



# PRÁCTICA 3 – Programación Dinámica

---

07/05/2022

---

Antonio Aranda Hernández  
Cristina Pérez Ordoñez  
Manuel Megías Briones

## Visión general

Esta práctica se basa en Programación Dinámica. Este algoritmo descompone el problema de forma recursiva aplicando un razonamiento inductivo y combina las soluciones.

A diferencia de divide y vencerás este algoritmo no es recursivo, sino que es iterativo y resuelve primero los problemas más pequeños guardando las soluciones parciales con el método *bottom-up*. Esto produce varias secuencias y únicamente al final se puede saber cual es la mejor. Por ello, es necesario la memorización para su reutilización.

## Objetivos

1. Estudio teórico
2. Diagrama de clases
3. ANEXO

## Hitos

Antes de todo recordar que en este caso en Double.MINValue en java devuelve un valor negativo, por lo tanto no nos valdría. Pero nosotros los tendremos en cuenta como el valor positivo de un doble que es 0 con 55 decimales y un 1 , que sería el mínimo valor positivo que podría obtener el doble.

### I. Estudio teórico

A la hora de resolver este problema basándonos en programación dinámica hacemos uso de una matriz cuya fila y columna 0 tendrán valor 0 (dado que representan el id del objeto y el peso de la mochila). Cada objeto tiene asociado un beneficio el cual se va a introducir en la matriz dependiendo de la capacidad de la mochila y del peso del objeto.

#### APARTADO A:

apartadoA()->O(n)

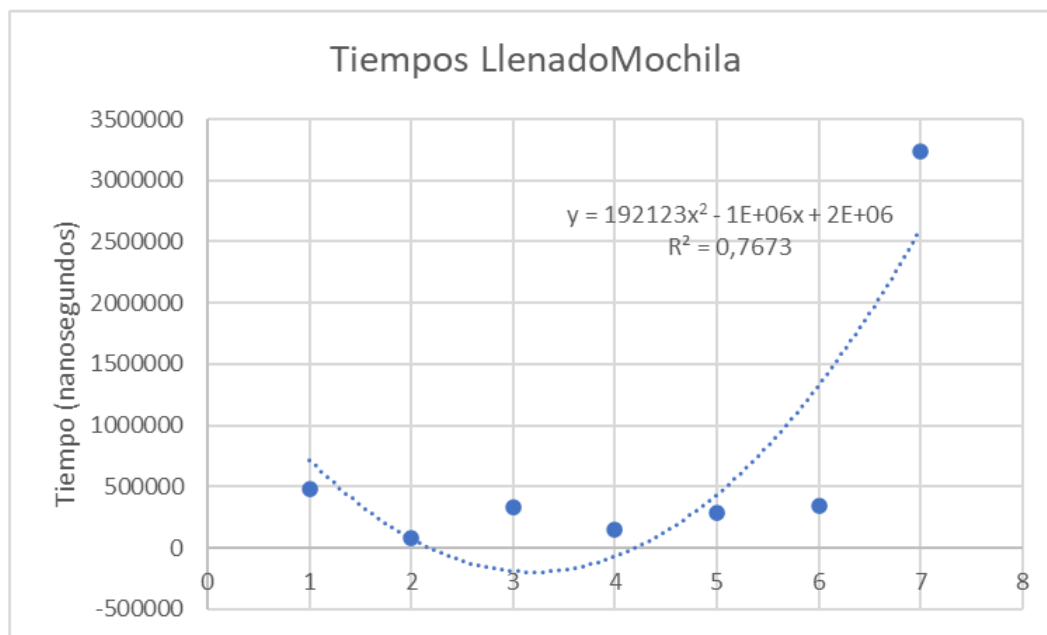
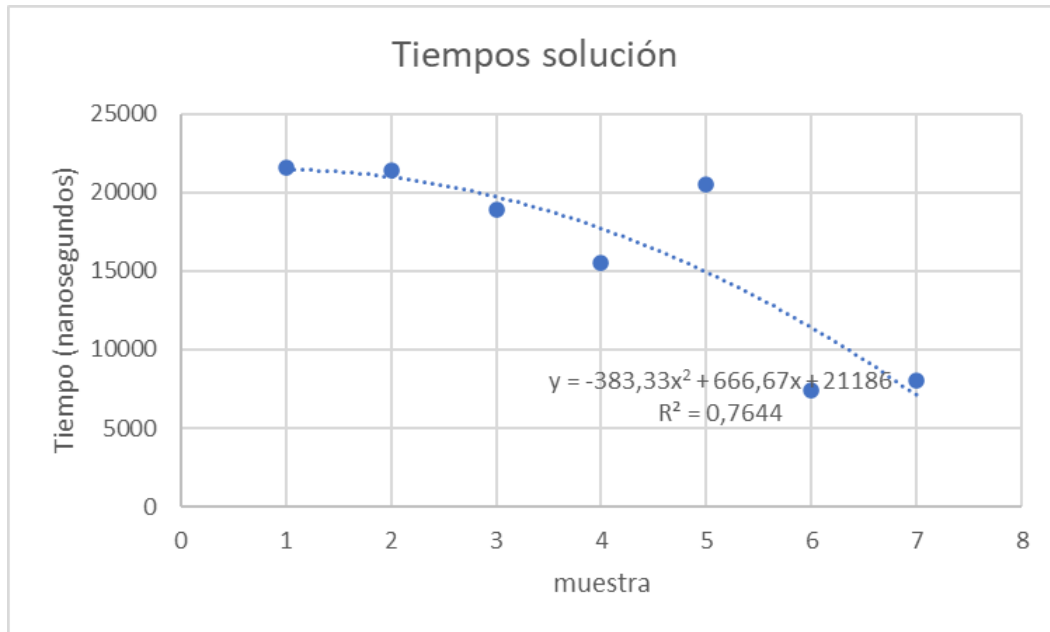
ProblemaMochila\_Llenado1()->O(n^2)

ToString->O(n^2)

Solucion1()->O(n)

lectura\_Archivos1()->O(n logn)

Las gráficas obtenidas nos muestran los siguientes órdenes:



Esto puede deberse a los procesos ya que son tiempos tan pequeños que cualquier proceso en ejecución del ordenador influye en ellos directamente.

