Práctica 03

Programación Dinámica: Una decisión bien tomada, un tesoro bien escogido.

Estructura de Datos y Algoritmos II

Rafael Expósito Mullor
Luis Díaz González
Universidad de Almería

INDICE

1. OBJETIVOS

- 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. INTRODUCCIÓN: PROGRAMACIÓN DINÁMICA

Existe una serie de problemas cuyas soluciones pueden ser expresadas recursivamente en términos matemáticos, y posiblemente la forma más natural de resolverlos es mediante un algoritmo recursivo. Sin embargo, el tiempo de ejecución de la solución recursiva, normalmente de orden exponencial, puede mejorarse substancialmente mediante la Programación Dinámica.

En el diseño Divide y Vencerás que vimos durante el primer tema de la asignatura veíamos como para resolver un problema lo dividíamos en subproblemas independientes, los cuales se resolvían de manera recursiva para combinar finalmente las soluciones y así resolver el problema original.
El inconveniente se presenta cuando los subproblemas obtenidos no son independientes, sino que existe solapamiento entre ellos, entonces es cuando una solución recursiva no resulta eficiente por la repetición de cálculos que eso conlleva.

La Programación Dinámica no sólo tiene sentido aplicarla por razones de eficiencia, sino porque además presenta un método capaz de resolver de manera eficiente problemas cuya solución ha sido abordada por otras técnicas habiendo fracasado. Donde tiene mayor aplicación la Programación Dinámica es en la resolución de problemas de optimización. En este tipo de problemas pueden aparecer distintas soluciones, cada una con un valor, y lo que se desea es encontrar la solución de valor óptimo (máximo o mínimo).

La solución de problemas aplicando esta técnica se basa en el principio enunciado por Bellman en 1957, el cual versa que:

“En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima”

Este principio puede parecer evidente pero no siempre es aplicable y, por tanto, es necesario verificar que se cumple para el problema en cuestión. Un ejemplo para el que no es aplicable este principio podría ser el caso de buscar el camino de coste máximo entre dos vértices de un grafo ponderado.

Para que un problema pueda ser abordado por esta técnica ha de cumplir las siguientes condiciones:

- La solución al problema ha de ser alcanzada a través de una secuencia de decisiones, una en cada etapa.

- Dicha secuencia de decisiones ha de cumplir el principio de Bellman que hemos citado anteriormente.

Por último, destacar que, en líneas generales, el diseño de un algoritmo de Programación Dinámica consta de los siguientes pasos:

1. Planteamiento de la solución como una sucesión de decisiones y verificación de que esta cumple el principio de Bellman.
2. Definición recursiva de la solución.
3. Cálculo del valor de la solución óptima mediante una tabla o matriz donde se almacenan soluciones a problemas parciales para reutilizar los cálculos.
4. Construcción de la solución óptima haciendo uso de la información contenida en la tabla anterior.

3. ESTUDIO DE LA IMPLEMENTACIÓN

La idea general del problema a resolver es la siguiente:

Una persona, a la que llamamos Pedrito, consigue llegar a un lugar concreto dentro de una cueva dónde se encuentra un tesoro. Esta persona se da cuenta que los objetos que conforman el tesoro no son muchos, pero sí puede apreciar que todos ellos son muy valiosos.
Dado su gran conocimiento sobre tesoros y al sopesar todos los objetos, puede tener una idea muy buena del peso y valor de cada uno de los objetos que conforman el tesoro.

Debido a lo peligroso que es el camino de vuelta y lo limitada que es la capacidad de su mochila, se enfrenta con la difícil tarea de que solo puede llevar consigo aquellos objetos que quepan en su mochila, la cual tiene una capacidad (peso que soporta) limitada.

Se pide implementar soluciones con el esquema de programación dinámica a los cuatro problemas que se plantean a continuación.

3.1. Problema 1: Mochila Clásica.

Suponiendo que los objetos no se pueden romper y que los pesos son exactos (número naturales positivos), determinar qué objetos debe elegir Pedrito para maximizar el valor total de lo que pueda llevarse en la mochila, sabiendo que los valores de los objetos que estima Pedrito son números reales positivos.

- Para entender cómo funciona el algoritmo que soluciona este problema e implementarlo posteriormente, vamos a plantear la siguiente hipótesis:

→ Contamos con una mochila de peso cuatro (P=4)
→ Valoramos cómo introducir 3 objetos, cada uno con un peso y valor distintos:

OBJETO 1 => p=1, v=20
OBJETO 2 => p=2, v=50
OBJETO 3 => p=3, v=60

Una vez contamos con estos datos debemos tener en cuenta que tenemos que crear una matriz de [N+1] filas y [P+1] columnas, siendo N el número de objetos y P el peso de nuestra mochila, resultando en lo siguiente:

CAPACIDAD						
OBJETOS		**0**	**1**	**2**	**3**	**4**
	0					
	1					
	2					
	3					

Nota: según la especificación del algoritmo, sabemos que el mejor valor será el que se encuentre en la posición marcada en verde en nuestra tabla.

