ORDENACIÓN SHELL

La ordenación Shell debe el nombre a su inventor, D. L. Shell. Se suele denominar también ordenación por inserción con incrementos decrecientes. Se considera que el método Shell es una mejora del método de inserción directa.
En el algoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el más pequeño, hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo.
El algoritmo de Shell modifica los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño, y con ello se consigue que la ordenación sea más rápida. Generalmente, se toma como salto inicial n/2 (siendo n el número de elementos), y luego se reduce el salto a la mitad en cada repetición hasta que sea de tamaño 1. El Ejemplo 1 ordena una lista de elementos siguiendo paso a paso el método de Shell.

Ejemplo 1
Obtener las secuencias parciales del vector al aplicar el método Shell para ordenar de modo creciente la lista:

6 5 2 3 4 0

El número de elementos que tiene la lista es 6, por lo que el salto inicial es 6/2 = 3. La siguiente tabla muestra el número de recorridos realizados en la lista con los saltos correspondiente.

Recorrido	*Salto*	*Intercambios*	*Lista*
1	3	(6,2),(5,4),(6,0)	2 4 0 3 5 6
2	3	(2,0)	0 4 2 3 5 6
3	3	ninguno	0 4 2 3 5 6
salto 3/2=1			
4	3	(4,2),(4,3)	0 2 3 4 5 6
5	1	ninguno	0 2 3 4 5 6

1. Algoritmo de ordenación Shell
Los pasos a seguir por el algoritmo para una lista de n elementos son:

1. Se divide la lista original en n/2 grupos de dos, considerando un incremento o salto entre los elementos de n/2.
2. Se clasifica cada grupo por separado, comparando las parejas de elementos, y si no están ordenados se intercambian.
3. Se divide ahora la lista en la mitad de grupos (n/4), con un incremento o salto entre los elementos también mitad (n/4), y nuevamente se clasifica cada grupo por separado.
4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un incremento o salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado.
5. El algoritmo termina cuando se llega a que el tamaño del salto es 1.

Por consiguiente, los recorridos por la lista están condicionados por el bucle,

```
intervalo ← n / 2
mientras (intervalo > 0) hacer
```

Para dividir la lista en grupos y clasificar cada grupo se anida este código,

```
desde i ← (intervalo + 1) hasta n hacer
    j ← i – intervalo
    mientras (j > 0) hacer
        k ← j + intervalo
        si (a[j] <= a[k]) entonces
            j ← 0
        sino
            Intercambio (a[j], a[k]);
            j ← j – intervalo
        fin _ si
    fin _ mientras
fin _ desde
```

donde se observa que se comparan pares de elementos de índice j y k, a[j], a[k]), separados por un salto de intervalo. Así, si n = 8, el primer valor de intervalo = 4, y los índices i = 5, j = 1, k = 6. Los siguientes valores que toman i = 6, j = 2, k = 7 y así hasta recorrer la lista. Para realizar un nuevo recorrido de la lista con la mitad de grupos, el intervalo se reduce a la mitad.

```
intervalo ← intervalo / 2
```

Y así se repiten los recorridos por la lista, mientras intervalo > 0.

2. Codificación del algoritmo de ordenación Shell

Al codificar el algoritmo es preciso considerar que Java toma como base en la indexación de arrays índice 0 y, por consiguiente, se han de desplazar una posición a la izquierda las variables índice respecto a lo expuesto en el algoritmo.

```
public static void ordenacionShell(int a[]) {
    int intervalo, i, j, k;
    int n = a.length;

    intervalo = n / 2;
    while (intervalo > 0) {
        for (i = intervalo; i < n; i++) {
            j = i - intervalo;
            while (j >= 0) {
                k = j + intervalo;
                if (a[j] <= a[k]) {
                    j = -1;    // par de elementos ordenado
                } else {
                    intercambiar(a, j, j + 1);
                    j -= intervalo;
                }
```

```
            }
        }
        intervalo = intervalo / 2;
    }
}
```

