# Examen de Algorítmica y Complejidad (Plan 2014)

21 de diciembre de 2023

**Nº matrícula:**_____ **Nombre:** _____

**Apellidos:** _____

**Problema** (**2.5 puntos**). Llega la Navidad y en casa de Santa Claus están en plena campaña de fabricación y organización del transporte de todos los juguetes que deben repartir. Sugarplum Mary, elfa ayudante de la señora Claus, se ha propuesto optimizar la distribución de los juguetes en los sacos. Cada juguete está empaquetado en una caja que tiene asociado un volumen concreto. Además, cada saco tiene asignado un volumen máximo de capacidad que no debe ser superado en ningún caso. Sugarplum Mary quiere **maximizar la cantidad de cajas** que se meten en un saco, asegurando que el volumen total de las cajas **no supera** el volumen máximo del saco. Para ello pide ayuda a los alumnos de Algorítmica y Complejidad de la ETSISI para que implementen un algoritmo en Java, basado en **programación dinámica**, que dada una lista de volúmenes de cajas, de las que se cuenta con un número ilimitado de cada una ya que son muchos los juguetes a empaquetar, y dado el volumen máximo que acepta un saco, encuentre el número máximo de cajas que podemos incluir en el saco sin superar su volumen máximo.

**Ejemplo**: Dada la siguiente lista de volúmenes de cajas y un volumen máximo de 27 para el saco:

| ***volumenCajas*** | 8 | 6 | 4 | 9 | 7 |
|---|---|---|---|---|---|

La solución sería incluir **6** cajas (5 de volumen 4 y 1 de volumen 7)

El algoritmo a implementar tendrá la siguiente cabecera:

```
int llenarSaco(int[] volumenCajas, int volSaco)
```

donde *volumenCajas* contendrá el volumen de los distintos tipos de cajas, *volSaco* indicará el volumen máximo del saco y el método deberá devolver el número máximo de cajas que se pueden meter en el saco.

a) (**0.25 puntos**) Define la **entrada**, la **salida** y la **semántica** de la función sobre la que estará basado el algoritmo de programación dinámica.

**maxCajas(i,j)**: número máximo de cajas que se pueden meter en el saco usando cajas de tipos 1..i sin sobrepasar el volumen máximo j del saco.

**Entrada**: dos valores enteros: i>=0, j>=0. El valor i=0 indica que no se dispone de ningún tipo de cajas. El valor j=0 indica que el volumen máximo que admite el saco es 0.

**Salida**: valor entero que indica el número máximo de cajas que se pueden meter en el saco

b) (**0.5 punto**) Expresa recursivamente la función anterior.

```
maxCajas(0,j) = 0  (i=0, no se dispone de cajas)
maxCajas(i,0) = 0  (j=0, el volumen máximo del saco es 0)

i>0 y j>0
  si volumenCajas[i-1]>j:
      maxCajas(i,j) = maxCajas(i-1, j)
  en caso contrario:
      maxCajas(i,j) = max {maxCajas(i-1, j) , 1+maxCajas(i, j-volumenCajas[i-1]) }
```

c) (**1.75 puntos**) Basándote en los anteriores apartados implementa el algoritmo en Java siguiendo el esquema de ***Programación Dinámica***, con una **complejidad**[1] **O(*N*·*K*) en tiempo y *O(K)* en memoria** (siendo *N* es el número de los tipos de cajas y *K* el volumen máximo que admite el saco).

```
int llenarSaco(int[] volumenCajas, int volSaco){
  int[][] maxCajas = new int[2][volSaco+1];
  for (int j=0; j<=volSaco; j++)
     maxCajas[0][j]=0;
  for (int i=0; i<=1; i++)
     maxCajas[i][0]=0;
  for (int i=1; i<=volumenCajas.length; i++){
     for (int j=1; j<=volSaco; j++){
        if (volumenCajas[i-1]>j)
            maxCajas[i%2][j] = maxCajas[(i-1)%2][j];
        else
            maxCajas[i%2][j] = Math.max(maxCajas[(i-1)%2][j],
                                        1+maxCajas[i%2][j-volumenCajas[i-1]]);
     }
  }
  return maxCajas[(volumenCajas.length)%2][volSaco];
}
```

[1] No cumplir con los requisitos pedidos conlleva **una puntuación de 0** en el apartado c).