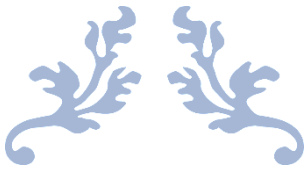

PRÁCTICA 2: ALGORITMOS APROXIMADOS: SELECCIÓN DE ACTIVIDADES PONDERADA

Algoritmos Avanzados. Grado en Ingeniería Informática.

10 DE NOVIEMBRE DE 2017
JORGE ARANDA GARCÍA
JOSE VICENTE BAÑULS GARCÍA

ÍNDICE

1. INTRODUCCIÓN

El objetivo de esta práctica es permitir a los alumnos una profundización en los algoritmos aproximados, el porqué son necesarios, así como su comparación con los algoritmos voraces. El problema propuesto en esta ocasión es similar al propuesto en la práctica anterior, pero en este caso se ha incluido un array en el que se indica un beneficio que se obtendría si se llevase a cabo la actividad correspondiente con ese índice. Precisamente al incluir ese array de beneficios, la función de maximización consiste en realizar un número de actividades compatibles que proporcionen el mayor beneficio posible. Para ello se han propuesto dos algoritmos voraces, que serán comparados con tres de aproximación. Los cinco serán detallados a continuación.

2. ENTORNO DE DESARROLLO

De nuevo, como sucedió en la anterior entrega, nos hemos decantado por Eclipse Java Neon y la versión utilizada de JDK ha sido JDK 1.8.0_60. La alta familiarización que tenemos con esta herramienta, además del buen resultado que obtuvimos en la práctica 1, nos han hecho seguir utilizándola, a pesar de la necesidad de tener que crear una clase Main para ejecutar el código, algo que no es necesario en la herramienta BlueJ. Además, debido a la necesidad de trabajar en paralelo, bien en la edición, bien en la edición del código hemos decidido utilizar una herramienta de control de versiones. En este caso hemos utilizado GitHub para el repositorio y SourceTree como herramienta de visualización de la misma debido a su facilidad de uso, así como de la integración que tiene con Eclipse.

3. ALGORITMOS

3.1. Algoritmo Voraz 1. Función de selección: Coger el máximo beneficio

El primer algoritmo propuesto tiene una función de selección que consiste en lo siguiente. Se coge la actividad que tenga mayor beneficio, la cual siempre va a formar parte de la solución. A partir de aquí se intenta coger la siguiente actividad de máximo beneficio, si es compatible con las seleccionadas anteriormente será parte de la solución final. El beneficio de la actividad seleccionada se suma al beneficio parcial.

Para ello, se ordena el array de beneficios mediante un array de índices. El método de ordenación escogido ha sido ShellShort (la justificación se encontrará en el anexo).

Usamos el método de ordenación para poder ir directamente a por la actividad de mayor beneficio, sin tener que recorrer el array de una forma tal que nos obligase a comprobar si la actividad candidata era compatible y además de máximo beneficio. Cuando se han explorado todas las actividades se muestra el beneficio total. La justificación del uso de esta función de selección es que al hablar de maximización lo primero que se viene a la cabeza es coger las actividades con mayor beneficio así como que cabe destacar que, a diferencia de la práctica en la que se iban escogiendo las actividades de forma que solo había que comprobar si la actividad candidata se solapaba justamente con la anterior seleccionada, en este caso al no haber ningún tipo de orden de tiempo de las actividades, es necesario comprobar que no se solape con ninguna de las escogidas anteriormente. Para ello se implementa una función auxiliar que se encarga de ello. La cabecera de este método auxiliar es:

```
private boolean esCompatible(int [] c, int [] f, boolean[] sol, int act){…}
```

La cual compara la tarea candidata con todas las que ya están incluidas en la solución. Esto se traduce en que si es la primera, va a ser escogida dado que será compatible con todas las seleccionadas (ninguna). Por motivos de eficiencia, forzamos dicho caso seleccionando la actividad de mayor beneficio directamente, ahorrándonos con ello un ciclo por el Array de soluciones.

A posteriori, pese a analizar cada vez el array completo, solo comprobamos la compatibilidad con los escogidos, por lo que las comparaciones para evitar los solapamientos se reducen a tan solo los que realmente importan.

De permitiremos utilizar una estructura de datos mas compleja, como podría ser un AllayList, el vector solución tendría un tamaño variable, no fijo como el array, reduciendo la complejidad considerablemente.