### **APARTADO C:**

apartadoC()-> $O(n^2)$

ProblemaMochila\_Llenado2()-> $O(n^2)$

ToString-> $O(n^2)$

Solucion2()-> $O(n)$

lectura\_Archivos2()-> $O(n \log n)$

#### **APARTADO D:**

Mochila\_voraz()-> $O(n^2)$

Solucion1()-> $O(n)$

lectura\_Archivos2()-> $O(n \log n)$

#### **A. Caso 1**

```
ArrayList<Objeto> objetos;
int maximo_peso;
int matriz_resultado[][];
/*
```

Hemos usado un arraylist de objetos, ya que la ordenación es mediante el sort(), que utiliza el mergesort, a la hora de ordenar la lista. Y un array de arrays, ya que es la forma más rápida si los datos se mantienen de forma ordenada, es decir, nunca se va a recorrer completamente.

```
public void ProblemaMochila_Llenado1() { //metodo iterativo
    for(int i = 1; i < this.objetos.size()+1; i++) { //recorremos los elementos columnas
        Objeto aux = objetos.get(i-1);
        for(int w=1; w < this.maximo_peso+1; w++) { // filas hasta 166 en este caso
```

En el llenado de la mochila, recorreremos con dos for, la matriz para el llenado por filas de la matriz. Hemos usado esta manera ya que vamos a ir calculando los beneficios de la mochila para un objeto con todos los posibles pesos.

```
        if(aux.getPeso() <= w) { //si el peso del objeto es menor que el peso que el peso maximo de esa columna
            if((aux.getBeneficio()+matriz_resultado[i-1][(w-aux.getPeso()) < 0 ? 0 : w-aux.getPeso()]) > matriz_resultado[i-1][w]) {
                this.matriz_resultado[i][w] = aux.getBeneficio()+matriz_resultado[i-1][(w-aux.getPeso()) < 0 ? 0 : w-aux.getPeso()];
            } else {
                this.matriz_resultado[i][w] = this.matriz_resultado[i-1][w]; //si no le ponemos la anterior
            } //else interno
        } else { //caso en el que aux.getpeso() > w
            this.matriz_resultado[i][w] = this.matriz_resultado[i-1][w]; //si no le ponemos la anterior
        }
    }
}
```

En el primer if es el caso de que el peso del objeto sea menor que el peso máximo de la columna, que comprobamos el siguiente if de que el beneficio de meterlo más el beneficio que ya tiene es mejor, si no le dejamos el anterior.

La búsqueda de la solución:

```
public ArrayList<Integer> Solucion1() {
    ArrayList<Integer> sol = new ArrayList<Integer>(); //array
    int i = this.objetos.size();
    int w = this.maximo_peso;
    while(i>0 && w>0) { //mientras que i y w no sean 0, es decir
        if(matriz_resultado[i][w] != matriz_resultado[i-1][w])
            sol.add(i);
            w = w - objetos.get(i-1).getPeso(); //el peso del
            i--;
        }else {
            i--;
        }
    }
    return sol;
}
```

Una vez que tenemos el beneficio máximo, nos posicionamos en la última fila y última columna de la matriz, y buscamos el último beneficio que no cambia, es decir, el beneficio del objeto que ha sido introducido. Restamos ese peso al peso que tenemos y así encontramos el siguiente peso. Es decir, si tenemos 80 pesos y le quitamos 40 que es el peso del objeto 4. sabemos que en el peso 40 estará la continuación de la solución y buscaremos en la columna 40 el último beneficio que es el máximo.

## B. Apartado 2

En el apartado B, nos encontramos con varios problemas, y es que los objetos de la mochila son infinitos.

Y es que no podríamos rellenar una tabla de valores infinitos, puesto que nunca pararemos de calcular valores.

Pero hemos pensado varias soluciones que quizás fuesen posibles, y es que si los objetos fueran infinitos pero estuviesen ordenados. Podríamos cortar por peso, en donde el peso de la mochila marcara las filas de la matriz. O si no estuvieran ordenadas por peso, marcaríamos un beneficio máximo. En el momento que lleguemos a un punto en donde después de X iteraciones no obtengamos beneficios mejores dejaríamos de buscar la inserción de objetos, ya que no existiría un conjunto solución que mejore ese beneficio.

## C. Apartado 3

```

Objetos_2 = new ArrayList<Objeto2>();
//por que son 64 bits para un double y mas uno de los 0
this.matriz_resultado = new int[objetos.size()][maximo_peso*53+1];
//un bucle para darle valor a los objetos, con los pesos mas pequeños, pero
//en el introduccimos los pesos con el el i mas la suma del minimo beneficio
for(int i=0;i<=objetos.size();i++) {
    for(int j =i; j<i+1;j+=Double.MIN_VALUE) { //j es el peso que tendra por
        objetos_aux.add(new Objeto2(j,objetos.get(i).getBeneficio(),i));
    }
}

```

En este caso el problema reside en que no existe memoria en nuestros ordenadores, para el almacenamiento de la tabla. Ahora en este caso la matriz resultado tiene  $53 \times \text{máximo peso}$ , ya que son divisibles y por tanto tiene que tener tantos pequeños objetos como posibles valores decimales que son 53 ya que es la mantisa del doble.

