento de la matriz.

Una forma de iterar a través de una matriz con un bucle for que verifica si el peso es diferente del último elemento

Una vez que comprueba la primera columna de la fila anterior, comprobará la anterior y así sucesivamente.

```

129
130● public List<Element> objects(double[][] A) {
131     List<Element> result = new ArrayList<Element>();
132     this.gananciaFinal = A[A.length - 1][A[0].length - 1]; // Valor máximo de la mochila
133     this.weightFinal = 0;
134     int j = A[0].length - 1; // Columna de partida
135     for (int i = A.length - 1; i > 0; i--) { // En cada iteración se sube de fila
136         if (A[i][j] != A[i - 1][j]) { // Si la celda superior es distinta, lo escogemos...
137             result.add(this.elements.get(i - 1));
138             this.weightFinal += this.elements.get(i - 1).getWeight();
139             j -= this.elements.get(i - 1).getWeight(); // Se desplaza la columna a la izquierda el peso del objeto
140         }
141     }
142     return result;
143 }

```

d) Método elementos con array

Su rol es buscar el objeto que tiene el menor peso, aquel cuyo peso ha sido previamente ordenado de menor a mayor, ubicado en la primera posición.

El siguiente algoritmo itera en un bucle siempre que la capacidad restante sea mayor que el peso mínimo de los objetos. Encuentra los artículos que más valor nos dan en función del espacio que queda en la mochila.

Al localizar el mejor candidato, se añade al *backpack* y se reduce su espacio.

```
183 private List<Element> elementos(double[] array) {
184     this.gananciaFinal = 0;
185     this.weightFinal = 0;
186     List<Element> result = new ArrayList<Element>();
187     int cap = this.capacity; // Capacidad restante
188     double pesoMin = this.elements.get(0).getWeight(); // Peso menor
189     int size = this.elements.size();
190     while (cap >= pesoMin) { // Puede quedar espacio sin ocupar
191         double gananciaMax = 0;
192         int index = -1; // Posición del objeto ganador
193         for (int i = size - 1; i >= 0; i--) { // Para cada objeto...
194             if (cap - this.elements.get(i).getWeight() >= 0) { // Valoraremos solo los objetos que quepan en el
195                 // espacio restante
196                 double aux = array[cap - (int) this.elements.get(i).getWeight()]
197                     + this.elements.get(i).getGanancia();
198                 if (aux > gananciaMax) {
199                     gananciaMax = aux;
200                     index = i;
201                 }
202             }
203         }
204         if (index == -1)
205             break; // Si no se encuentra un candidato factible, se finaliza la búsqueda
206         this.gananciaFinal += this.elements.get(index).getGanancia();
207         this.weightFinal += this.elements.get(index).getWeight();
208         result.add(this.elements.get(index));
209         cap -= this.elements.get(index).getWeight(); // Decrementamos el espacio restante de la mochila
210     }
211     return result;
212 }
```

e) Método backpackUnlimitedCapacity

Creamos un array con tamaño `this.capacity+1`, de esta manera el array determina el mayor valor que se puede obtener con todos los elementos por debajo de la capacidad del *backpack*.

Los elementos se clasifican en orden ascendente por el peso y luego se evalúan como los mejores, si el valor en la matriz está en una posición o si el valor de la celda se mueve con el peso del elemento izquierdo más su valor propio.

```
167
168 public List<Element> backpackUnlimitedCapacity() {
169     this.elements.sort(new ElementWeightComparator()); // Ordenar ascendente por peso
170     int n = this.elements.size();
171     double[] array = new double[this.capacity + 1]; // Array
172     for (int i = 0; i <= this.capacity; i++) {
173         for (int j = 0; j < n; j++) {
174             if (this.elements.get(j).getWeight() <= i) {
175                 array[i] = Math.max(array[i],
176                     array[i - (int) this.elements.get(j).getWeight()] + this.elements.get(j).getGanancia());
177             }
178         }
179     }
180     return this.elementos(array); // ¿Como recuperamos los objetos?
181 }
```

f) Método backpackGreedy

En primero hay que ordenar de forma descendente en función de la relación establecida entre valor y pesos.

Si seleccionamos un objeto entero con el 100% del peso.