PRIMER PASO

Inicializamos la matriz. Los valores de la fila y columna 0 serán ceros. Si lo pensamos, esto es porque no podemos valorar el introducir ningún objeto cuando la capacidad de nuestra mochila es cero o bien el caso en el que no estamos introduciendo ningún objeto y por tanto no estamos haciendo nada.

CAPACIDAD						
OBJETOS		**0**	**1**	**2**	**3**	**4**
	0	0	0	0	0	0
	1	0				
	2	0				
	3	0				

SEGUNDO PASO

El algoritmo irá rellenando la matriz por filas, introduciendo en cada posición el valor máximo entre dos posibles valores: aquel que haya en la fila superior de la columna en la que nos encontramos y aquel que corresponde a esa fila más el valor en la fila anterior desplazado en columna hacia la izquierda según su peso.

Describir el procedimiento es más complicado que ponerlo en práctica gráficamente. Para simplificarlo, podemos denotar el procedimiento de la siguiente forma:

M[i - 1] [j]

V[i] + M[i - 1] [j - p[i]]

MAX

Siendo M nuestra matriz, V el valor del objeto que queremos introducir y p el peso del objeto.

Vamos a ejemplificar cómo se introduciría el primer elemento al trabajar con primer objeto con p=1 y v=20.

M[1-1][1] = M[0][1] = 0
V[i] + M[1-1][1-1] = 20 + 0

MAX (0, 20) = 20

CAPACIDAD						
OBJETOS		**0**	**1**	**2**	**3**	**4**
	0	0	0	0	0	0
	1	0				
	2	0				
	3	0				

Además, es importante comprobar el peso en cada caso, es decir, en este mismo ejemplo cuando queramos introducir un tercer objeto, pero nos encontremos en una situación en la que la capacidad de la mochila es menor, lógicamente no podremos introducir ese objeto y nuestra matriz permanecerá con los valores previos.

De esta forma, la matriz final sería la siguiente:

CAPACIDAD						
OBJETOS		**0**	**1**	**2**	**3**	**4**
	0	0	0	0	0	0
	1	0	20	20	20	20
	2	0	20	50	70	70
	3	0	20	50	70	80

Hasta este momento ya podríamos asegurar lo siguiente: en cada celda tenemos la solución óptima para una mochila de capacidad y número de objetos concreta. Para una mochila de capacidad 4 y con los 3 objetos de este ejemplo, el resultado óptimo está en la celda marcada en verde.

DETERMINAR QUÉ OBJETOS SE ELIGEN (RESCATAR ELEMENTOS DE LA MATRIZ)

Ya sabemos el valor máximo que podemos conseguir con los 3 objetos de los que disponemos, pero no qué objetos van a introducirse en la mochila finalmente pues no todos ellos caben.

El procedimiento es el siguiente:

1. Empezamos a comprobar desde la última posición. En nuestro caso la posición (3, 4) marcada en verde.

2. Vamos comparando valores. Si el valor de la fila superior **es distinto**, ese objeto se introduce en la mochila y, a continuación, nos desplazamos el peso del objeto hacia la izquierda.

3. Si **no es distinto** seguimos comprobando hacia arriba.

CAPACIDAD						
OBJETOS		**0**	**1**	**2**	**3**	**4**
	0	0	0	0	0	0
	1	0	20	20	20	20
	2	0	20	50	70	70
	3	0	20	50	70	80

Siguiente el recorrido marcado:

80 != 70 → Se introduce el OBJETO 3 (p=3, v=60) y nos desplazamos el peso del objeto a la izquierda.
20 = 20 → El OBJETO 2 NO se introduce y miramos en la posición de arriba.
20 != 0 → Se introduce el OBJETO 1 (p=1, v=20).

3.2. Problema 2. Mochila Ilimitada.

Pedrito descubre que hay otra cueva mucho más grande, cuyos objetos no alcanza a contar. Lo que si puede observar es hay una cantidad inagotable de objetos preciosos de tipos (clases) distintos. Cada objeto de un tipo es indivisible (no se puede romper), tiene un cierto peso (natural positivo) y un cierto valor (real positivo) que Pedrito conoce perfectamente.
Determinar qué cantidad de objetos de cada tipo debe coger Pedrito para maximizar el valor total de lo que puede llevarse en su mochila.

- Para resolver este problema debemos tener claro que, a diferencia de lo que ocurría en el algoritmo anterior, ahora va a ser posible introducir varias veces un mismo objeto.
En este caso, en lugar de utilizar una matriz usaremos un array, siendo la capacidad de la mochila más 1 el tamaño de esta estructura.

Para entender cómo funciona el algoritmo que soluciona este problema e implementarlo posteriormente, vamos a plantear la siguiente hipótesis:

1. Tenemos una mochila de capacidad 5
2. Valoramos como y cuantos objetos introducir partiendo de estos dos tipos:

   OBJ 1 => p= 1, v =5
   OBJ 2 => p=2, v =11

PRIMER PASO

Partimos de un array de tamaño = 6 (capacidad mochila + 1) con un valor de cero en cada posición.