También en nuestra solución para poder utilizar de nuevo los objetos , es decir puede ser el objeto  $n$  tantas veces como posibles divisiones tenga. Hemos creado una variable que te crea tantas veces el mismo id para ese objeto como instancias tenga.

```

int id; //en este caso creamos una id, ya que queremos qu
public Objeto2(int peso,int beneficio,int id) {
    this.peso = peso;
    this.beneficio = beneficio;
    this.id = id;
}

```

Es decir, siempre que se esté recorriendo 1 tendrá la id 1.

Cosa que se ha eliminado para el apartado D , donde cada objeto tiene su propia ID.

Como podemos ver en el informe de errores , no podemos por nuestros recursos y nuestro tiempo la ejecución de este programa:

```

1#
2# There is insufficient memory for the Java Runtime Environment to continue.
3# Native memory allocation (mmap) failed to map 33554432 bytes for GL virtual space
4# Possible reasons:
5#   The system is out of physical RAM or swap space
6# Possible solutions:
7#   Reduce memory load on the system
8#   Increase physical memory or swap space
9#   Check if swap backing store is full
10# Decrease Java heap size (-Xmx/-Xms) probado y no funciona
1# Decrease number of Java threads
2# Decrease Java thread stack sizes (-Xss)
3# Set larger code cache with -XX:ReservedCodeCacheSize=
4# This output file may be truncated or incomplete.
5#
6# Out of Memory Error (os_windows.cpp:3532), pid=17752, tid=21888
7#
8# JRE version: Java(TM) SE Runtime Environment (17.0.2+8) (build 17.0.2+8-LTS-86)
9# Java VM: Java HotSpot(TM) 64-Bit Server VM (17.0.2+8-LTS-86, mixed mode, sharing, tiered, compressed class ptrs, gl gc, windows-amd64)
10# No core dump will be written. Minidumps are not enabled by default on client versions of Windows
1#
2#
3----- S U M M A R Y -----
4
5Command Line: -Xms10000M -Xmx100000M -Dfile.encoding=Cp1252 -XX:+ShowCodeDetailsInExceptionMessages Program.saldia
6
7Host: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, 12 cores, 31G, Windows 11, 64 bit Build 22000 (10.0.22000.613)
8Time: Thu May 5 20:45:20 2022 Hora de verano romance elapsed time: 110.493984 seconds (0d 0h 1m 50s)
9
10----- T H R E A D -----
1#
2Current thread (0x000002689828edd0): JavaThread "main" [_thread_in_vm, id=21888, stack(0x000000c46d500000,0x000000c46d600000)]
3#
4Stack: [0x000000c46d500000,0x000000c46d600000]
5Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
6C:\Program Files\Java\jdk-17.0.2\bin\java.exe

```

## D. Apartado 4

Los algoritmos voraces y la programación dinámica sirven para resolver problemas similares pero su forma de trabajar es muy distinta, siendo preferible siempre uno por encima de otro, ambos dan con soluciones pero por lo general los algoritmos voraces suelen fallar más a la hora de dar soluciones óptimas, esto no quiere decir que no las alcancen, lo hacen, pero depende de la naturaleza del programa que les sea más fácil o más difícil llegar a ella.

Los resultados de los algoritmos de programación dinámica donde el gasto computacional va a ser de  $O(n)$  para recorrer la matriz en busca de la solución y otro  $O(n^2)$  para rellenar la tabla.

Llenado de la tabla:

```

public void ProblemaMochila_Llenado2() { //metodo iterativo
    for(int i = 1; i < this.objetos.size(); i++) { //recorremos los elementos columnas
        Objeto2 aux = objetos.get(i-1);
        for(int w = 1; w < this.maximo_peso + 1; w++) { // va sumando el valor minimo de un double
            if(aux.getPeso() <= w) { //si el peso del objeto es menor que el peso que el peso maximo de esa columna
                if((aux.getBeneficio() + matriz_resultado[i-1][(w-aux.getPeso()) < 0 ? w-aux.getPeso() : w]) > matriz_resultado[i-1][w]) { //si la
                    this.matriz_resultado[i][w] = aux.getBeneficio() + matriz_resultado[i-1][(w-aux.getPeso()) < 0 ? w-aux.getPeso() : w]; //inserta
                } else {
                    this.matriz_resultado[i][w] = this.matriz_resultado[i-1][w]; //si no le ponemos la anterior
                } //else interno
            } else { //caso en el que aux.getPeso() > w
                this.matriz_resultado[i][w] = this.matriz_resultado[i-1][w]; //si no le ponemos la anterior
            }
        } //bucle que recorre los objetos
    }
}

```

### Búsqueda de la solución:

```

public ArrayList<Integer> Solucion2() {
    ArrayList<Integer> sol = new ArrayList<Integer>(); //array solucion
    int i = this.objetos_aux.size();
    int w = this.maximo_peso;
    while(i > 0 && w > 0) { //mientras que i y w no sean 0, es decir los objetos o el peso
        if(matriz_resultado[i][w] != matriz_resultado[i-1][w]) { //si el resultado de la anterior es distinto significa que hay que añadir ese objeto
            sol.add(objetos_aux.get(i-1).getId());
            w = w - objetos.get(i-1).getPeso(); //el peso del siguiente objeto a explorar
            i--;
        } else {
            i--;
        }
    }
    return sol;
}
/*

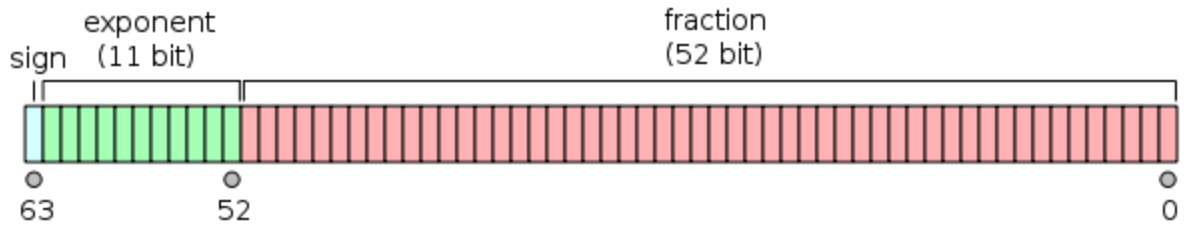
```