La cabecera del método que se corresponde con este algoritmo es:

```
public int seleccionActividadesPonderadoVoraz (int [] c, int [] f, int [] b){…}
```

Donde c, es el array de tiempos de comienzos, f, el array de tiempos de fin y b el array de beneficios de cada actividad.

Cabe destacar que la complejidad de este método depende directamente de los dos bucles anidados que hay. El primero es el que selecciona la actividad de mayor beneficio, el segundo, dentro de la función esCompatible(…), que comprueba que la actividad candidata sea compatible con todas las seleccionadas. Por lo tanto, la complejidad es de $O(n^2)$.

3.2. Algoritmo Voraz 2. Función de selección: Coger la máxima tasa de beneficio/duración

Para esta actividad la función de selección consiste en escoger la actividad que tenga mayor tasa de beneficio/duración. Estas tasas se almacenan en un array, en la que cada posición i del array almacena lo siguiente: b[i]/f[i]-c[i].

Este array se ordena como se ha hecho en el apartado anterior para no tener que comprobar si es el que tiene la mayor tasa, simplemente se accede a la actividad deseada. Si ésta es compatible con todas las actividades seleccionadas anteriormente, o si es la primera, se selecciona y se almacena en el array de solución final. La justificación de esta función de selección consiste en ofrecer una visión distinta que no dependa solo del beneficio, sino que también se tenga en cuenta de alguna forma la duración de las actividades. Además, queríamos comprobar, como ocurre en otros problemas como en el problema de la mochila, en el que al ir escogiendo el objeto con mayor tasa beneficio/peso, se obtenía la solución óptima. En efecto, en algunos casos esta función de selección obtiene la solución óptima, aunque cabe destacar que ni este algoritmo ni el anterior garantizan la solución óptima y pueden dar salidas totalmente distintas para una misma entrada. Por ejemplo:

Método	Datos de entrada	Beneficio	Actividades seleccionadas
seleccionActividadesPonderadoVoraz	{0,2,4,1,7,6}, {3,7,6,5,9,8}, {2,7,4,4,2,2};	9	{1,5}
seleccionActividadesPonderadoVoraz2	{0,2,4,1,7,6}, {3,7,6,5,9,8}, {2,7,4,4,2,2};	8	{0,2,5}

Aquí se puede observar que el método que efectivamente devuelve la solución óptima es seleccionActividadesPonderadoVoraz, es decir el que selecciona aquellas actividades con mayor beneficio. Hay que tener en cuenta que obtiene una solución óptima incluso escogiendo menos cantidad de actividades que el segundo algoritmo.

Sin embargo, se puede dar el caso contrario, aquí se muestra un ejemplo que lo prueba:

Método	Datos de entrada	Beneficio	Actividades Seleccionadas
seleccionActividadesPonderadoVoraz	{0,0,1,3,4,5}, {5,1,3,4,5,8}, {60,50,50,50,50,50}	110	{0,5}
seleccionActividadesPonderadoVoraz2	{0,0,1,3,4,5}, {5,1,3,4,5,8}, {60,50,50,50,50,50}	250	{1,2,3,4,5}

En este caso, se puede observar cómo el algoritmo que obtiene una solución óptima es seleccionActividadesPonderadoVoraz2, el algoritmo que escoge las actividades con mayor tasa beneficio/duración, y cómo el primer algoritmo no es capaz de obtener una solución ni siquiera cercana.

La cabecera que se corresponde con este algoritmo es la siguiente:

```
public int seleccionActividadesPonderadoVoraz2 (int []c, int [] f, int [] b){…}
```

donde los array que se le pasan como parámetros son exactamente los mismos que para el método anterior.

La complejidad de este algoritmo, cabe destacar la similitud que tiene con el anterior, exceptuando algún cálculo aritmético, es de $O(n^2)$. Hay dos bucles anidados uno que recorre las actividades candidatas y otro que comprueba si dicha actividad candidata es compatible con todas las seleccionadas anteriormente.

3.3. Algoritmo aproximado 1: Función de aproximación: Coger la actividad de máximo beneficio, comprobando que empiece después de que haya terminado la anterior seleccionada

En este momento comienza el real objetivo de la práctica. La implementación de algoritmos aproximados. Estos algoritmos son necesarios porque hay problemas que debido a su complejidad o debido al propio tamaño del problema, los algoritmos voraces no son capaces de abarcarlos, por ello nos conformamos con soluciones cercanas a la óptima.