0	1	2	3	4	5
0	0	0	0	0	0

SEGUNDO PASO

Lo primero que debemos tener claro es que la posición cero del array siempre permanecerá con el mismo valor, pues es imposible introducir ningún objeto en una mochila si no tiene capacidad.

Para ir rellenando el array debemos tener en cuenta que el peso del objeto que estamos tratando de introducir sea menor o igual a la capacidad de la mochila.
A continuación, iremos mirando si el valor del peso del objeto más el valor que haya en la posición desplazada el valor del peso a la izquierda es mayor que el valor de la posición actual.

Esto podría denotarse de la siguiente manera:

A[i]
V[i] + A[i-p[i]] } MAX

Siendo A la posición en nuestra matriz, V el valor del objeto que queremos introducir y p el peso del objeto.

Mostramos gráficamente este procedimiento para las dos primeras posiciones del array:

Recordamos que:

OBJ1 (p=1, v=5)	OBJ2 (p=2, v=11)

CAPACIDAD MOCHILA

0	1	2	3	4	5
0	0	0	0	0	0

Para OBJ1 → ¿Es mayor el valor en A[1] o el valor en A[1-1] más el valor del OBJ1 v=5?
(0+5) > 0, introducimos el valor 5 en la primera posición.

CAPACIDAD MOCHILA

0	1	2	3	4	5
0	5	0	0	0	0

Como para una capacidad de 1 no es posible introducir más objetos, el objeto 2 no se comprueba y pasamos a mirar la siguiente posición (A[2]) del array, la cual indica una capacidad 2 de la mochila.

Para OBJ1 → ¿Es mayor el valor en A[2] o el valor en A[2-1] más el valor del OBJ1 v=5?
(5+5) > 0, introducimos el valor 10 en la segunda posición.

CAPACIDAD MOCHILA

0	1	2	3	4	5
0	5	10	0	0	0

Para OBJ2 → ¿Es mayor el valor ahora presente en A[2] o el valor en A[2-2] más el valor del OBJ1 v=11?
(0+11) > 10, introducimos el valor 11 en la segunda posición, sustituyendo el valor anterior.

CAPACIDAD MOCHILA

0	1	2	3	4	5
0	5	11	0	0	0

.
.
.

Siguiendo con este procedimiento llegamos a este array final:

CAPACIDAD MOCHILA

0	1	2	3	4	5
0	5	11	16	22	27

→ Hasta este punto ya sabemos que el beneficio máximo que podemos obtener es 27.

DETERMINAR QUE OBJETOS SE ELIGEN (RESCATAR ELEMENTOS DEL ARRAY)

Iremos recorriendo los objetos mientras que la capacidad de la mochila tiene sea superior o igual al peso del objeto menor, que será el que marque cuando nos quedamos sin espacio en la mochila.

De esta forma, podría quedarse espacio sin ocupar pero no caber más objetos. Por ejemplo, al tener una mochila a la que le queda una capacidad libre de 3 pero el objeto más pequeño que podemos introducir tiene peso 5.

La idea es ir valorando primero los objetos que más peso tienen, ya que estos son lo que mayor valor suelen tener. Si la capacidad de la mochila menos el peso del objeto que estamos valorando es mayor o igual que cero significa que sí entra en la mochila y podríamos considerarlo.

Utilizamos una variable que nos dice el valor que obtendría si cogiese lo que hay en el array en la posición capacidad de la mochila restante menos el peso del objeto, más el valor del objeto (A[c-p[i] + v[i]), lo comparamos con el valor de otra variable que nos indica el valor máximo, si es mayor actualizamos ese valor máximo y guardamos la posición del objeto.
Repetimos este proceso con los demás objetos hasta encontrar la mejor opción, introducirla en la mochila y decrementar la capacidad de esta.

En nuestro ejemplo:

0	1	2	3	4	5
0	5	11	16	22	27

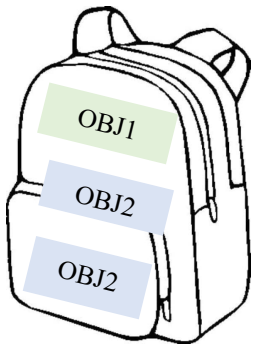
OBJ1 (p=1, v=5) OBJ2 (p=2, v=11)

Capacidad inicial de la mochila = 5

Cogeríamos el objeto de mayor peso (OBJ2), como su peso es 2 y 2<5 podemos introducir el objeto y decrementamos la capacidad de la mochila (5-2=3)

Volveríamos a coger el OBJ2, como su peso es 2 y 2<3 podemos introducir el objeto y decrementamos la capacidad de la mochila (3-2=1)

Como la capacidad de la mochila sigue siendo superior o igual al peso del objeto menor (OBJ1) podemos continuar introduciendo elementos. Esta vez cogeríamos el OBJ1, y la mochila quedaría llena.



