Teoría de algoritmos 1 – 75.29

Trabajo Práctico Nº: 3

Integrantes:		
Padrón	**Nombre y Apellido**	**Email**
83.989	Alejo Vinjoy	avinjoy@gmail.com
84.623	Santiago Nicolás Risaro Sesar	nrisaro@gmail.com

Para uso de la cátedra
Primera entrega
Corrector
Observaciones
Segunda entrega
Corrector
Observaciones

Contenido

Enunciado

75.29 Teoría de Algoritmos I
Trabajo Práctico Nº 3
Problema del Empaquetamiento

Fecha de entrega: 27 de noviembre de 2013

Definición: Dado un conjunto de n objetos cuyo tamaño son {T1, T2, … Tn} , con Tiε(0, 1] se debe empaquetarlos usando la cantidad mínima de envases de capacidad 1.

Se pide

1) Demostrar que el problema del empaquetamiento es NP Completo (para lo cual se debe utilizar alguno de los problemas NP Completos vistos en clase).

2) Programar un algoritmo por fuerza bruta que busque la solución exacta del problema. Analizar el orden del mismo. Realizar mediciones empíricas tomando el tiempo que demora cada corrida, graficar en función de n y comparar con la curva teórica. Analizar los resultados de las mediciones.

3) Dado el siguiente algoritmo: Se abre el primer envase y se empaqueta el primer objeto, luego por cada uno de los objetos restantes se prueba si cabe en el envase actual que está abierto, si es así se lo empaqueta en el mismo envase y se continúa con el siguiente objeto, si no cabe se cierra el envase actual y se abre uno nuevo que pasa a ser el envase actual y se empaqueta el objeto y continúa con el próximo hasta lograr empaquetar todos los objetos.

Este algoritmo sirve como una aproximación para resolver el problema del empaquetamiento.

Se pide implementar este algoritmo, analizar el orden y analizar que tan buena es la aproximación.

Para analizar que tan buena es la aproximación se usa la siguiente fórmula: Sea I una instancia cualquiera del problema del empaquetamiento, sea z(I) la solución óptima para esa instancia y sea A(I) la solución aproximada entonces se define $\frac{A(I)}{Z(I)} \leq R(A)$ para todas las instancias I. Calcular R(A) para la aproximación dada (y demostrar que la cota está bien calculada). Realizar mediciones empíricas utilizando el algoritmo exacto del punto anterior y el algoritmo aproximado de este punto con el objeto de verificar que se cumple la relación.

Ejemplo:

T={0,4; 0,8; 0,5; 0,1; 0,7; 0,6; 0,1; 0,4; 0,2; 0,2}

Solución exacta:

E1={0,5; 0,4; 0,1}
E2={0,8; 0,2}
E3={0,7; 0,2; 0,1}
E4={0,6; 0,4}
Total 4 envases.

Solución aproximada:

E1={0,4}
E2={0,8}
E3={0,5; 0,1}
E4={0,7}
E5={0,6; 0,1}
E6={0,4; 0,2; 0,2}
Total 6 envases

Datos de entrada:
Los datos vendrán en archivos de textos con el siguiente formato:
<n>
<línea en blanco>
<T1>
<T2>
...
<Tn>
<EOF>

Invocación:
tdatp3 <E> | <A> <datos.txt>
donde el parámetro E indica calcular la solución exacta y el parámetro A calcular la solución aproximada.

Formato de salida:
La salida será por pantalla con el siguiente formato y con el encabezado establecido en las normas para la presentación de los TPs:

<Solución Exacta> | <Solución Aproximada>: #Envases
<Tiempo de ejecución en mseg>.

Consideraciones para realizar las pruebas empíricas de medición de tiempo:
Se recomienda realizar varias corridas con distintos conjuntos de datos del mismo tamaño y promediar los tiempos medidos para obtener un punto a graficar. Repetir para valores de n crecientes hasta valores que sean manejables con el hardware donde se realiza la prueba.

