

Clase 32

Aprendizaje activo: ordenación

¿Por qué es interesante ordenar?

- **La ordenación es una operación que tiene lugar en muchos programas grandes.**
- **Hay muchas formas de ordenar y los informáticos han dedicado mucho esfuerzo a investigar el desarrollo de algoritmos de ordenación eficaces.**
- **Algunos algoritmos incluyen un número importantes de principios de diseño de algoritmos, algunos de ellos muy intuitivos.**
- **En esta clase, analizaremos una serie de algoritmos de ordenación e intentaremos descubrir las ventajas de cada uno.**

Ordenar

- La ordenación implica que los elementos se establecen en un orden determinado. Debemos utilizar el mismo enfoque que empleamos en los árboles de búsqueda binarios.
- Esto es, ordenaremos los objetos y siempre que invoquemos un orden,
 - proporcionaremos un `Comparator` para ordenar los elementos necesarios o bien
 - la rutina de ordenación podrá asumir que los objetos que deben ordenarse pertenecen a una clase como `Integer` que implementa `Comparable` y posee una ordenación nativa.

3

La interfaz `Sort`

- Hay veces en las que queremos ordenar sólo una parte de la colección.
- Teniendo esto en cuenta, definiremos una interfaz para la ordenación. La interfaz `Sort` debe contar con dos métodos, uno para ordenar un *array* completo de objetos y otro para ordenar la parte de un *array* especificada por elementos de inicio y fin (un intervalo real cerrado, no semiabierto):

```
public interface Sort {  
    public void sort(Object[] d,int start,int end);  
    public void sort(Object[] d);  
}
```

4

Uso de la interfaz Sort

- Esta interfaz convierte los algoritmos de ordenación en clases.
- Debemos crear una instancia de la clase antes de poder utilizarla para realizar una ordenación.
- Cada clase de ordenación tendrá dos constructores. El constructor predeterminado ordenará los elementos con orden nativo. El otro constructor tomará un `Comparator` como único argumento.
- A continuación se incluye un fragmento de código de ejemplo que crea un `InsertionSort` para ordenar un *array* de enteros:

```
Integer [] iArray = new Integer[ 100 ];  
. . .          // inicializa iArray  
Sort iSort = new InsertionSort();  
iSort.sort( iArray );
```

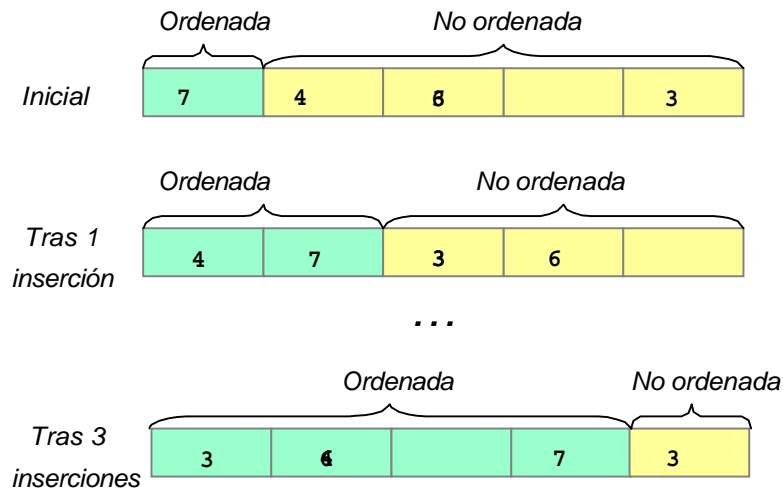
5

Orden de inserción

- La ordenación de inserción es una versión formalizada de la forma en que muchos de nosotros ordenamos las cartas. En una ordenación de inserción, los elementos se seleccionan de uno en uno desde una lista no ordenada y se insertan en una ordenada.
- Es un buen ejemplo de algoritmo incremental o iterativo. Es sencillo e intuitivo, pero podemos hacer algo más para mejorarlo.
- Para ahorrar memoria y evitar copias innecesarias, ordenaremos el *array* en su ubicación.
- Para ello, utilizaremos el extremo inferior del *array* para ampliar la lista ordenada comparándola con el extremo superior del *array sin ordenar*.

6

Diagrama de ordenación de inserción



7

Descarga de simulaciones

- Vaya a la página del material de clase en el sitio web de clase y descargue el archivo **Sorting.zip** en el directorio Archivos de apoyo con Internet Explorer, no con Netscape.
- Haga doble clic en el archivo descargado y pulse el botón Extraer. Utilice el cuadro de diálogo del archivo para seleccionar la ubicación en la que desea descomprimir las simulaciones que vamos a utilizar hoy.

8

Ejecute la simulación InsertionSort

- Desplácese hasta el archivo zip sin descomprimir y haga doble clic en el archivo `InsertionSort.jar` en el Explorador de Windows, no en Forte. Debería aparecer la simulación que utilizaremos para analizar el algoritmo.
- Escriba una serie de números y pulse Enter después de cada uno. Éstos serán los números que vamos a ordenar.
- Pulse `start` y, después, `stepInto` para ir paso a paso en el código.
 - `reset` reiniciará la ordenación actual.
 - `new` le permite especificar una nueva ordenación.