El primer algoritmo aproximado propuesto, va a ser muy parecido al primer voraz. El algoritmo utiliza ShellShort para ordenar el array de beneficios y se va escogiendo la actividad de mayor beneficio. Hasta aquí todo igual, el verdadero cambio viene ahora. Con la intención de escoger el número de actividades escogidas, se limita dicha elección a coger aquellas que tengan un comienzo mayor al tiempo de finalización de la escogida anteriormente. De este modo algunas actividades que serían escogidas por el algoritmo voraz, no se tendrían en cuenta.

La cabecera que se corresponde con este algoritmo es:

```
public int seleccionActividadesPonderada(int [] c, int [] f, int [] b){…}
```

La complejidad de este método depende únicamente del método de ordenación escogido que en ningún caso va a ser superior a $O(n^2)$ y normalmente es inferior, lo cual

supone una mejora sustancial con respecto al algoritmo voraz y más tratándose de tamaños de entrada grandes, para los que se usan estos algoritmos.

3.4. Algoritmo aproximado 2: Función de aproximación: Coger la actividad con mayor tasa beneficio/duración y comienzo mayor al tiempo de fin de la anterior seleccionada

El segundo algoritmo se relaciona con el segundo voraz. Sus funciones de selección son exactamente las mismas, se escoge la actividad con mayor tasa beneficio/duración. Sin embargo, como ocurre en el apartado anterior, para el algoritmo aproximado solo se seleccionan aquellas que comiencen después de que la seleccionada anteriormente haya terminado.

Para ello se ordenan las actividades utilizando ShellShort para no tener que comprobar cada vez si la actividad candidata es realmente la que se quiere mirar.

La cabecera del método para este algoritmo es:

```
public int seleccionActividadesPonderada2(int []c, int [] f, int [] b){…}
```

Como ocurre en el apartado anterior la complejidad se reduce puesto que depende del método de ordenación. Como ya se ha comentado en apartados anteriores no va a ser superior a $O(n^2)$ y normalmente es inferior.

3.5. Algoritmo aproximado 3: Función de aproximación. Coger las actividades de mayor beneficio mientras no se supere la media de éstos.

Como tercer algoritmo aproximado se vuelve a tener en cuenta. Se ordenan los beneficios, y se van escogiendo las actividades siempre y cuando la suma de las seleccionadas no supere la media de los beneficios. Es decir, si la media de beneficios es 15, el acumulado es 11 y la actividad candidata tiene un beneficio de 3 se podría escoger. A partir de este momento sólo se podrían seleccionar actividades que tengan un beneficio de 1. El motivo para implementar esta función de aproximación es que queríamos una función que para nada fuese realista, puesto que este algoritmo puede devolver actividades que se solapen. Además, por lo general, las soluciones van a distar más o menos la mitad del beneficio obtenido por el algoritmo parcial, incluso más, dependiendo de la/s primera/s selección/es.

La cabecera que se corresponde con este algoritmo es:

```
public int seleccionActividadesPonderada3(int [] c, int [] f , int [] b){…}
```

Como ocurre en los apartados anteriores la complejidad depende del algoritmo ordenación. Por lo tanto, solo en algunos casos va a llegar a ser o(n^2).

3.6. Algoritmo aproximado 4: Función de aproximación: Coger las actividades de mayor tasa beneficio/duración mientras no se supere la media de los beneficios.

Siguiendo con el ejemplo del apartado anterior, y en contraste con el segundo algoritmo voraz, se ha implementado este algoritmo cuya función de aproximación consiste en, una vez ordenadas las tasas beneficio/duración, se van escogiendo las mayores, mientras no se supere el beneficio medio. Exactamente tal y como ocurre en el apartado anterior, solo que, en lugar de escoger la actividad de mayor beneficio, se escoge la actividad con mayor tasa beneficio/duración, que no tiene por qué coincidir.

Al igual que en el apartado anterior se pueden obtener actividades que se solapen, por lo que los resultados no son demasiado realistas, simplemente se busca un primer acercamiento.

Su cabecera es:

```
public int seleccionActividadesPonderada4(int [] c,int [] f, int [] b){…}
```

La complejidad, como en los apartados anteriores depende del algoritmo de ordenación.

3.7. Comparación de tiempos

Para la comparación de tiempos se ha usado System.nanoTime() que ofrece Java.