3.3. Problema 3. Mochila con pesos decimales.

Suponiendo que los objetos no se pueden romper, pero los pesos pueden no ser exactos (los pesos pueden ser número reales positivos). Indicar cómo habría que modificar el algoritmo del primer aparatado para que Pedrito se pudiera llevar en su mochila el máximo de riquezas.

- Una de las limitaciones del algoritmo clásico de la mochila es que solo permite ser aplicado cuando el peso de los objetos es de tipo entero.
Una idea sería tratar los pesos de tipo entero despreciando la parte decimal, es decir redondeando, pero eso haría que nuestra solución no fuese óptima sino aproximada, que no es lo que pretendemos inicialmente para resolver este problema.

Tras analizar la situación, vemos con un ejemplo como podríamos plantear una modificación para el algoritmo clásico. Supongamos que tenemos una capacidad de 20’25 y que disponemos de tres objetos con los siguientes pesos: 5’72, 3’67, 8, 4’1. Todos estos valores se pueden expresar con el mayor número de decimales de forma que tendríamos:

[20’250, 5’720, 3’670, 8’000, 4’100]

Multiplicando todos los valores por un factor 10^x, donde x sea el número máximo de decimales, el resultado sería el siguiente:

[20250, 5720, 3670, 8000, 4100]

Ahora que los valores no tienen decimales, podemos utilizarlos como números enteros y aplicar el mismo algoritmo que definimos en el problema 1 con alguna que otra modificación.

Será necesario implementar un método para determinar cuántos decimales tienen como máximo todos los valores, el cual tendrá una estructura similar a la siguiente:

```
private int numDec(double d) {
        if(d == (int)d) return 0;
        String text = Double.toString(Math.abs(d));
        int enteros = text.indexOf('.');
        if(enteros == -1) return 0;
        int decimales = text.length() - enteros- 1;
        return decimales;
}
```

Con el método anterior ya podemos saber cuántos decimales tiene un número, ahora tenemos que obtener el máximo número de decimales entre el peso o la capacidad de la mochila y todos los objetos. Para ello el método utilizado tiene una estructura parecida a la siguiente:

```
private int maxNumDec() {
        int max = numberOfDecimals(capacity);
        for (Object ob : objects) {
                int n = numberOfDecimals(ob.getWeight());
                max = n > max ? n : max;
        }
        return (int) Math.pow(10, max);
}
```

Si aplicamos este método al cálculo de la capacidad y peso de los objetos en el algoritmo clásico conseguimos resolver este problema.

3.4. Problema 4. Mochila fraccionaria.

Imaginemos el caso de que los objetos valiosos que encuentra Pedrito en la cueva se pueden romper (fraccionar), ¿sería sencillo encontrar una solución con programación dinámica para que Pedrito determine qué objetos elegir para maximizar el valor total de lo que puede llevarse en su mochila? ¿Sería más sencillo realizarlo con un algoritmo greedy?

De ser así, implementarlo y exponer las principales diferencias entre los dos esquemas algorítmicos (programación dinámica y greedy).

- En clase hemos visto que existen dos algoritmos de la mochila: la mochila 0-1 donde los elementos son indivisibles y hay dos posibles opciones, escoger o no escoger cada elemento. Por otro lado, tenemos la mochila 0/1 o “fraccionaria” donde los elementos sí son divisibles y se pueden escoger partes de ellos.

Al igual que para resolver el problema de la mochila clásica la mejor alternativa es utilizar la programación dinámica, si buscamos obtener la mejor y más sencilla solución posible al problema de la mochila fraccionaria la mejor opción es aplicar el esquema voraz.

Las principales diferencias entre el esquema de programación dinámica y voraz las mostramos en la siguiente tabla:

CRITERIO	MÉTODO VORAZ	MÉTODO DINÁMICO
Ocurrencia	Se elige la opción que parece la mejor en el paso actual para construir una solución óptima.	Se considera la solución del problema actual y la solución de los subproblemas previamente resueltos para construir una solución global.
Toma de decisiones	Genera una única secuencia de decisión. Por lo tanto, el resultado del algoritmo puede depender de las elecciones realizadas hasta ahora y no de las elecciones futuras o de todas las soluciones al subproblema.	Genera muchas secuencias de decisión, ya que considera todos los casos posibles evaluando los subproblemas al principio, para obtener el resultado óptimo.
Cálculo de la solución	Calculamos la solución de manera iterativa o hacia adelante sin volver a visitar o retroceder a las opciones o soluciones anteriores.	Construye su solución siguiendo un enfoque ascendente o descendente sintetizándolos a partir de sub-soluciones óptimas más pequeñas.

Tabla. Esquema voraz vs Programación dinámica.

Para resolver el problema hay que plantear el hecho de que es necesario seguir una **heurística**, la cual será la **relación entre el valor y el peso del objeto**. Poniendo un ejemplo, esto sería:

OBJETO	PESO	VALOR	RELACIÓN
1	50	109	v/p= 2’18
2	30	59	v/p= 1’96
3	20	45	v/p= 2’25

Tabla. Ejemplo de objetos con su relación v/p.