Realmente la búsqueda de la solución una vez se tengan los datos es bastante rápida por ello estos algoritmos pueden llegar a ser mejor, además de para computadores donde espacio no sea un problema, pero si el gasto computacional.

Pero estos algoritmos conllevan un problema, y es que el orden espacial cuando el problema lleva al almacenamiento de grandes datos, no es trivial. Por muy pequeño que sea el coste del cálculo de los datos, nos encontramos con una falta de espacio muy difícil de solucionar, o prácticamente imposible. Como hemos visto en el caso de calcular objetos en los cuales el peso se puede fragmentar, nos hemos encontrado con falta de espacio, inclusive para el tamaño de la mochila con 5 elementos y un peso máximo de 21 kilos. Ya que supone una matriz de  $(21 \times 53) \times 10$ , lo cual supone un orden espacial bastante



grande.



Ya que en el caso del doble la mantisa(decimal), supone 53 bits, por cada uno de los pesos.

Sin embargo, para el mismo problema utilizando un algoritmo voraz, no nos encontramos con ese problema, siempre y cuando no utilicemos una forma recursiva, ya que si utilizamos la recursividad obtendremos un desbordamiento de la pila y nos encontraremos con el mismo problema. Aunque la solución para greedy sin utilizar recursividad, en nuestro caso depende mucho de la ordenación de los candidatos. Nosotros hemos usado un mergesort para la ordenación en un arrayList, que es de  $O(n \log n)$ . Y necesitamos en nuestro caso dos bucles, para crear objetos que sean fraccionados. Cabe recordar que el algoritmo de programación dinámica también utiliza esa ordenación, ya que para el llenado de la tabla deben de estar ordenados.

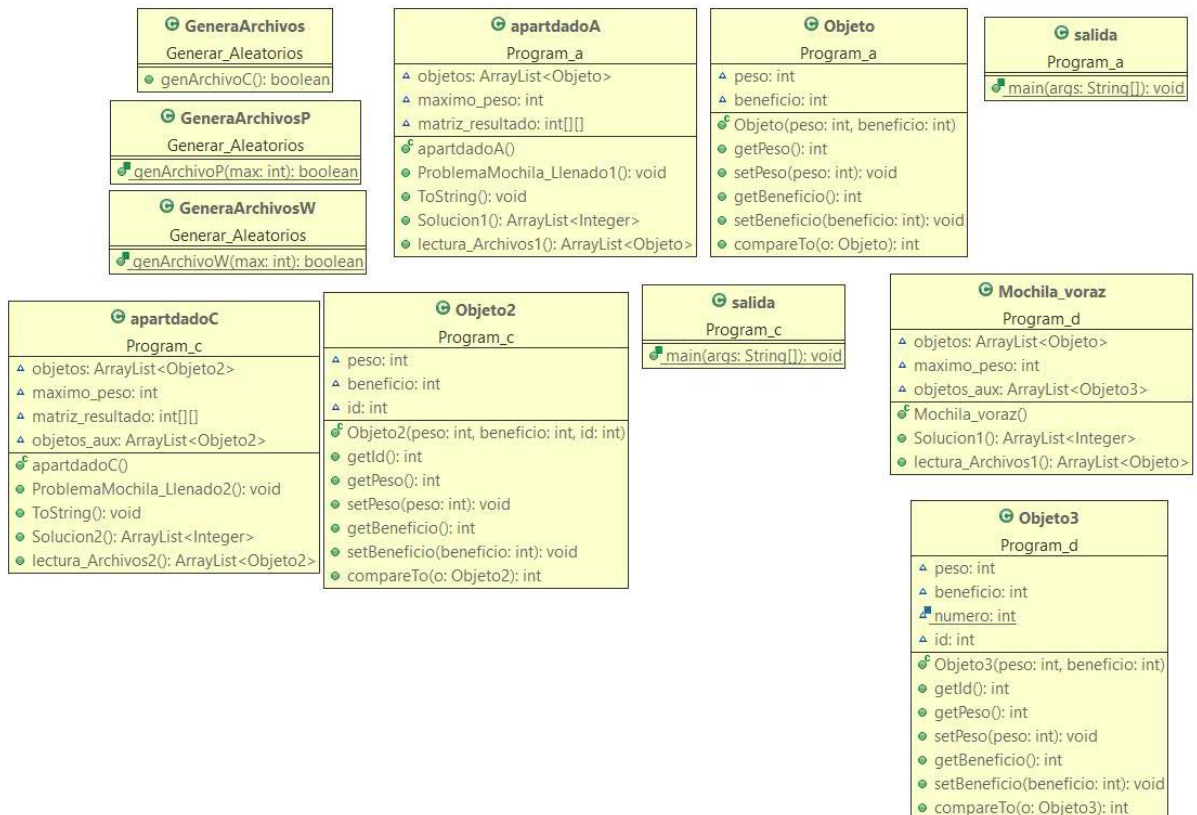
```
objetos = this.lectura_Archivos1(); //inicializa todas las variables
objetos_aux = new ArrayList<Objeto3>();
for(int i=0;i<=objetos.size();i++) {
    for(int j =i; j<i+1;j+=Double.MIN_VALUE) { //j es el peso que tendra
        objetos_aux.add(new Objeto3(j,objetos.get(i).getBeneficio()));
    }
}
}
```