3. Análisis del algoritmo de ordenación Shell

A pesar de que el algoritmo tiene tres bucles anidados (while-for-while) es más eficiente que el algoritmo de inserción y que cualquiera de los algoritmos simples analizados en los apartados anteriores. El análisis del tiempo de ejecución del algoritmo Shell no es sencillo. Su inventor, Shell, recomienda que el intervalo inicial sea n/2 y continuar dividiendo el intervalo por la mitad hasta conseguir un intervalo 1 (así se hace en el algoritmo y en la codificación expuestos). Con esta elección se puede probar que el tiempo de ejecución es $0(n^2)$ en el peor de los casos, y el tiempo medio de ejecución es $0(n^{3/2})$. Posteriormente, se han encontrado secuencias de intervalos que mejoran el rendimiento del algoritmo. Un ejemplo de ello consiste en dividir el intervalo por 2.2 en lugar de por la mitad. Con esta nueva secuencia de intervalos se consigue un tiempo medio de ejecución de complejidad menor de $0(n^{5/4})$. Nota de programación La codificación del algoritmo Shell con la mejora de hacer el intervalo igual al intervalo anterior dividido por 2.2 puede hacer el intervalo igual a 0. Si esto ocurre, se ha de codificar que el intervalo sea igual a 1, en caso contrario, no funcionará el algoritmo.

```
intervalo = (int) intervalo / 2.2;
intervalo = (intervalo == 0) ? 1 : intervalo;
```

MÉTODOS DE ORDENACIÓN BINSORT Y RADIXSORT

Estos métodos de ordenación utilizan *urnas* para depositar en ellas los registros en el proceso de ordenación. En cada recorrido de la lista a ordenar se deposita en una *urna*$_i$ aquellos registros cuya clave tienen una cierta correspondencia con el índice *i* .

Método de ordenación Binsort (*ordenación por urnas*)

Este método, también llamado *clasificación por urnas*, se propone conseguir funciones tiempo de ejecución de complejidad menor de `O(n log n)` para ordenar una lista de `n` elementos, siempre que se conozca alguna relación del campo *clave* de los elementos respecto de las urnas.

Dado un array `v[]` de registros, se desea ordenar respecto un campo clave de tipo entero, además se sabe que los valores de las claves se encuentran en el rango de `1` a `n`, sin claves duplicadas y siendo `n` el número de elementos. En estas circunstancias ideales es posible ubicar los registros ordenados en un array auxiliar `t[]` mediante este único bucle:

```
for i = 1 to n do
  t[v[i].clave]= v[i];
```

Sencillamente, determina la posición del registro que le corresponde según el valor del campo clave. El bucle lleva un tiempo de ejecución de complejidad lineal `O(n)`.

Esta ordenación tan sencilla que se ha expuesto es un caso particular del método de ordenación por urnas (binsort). Este método utiliza urnas, cada urna contiene todos los registros con una misma clave.

El proceso consiste en examinar cada registro `r` a clasificar y situarle en la urna `i`, coincidiendo `i` con el valor del campo clave de `r`. En la mayoría de los caso en que se utilice el algoritmo será necesario guardar más de un registro en una misma urna por tener claves repetidas. Entonces estas urnas hay que concatenarlas en el orden de menor índice de urna a mayor, así quedará el array en orden creciente respecto al campo clave. En la Figura 6.3 se muestra un vector de `m` urnas. Las urnas están representadas por listas enlazadas.

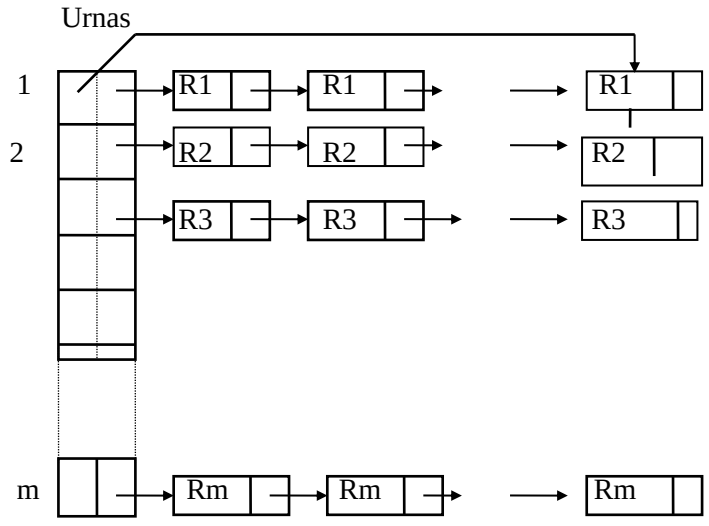

Figura 6.3 Estructura formada por `m` urnas

Algoritmo de ordenación Binsort.