Por otro lado, si un objeto está parcialmente seleccionado, se le asigna un valor entre 0 y 1, y cuando ocurra el algoritmo, terminará, porque no hay más espacio.

```
232 public List<Element> backpackGreedy() {
233     this.elements.sort(new ElementWeightComparator()); // Orden descendente por relacion valor-peso
234     this.gananciaFinal = 0;
235     this.weightFinal = 0;
236     List<Element> result = new ArrayList<Element>();
237     for (Element e : this.elements) {
238         if (this.weightFinal + e.getWeight() <= this.capacity) { // Si cabe entero
239             e.setAmount(1);
240             result.add(e);
241             this.weightFinal += e.getWeight();
242             this.gananciaFinal += e.getGanancia();
243             if (this.weightFinal == this.capacity)
244                 break; // Si se alcanza la capacidad máxima
245         } else { // Si no cabe entero
246             e.setAmount((this.capacity - this.weightFinal) / e.getWeight());
247             result.add(e);
248             this.weightFinal += e.getWeight() * e.getAmount();
249             this.gananciaFinal += e.getGanancia() * e.getAmount();
250             break; // Ya no cabe mas
251         }
252     }
253     return result;
254 }
```

3.2 ElementWeightComparator

```
1 package org.eda2.practica3;
2
3 import java.util.Comparator;
4
5 public class ElementWeightComparator implements Comparator<Element>{
6
7     @Override
8     public int compare(Element element1, Element element2) {
9         return Double.compare(element1.getWeight(), element2.getWeight());
10    }
11
12 }
13
```

4. Estudio Teórico

4.1. Método de backpack o mochila

```

114 public List<Element> backpack() {
115     this.elements.sort(new ElementWeightComparator()); // Ordenar ascendente por peso n.logn
116     int row = this.elements.size() + 1; O(1)
117     int column = this.capacity + 1; O(1)
118     double[][] A = new double[row][column]; // Matriz O(1)
119     for (int i = 1; i <= this.elements.size(); i++) {
120         for (int j = 1; j <= this.capacity; j++) {
121             if (this.elements.get(i - 1).getWeight() <= j) {
122                 A[i][j] = Math.max(A[i - 1][j], this.elements.get(i - 1).getGanancia()
123                     + A[i - 1][j - (int) this.elements.get(i - 1).getWeight()]);
124             } else {
125                 A[i][j] = A[i - 1][j];
126             }
127         }
128     }
129     return objects(A); // Recupera los elementos que formarán parte de la mochila O(n)
130 }

```

Nº elementos --> n
Capacidad --> m

$$T(n) = c + \sum_{i=1}^n \left(\sum_{j=1}^{cap} c \right) = c + \sum_{i=1}^n (cap - 1 + 1) = c + \sum_{i=1}^n (c * m) = c + (n - 1 + 1) = c + n * m * c$$

$$\max(c, c + n * m * c) = c * n * m \rightarrow O(n * m)$$

```

131
132 public List<Element> objects(double[][] A) {
133     List<Element> result = new ArrayList<Element>();
134     this.gananciaFinal = A[A.length - 1][A[0].length - 1]; // Valor máximo de la mochila O(1)
135     this.weightFinal = 0; O(1)
136     int j = A[0].length - 1; // Columna de partida O(1)
137     for (int i = A.length - 1; i > 0; i--) { // En cada iteración se sube de fila
138         if (A[i][j] != A[i - 1][j]) { // Si la celda superior es distinta, lo escogemos...
139             result.add(this.elements.get(i - 1));
140             this.weightFinal += this.elements.get(i - 1).getWeight();
141             j -= this.elements.get(i - 1).getWeight(); // Se desplaza la columna a la izquierda el peso del objeto
142         }
143     }
144     return result;
145 }

```

$$m.length - 1 \rightarrow n \quad \sum_{i=1}^n c = c * n \rightarrow O(n)$$

$$\max(n, m, n) = n.m \rightarrow O(n)$$

- Si m es constante, según el análisis del algoritmo, el orden de complejidad será como O(nm)
- Si m > n → el orden de complejidad será como O(nm)

Es decir que el consumo de memoria en los dos es mn que es el tamaño de la matriz A