Estrategia de Resolución

1) Demostrar que el problema del empaquetamiento es NP Completo (para lo cual se debe utilizar alguno de los problemas NP Completos vistos en clase).

Para demostrar lo pedido nos basaremos en la siguiente propiedad :

Si Y ***es un problema NP-Completo,*** X ***es un problema en NP que cumple con la propiedad:*** $Y \leq X$***, entonces X es NP - Completo.***

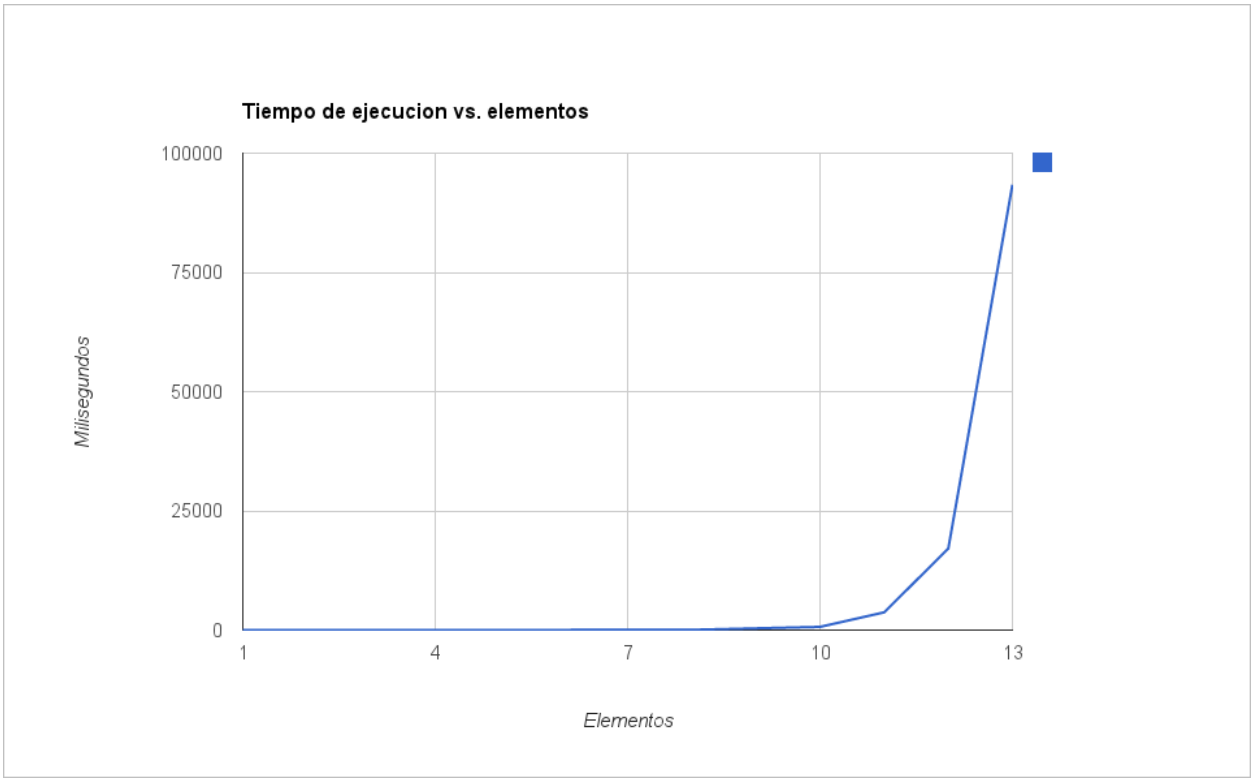
Basta con tomar como Y al problema ya visto de la suma de subconjuntos y a X como nuestro problema dado para lograr aplicar esta propiedad.