Ya que la siguiente parte del algoritmo se encarga de analizar posibles candidatos que son del  $O(n^2)$ .

```
public ArrayList<Integer> Solucion1(){
    ArrayList<Integer> sol = new ArrayList<Integer>();//array solucion, no se inicializa, ya que esta a false por defecto, y con el maxim
    int peso = 0;
    while(peso!=maximo_peso) {
        //funcion de seleccion(heuristica), generada con el compareTo
        if(peso+objetos_aux.get(0).getPeso()<=maximo_peso) { //saco el primero ya que es el mejor, segun la funcion de seleccion
            sol.add(objetos_aux.get(0).getId()); //colocamos el id en el conjunto solucion
            objetos_aux.remove(objetos_aux.get(0)); //lo eliminamos de la lista de candidatos
        } else {
            peso = maximo_peso; //si llega aqui es que tenemos la solucion, recuerda la heuristica está en el compareTo
        }
    }
    return sol;
}
/*
```

Con lo cual para datos también grandes , sigue siendo un problema, pero es mucho mejor que el caso de la programación dinámica.

## II. Diagrama de clases



## III. ANEXO

```
package Generar_Aleatorios;
```

```
import java.io.BufferedWriter;
```

```
import java.io.File;
```

```
import java.io.FileWriter;
```

```
import java.io.IOException;
```

```

import java.io.PrintWriter;

/**
 * Para la generación de archivos aleatorios hemos seguido la estructura
 * necesaria para la ejecución de los métodos implementados.
 * Hemos usado clases de java para la escritura del archivo en formato .txt
 * Dado el parámetro n por el usuario, se genera un archivo con n nodos y sus
 * respectivas aristas con pesos aleatorios usando Math.Random con
 * un valor entre 0 y 1000.
 * @param nombre : nombre para la generación del archivo
 * @param n : entero para representar el número de nodos que vamos a usar en
 * el fichero
 * @param i : nodo origen
 * @param j : nodo destino
 * @return booleano true si se ha realizado correctamente el método y false para
 * el caso contrario
 */

public class GeneraArchivos {

    public boolean genArchivoC() {
        String directorioEntrada = System.getProperty("user.dir") +
        File.separator + "src" + File.separator + "dataset" + File.separator;

        try(FileWriter fw = new
        FileWriter(directorioEntrada+"c"+"c.txt", true);
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter out = new PrintWriter(bw))
        {

```

```
        int max=(int)(Math.random()*10000+1);
        out.println(max);
        GeneraArchivosW.genArchivoW(max);
        GeneraArchivosP.genArchivoP(max);
        return true;

    } catch (IOException e) {
        //añadir excepción
        return false;
    }

}

}

package Generar_Aleatorios;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

/**
 * Para la generación de archivos aleatorios hemos seguido la estructura
 necesaria para la ejecución de los métodos implementados.
 * Hemos usado clases de java para la escritura del archivo en formato .txt

```

```

* Dado el parámetro n por el usuario, se genera un archivo con n nodos y sus
respectivas aristas con pesos aleatorios usando Math.Random con
* un valor entre 0 y 1000.
* @param nombre : nombre para la generación del archivo
* @param n : entero para representar el número de nodos que vamos a usar en
el fichero
* @param i : nodo origen
* @param j : nodo destino
* @return booleano true si se ha realizado correctamente el método y false para
el caso contrario
*/

```

```

public class GeneraArchivosP {

    public static boolean genArchivoP(int max) {

        String directorioEntrada = System.getProperty("user.dir") +
File.separator + "src" + File.separator + "dataset" + File.separator;

        try(FileWriter fw = new
FileWriter(directorioEntrada+"w"+"txt", true);

            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter out = new PrintWriter(bw))
        {

            for(int i=0;i<max;i++) {

out.println(Math.random()*10000+1);

            }

            return true;

```

```
        } catch (IOException e) {
            //añadir excepción
            return false;
        }

    }

}

package Generar_Aleatorios;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

/**
 * Para la generación de archivos aleatorios hemos seguido la estructura
 * necesaria para la ejecución de los métodos implementados.
 * Hemos usado clases de java para la escritura del archivo en formato .txt
 * Dado el parámetro n por el usuario, se genera un archivo con n nodos y sus
 * respectivas aristas con pesos aleatorios usando Math.Random con
 * un valor entre 0 y 1000.
 * @param nombre : nombre para la generación del archivo
 * @param n : entero para representar el número de nodos que vamos a usar en
 * el fichero