De esta forma, ya no cogeríamos simplemente los objetos de mayor valor, sino los objetos con mayor relación valor/peso. Además de esto, podemos ordenar los elementos de forma descendente por esta relación para que los mejores objetos de encuentren primero.

Para aplicar esto usaremos un atributo “cantidad”, estableciéndola a 1 si representa el total del objeto o entre 0 y 1 si escogemos una parte del objeto.

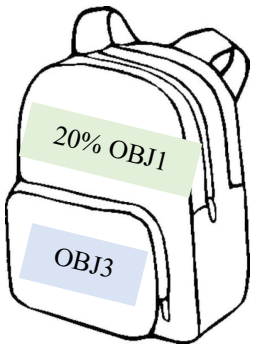
Siguiendo con el ejemplo de la tabla anterior, para una capacidad de mochila de 22, lo que haríamos sería lo siguiente:

1. Ordenamos los objetos de forma descendente según su relación valor/peso

OBJ3 (2’25) > OBJ1 (2’18) > OBJ2 (1’96)

2. Cogemos el objeto con mayor relación (OBJ3) y comprobamos que su peso (20) es menor que la capacidad de la mochila (30). Añadimos el objeto a la mochila.

3. La capacidad de la mochila ahora es de 30-20=10. Cogemos el siguiente objeto (OBJ1) y comprobamos su peso (50) y el de la mochila (10). Como el objeto no centra en nuestra mochila, lo fraccionamos hasta que tenga un peso 10 (10/50) y lo introducimos.



4. ESTUDIO TEÓRICO

- En esta sección vamos a intentar analizar los órdenes de complejidad teóricos de los distintos algoritmos para así comparar cuál se comportaría de una forma más eficiente de forma general. A continuación, se muestra el algoritmo que resuelve el **problema 1 (mochila clásica) y el problema 3 (mochila con pesos decimales)** para ir viendo el recorrido más claro. Este algoritmo incluye el uso del *mcd* para intentar reducir el tamaño de la matriz resultante lo máximo posible en ambos casos.

```
public ArrayList<Objeto> mochila() {
	this.objetos.sort(null);  → → → nlogn
	int n = this.objetos.size();
	int dec = numeroMaximoDecimales();
	int mcd = mcd(dec);
	double[][] B = new double[n + 1][(capacidad * dec)/mcd + 1];
	for (int i = 1; i <= n; i++) { → → → ∑
		for (int j = 1; j <= (capacidad * dec)/mcd; j++) { → → → ∑
		if ((this.objetos.get(i - 1).getPeso() * dec)/mcd <= j) {
			B[i][j] = Math.max(B[i - 1][j], this.objetos.get(i - 1).getValor()
				+ B[i - 1][j - (int) (this.objetos.get(i - 1).getPeso() * dec)/mcd]);
		} else {
			B[i][j] = B[i - 1][j];
		}
		}
	}
		return RecuperarObjetos(B, mcd, dec); → → → n
}
```

El algoritmo se compone de una ordenación que ya sabemos que es de O(nlogn), dos bucles for anidados que tendremos que resolver mediante una sumatoria y una llamada al método que se encarga de elegir los objetos, formado por un bucle for (O(n)). Todas las demás operaciones son de O(1).

Para resolver la sumatoria en el caso de los dos bucles for anidades mencionados hace un momento realizamos el siguiente cálculo:

$$c + \sum_{i=1}^{n}(\sum_{j=1}^{cap} c) = c + \sum_{j=1}^{cap}(cap - 1 + 1)c = c + \sum_{j=1}^{cap} cap * c = \; c + (n - 1 + 1)cap * c = c + n * cap * c \rightarrow O(n * cap)$$

De esta forma tenemos que: **T(n) = nlogn + nCap + n**.
Por tanto, en el mejor de los casos el orden sería nlogn y en el peor (capacidad de la mochila muy grande) podría llegar a ser n^2. Por norma general se suele indicar el orden de los algoritmos en el peor de los casos, por lo que afirmaríamos que el algoritmo es finalmente de **$O(n^2)$.**

Para el caso del **problema 2 (mochilaIlimitada)** donde tenemos recursos ilimitados de cada objeto, la estructura del algoritmo es similar, pero rellenando un array en lugar de una matriz, por tanto, llegamos a la conclusión de que el orden de complejidad es también similar, **$O(n^2)$** para el peor caso. A continuación, mostramos el código principal del algoritmo de mochila ilimitada para mostrar su estructura:

```
public ArrayList<Objeto> mochilaIlimitada() {
	this.objetos.sort(null); → → → nlogn
	int n = this.objetos.size();
	double[] array = new double[capacidad + 1];
	for (int i = 0; i <= capacidad; i++) {
		for (int j = 0; j < n; j++) {
			if (this.objetos.get(j).getPeso() <= i) {
				array[i] = Math.max(array[i],
				array[i - (int) this.objetos.get(j).getPeso()] +
				this.objetos.get(j).getValor());
			}
		}
	}
		return recuperarIlimitados(array);
}
```