El problema de la suma de los subconjuntos se puede ver como una reducción de nuestro problema de empaquetamiento, ya que se podría ver a los subconjuntos a formar como los envases a llenar donde, a su vez, la meta que tenemos que alcanzar es que sumen uno. Además sabemos que el problema de empaquetamiento es un problema cuya solución corre en tiempo no polinomial. Entonces como ya sabemos que el problema de suma de subconjuntos es NP - Completo, por la propiedad antes enunciada queda demostrado lo pedido.

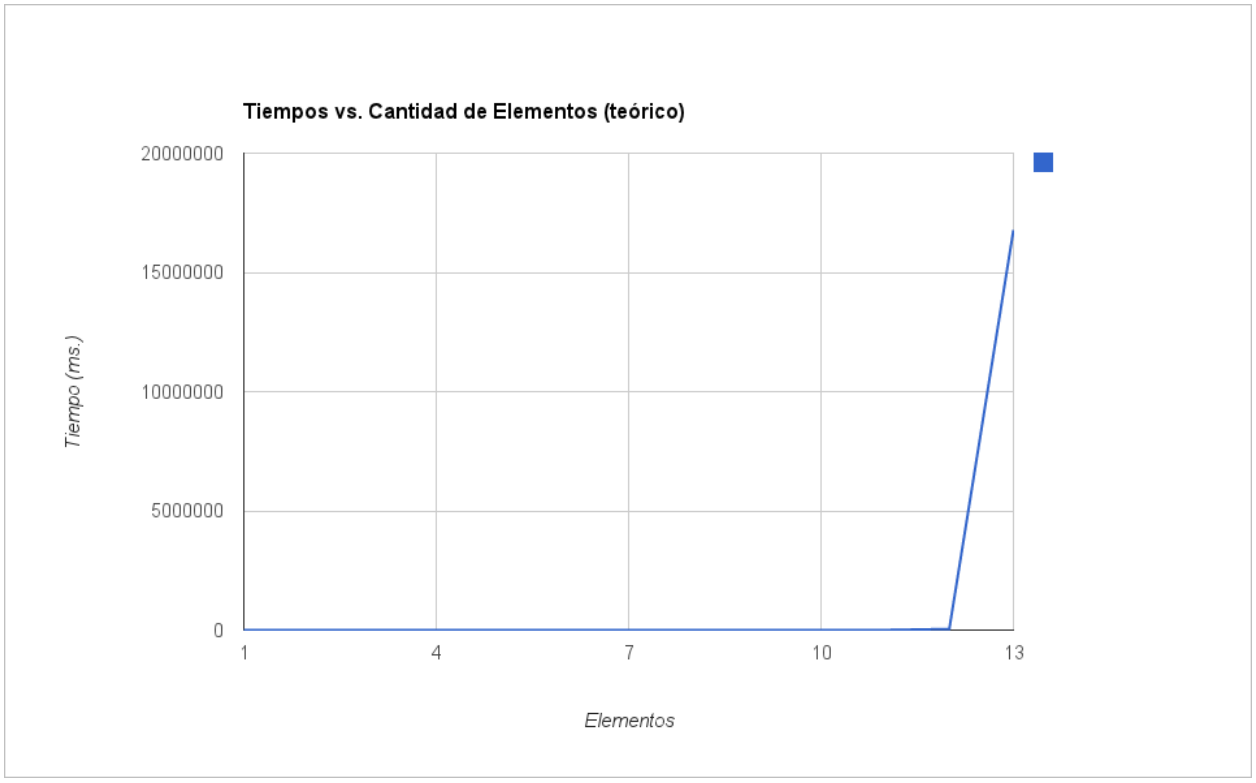
2) Se Implementó el algoritmo exacto, por fuerza bruta obteniéndose los siguiente resultados:

Solución Exacta		
Paquetes	**Elementos**	**Tiempos (Miliseg)**
1	1	0
2	2	0
3	4	4
4	8	60
5	10	675
5	11	3725
6	12	17097
8	13	93384

Los cuales fueron volcados al siguiente gráfico, donde se puede ver claramente la naturaleza exponencial del algoritmo exacto:



A su vez, podemos observar que se parece bastante a su forma teórica O (m ^ n):



Como conclusión podemos ver que el algoritmo nos sirve para valores de n muy chicos. Dado que para valores de n (cantidad de elementos) apenas mayores a 16 los tiempos se disparan, por lo cual la solución exacta deja de ser viable en términos computacionales.