```

```
* @param i : nodo origen
* @param j : nodo destino
* @return booleano true si se ha realizado correctamente el método y false para
el caso contrario
*/
```

```
public class GeneraArchivosW {

    public static boolean genArchivoW(int max) {
        String directorioEntrada = System.getProperty("user.dir") +
        File.separator + "src" + File.separator + "dataset" + File.separator;

        try(FileWriter fw = new
        FileWriter(directorioEntrada+"p"+"p.txt", true);
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter out = new PrintWriter(bw))
        {
            for(int i=0;i<max;i++) {

                out.println(Math.random()*10000+1);

            }

            return true;

        } catch (IOException e) {
            //añadir excepción
            return false;
        }
    }
}
```

```
    }

}

package Program_a;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

import Program_c.Objeto2;

public class apartdadoA {
    ArrayList<Objeto> objetos;
    int maximo_peso;
    int matriz_resultado[][];
    /*
     * Tenemos los pesos, beneficios y peso maximo
     * recuerda que la posicion de matriz resultado [0] [o] son todo 0
     */
    public apartdadoA() throws FileNotFoundException {
        objetos = this.lectura_Archivos1(); //inicializa todas las variables
        this.matriz_resultado = new int[objetos.size()+1][maximo_peso+1];
        //ya que son los objetos que hay mas uno, mas el maximo peso mas 1, ya que
        las primeras columnas son 0
    }
    /*
```



```

    * k van a ser los elementos y j el w , el peso
    */
    //No es necesario inicializar a 0, ya que java deja por defecto los enteros a
0
    //array[fila,columna]
    public void ProblemaMochila_Llenado1() { //metodo iterativo
        for(int i = 1; i < this.objetos.size() + 1; i++) { //recorremos los elementos
columnas
            Objeto aux = objetos.get(i-1);
            for(int w=1; w < this.maximo_peso+1; w++) { // filas hasta 166
en este caso
                if(aux.getPeso() <= w) { //si el peso del objeto es
menor que el peso que el peso maximo de esa columna

                    if((aux.getBeneficio() + matriz_resultado[i-1][(w-aux.getPeso()) < 0 ? 0 : w-aux.getPeso()]) > matriz_resultado[i-1][w]) { //si la resta es -1 devolvemos 0 y aqui
comprobamos que el beneficio es menos para insertarlo

                        this.matriz_resultado[i][w] =
aux.getBeneficio() + matriz_resultado[i-1][(w-aux.getPeso()) < 0 ? 0 : w-aux.getPeso()
]; //insertamos el beneficio de introducir el objeto

                    } else {

                        this.matriz_resultado[i][w] =
this.matriz_resultado[i-1][w]; //si no le ponemos la anterior

                    } //else interno

                } else { //caso en el que aux.getpeso() > w

                    this.matriz_resultado[i][w] =
this.matriz_resultado[i-1][w]; //si no le ponemos la anterior

                }

            }

        } //bucle que recorre los objetos

```

```

        }
    }
    public void ToString() {
        for(int i = 0;i<=objetos.size();i++) {
            for(int j = 0;j<=maximo_peso;j++) {
                System.out.print(matriz_resultado[i][j]+" ");
            }
            System.out.println("\n");
        }
    }

    public ArrayList<Integer> Solucion1(){
        ArrayList<Integer> sol = new ArrayList<Integer>();//array solucion
        int i = this.objetos.size();
        int w = this.maximo_peso;
        while(i>0 && w>0) { //mientras que i y w no sean 0, es decir los objetos
o el peso
            if(matriz_resultado[i][w]!= matriz_resultado[i-1][w]) { //si el resultado
de la anterior es distinto significa que hay que añadir ese objeto
                sol.add(i);
                w = w - objetos.get(i-1).getPeso();//el peso del siguiente
objeto a explorar
                i --;
            }else {
                i--;
            }
        }
        return sol;
    }

```

```

    }
    /*
    * En la lectura de archivos cargamos los archivos
    */

    public ArrayList<Objeto> lectura_Archivos1() throws
FileNotFoundException{

        String directorioPeso = System.getProperty("user.dir") +
File.separator + "src"+File.separator+"datos"+File.separator+"p02_w.txt";

        String directorioBeneficio = System.getProperty("user.dir") +
File.separator + "src"+File.separator+"datos"+File.separator+"p02_p.txt";

        String directorioMax = System.getProperty("user.dir") +
File.separator + "src"+File.separator+"datos"+File.separator+"p02_c.txt";

        ArrayList<Objeto> lista = new ArrayList<Objeto>();
        File docMax = new File(directorioMax);
        File docPeso = new File(directorioPeso);
        File docBeneficio = new File(directorioBeneficio);
        Scanner objPeso = new Scanner(docPeso);
        Scanner objBeneficio = new Scanner(docBeneficio);
        Scanner objMax = new Scanner(docMax);
        this.maximo_peso = Integer.parseInt(objMax.nextLine().trim());
        while (objPeso.hasNextLine()) {
            int peso = Integer.parseInt(objPeso.nextLine().trim());
            int beneficio
=Integer.parseInt(objBeneficio.nextLine().trim());
            lista.add(new Objeto(peso,beneficio));
        }

        lista.sort(null);

        //ordenar las colas
        objBeneficio.close();

```

```
        objPeso.close();
        objMax.close();
        return lista;
    }
}
```

```
package Program_a;
```

```
public class Objeto implements Comparable<Objeto>{
    int peso;
    int beneficio;

    public Objeto(int peso,int beneficio) {
        this.peso = peso;
        this.beneficio = beneficio;
    }

    public int getPeso() {
        return peso;
    }

    public void setPeso(int peso) {
        this.peso = peso;
    }

    public int getBeneficio() {
        return beneficio;
    }
}
```

```
        public void setBeneficio(int beneficio) {
            this.beneficio = beneficio;
        }
    }

    /*
     * Compara el objeto segun el peso
     */

    @Override
    public int compareTo(Objeto o) {
        if(this.getPeso()<o.getPeso()) return -1;
        if(this.getPeso()>o.getPeso()) return 1;
        return 0;
    }
}

package Program_a;

import java.io.FileNotFoundException;
import java.util.ArrayList;

public class salida {

    public static void main(String[] args) throws FileNotFoundException {
        apartdadoA o = new apartdadoA();
        o.ProblemaMochila_Llenado1();
        o.ToString();
        ArrayList<Integer> sol = o.Solucion1();
    }
}
```

```
        for(int i : sol) {
            System.out.println(i);
        }
    }
}

package Program_c;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

public class apartdadoC {
    ArrayList<Objeto2> objetos;
    int maximo_peso;
    int matriz_resultado[][];
    ArrayList<Objeto2> objetos_aux;//objetos con los pesos doubles
    /*
     * Tenemos los pesos, beneficios y peso maximo
     * recuerda que la posicion de matriz resultado [0] [0] son todo 0
     */
    public apartdadoC() throws FileNotFoundException {
        objetos = this.lectura_Archivos2(); //inicializa todas las variables
        objetos_aux = new ArrayList<Objeto2>();
        //por que son 64 bits para un double y mas uno de los 0
        this.matriz_resultado = new int[objetos.size()][maximo_peso*64+1];
        //multiplicamos por 2^6 = 64
    }
}
```