Se han ejecutado todos los algoritmos con los siguientes datos de entrada, en un bucle 100 veces y se ha calculado la media para obtener un dato significativo(En este caso se han ignorado las impresiones de las soluciones puesto que no son propiamente del algoritmo si no de la presentación de la práctica):

```
int [] com = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};

int [] fin ={4,3,5,6,6,7,10,12,11,14,13,17,16,15,20,22,18,21,30 ,22};

int [] ben = {3,1,4,5,6,8,9,1,4,3,11,3,7,4,5,8,10,15,3,7};
```

Los resultados obtenidos son los siguientes:

seleccionActividadesPonderadoVoraz: 20819 ns.

seleccionActividadesPonderadoVoraz2: 11586 ns.

seleccionActividadesPonderada: 6979 ns.

seleccionActividadesPonderada2: 6150 ns.

seleccionActividadesPonderada3: 2313 ns.

seleccionActividadesPonderada4: 3039 ns.

Cabe destacar la diferencia notable de tiempos que hay entre los algoritmos voraces y los aproximados que, en el mejor de los casos es de casi el doble. Los resultados obtenidos son los esperados al ser un array de un tamaño considerable, se prevé que los algoritmos aproximados tarden menos, puesto que no tienen en cuenta todas las restricciones. Además, los dos últimos, como comparan solo hasta la "mitad", son mucho más rápidos aún. Podríamos afirmar que los algoritmos aproximados no solo devuelven una solución aproximada como se comprobará en el apartado de resultados sino que además hemos conseguido una mejora sustancial de tiempo.

3.8. Comparación con algoritmos Práctica 1.

Es relevante recordar que los algoritmos implementados para la práctica 1, no tienen un array de beneficios y su objetivo es maximizar el número de actividades al que se asiste.

Siguiendo el mismo procedimiento que el realizado con los algoritmos de la práctica 2, ejecutar un bucle 100 veces, sin las impresiones por pantalla, para los mismos datos de entrada:

```
int [] com = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};

int [] fin = {4,3,5,6,6,7,10,12,11,14,13,17,16,15,20,22,18,21,30 ,22};
```

Los resultados obtenidos son los siguientes, se ha ignorado el algoritmo en el que el usuario introducía los datos ordenados puesto que no es relevante, ya que la mayor parte de tiempo lo llevaría el usuario ordenando los tiempos.

seleccionActividadesMejorado: 10421 ns. (Este método ordenaba internamente el array de tiempos de fin)

seleccionActividadesSinOrden: 31253 ns. (Este método no utilizaba ningún tipo de ordenación, buscaba la siguiente actividad candidata).

En este caso vamos a comparar nuestros algoritmos voraces de la práctica 2, con el implementado en el método seleccionActividadesMejorado, puesto que todos estos métodos incluyen, de una forma u otra, algún tipo de ordenación, a pesar de que ordenan array distintos:

seleccionActividadesPonderadoVoraz: 20819 ns.

seleccionActividadesPonderadoVoraz2: 11586 ns.

seleccionActividadesPonderada: 6979 ns.

seleccionActividadesPonderada2: 6150 ns.

seleccionActividadesPonderada3: 2313 ns.

seleccionActividadesPonderada4: 3039 ns.

En relación con los algoritmos de la práctica 2, tiene un tiempo mayor que los algoritmos aproximados, lo cual reafirma la correcta implementación de éstos. Con respecto a los voraces tiene tiempos similares, puesto que la complejidad añadida no es demasiada, simplemente un array más.

4. RESULTADOS

En este apartado se van a mostrar los resultados obtenidos

4.1. Grado de optimalidad de los algoritmos voraces

En este apartado se va a medir el grado de optimalidad de los dos algoritmos voraces implementados. Para ello se ha calculado el porcentaje de casos óptimos que devuelve cada uno de los algoritmos. Para ello, en la clase main, se ha ejecutado el siguiente código:

```
int optimosAlg1=0;
int optimosAlg2=0;

Random n = new Random();
for (int i = 0; i < 100; i++) {

	int[] cs = new int [TAMAÑO-DATOS-ENTRADA];
	int[] fs = new int [TAMAÑO-DATOS-ENTRADA];
	int[] bs = new int [TAMAÑO-DATOS-ENTRADA];
	for (int j = 0; j < TAMAÑO-DATOS-ENTRADA; j++) {
		cs[j] = n.nextInt(100);
		int n1 = n.nextInt(100);
		while (true) {
			if (n1 > cs[j]) {
				fs[j] = n1;
				break;
			}else {
				n1 = n.nextInt(101);
			}
		}
		bs[j] = n.nextInt(100);
	}

 int a = alg.seleccionActividadesPonderadoVoraz(cs, fs,bs);

 int b =alg.seleccionActividadesPonderadoVoraz2(cs,fs,bs);

	if (a>b){
		optimosAlg1++;
	}else if (a<b){
```

```
                optimosAlg2++;
            }else if (a==b){
                optimosAlg1++;
                optimosAlg2++;
            }

        }
```