3) Implementamos la solución aproximada delineada en el enunciado y la probamos con las mismas cantidades y repeticiones que para la solución exacta, los resultados son estos

Solución Aproximada			
Paquetes	**Elementos**	**Tiempos (Miliseg)**	**Tiempos (Nanoseg)**
1	1	0	14760
2	2	0	17966
3	4	0	23599
6	8	0	34201
6	10	0	38534
9	11	0	41737
11	12	0	46151
11	13	0	47733

Como puede verse, para esa baja cantidad de elementos, la ejecución es tán rápida que no puede ser medida en términos de milisegundos. Viendo los tiempos en nanosegundos se nota una “linealidad” de la solución.

Para corroborar que la solución es lineal hicimos otra serie de mediciones, con cantidades elevadas de elementos. Cabe destacar que, con los recursos que poseemos, no es factible ejecutar el algoritmo por fuerza bruta para estas cantidad de elementos.

Solución Aproximada		
Paquetes	**Elementos**	**Tiempos (Miliseg)**
6151	8192	5
12408	16384	9
24729	32768	10
49177	65536	17
98638	131072	28
197768	262144	60
394963	524288	143
790059	1048576	287

Aquí puede verse la naturaleza lineal del algoritmo, acentuada cuanto más grande es el conjunto de elementos a tratar. Como era de esperarse, a medida que la cantidad de elementos se duplica, el tiempo de ejecución también lo hace.

Comparando la solución óptima con la aproximada podemos ver cuán buena es la aproximación y si respeta la cota teórica.

Elementos	Paquetes (Sol. Exacta)	Paquetes (Sol. Aprox)	A(I)/Z(I)
1	1	1	1
2	2	2	1
4	3	3	1

8	4	6	1.5
10	5	6	1.2
11	5	9	1.8
12	6	11	1.833
13	8	11	1.375

Demostración de la cota R(A)

Basado en http://en.wikipedia.org/wiki/Bin_packing_problem#First-fit_algorithm

Podemos ver empíricamente que el número de envases utilizado por este algoritmo de aproximación es no más que que el doble del óptimo. Esto se debe a la observación de que en cualquier momento, es imposible que 2 envases estén llenos hasta la mitad. Si esto no fuera posible, significaría que en algún punto un solo envase estaría lleno hasta la mitad y otro envase debería ser abierto para contenter a la otra mitad de los elementos. Pero como el primero ya tiene por lo menos la mitad de su espacio ocupado, el algoritmo no abrirá un nuevo envase para los elementos que sean menores a la mitad. Solamente cuando se llene un envase más de la mitad o cuando un elemento sea mayor a la mitad libre del envase el algoritmo abrirá un nuevo envase.

Por lo tanto si tenemos *B envases,* por lo menos *B* - 1 envases son más de la mitad. Por lo tanto $\sum_{i=1}^{n} a_i > \frac{B-1}{2} V$. Y como $\frac{\sum_{i=1}^{n} a_i}{V}$ es un límite inferior de la *OPT* valor óptimo, tenemos que *B* -. 1 <2 *OPT* y por lo tanto ***B* ≤ 2 *OPT***

Análisis de Orden

Realizado en los comentarios de los métodos en el código fuente.

A modo de resumen, se indica aquí que el tiempo de la solución por fuerza bruta es O(m^n) y que la solución aproximada tiene un costo lineal, O(n).

