20 | Ordenamientos

Meta

Que el alumno comprenda e implemente algoritmos de ordenamiento.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Implementar algoritmos de ordenamiento en un lenguaje orientado a objetos
- Identificar los mejores y peores casos de un algoritmo de ordenamiento.

Código Auxiliar 20.1: Ordenamientos

https://github.com/computacion-ciencias/ed-ordenamientos-cs

Antecedentes

Una tarea frecuente consiste en ordenar los elementos almacenados en un arreglo, al cual se considera lleno. Para ello existen varios algoritmos basados en comparaciones.

Ordenamiento burbuja

En este ordenamiento se puede decir que cada dato emerge hacia su posición final como una burbuja, de modo que, al final de cada iteración, hay un dato más en su posición correcta hacia la derecha del arreglo.

Algoritmo 4 OrdenamientoBurbuja

```
1: function BURBUJA(arreglo)
2:     for i ∈ [0, penúltimo(arreglo)] do
3:         for j ∈ [0, penúltimo(arreglo) − i] do
4:             if arreglo[j] > arreglo[j + 1] then
5:                 INTERCAMBIA(arreglo, j, j + 1)
6:             end if
7:         end for
8:     end for
9:     return arreglo ordenado
10: end function
```

Observa que, en esta versión, los dos ciclos se ejecutan completos independientemente del orden en que estén los datos. Se puede mejorar la implementación si se agrega una bandera booleana para detectar si ya no se hicieron modificaciones al terminar la ejecución del ciclo más interno y romper el ciclo en cuanto el arreglo esté ordenado, sin realizar ya más comparaciones. Con esta variación sí habrá mejor y peor caso.

Ordenamiento por selección

Este algoritmo ordena el arreglo de izquierda a derecha. Para cada casilla i recorre el resto del arreglo buscando al dato más pequeño para colocarlo ahí.

Ordenamiento por inserción

Este algoritmo también ordena el arreglo de izquierda a derecha, pero tomando al dato en la casilla siguiente e intercambiándolo casilla por casilla con el dato a su izquierda, hasta que éste sea menor o igual que él. De este modo no necesita recorrer todo un lado del arreglo para colocar al dato en su lugar, si éste ya se encontraba cerca de la posición correcta.

Ordenamiento por fusión

Mejor conocido por su nombre en inglés *Merge Sort* este argumento utiliza la estrategia divide y vencerás. Se realiza en dos etapas:

1. Divide el arreglo en dos mitades y estas en mitades, recursivamente hasta obtener arreglos de tamaño uno. Esto porque los arreglos de un elemento ya están ordenados.

2. Combina de regreso los arreglos fusionando pares de listas ordenadas, pues esta tarea se puede realizar en tiempo del orden de la longitud de las lista.

Se implementa de manera recursiva. En el caso de `Java`, como no es posible subdividir un arreglo en subarreglos, se utilizan pares de índices para indicar las posiciones inicial y final de cada subarreglo en las llamadas recursivas.

Ordenamiento rápido

Este ordenamiento, llamado *Quick Sort* en inglés, utiliza divide y vencerás, pero también incluye un elemento aleatorio. Su funcionamiento se basa en la elección al azar de un valor *pivote* en alguna casilla del arreglo. A continuación se procede a colocar los elementos menores al pivote a su izquierda y los mayores, a su derecha. Consecuentemente, el pivote ya ha quedado colocado en su lugar.

El procedimiento se repite recursivamente en cada subarreglo a izquierda y derecha, hasta obtener arreglos de longitud uno, pues estos ya están ordenados.

El ordenamiento rápido pierde la apuesta si elige al más pequeño o al más grande de los valores en el arreglo, pues entonces sólo él quedará en su lugar y no logrará dividir el problema.

Si en cada elección el pivote divide al arreglo aproximadamente en mitades, la complejidad de este algoritmo es $\mathcal{O}(n\log_2(n))$, pero si falla, será $\mathcal{O}(n^2)$.

Desarrollo

En esta práctica se programarán objetos capaces de ordenar arreglos de objetos tipo `IComparable<T>`. Estos objetos pertenecerán a clases que implementarán la interfaz `IOrdenador<C>` en el espacio de nombres `ED.Ordenamientos`. Observa que es una interfaz genérica. Cada clase representará a un algoritmo de ordenamiento y por lo tanto se llamará igual que él, pero con la terminación *Sorter* (ej. `BubbleSorter` para *BubbleSort*). De este modo programarás una clase por cada algoritmo. El ordenamiento se realizará de forma ascendente (de menor a mayor).

Los algoritmos de ordenamiento a implementar serán los siguientes:

- BubbleSort.
- SelectionSort.
- InsertionSort.
- MergeSort.
- QuickSort. En este caso se implementará tomando como pivote el primer elemento del arreglo, para que sea más fácil generar el peor caso.

Además para cada ordenamiento se debe implementar el método que devuelve un arreglo de enteros, el cual representará el peor caso y otro para el mejor caso, en términos de complejidad, para cada uno de los algoritmos mencionados previamente. Cuando la complejidad sea la misma para todos los casos, bastará con devolver cualquier arreglo del tamaño indicado.

Preguntas

1. Explique cómo generó cada uno de los peores casos y por qué es el peor caso para ese algoritmo, además de mencionar el orden de la complejidad del peor caso.
2. Explique cuáles son los mejores casos para los mismos algoritmos y cuál es su complejidad.
3. ¿En qué algoritmos la complejidad en el peor y el mejor caso es la misma?¿Cuál es ésta?
4. ¿En qué algoritmos difiere? Mencione sus complejidades en el mejor y peor caso.