Por último, para el caso del algoritmo que resuelve el **problema 4 con el esquema voraz (mochilaGreedy)** la estructura se muestra en la siguiente página:

```
public ArrayList<Objeto> mochilaGreedy() {
	this.objetos.sort(new ComparadorGreedy()); → → → nlogn
	valorFinal = 0;
	pesoFinal = 0;
	ArrayList<Objeto> resultados = new ArrayList();
	for (Objeto obj : objetos) { → → → → → → → → → n
		if (pesoFinal + obj.getPeso() <= capacidad) {
			obj.setCantidad(1);
			resultados.add(obj);
			pesoFinal += obj.getPeso();
			valorFinal += obj.getValor();
			if (pesoFinal == capacidad)
				break;
		} else {
			obj.setCantidad((capacidad - pesoFinal) /
obj.getPeso());
			resultados.add(obj);
			pesoFinal += obj.getPeso() * obj.getCantidad();
			valorFinal += obj.getValor() * obj.getCantidad();
			break;
		}
	}
	return resultados;
}
```

Como puede observarse, contamos con un proceso de ordenación (O(nlogn)) y un bucle for que contiene operaciones de O(1), dando como resultado total un O(n). Teniendo todo esto en cuenta podríamos decir que el **orden teórico del algoritmo Greedy es O(nlogn).**

Concluyendo, podemos decir que el algoritmo Greedy será más rápido y eficiente que los dinámicos, y esto tiene sentido puesto que estos últimos evalúan todas las posibilidades para encontrar la solución óptima.

5. ESTUDIO EXPERIMENTAL

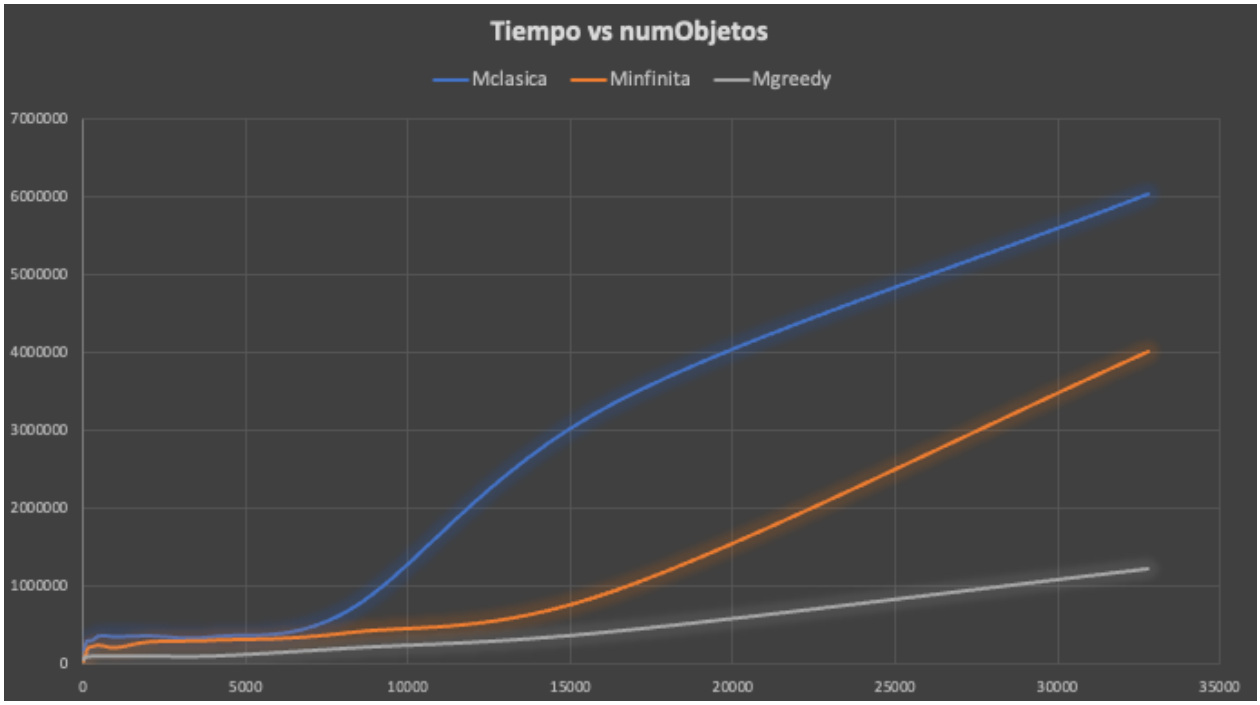
- En esta sección vamos a realizar una comparación de los tiempos obtenidos en la ejecución de los algoritmos en distintas situaciones.
Para hacer estas pruebas hemos creado un constructor de mochilas, de tal forma que le pasaremos una capacidad y un número de objetos por parámetro, además de crear nuevos objetos con un peso y valor aleatorios. La estructura de este constructor es la siguiente:

```
public Mochila(int capacidad, int numObjetos) {
	this.capacidad = capacidad;
	this.objetos = new ArrayList<Objeto>();
	for (int i = 1; i <= numObjetos; i++) {
		int peso = (int) (Math.random()*capacidad+1);
		int valor = (int) (Math.random()*(capacidad*2)+1);
		this.objetos.add(new Objeto("objeto"+i,peso , valor));
	}
}
```