Indicaciones para la ejecución

Se asume que en la PC donde va a ejecutarse el programa se encuentra instalada y correctamente configurada una versión del JRE. Este trabajo práctico fue desarrollado utilizando la versión 1.6 de java y testeado con el JRE oficial, provisto por Oracle. Si bien no fue probado con otra configuración, el trabajo debería ejecutarse sin problemas con máquinas virtuales versión 1.7 y/o no oficiales. Debido a los cambios en la API entre la versión 1.5 y 1.6 de java no hay garantías de que pueda ejecutarse correctamente con versiones anteriores a 1.6.

Para la compilación se utilizó Maven 3.

Se asume que existe una carpeta $HOME/tda y que en ella se han descargado los archivos a utilizar.

Ejecución

1. Posicionarse en el directorio donde se encuentra el archivo tdatp3.jar

```
cd $HOME/tda
```

2. Ejecutar el programa

```
java -jar tdatp3.jar <TIPOSOLUCION> <DATOS> <CANTIDAD_REP>
```

<TIPOSOLUCION> E para la solución exacta, A para la alternativa.

<DATOS> es un parámetro con la ruta del archivo que contiene los datos a cargar. También puede ser -Pnum donde num indica la cantidad de números generados aleatoriamente.

<CANTIDAD_REP> -Vnum donde num indica la cantidad de repeticiones de la corrida.

Compilación

1. Posicionarse en la carpeta donde se descargará el código fuente

```
cd $HOME/tda
```

2. Descargar una copia del código fuente

```
svn checkout
http://teoria-de-algoritmos-2013-2.googlecode.com/svn/tags/Entrega_3_2013_11_26/TeoriaDeAlgoritmos
```

3. Poscionarse en la carpeta recién descargada

```
cd TeoriaDeAlgoritmos
```

4. Ejecutar maven

```
mvn clean package
```

5. El ejecutable se encuentra en la carpeta target

Conclusiones

Podemos concluir que para los algoritmos NP-Completos, aquellos cuya resolución exacta no se puede calcular dentro de un tiempo razonable, lo mejor que podemos hacer es determinar aproximaciones y heurísticas que optimicen el tiempo. Si bien esto no es lo mejor, hay optimizaciones para los problemas NP-Completos que logran un balance entre la solución obtenida (que no necesariamente es la exacta en todos los casos) y el tiempo insumido.

Código Fuente

Clase CargadorDeElementos

```
public class CargadorDeElementos implements FuenteDeDatos {

    private final BufferedReader reader;
    private List<Float> elementos;

    public CargadorDeElementos(BufferedReader reader) {
       this.reader = reader;
    }

    @Override
    public List<Float> obtenerDatos() {

       if (elementos == null) {
             elementos = new ArrayList<Float>();
             cargarElementos();
       }

       return elementos;
    }

    private void cargarElementos() {

       try {
             leerHeader();

             String linea = reader.readLine();

             while (linea != null) {

                   float elemento = Float.valueOf(linea);
                   elementos.add(elemento);

                   linea = reader.readLine();
             }

       } catch (IOException e) {
             e.printStackTrace();
       }
    }

    private void leerHeader() throws IOException {
       reader.readLine();
       reader.readLine();
    }
```

```
}
```

Clase Empaquetamiento

```
public class Empaquetamiento {

    private static Empaquetamiento instancia;

    public static void main(String[] args) {

       System.out.println("Teoría de algoritmos - TP 3");
       System.out.println("Autores:");
       System.out.println("Alejo Vinjoy - 83.989");
       System.out.println("Santiago Nicolas Risaro Sesar - 84.623");

       Empaquetamiento instancia = getInstancia();

       String tipoDeSolucion = args[0];
       String tipoDeFuente = args[1];

       int cantidadDeEjecuciones = calcularCantidadDeEjecuciones(args);

       instancia.ejecutar(SelectorDeSolucion.obtenerSelector(tipoDeSolucion),
SelectorDeFuente.obtenerFuente(tipoDeFuente), cantidadDeEjecuciones);

    }

    private static int calcularCantidadDeEjecuciones(String[] args) {
       int cantidadDeEjecuciones = 1;
```

```
        if (args.length > 2) {
              String ejecuciones = args[2].substring(2);
              cantidadDeEjecuciones = Integer.valueOf(ejecuciones);
        }
        return cantidadDeEjecuciones;
    }

    private static Empaquetamiento getInstancia() {

        if (instancia == null) {
              instancia = new Empaquetamiento();
        }

        return instancia;
    }

    public static void setInstancia(Empaquetamiento instancia) {
        Empaquetamiento.instancia = instancia;
    }

    public void ejecutar(SelectorDeSolucion selectorDeSolucion, FuenteDeDatos
fuente, int cantidadDeEjecuciones) {

        Solucion solucion = selectorDeSolucion.obtenerSolucion(fuente);
        solucion.ejecutar(cantidadDeEjecuciones);

        System.out.println(selectorDeSolucion.getMensaje() + ": " +
solucion.getEnvases());
        System.out.println(solucion.getTiempoPromedio());
    }
```

```
}
```