El algoritmo se aplica sobre listas de registros con un campo clave cuyo rango de valores es relativamente pequeño respecto al número de registros. Normalmente el campo clave es de tipo entero, en el rango `1 .. m`. Son necesarias `m` urnas por ello se declara un vector de `m` urnas. Las urnas se representan mediante listas enlazadas, cada elemento de la lista contiene un registro cuyo campo clave se corresponde con el índice de la urna en la que se encuentra, normalmente la clave será igual al índice de la urna. Así en la urna `1` se sitúan los registros cuyo campo clave es `1`, en la urna `2` los registros cuyo campo clave es `2`, y así sucesivamente en la urna *i* se sitúan los registros cuyo campo clave es igual a *i* .

La primera acción del algoritmo consiste en distribuir los registros en las diversas urnas. Una vez realizada la distribución, es necesario concatenar las listas enlazadas para formar un única lista con los registros en orden creciente; por último, se recorre la lista asignando cada nodo al vector, y de esa manera el vector de registros queda en orden respecto al campo clave. La Figura 6.4 se muestra cómo realizar la concatenación.

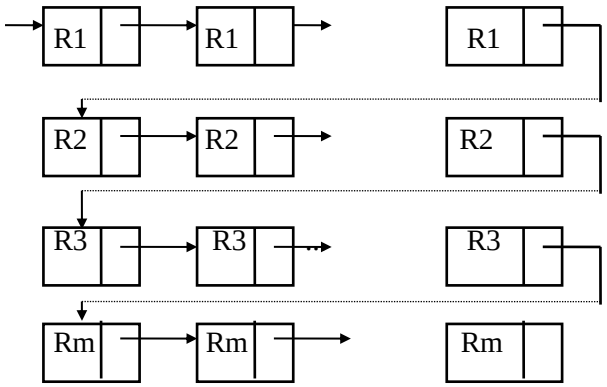

Figura 6.4 Concatenación de urnas representadas por listas enlazadas.

Los pasos que sigue el algoritmo expresado en seudocódigo para un vector de *n* registros:

```
OrdenacionBinsort(vector, n)
inicio
  CrearUrnas(Urnas);
    {Distribución de registros en sus correspondientes urnas}
  desde j = 1 hasta n hacer
    AñadirEnUrna(Urnas[vector[j].clave], vector[j]);
  fin_desde
    {Concatena las listas que representan a las urnas
      desde Urna_i hasta Urna_m}
  i = 1;   {búsqueda de primera urna no vacía}
  mientras EsVacia(Urnas[i]) hacer
    i = i+1
  fin_mientras
  desde j = i+l a m hacer
    EnlazarUrna(Urnas[i], Urnas[j]);
  fin_desde
    {recorre las lista(urnas) resultado de la concatenación}
  j = 1;
  dir = <frente Urnas[i]>;
```

```
    mientras dir <> nulo hacer
      vector[j] = <registro apuntado por dir>;
      j = j+i;
      dir = Sgte(dir)
    fin_mientras
  fin
```

Método de ordenación RadixSort (*ordenación por residuos*)

Este método de ordenación es un caso particular del algoritmo de clasificación por urnas. La idea intuitiva de método de ordenación por *residuos* se encuentra en los pasos que se siguen a la hora de ordenar, de forma manual, un conjunto de fichas; consiste en formar diversos *montones* de fichas, cada uno caracterizado por tener sus componentes un mismo dígito (letra, si es ordenación alfabética) en la misma posición, inicialmente primera posición que son las unidades; estos montones se recogen en orden ascendente, desde el montón del dígito 0 al montón del dígito 9. Entonces las fichas están ordenadas respecto a las unidades. A continuación se vuelve a distribuir las fichas en montones, según el dígito que esté en la segunda posición (*decenas*). El proceso de distribuir las fichas por montones y posterior acumulación en orden se repite tantas veces como número de dígitos tiene la ficha de mayor valor.
Se desea ordenar la siguiente lista de fichas, aplicando los pasos del algoritmo *RadixSort*; las fichas están identificadas por un campo entero de tres dígitos:

```
345, 721, 425, 572, 836, 467, 672, 194, 365, 236, 891, 746, 431,

834, 247, 529, 216, 389
```