4.2. Método backpackUnlimitedCapacity

```

170 public List<Element> backpackUnlimitedCapacity() {
171     this.elements.sort(new ElementWeightComparator()); // Ordenar ascendente por peso n.logn
172     int n = this.elements.size();
173     double[] array = new double[this.capacity + 1]; // Arr O(1)
174     for (int i = 0; i <= this.capacity; i++) {
175         for (int j = 0; j < n; j++) {
176             if (this.elements.get(j).getWeight() <= i) {
177                 array[i] = Math.max(array[i],
178                     array[i - (int) this.elements.get(j).getWeight()] + this.elements.get(j).getGanancia());
179             }
180         }
181     }
182     return this.elementos(array); // ¿Como recuperamos los objetos?
183 }

```

Nº elementos --> n
Capacidad --> m

$$\sum_{i=0}^m \left(\sum_{j=0}^n (c) \right) = c * n * m$$

max (nlogn, c.n.m)

* Si m es constante el max es nlogn
* Si m > n el max es c.n.m

```

185 private List<Element> elementos(double[] array) {
186     this.gananciaFinal = 0;
187     this.weightFinal = 0;
188     List<Element> result = new ArrayList<Element>();
189     int cap = this.capacity; // Capacidad restante
190     double pesoMin = this.elements.get(0).getWeight(); // Peso menor
191     int size = this.elements.size();
192     while (cap >= pesoMin) { // Puede quedar espacio sin ocupar
193         double gananciaMax = 0;
194         int index = -1; // Posición del objeto ganador
195         for (int i = size - 1; i >= 0; i--) { // Para cada objeto...
196             if (cap - this.elements.get(i).getWeight() >= 0) { // Valoraremos solo los objetos que quepan en el
197                                                         // espacio restante
198                 double aux = array[cap - (int) this.elements.get(i).getWeight()]
199                     + this.elements.get(i).getGanancia();
200                 if (aux > gananciaMax) {
201                     gananciaMax = aux;
202                     index = i;
203                 }
204             }
205         }
206         if (index == -1)
207             break; // Si no se encuentra un candidato factible, se finaliza la búsqueda
208         this.gananciaFinal += this.elements.get(index).getGanancia();
209         this.weightFinal += this.elements.get(index).getWeight();
210         result.add(this.elements.get(index));
211         cap -= this.elements.get(index).getWeight(); // Decrementamos el espacio restante de la mochila
212     }
213     return result;
214 }

```

while -> O(n)
 for -> O(n) => O(n*m) -> O(n)^2

Después de analizar el estudio teórico, se deduce que el orden de complejidad es O(n^2).

4.3. Método backpackGreedy

```

234 public List<Element> backpackGreedy() {
235     this.elements.sort(new ElementWeightComparator()); // Orden descendente por relacion valor-peso nlogn
236     this.gananciaFinal = 0;
237     this.weightFinal = 0;
238     List<Element> result = new ArrayList<Element>();
239     for (Element e : this.elements) {
240         if (this.weightFinal + e.getWeight() <= this.capacity) { // Si cabe entero
241             e.setAmount(1);
242             result.add(e);
243             this.weightFinal += e.getWeight();
244             this.gananciaFinal += e.getGanancia();
245             if (this.weightFinal == this.capacity)
246                 break; // Si se alcanza la capacidad máxima
247         } else { // Si no cabe entero
248             e.setAmount((this.capacity - this.weightFinal) / e.getWeight());
249             result.add(e);
250             this.weightFinal += e.getWeight() * e.getAmount();
251             this.gananciaFinal += e.getGanancia() * e.getAmount();
252             break; // Ya no cabe mas
253         }
254     }
255     return result;
256 }

```

Diagrama de complejidad:

- $O(1)$ para la línea 240 (dentro del bucle).
- $O(1)$ para la línea 247 (dentro del bucle).
- $O(n)$ para el bucle for (líneas 239-254).

$$\text{Nº elementos} = n \quad T(n) = n \log n + c + \sum_0^n (c) = n \log n + c + c * n$$

$$\max(O(n \log n), O(n)) = O(n \log n)$$

Después de analizar el estudio teórico, se deduce que el orden de complejidad es $O(n \log n)$.