9

Preguntas sobre InsertionSort

Utilice el simulador para analizar las siguientes preguntas y, cuando crea conocer las respuestas, háganoslo saber:

- ¿Cuántos elementos hay que mover en el bucle interno `for` si ejecuta `InsertionSort` en una lista ya ordenada? ¿Se ejecuta en un tiempo $O(1)$, $O(n)$ o $O(n^2)$?
- ¿Qué orden de elementos se producirá en el caso de peor rendimiento? ¿En este caso, se ejecuta en un tiempo $O(1)$, $O(n)$ o $O(n^2)$? ¿Por qué?
- En el caso habitual, ¿hay más puntos en común con el mejor caso o con el peor caso?

10

QuickSort

- Quicksort es un algoritmo clásico y sutil inventado originalmente por C. A. R. Hoare en 1962.
- Actualmente, hay variaciones infinitas del algoritmo original y está considerado como el mejor algoritmo general de ordenación para aplicaciones industriales.

11

Estrategia de QuickSort

- QuickSort es un algoritmo basado en la filosofía “divide y vencerás”. Realiza una ordenación recursiva mediante una operación denominada *partición* que divide el *array* en partes cada vez más pequeñas.
- La idea básica de este algoritmo es elegir un elemento para ordenar en una parte del *array* (llamado *pivote*). A continuación, el algoritmo *particiona* el *array* en relación al *pivote*.
- Particionar significa dividir el *array* en dos *subarrays*. El de la izquierda con elementos menores o iguales que el *pivote*, y el de la derecha con elementos mayores o iguales que el *pivote*. El algoritmo intercambiará elementos para lograr este estado.

12

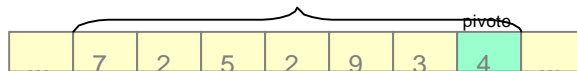
Estrategia de QuickSort, 2

- Tenga en cuenta que los *subarrays* no están ordenados
- En la versión que utilizaremos hoy (pero no en todas las implementaciones de QuickSort), moveremos el pivote entre las dos particiones.
- Quicksort se aplicará recursivamente en los *subarrays*.
- El algoritmo finalizará cuando los *subarrays* de un elemento se ordenen trivialmente.

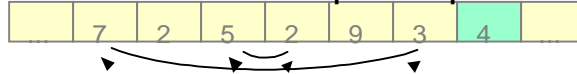
13

Partición sencilla de QuickSort

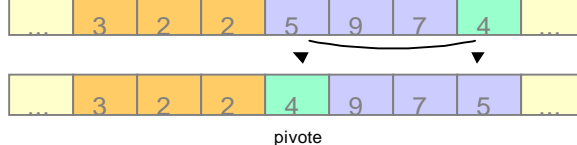
1. Seleccione el último elemento del segmento del *array* actual para que sea el pivote:



2. Intercambie los elementos para cumplir la condición de partición:



3. Intercambie el pivote con el primer elemento de la mitad superior de la partición:



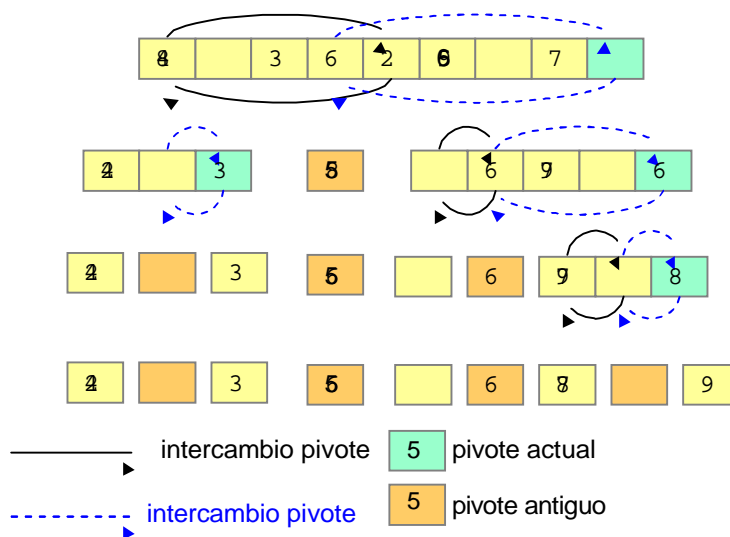
14

Partición

- La partición es el paso clave en Quicksort.
- En nuestra primera versión, el *pivote* se selecciona para que sea el último elemento del (*sub*)array que se va a ordenar.
- Recorremos el (*sub*)array desde el extremo izquierdo utilizando el índice *low* para buscar un elemento \geq el pivote.
- Cuando encontramos uno, recorremos el extremo izquierdo utilizando el índice *high* para buscar un elemento \leq el pivote.
- Si $low \leq high$, los intercambiamos y comenzamos a buscar otro par de elementos intercambiables.
- Si $low > high$, hemos terminado e intercambiamos *low* con el pivote, que ahora se encuentra entre las dos particiones.