//un bucle para darle valor a los objetos, con los pesos mas pequeños, pero como los objetos no se pueden repetir

//en el introduccimos los pesos con el el i mas la suma del minimo beneficio y damos lugar a que no solo un objeto sea el que se meta si no mitad de muchos, y tenemos en cuenta que el beneficio es el mismo

```
for(int i=0;i<=objetos.size();i++) {
    for(int j =i; j<i+1;j+=Double.MIN_VALUE) { //j es el peso que
        tendra por ejemplo en el primer caso 1, mas los decimales que tenga el minimo
        double
```

```
        objetos_aux.add(new
        Objeto2(j,objetos.get(i).getBeneficio(),i));
    }
}
```

```
}
/*
```

\* aqui en vez de que tenga en cuenta cada uno de los objetos , creamos tantos objetos como valores maximos haya

```
*/
```

```
public void ProblemaMochila_Llenado2() { //metodo iterativo
```

```
    for(int i = 1;i<this.objetos.size()+1;i++) { //recorremos los elementos
    columnas
```

```
        Objeto2 aux = objetos.get(i-1);
```

```
        for(int w=1; w<this.maximo_peso+1;w++) { // va sumando el
        valor minimo de un double
```

```
            if(aux.getPeso()<=w) { //si el peso del objeto es
            menor que el peso que el peso maximo de esa columna
```

```
            if((aux.getBeneficio()+matriz_resultado[i-1][(w-aux.getPeso())<0?0:w-aux.getPes
            o()])> matriz_resultado[i-1][w]) { //si la resta es -1 devolvemos 0 y aqui
            comprobamos que el beneficio es menos para insertarlo
```

```

        this.matriz_resultado[i][w] =
aux.getBeneficio()+matriz_resultado[i-1][(w-aux.getPeso())<0]?0:w-aux.getPeso()
]; //insertamos el beneficio de introducir el objeto

        }else {

            this.matriz_resultado[i][w] =
this.matriz_resultado[i-1][w]; //si no le ponemos la anterior

            }//else interno

        }else { //caso en el que aux.getpeso()>w

            this.matriz_resultado[i][w] =
this.matriz_resultado[i-1][w]; //si no le ponemos la anterior

            }

        }//bucle que recorre los objetos

    }

}

public void ToString() {

    for(int i = 0;i<=objetos.size();i++) {

        for(int j = 0;j<=maximo_peso;j++) {

            System.out.print(matriz_resultado[i][j]+" ");

        }

        System.out.println("\n");

    }

}

public ArrayList<Integer> Solucion2(){

    ArrayList<Integer> sol = new ArrayList<Integer>();//array solucion

    int i = this.objetos_aux.size();

    int w = this.maximo_peso;

    while(i>0 && w>0) {//mientras que i y w no sean 0, es decir los objetos
o el peso

```



if(matriz\_resultado[i][w]!= matriz\_resultado[i-1][w]) { //si el resultado de la anterior es distinto significa que hay que añadir ese objeto

sol.add(objetos\_aux.get(i-1).getId());

w = w - objetos.get(i-1).getPeso();//el peso del siguiente objeto a explorar

i --;

}else {

i--;

}

}

return sol;

}

/\*

\* En la lectura de archivos cargamos los archivos

\*/

public ArrayList<Objeto2> lectura\_Archivos2() throws  
FileNotFoundException{

String directorioPeso = System.getProperty("user.dir") +  
File.separator + "src" + File.separator + "datos" + File.separator + "p02\_w.txt";

String directorioBeneficio = System.getProperty("user.dir") +  
File.separator + "src" + File.separator + "datos" + File.separator + "p02\_p.txt";

String directorioMax = System.getProperty("user.dir") +  
File.separator + "src" + File.separator + "datos" + File.separator + "p02\_c.txt";

ArrayList<Objeto2> lista = new ArrayList<Objeto2>();

File docMax = new File(directorioMax);

File docPeso = new File(directorioPeso);

File docBeneficio = new File(directorioBeneficio);

Scanner objPeso = new Scanner(docPeso);

Scanner objBeneficio = new Scanner(docBeneficio);

```

        Scanner objMax = new Scanner(docMax);

        this.maximo_peso =
Integer.parseInt(objMax.nextLine().trim());

        while (objPeso.hasNextLine()) {

            int peso =
Integer.parseInt(objPeso.nextLine().trim());

            int beneficio
=Integer.parseInt(objBeneficio.nextLine().trim());

            lista.add(new Objeto2(peso,beneficio,0));//0 porque
aqui el id nos da igual, es una forma de castearlo

        }

        lista.sort(null);

        //ordenar las colas
        objBeneficio.close();
        objPeso.close();
        objMax.close();

        return lista;

    }
}

```

```
package Program_c;
```

```

public class Objeto2 implements Comparable<Objeto2>{

    int peso;

    int beneficio;

    int id;//en este caso creamos una id, ya que queremos que sean el mismo
objeto

    public Objeto2(int peso,int beneficio,int id) {

        this.peso = peso;