Interface FuenteDeDatos

```
public interface FuenteDeDatos {

    List<Float> obtenerDatos();
}
```

Clase GeneradorDeCasosRandom

```
public class GeneradorDeCasosRandom implements FuenteDeDatos{

    private final int cantidadDeElementos;
    private final List<Float> elementos;
    private Random random;
    private DecimalFormat format;

    public GeneradorDeCasosRandom(int cantidadDeElementos) {
       this.cantidadDeElementos = cantidadDeElementos;
       this.elementos = new ArrayList<Float>(cantidadDeElementos);
       random = new Random();
       Locale.setDefault(new Locale("us"));
       format = new DecimalFormat("#.#");

    }

    @Override
```

```
    public List<Float> obtenerDatos() {


        this.elementos.clear();

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


                float numeroRandom = random.nextFloat() + 0.1F;


                if (numeroRandom > 1.0F) {

                        numeroRandom = 1.0F;

                }


                String formateado =
String.format(Locale.US,"%s",format.format(numeroRandom));

                float numeroFormateado = Float.valueOf(formateado);


                elementos.add(numeroFormateado);

        }

        return elementos;

    }


}
```

Clase SelectorDeFuente

```
public class SelectorDeFuente {

    public static FuenteDeDatos obtenerFuente(String parametro) {

       FuenteDeDatos fuenteDeDatos = null;

       if (parametro.endsWith(".txt")) {

             fuenteDeDatos = construirCargador(parametro);

       } else {

             fuenteDeDatos = construirGenerador(parametro);
       }

       return fuenteDeDatos;
    }

    public static FuenteDeDatos construirCargador(String parametro) {

       FuenteDeDatos fuenteDeDatos = null;
       BufferedReader reader;

       try {
             reader = new BufferedReader(new FileReader(parametro));
             fuenteDeDatos = new CargadorDeElementos(reader);
```

```
        } catch (FileNotFoundException e) {
              e.printStackTrace();
        }
        return fuenteDeDatos;
    }

    public static FuenteDeDatos construirGenerador(String parametro) {

        FuenteDeDatos fuenteDeDatos;
        String cantidadElementosStr = parametro.substring(2);
        int cantidadElementos = Integer.valueOf(cantidadElementosStr);

        fuenteDeDatos = new GeneradorDeCasosRandom(cantidadElementos);
        return fuenteDeDatos;
    }

}
```

Enumerado SelectorDeSolucion

```
public enum SelectorDeSolucion {

    SOLUCION_ALTERNATIVA {

        @Override
        public Solucion obtenerSolucion(FuenteDeDatos fuente) {
              return new SolucionAlternativa(fuente);
        }
```

```
        @Override
        public String getMensaje() {
              return "Solución Aproximada";
        }
    },
    SOLUCION_EXACTA {
        @Override
        public Solucion obtenerSolucion(FuenteDeDatos fuente) {
              return new SolucionExacta(fuente);
        }

        @Override
        public String getMensaje() {
              return "Solución Exacta";
        }
    };

    private static Map<String, SelectorDeSolucion> selectores = new
HashMap<String, SelectorDeSolucion>();

    static {

        selectores.put("A", SOLUCION_ALTERNATIVA);
        selectores.put("E", SOLUCION_EXACTA);
    }

    public static SelectorDeSolucion obtenerSelector(String tipoDeSolucion) {
        return selectores.get(tipoDeSolucion);
    }
```

```
    public abstract Solucion obtenerSolucion(FuenteDeDatos fuente);

    public abstract String getMensaje();

}
```