Para una primera prueba, en nuestra clase *MainTiempos* creamos una mochila de capacidad fija de 80 y con un total de objetos variable mediante un bucle for. El código utilizado para esta medición es básicamente el siguiente:

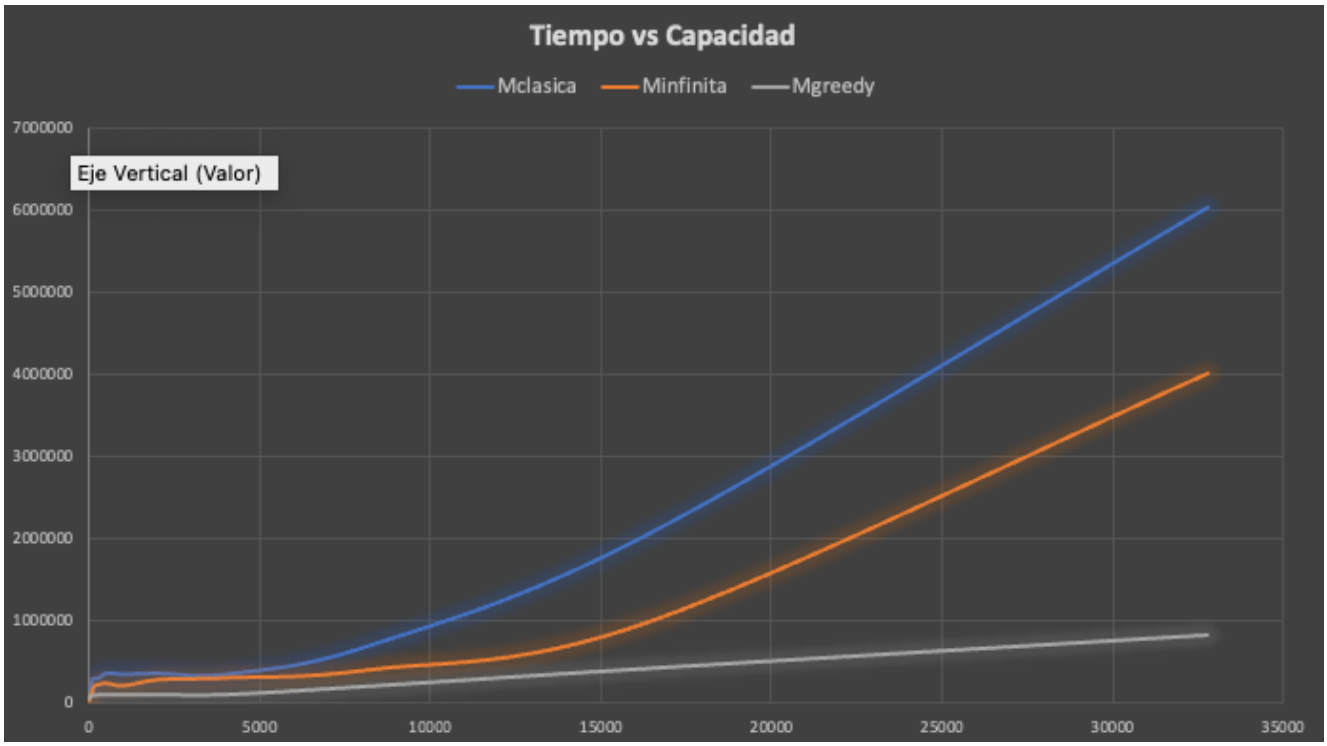
```
for (int i = 1; i <= 8192; i = i * 2) {

            Mochila m = new Mochila(80 ,i);

            empieza = System.nanoTime();
            m.mochila();
            acaba = System.nanoTime();
            tMochilaC = acaba - empieza;
            empieza = System.nanoTime();
            m.mochilaIlimitada();
            acaba = System.nanoTime();
            tMochilaI = acaba - empieza;
            empieza = System.nanoTime();
            m.mochilaGreedy();
            acaba = System.nanoTime();
            tMochilaG = acaba - empieza;
}
```

Los resultados obtenidos se reflejan en la siguiente gráfica:



Gráfica 2. Tiempos obtenidos para una mochila de capacidad fija y número de objetos variables.

Para una segunda prueba, en nuestra clase *MainTiempos* creamos una mochila de capacidad variable mediante un for y con un total de objetos fijo de 80.
Los resultados obtenidos se muestran en la siguiente gráfica:



Gráfica 2. Tiempos obtenidos para un mismo número de objetos y distintas capacidades de mochila.

Como conclusión, en ambas gráficas vemos como se la tendencia de los algoritmos es cumplir con los órdenes descritos en el estudio teórico, siendo el algoritmo Greedy el más rápido como ya estaba previsto.