```

```
        this.beneficio = beneficio;
        this.id = id;
    }
    public int getId() {
        return id;
    }
    public int getPeso() {
        return peso;
    }

    public void setPeso(int peso) {
        this.peso = peso;
    }

    public int getBeneficio() {
        return beneficio;
    }

    public void setBeneficio(int beneficio) {
        this.beneficio = beneficio;
    }
}

/*
 * Compara el objeto segun el peso
 */

@Override
public int compareTo(Objeto2 o) {
    if(this.getPeso()<o.getPeso()) return -1;
```

```
        if(this.getPeso()>o.getPeso()) return 1;
        return 0;
    }

}

package Program_c;

import java.io.FileNotFoundException;
import java.util.ArrayList;

public class salida {

    public static void main(String[] args) throws FileNotFoundException {
        apartdadoC o = new apartdadoC();
        o.ProblemaMochila_Llenado2();
        o.ToString();
        ArrayList<Integer> sol = o.Solucion2();
        for(int i : sol) {
            System.out.println(i);
        }
    }
}

package Program_d;

import java.io.File;
import java.io.FileNotFoundException;
```

```
import java.util.ArrayList;
import java.util.Scanner;

import Program_a.Objeto;

public class Mochila_voraz {
    ArrayList<Objeto> objetos;
    int maximo_peso;
    ArrayList<Objeto3> objetos_aux;

    public Mochila_voraz() throws FileNotFoundException {
        objetos = this.lectura_Archivos1(); //inicializa todas las variables
        objetos_aux = new ArrayList<Objeto3>();
        for(int i=0;i<=objetos.size();i++) {
            for(int j =i; j<i+1;j+=Double.MIN_VALUE) {j es el peso que tendra
por ejemplo en el primer caso 1, mas los decimales que tenga el minimo double
                objetos_aux.add(new
Objeto3(j,objetos.get(i).getBeneficio()));
            }
        }
    }

    public ArrayList<Integer> Solucion1(){
        ArrayList<Integer> sol = new ArrayList<Integer>();//array solucion, no se
inicializa, ya que esta a false por defecto,y con el maximo valor del doble
        int peso = 0;
        while(peso!=maximo_peso) {
            //funcion de seleccion(heuristica), generada con el compare to
```

```

        if(peso+objetos_aux.get(0).getPeso())<=maximo_peso) { //saco el primero
ya que es el mejor, segun la funcion de seleccion

```

```

        sol.add(objetos_aux.get(0).getId()); //colocamos el id en el
conjunto solucion

```

```

        objetos_aux.remove(objetos_aux.get(0)); //lo eliminamos de
la lista de candidatos

```

```

    }else {

```

```

        peso = maximo_peso; //si llega aqui es que tenemos la solucion ,
recuerda la heuristica está en el comapareTo

```

```

    }

```

```

}

```

```

    return sol;

```

```

}

```

```

/*

```

```

    * En la lectura de archivos cargamos los archivos

```

```

    */

```

```

    public ArrayList<Objeto> lectura_Archivos1() throws FileNotFoundException{
        String directorioPeso = System.getProperty("user.dir") + File.separator
+"src"+File.separator+"datos"+File.separator+"p02_w.txt";

```

```

        String directorioBeneficio = System.getProperty("user.dir") + File.separator
+"src"+File.separator+"datos"+File.separator+"p02_p.txt";

```

```

        String directorioMax = System.getProperty("user.dir") + File.separator
+"src"+File.separator+"datos"+File.separator+"p02_c.txt";

```

```

        ArrayList<Objeto> lista = new ArrayList<Objeto>();

```

```

        File docMax = new File(directorioMax);

```

```

        File docPeso = new File(directorioPeso);

```

```

        File docBeneficio = new File(directorioBeneficio);

```

```

        Scanner objPeso = new Scanner(docPeso);

```

```

        Scanner objBeneficio = new Scanner(docBeneficio);

```

```

Scanner objMax = new Scanner(docMax);
this.maximo_peso = Integer.parseInt(objMax.nextLine().trim());
while (objPeso.hasNextLine()) {
    int peso = Integer.parseInt(objPeso.nextLine().trim());
    int beneficio = Integer.parseInt(objBeneficio.nextLine().trim());
    lista.add(new Objeto(peso,beneficio));
}

lista.sort(null);
//ordenar las colas
objBeneficio.close();
objPeso.close();
objMax.close();
return lista;
}
}

package Program_d;

public class Objeto3 implements Comparable<Objeto3>{
    int peso;
    int beneficio;

    static int numero = 0;//en este caso creamos una id, ya que queremos que sean
    el mismo objeto

    int id;

    public Objeto3(int peso,int beneficio) {
        this.peso = peso;
        this.beneficio = beneficio;
        this.id = numero; //vamos a generar una id para cada mochila
    }
}

```

```
        numero++;
    }
    public int getId() {
        return id;
    }
    public int getPeso() {
        return peso;
    }

    public void setPeso(int peso) {
        this.peso = peso;
    }


    public int getBeneficio() {
        return beneficio;
    }

    public void setBeneficio(int beneficio) {
        this.beneficio = beneficio;
    }
}

/*
 * Compara el objeto segun el peso, nos va a devolver el objeto mas prometedor,
ya que tiene menor peso y mayor beneficio
 */

@Override
public int compareTo(Objeto3 o) {
    if((this.getPeso()<o.getPeso() && this.getBeneficio()>o.getBeneficio()))
return -1;
```





```
        if(this.getPeso()>o.getPeso() && this.getBeneficio()>o.getBeneficio())
return 1;
        return 0;
    }

}
```