Clase Solucion

```
public abstract class Solucion{

    protected List<Float> elementos;
    private FuenteDeDatos fuente;
    private int cantidadDeEjecuciones;
    private List<Long> ejecuciones;

    public Solucion(FuenteDeDatos fuente) {
       this.fuente = fuente;
       ejecuciones = new ArrayList<Long>();
    }

    public List<Float> getElementos() {
       return elementos;
    }

    public abstract Integer getEnvases();

    public void ejecutar(int cantidadDeseada) {

       for (int i = 0; i < cantidadDeseada; i++) {
```

```
            this.elementos = fuente.obtenerDatos();

            long millisAntes = System.currentTimeMillis();
            aplicarAlgoritmo();

            ejecuciones.add(System.currentTimeMillis() - millisAntes);
            cantidadDeEjecuciones++;
      }

}

public abstract void aplicarAlgoritmo();

public Integer getCantidadDeEjecuciones() {
      return cantidadDeEjecuciones;
}

public List<Long> getEjecuciones() {
      return ejecuciones;
}

public Long getTiempoPromedio() {

      long tiempo = 0L;

      for (long ejecucion : ejecuciones) {
            tiempo += ejecucion;
      }
```

```
        return tiempo/cantidadDeEjecuciones;
    }


}
```

Clase SolucionAlternativa

```
public class SolucionAlternativa extends Solucion {

    private Map<Integer, Float> envases;

    public SolucionAlternativa(FuenteDeDatos fuente) {

        super(fuente);
        this.envases = new HashMap<Integer, Float>();
    }

    public SolucionAlternativa(List<Float> elementos) {

        super(null);
        this.elementos = elementos;
        this.envases = new HashMap<Integer, Float>(elementos.size());
    }

    @Override
    public Integer getEnvases() {
        return envases.size();
    }
```

```
    /**
     * Recorre uno por uno los elementos y los va ubicando siempre en el
último envase.
     * Dado que no vuelve a recorrer la lista de envases, y siempre trabaja
con el último,
     * y que agregar un nuevo envase, o evaluar si un elemento entra en un
envase, tienen costo constante,
     * el tiempo de ejecución de este algoritmo es lineal respecto de la
cantidad de elementos, O(n).
     */
    @Override
    public void aplicarAlgoritmo() {

       envases.clear();

       int envaseActual = 1;
       envases.put(envaseActual, 0F);

       for (Float elemento : elementos) {

             Float elementosEnElEnvase = envases.get(envaseActual);

             if (elementosEnElEnvase + elemento <= 1) {
                   envases.put(envaseActual, elementosEnElEnvase + elemento);
             } else {
                   envaseActual++;
                   envases.put(envaseActual, elemento);
             }
       }
    }
```

```
}
```