Atendiendo al dígito de menor peso (unidades) las fichas se distribuyen en *montones* del `0` al `9`:

				216		
431			*365*	*746*		
891	*672*	*834*	*425*	*236*	*247*	*389*
721	*572*	*194*	*345*	*836*	*467*	*529*
1	*2*	*4*	*5*	*6*	*7*	*9*

Recogiendo los *montones* en orden ascendente la lista de fichas es la siguiente:

```
721, 891, 431, 572, 672, 194, 834, 345, 425, 365, 836, 236, 746, 216,
467, 247, 529, 389
```

Esta lista ya está ordenada respecto al dígito de menor peso, respecto a las unidades. Pues bien, ahora de nuevo se distribuye la secuencia de fichas en *montones* respecto al segundo dígito:

		236					
	529	*836*	*247*				
	425	*834*	*746*	*467*	*672*		*194*
216	*721*	*431*	*345*	*365*	*572*	*389*	*891*
1	*2*	*3*	*4*	*6*	*7*	*8*	*9*

Recogiendo los *montones* en orden ascendente, la lista de fichas es la siguiente:

```
216, 721, 425, 529, 431, 834, 836, 236, 345, 746, 247, 365, 467, 572,
672, 389, 891, 194
```

En este momento las fichas ya están ordenadas respecto a los dos últimos dígitos, es decir respecto a las decenas. Por último, se distribuye las fichas en *montones* respecto al tercer dígito:

	247	*389*	*467*				*891*
	236	*365*	*431*	*572*		*746*	*836*
194	*216*	*345*	*425*	*529*	*672*	*721*	*834*
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*

Recogiendo de nuevo los *montones* en orden ascendente, la lista de fichas ya está ordenada:

```
194, 216, 236, 247, 345, 365, 389, 425, 431, 467, 529, 572, 672, 721,
746, 834, 836, 891
```

Algoritmo de ordenación RadixSort.

La idea clave de la ordenación RadixSort es clasificar por urnas, primero respecto al dígito de menor peso, (unidades); después respecto al dígito del siguiente peso (decenas), y así sucesivamente se continúa hasta alcanzar el dígito más significativo. Una vez terminada la clasificación respecto al dígito mas significativo la lista está ordenada. En cada paso hay que unir las urnas en orden ascendente, desde la urna `0` a la urna `9`; consiste, simplemente, en enlazar el *final* de una urna con el *frente* de la siguiente.
Al igual que en el método de *BinSort*, las urnas se representan mediante un array de listas enlazadas. Se ha de disponer de tantas urnas como dígitos, `10` urnas, numeradas de `0` a 9. Si la clave respecto a la que se ordena es alfabética, habrá tantas urnas como letras distintas, desde la urna que representa a la letra *a* hasta la *z*.
El algoritmo que se escribe, en primer lugar determina el número máximo de dígitos que puede tener una clave. Un bucle de tantas iteraciones como el máximo de dígitos, realiza las acciones de distribuir por urnas los registros, concatenar...
La distribución por urnas exige obtener el dígito del campo clave que se encuentra en la posición definida por el bucle externo, dicho dígito será el índice de la urna.

```
OrdenacionRadixsort(vector, n)
inicio
  { cálculo el número máximo de dígitos: ndig }
  ndig = 0;
  temp = maximaClave;
  mientras (temp > 0) hacer
    ndig = ndig+1
    temp = temp / 10;
  fin_mientras
  peso =1   { permite obtener los dígitos de menor a mayor peso}
  desde i = 1 hasta ndig hacer
    CrearUrnas(Urnas);
    desde j = 1 hasta n hacer
      d = (vector[j] / peso) modulo 10;
      AñadirEnUma(Urnas[d], vector[j]);
    fin_desde
    { búsqueda de primera urna no vacía: j }
    desde r = j+1 hasta M hace    { M: número de urnas }
      EnlazarUma(Urnas[r], Urnas[j]);
    fin_desde
```

```
      {Se recorre la lísta-urna resultante de la concatenación}
    r = 1;
    dir = frente(Urna[j]);
    mientras dir <> nulo hacer
      vecto[r] = dir.registro;
      r = r+1;
      dir = siguiente(dir)
    fin_mientras
    peso = peso * 10;
  fin_desde
fin_ordenacion
```