15

Partición recursiva



Simulación de Quicksort

- Haga doble clic en el archivo `QuickSort.jar`.
- Funciona igual que la simulación `InsertionSort`.

17

Preguntas de QuickSort

Utilice el simulador para analizar las siguientes preguntas y, cuando crea conocer las respuestas, háganoslo saber:

- ¿Por qué permanecen los índices `low` y `high` dentro del *subarray*?
- ¿Cómo podemos estar seguros de que el procedimiento ha terminado y de que el *subarray* está correctamente particionado?
¿Cómo puede `low` detenerse en un elemento desordenado sin que se produzca un intercambio? ¿Qué ocurre si `low` pasa un `high`? ¿Por qué `low` no estará desordenado?

18

Preguntas de QuickSort, 2

- ¿Qué ocurre si el pivote es el elemento más grande o más pequeño del *subarray*?
- ¿Es Quicksort más o menos eficaz si el pivote siempre es el elemento más grande o más pequeño del (*sub*)array? ¿Qué datos de entrada harían que esto ocurriese?
- ¿Por qué intercambiar *high* y *low* cuando los dos son iguales que el pivote? ¿No es esto innecesario? ¿Qué ocurrirá si intenta ordenar un *array* con todos los elementos iguales?

19

Rendimiento de Quicksort

En general, Quicksort ofrece un rendimiento $O(n \log n)$, pero en los casos descritos anteriormente, su rendimiento será menor: ordenación de inserción ($O(n^2)$) o incluso peor, ya que la ordenación de inserción puede ordenar un conjunto preordenado en $O(n)$.

20

Un Quicksort mejor

- La elección del pivote resulta crucial para el rendimiento de Quicksort.
- El pivote ideal es la media del *subarray*, es decir, el miembro de la mitad del *array* ordenado. Pero no podemos encontrar la media sin ordenar primero.
- Resulta que la media del primer elemento, del medio y del último de cada *subarray* es un buen sustituto para la media. Garantiza que cada parte de la partición tendrá, al menos, dos elementos siempre y cuando el array tenga, como mínimo, cuatro. Pero su rendimiento suele ser mucho mejor y no existen casos naturales que produzcan un comportamiento del peor de los casos.
- Intente ejecutar `MedianQuickSort` desde `MedianQuickSort.jar`.

21

CountingSort

- `InsertionSort` y `Quicksort` ordenan comparando elementos. ¿Existe otra forma de realizar esta acción?
- Asumamos que estamos ordenando datos con un conjunto limitado de claves de enteros que oscila entre 0 y `range-1`.
- `CountingSort` ordena los datos en un *array* temporal llevando a cabo un “censo” de las claves y creando un directorio en el que cada clave debe insertarse.

22

Pasos del algoritmo CountingSort

1. Copia los números que se van a ordenar en un *array* temporal.
2. Inicializa a 0 un *array* indexado por valores de claves (un histograma de claves).
3. Itera sobre el *array* que se va a ordenar contando la frecuencia de cada clave.
4. Ahora calcula el histograma acumulativo de cada valor de la clave, k . El primer elemento, $k=0$, coincide con la frecuencia de la clave k . El segundo, $k=1$, es la suma de la frecuencia para $k=0$ y $k=1$. El tercero es la suma del segundo más la frecuencia para $k=2$. Y así sucesivamente.

23

Pasos del algoritmo CountingSort, 2

5. El primer elemento del histograma acumulativo contiene el número de elementos del *array* original con valores ≤ 0 . El segundo, los que son ≤ 1 . Crean bloques de valores en el *array* ordenado.
6. Comenzando por el último elemento del *array* original y subiendo hasta el primero, busca su clave en el histograma acumulativo hasta encontrar su destino en el *array* ordenado. Será el valor de histograma $- 1$.
7. Disminuye la entrada del histograma acumulativo para que la siguiente clave no se almacene sobre la primera.

24

CountingSort

rango = 10

4		7	6	4	8		5	8	2
---	--	---	---	---	---	--	---	---	---

frecuencias

0	1	2	3	4	5		7	8	9
0	0	1	2	2	1		1	2	0

histograma acumulativo

0	1	2	3	4	5		7	8	9
0	0	1	3	5	6		8	10	10

No hay claves 0, 1 ó 9

La clave 2 tiene lugar 1 vez y debería ocupar la posición 0.

La clave 3 tiene lugar 2 veces y debería ocupar las posiciones 1 y 2.

La clave 4 tiene lugar 2 veces y debería ocupar las posiciones 3 y 4.

Etc.

Preguntas de CountingSort

- Haga doble clic en el archivo `CountingSort.jar`.
- Observe que, dado que `CountingSort` requiere la especificación de un rango (o dos recorridos en los datos), no se puede implementar con nuestra interfaz `Sort`.
- Experimente con la simulación. Escriba los números que quiere ordenar primero. A continuación, escriba el rango en el campo `range` y pulse el botón `start`. Utilice `stepinto` para trazar el código.
- ¿Tiene este método algún caso mejor o peor?
- ¿Ordena en un tiempo $O(1)$, $O(n)$ o $O(n \log n)$? ¿Por qué?