Clase SolucionExacta

```
public class SolucionExacta extends Solucion{

    private double[] espacioLibreEnvase;
    private boolean[][] estaEnvaseConItem;

    public SolucionExacta(FuenteDeDatos fuente) {
       super(fuente);
       this.espacioLibreEnvase = null;
    }

    public SolucionExacta(List<Float> elementos) {
       super(null);
       this.elementos = elementos;
       this.espacioLibreEnvase = new double[elementos.size()];
    }

    public SolucionExacta(List<Float> itemSize, int cantidadEnvases) {
       super(null);
       this.elementos = itemSize;
       this.espacioLibreEnvase = new double[cantidadEnvases];

       for (int i = 0; i < cantidadEnvases; i++) {
             this.espacioLibreEnvase[i] = 1.0F;
       }
```

```
        this.estaEnvaseConItem = new
boolean[cantidadEnvases][this.elementos.size()];
    }

    public List<Float> getItems() {
        return elementos;
    }

    public void setItems(List<Float> items) {
        this.elementos = items;
    }

    public double[] getEspacioLibreEnvase() {
        return espacioLibreEnvase;
    }

    public void setEspacioLibreEnvase(double[] espacioLibreEnvase) {
        this.espacioLibreEnvase = espacioLibreEnvase;
    }

    public boolean[][] getEstaEnvaseConItem() {
        return estaEnvaseConItem;
    }

    public void setEstaEnvaseConItem(boolean[][] estaEnvaseConItem) {
        this.estaEnvaseConItem = estaEnvaseConItem;
    }

    public void setCantidadEnvases(int cantEnvases) {
```

```
        this.espacioLibreEnvase = new double[cantEnvases];
        for (int i = 0; i < cantEnvases; i++) {
              this.espacioLibreEnvase[i] = 1.0F;
        }
        this.estaEnvaseConItem = new
boolean[cantEnvases][this.elementos.size()];
    }

    /**
     * Algoritmo de fuerza bruta para la inserción de un elemento en los
envases.
     * Orden O (m^n) donde n es la cantidad de elementos a insertar y m la
cantidad de
     */
    public boolean pack(int item) {
        // Mostrar la solución si terminamos
        if (item == elementos.size()) {
              return true;
        }

        // sino seguimos buscando los elementos
        for (int i = 0; i < espacioLibreEnvase.length; i++) {
              if (round(espacioLibreEnvase[i],2) >=
round(elementos.get(item),2)) {
                    estaEnvaseConItem[i][item] = true; // insertarlo en el
envase
                    espacioLibreEnvase[i]=round(espacioLibreEnvase[i] -=
elementos.get(item),2);
                    if (pack(item+1)) //Sigo con el proximo elemento
                          return true;
                    espacioLibreEnvase[i]=round(espacioLibreEnvase[i] +=
elementos.get(item),2); // No, entró el anterior, lo sacamos
```

```
                    estaEnvaseConItem[i][item] = false;
                }
            }
            return false;
        }

        private void showResults() {
            for (int i = 0; i < espacioLibreEnvase.length; i++) {
                System.out.println("bag" + i);
                for (int j = 0; j < elementos.size(); j++)
                    if (estaEnvaseConItem[i][j] == true)
                        System.out.println("item" + j + "(" +
elementos.get(j)
                                + ") ");
            }
        }

        /**
         * Redondeo para manejo de floats
         */
        public static double round(double value, int places) {
          if (places < 0) throw new IllegalArgumentException();
          BigDecimal bd = new BigDecimal(value);
          bd = bd.setScale(places, BigDecimal.ROUND_HALF_UP);
          return bd.doubleValue();
        }

        @Override
        /**
         * Cantidad de envases utilizados.
```

```
     */
    public Integer getEnvases() {
       Integer envases =0;
       if (this.espacioLibreEnvase != null)
             envases= this.espacioLibreEnvase.length;
       return envases;
    }

    /**
     * Aplica el algoritmo de la solucion exacta.
     * Va probando por fuerza bruta y de 1 en adelante la cantidad de envases a llenar para
     * obtener el mínimo.
     * Es de orden exponencial O(m x (n^n)) donde m es la cantidad de envases y n la cantidad de elementos
     */
    @Override
    public void aplicarAlgoritmo() {
       //Itero por la cantidad de envases hasta encontrar la que insume menos envases.
       //Asume que los elementos ya están cargados.
       boolean flag=false;
       if (!elementos.isEmpty()) {
             for (int cantEnvases=0; cantEnvases < this.elementos.size() && !flag; cantEnvases++) {
                   this.setCantidadEnvases(cantEnvases+1);
                   flag=this.pack(0);
             }
       }
    }
}
```