Este código básicamente lo que hace es generar datos de entrada válidos, en un rango de 0 a 99, que se va a ejecutar 100 veces. Hemos comprobado que dependiendo del valor de TAMAÑOS-DATOS-ENTRADA, es decir, el tamaño de los array, el grado de optimalidad varía. Por ejemplo, para un TAMAÑO-DATOS-ENTRADA = 10, el número de veces que devuelve seleccionActividadesPonderadoVoraz es 67, mientras que para seleccionActividadesPonderadoVoraz2 es 64. Es decir, 67% y 64% de óptimos. Sin embargo, para TAMAÑO-DATOS-ENTRADA = 100, el número de óptimos devuelto por el primer algoritmo es del 5%, mientras que del segundo es del 95%. Esto se debe a que, a mayor número de actividades, el rango de actividades escogidas por el segundo algoritmo es mayor, con lo cual tiene mayor posibilidad de alcanzar el óptimo.

4.2. Exposición de resultados

Los datos de entrada que se corresponden con cada una de las ejecuciones son:

Ejec1:
```
int [] com = {0,0,1,3,4,5};
int [] fin = {5,1,3,4,5,8};
int [] ben = {60,50,50,50,50,50};
```

Ejec2:
```
int [] com2 = {0,2,4,1,7,6};
int [] fin2 = {3,7,6,5,9,8};
int [] ben2 = {2,7,4,4,2,2};
```

Ejec3:
```
int [] com3 = {3,7,1,5,4,8,2,11,9};
int [] fin3 = {5,8,4,7,8,10,3,14,15};
int [] ben3 = {4,3,5,1,6,3,4,6,2};
```

Ejec4:
```
int [] com4 = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
int [] fin4 = {4,3,5,6,6,7,10,12,11,14,13,17,16,15,20,22,18,21,30,22};
int [] ben4 = {3,1,4,5,6,8,9,1,4,3,11,3,7,4,5,8,10,15,3,7};
```

Ejec5:
```
int [] com5 = {3,8,11,10};
int [] fin5 = {13,11,22,40};
int [] ben5 = {4,7,3,10};
```

Ejec6:
```
int [] com6 =
{12,16,11,15,2,3,17,6,5,14,17,19,6,16,2,19,11,12,11,8,6,7,13,6,14,0,19,1,7,1,
1,7,16,9,0,4,19,1,1,3,14,14,9,18,15,4,10,16,4,7};
int [] fin6 =
{16,20,17,19,7,14,20,9,17,18,18,20,17,19,5,20,14,14,13,12,16,18,17,15,16,11,2
0,15,16,6,16,8,18,19,17,6,20,2,10,12,18,17,12,19,20,19,18,18,14,13};
int [] ben6 =
{0,6,7,15,0,17,11,8,2,17,9,5,16,15,2,12,0,0,2,12,17,3,4,14,5,4,1,9,8,16,9,18,
15,6,15,6,10,18,1,19,2,3,2,14,7,5,19,6,14,18};
```

Ejec7:
```
int [] com7 = {0,3};
```

```
int [] fin7 = {4,20};
int [] ben7 = {10,11};
```

Ejec8:

```
int [] com8 = {15,14,13,12,11,10,9,7,5,3,2,1,0};
int [] fin8 = {17,18,15,19,14,11,13,9,8,10,6,3,3};
int [] ben8 = {4,5,6,1,7,4,8,2,9,11,14,4,2};
```

	seleccionActividadesPonderadoVoraz	seleccionActividadesPonderadoVoraz2	seleccionActividadesPonderada
ejec 1	110	250	110
ejec 2	9	8	9
ejec 3	20	21	12
ejec 4	74	71	15
ejec 5	10	10	10

	seleccionActividadesPonderada2	seleccionActividadesPonderada3	seleccionActividadesPonderada4
ejec 1	150	50	50
ejec 2	6	2	2
ejec 3	13	3	3
ejec 4	25	5	5
ejec 5	10	4	4
ejec 6	62	8	8
ejec 7	10	10	10
ejec 8	14	5	5

Las tablas están divididas por cuestiones de legibilidad.