6. ANEXO

Lista de archivos entregados para esta práctica

1. **Objeto.java** → Representa los objetos de la mochila.
2. **Mochila.java** → Se resuelven los distintos problemas planteados con el algoritmo de la mochila.
3. **ComparadorGreedy.java** → Clase utilizada por el algoritmo greedy. Creada para ordenar descendentemente los objetos por su relación valor/peso.
4. **Main.java** → Clase para comprobar el correcto funcionamiento del algoritmo de la mochila con los archivos de datos dados.
5. **MainTiempos.java** → clase creada para la medición de tiempos de ejecución de los algoritmos.
6. **TestPractica03.java** → test Junit.
7. **Archivos de prueba.txt**→ p01_x.txt y p02_x.txt
7. **Practica03 – programación Dinámica.pdf** → Este mismo informe.

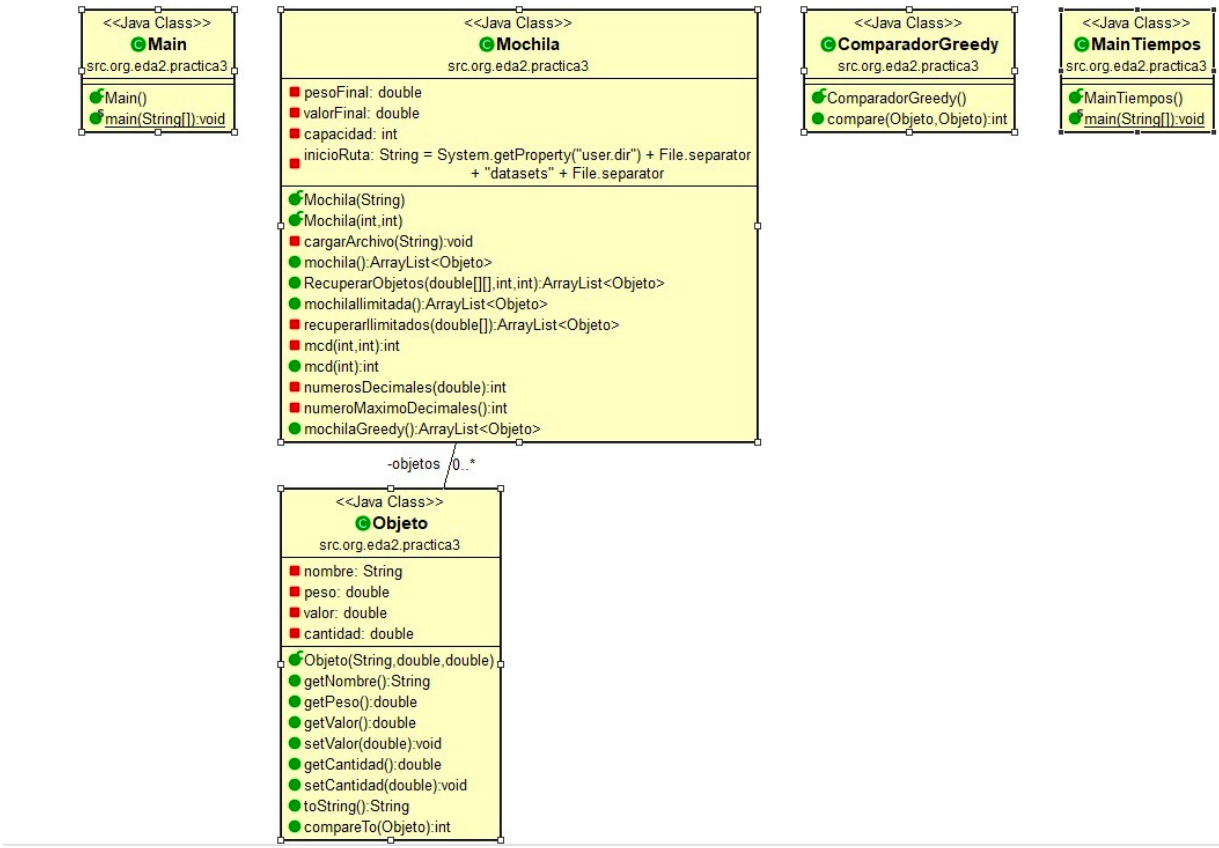


Imagen. Diagrama de clases.

7. Fuentes

[1] Programación dinámica en Java, disponible en https://pharos.sh/programacion-dinamica-en-java/

[2] The *unbounded* knapsack problem, disponible en http://www.mathcs.emory.edu/~cheung/Courses/253/Syllabus/DynProg/knapsack2.html

[3] Programación dinámica y su aplicación a diferentes problemas, disponible en http://www.lcc.uma.es/~av/Libro/CAP5.pdf

[4] Diferencia entre programación codiciosa y dinámica, disponible en https://aprendiendoaprogramar.es/blog/diferencia-entre-programacion-codiciosa-y-dinamica/

[5] Comparación de técnicas algorítmicas básicas con el problema de la Mochila, disponible en https://elbauldelprogramador.com/comparacion-de-tecnicas-algoritmicas-basicas-con-el-problema-de-la-mochila/