Se puede observar cómo los algoritmos aproximados sirven básicamente para realizar un acercamiento a la solución, aunque en algunos casos son capaces de encontrar una solución óptima, como hacen alguno de sus correspondientes voraces. Sin embargo, a medida que los tamaños de entrada aumentan, se puede observar cómo los algoritmos no son capaces de obtener la solución óptima, aunque son capaces de obtener una buena aproximación como ocurre en la ejecución 8, o incluso en la número 1.

A modo de resumen se presenta la siguiente tabla que muestra el porcentaje de soluciones óptimas y subóptimas, así como otros datos. Esta tabla resumen ha sido generada con la herramienta OptimEx, al igual que las tablas de resultados.

Medida	seleccionActividadesPonderada	seleccionActividadesPonderadoVoraz	seleccionActividadesPonderadoVoraz2
Núm. ejecuciones	8	8	8
% subóptimas	62,50 %	37,50 %	50,00 %
% óptimas	37,50 %	62,50 %	50,00 %
% superóptimas	0,00 %	0,00 %	0,00 %
% desviación media	No hay Optimo	No hay Optimo	No hay Optimo
% desviación máxima superóptima	No hay Optimo	No hay Optimo	No hay Optimo
% desviación máxima subóptima	No hay Optimo	No hay Optimo	No hay Optimo

Medida	seleccionActividadesPonderada2	seleccionActividadesPonderada3	seleccionActividadesPonderada4
Núm. ejecuciones	8	8	8
% subóptimas	87,50 %	100,00 %	100,00 %
% óptimas	12,50 %	0,00 %	0,00 %
% superóptimas	0,00 %	0,00 %	0,00 %
% desviación media	No hay Optimo	No hay Optimo	No hay Optimo
% desviación máxima superóptima	No hay Optimo	No hay Optimo	No hay Optimo
% desviación máxima subóptima	No hay Optimo	No hay Optimo	No hay Optimo

Las tablas están divididas por cuestiones de legibilidad.

5. CONCLUSIONES

Tras analizar los distintos resultados se puede apreciar de manera clara la diferencia entre los algoritmos aproximados y los voraces. Mientras que los primeros son evidentemente más rápidos, dependen en gran medida de como relajes las restricciones para ofrecer más o menos eficiencia. También podemos ver como no coinciden las actividades escogidas cuando no tenemos en cuenta los beneficios a cuando si los tenemos en cuenta.

Esta afirmación parece algo obvia, sin embargo, lo que descubrimos realizando la práctica es que los algoritmos que se centran en el beneficio puro tienden a ofrecer mejores resultados a menor número de actividades, mientras que los que van a por la tasa de beneficio/duración alcanzan mejores resultados en espacios muestras más grandes, lo que nos lleva a la conclusión de que a una mayor heterogeneidad en los datos, como tendería a ocurrir en la vida real, es mejor usar los que optimizan tasas.

Por último, frente a los resultados que distan de la optimalidad en los aproximados, son eficaces en cuanto a su propósito, encontrar soluciones “validas” en tiempos razonables frente a ingentes cantidades de datos de difícil procesado.

6. ANEXOS

De nuevo, como en la anterior entrega, creemos conveniente explicar por qué nos decantamos por usar la ordenación Shell, en detrimento de otras opciones. El principal motivo es que es una de las pocas opciones que tienen una complejidad menor de, en la mayoría de los casos, inferior a $O(n^2)$.
Este método compara cada elemento con el que está a un cierto número de posiciones, llamado salto, en lugar de compararlo con el que está justo a su lado. Este salto es variable, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera). Se van dando pasadas con el mismo salto hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

Gracias a su implementación interna, este método apenas se ve afectado por el grado de ordenación de los datos de entrada.

Por último, con respecto a la entrega de la práctica 1, se ha implementado un método, además del que ya había, que es exactamente igual pero que trata un array de double en lugar de un array de int. Además, en estas implementaciones se ha cambiado la comparación para que el algoritmo sea capaz de ordenar de mayor a menor, lo cual nos resultaba más ventajoso que recorrer el array del revés o darle la vuelta.