Práctica Número 1
Análisis teórico y experimental de la eficiencia

Objetivo

Trataremos a lo largo de la práctica de realizar un análisis experimental de la eficiencia de los métodos de ordenación por inserción y ordenación rápida.

1. Introducción

Es bien conocido que ordenar un vector de n componentes de un tipo ordenado tiene un coste asintótico entre $n \log n$ y n^2 dependiendo del algoritmo que se utilice. La ordenación rápida o *quickSort* tiene un coste en el caso peor $\mathcal{O}(n^2)$ y un coste $\mathcal{O}(n \log n)$ en el caso medio. Los algoritmos de ordenación por selección e inserción tienen un coste asintótico $\mathcal{O}(n^2)$. Existen algoritmos como la ordenación por conteo o *countingSort* que llevan tiempo lineal, pero exigen conocer el rango de los elementos a ordenar lo cuál no siempre es posible, o es un rango tan grande que hace perder el espíritu del método.

El algoritmo de ordenación por inserción ya se introdujo en la hoja de problemas 1. La descripción de quickSort es la siguiente.

2. El algoritmo de ordenación rápida.

2.1. Descripción

El ordenamiento rápido (quicksort) es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a $n \log n$. Esta es la técnica de ordenamiento más rápida conocida. Fue desarrollada por C. Antony R. Hoare en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos).

El algoritmo fundamental es el siguiente:

1. Elegir un elemento del vector de elementos a ordenar, al que llamaremos pivote.
2. Reposicionar los demás elementos del vector a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada. Existen varias maneras de obtener el pivote. En la implementación que se proporciona en el material de la práctica hay una de ellas.
3. El vector queda separado en dos subvectores, uno formado por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
4. Repetir este proceso de forma recursiva para cada subvector mientras éste contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

- En el mejor caso, el pivote termina en el centro del vector, dividiéndolo en dos subvectores de igual tamaño. En este caso, el orden de complejidad del algoritmo es $\mathcal{O}(n \log n)$.
- En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $\mathcal{O}(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas.
- En el caso promedio, el orden es $\mathcal{O}(n \log n)$.

No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote.

Demostración tramposa.

Vamos a suponer que el número total de elementos a ordenar es potencia de dos, es decir, $n = 2k$. de Aquí podemos ver que $k = \log_2(n)$, donde k es el número de divisiones que realizará el algoritmo.

En la primera fase del algoritmo habrá n comparaciones; en la segunda fase, el algoritmo creará dos sublistas aproximadamente de tamaño $n/2$. El número total de comparaciones de estas dos sublistas es: $2(n/2) = n$. En la tercera fase el algoritmo procesará 4 sublistas más, por tanto el número total de comparaciones en esta fase es $4(n/4) = n$.

En conclusión, el número total de comparaciones que hace el algoritmo es:

$$n + n + n + ... + n = kn,$$

donde $k = \log_2(n)$, por tanto la complejidad es $\mathcal{O}(n \log_2 n)$ □

2.2. Técnicas de selección del pivote en tiempo $\mathcal{O}(n)$

Una idea preliminar para ubicar el pivote en su posición final sería contar la cantidad de elementos menores que él, y colocarlo una posición más arriba, moviendo luego todos esos elementos menores que él a su izquierda, para que pueda aplicarse la recursividad.

Existe, no obstante, un procedimiento mucho más efectivo. Se utilizan dos índices: i, al que llamaremos índice izquierdo, y j, al que llamaremos índice derecho. El algoritmo es el siguiente:

1. Recorrer la lista simultáneamente con i y j: por la izquierda con i (desde el primer elemento), y por la derecha con j (desde el último elemento).
2. Cuando lista[i] sea mayor que el pivote y lista[j] sea menor, se intercambian los elementos en esas posiciones.
3. Repetir esto hasta que se crucen los índices.
4. El punto en que se cruzan los índices es la posición adecuada para colocar el pivote, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

3. Descripción del material suministrado

Para realizar la practica se suministra un paquete de clases Java que contiene métodos diversos métodos de ordenación ya implementados. Se compone de las siguientes clases.

Sort es una interfaz con métodos para ordenar todo el vector o un subvector que se especifica mediante su comienzo y su final.

El siguiente fragmento de código permitiría ordenar por Quicksort un array de 100 enteros:

```
Integer     a[] = new Integer [100];       //crea el array
...                                        //inicializa el array
Comparator<Integer>  c   = new EnterosComparador();
Sort<Integer>        in  = new InsertionSort<Integer>(c);
in.sort(a);                                //ordena el array
```

InsertionSort Es una clase que implementa Sort utilizando el algoritmo de inserción como base para la implementación de los métodos. Es genérica, es decir trabaja con un tipo genérico T. La clase InsertionSort está parametrizada por el comparador a utilizar, mediante un atributo de la interfaz Comparator. Para utilizarla debe implementarse la interfaz Comparator y deben instanciarse los objetos de la clase. Alternativamente, si el tipo T permite comparaciones (implementa la interfaz `Comparable<T>`), puede utilizarse la comparación por defecto (`compareTo`) de dicho tipo.

QuickSort La clase QuickSort responde a una descripción similar a la de InsertionSort, es decir es genérica y puede utilizarse con un comparador concreto o con el comparador por defecto del tipo genérico.

4. Actividad a realizar

1. Descargar las clases suministradas en el material en un proyecto Java.

2. Para realizar nuestros experimentos trabajaremos con vectores aleatorios de números enteros. Por ello lo primero que hacemos es implementar la interfaz Comparator en una clase que denominaremos EnterosComparador. Para ello puede utilizarse el siguiente código (parcialmente realizado) u otro que se desee:

```
/**
 * Implementa la comparacion entre enteros a través de la interfaz Comparator
 */
  import java.util.*;
  public class EnterosComparador implements Comparator<Integer>
  {
    public int compare(Integer n, Integer m)
    {
      // retorna 1 si n representa un entero menor que m
      // retorna -1 si n representa un entero mayor que m
      // retorna 0 en caso contrario
      // COMPLETAR
    }
  }
```

3. Modificar la implementación de las clases QuickSort e InsertionSort para que los métodos de ordenación, además de ordenar retornen un valor entero que contiene el número de instrucciones elementales (coste constante) realizadas para obtener el vector ordenado a partir de los datos. No olvidar a su vez modificar la interfaz Sort.

4. Añadir un método a la interfaz Sort con la siguiente cabecera y especificación:

```
/**
 * Retorna una estimacion del tiempo medio de ordenar vectores de n
 * componentes usando num ejemplos aleatoriamente generados
 */
  public int tMedSort(int n, int num);
```

5. Implementar el método anterior en cada clase. A modo de ejemplo se da el código para InsertionSort. Si se usa este código hay que entenderlo, documentarlo y mejorarlo.

```
  public int tMedSort( int n, int num ){
    int     estimacion = 0;
    Integer vector[]   = new Integer[n];
    for ( int j=1; j<=num; j++ ){
```

```
      int    a[] = new int[n];
      Random rnd = new Random();

      // genera aleatoriamente vector de tamanio n
      for( int i=0; i<a.length; i++ ){
        a[i]      = rnd.nextInt();
        vector[i] = new Integer(a[i]);
      }

      // ordena el vector generado
      Comparator<Integer> c  = new EnterosComparador();
      Sort<Integer>       in = new InsertionSort<Integer>(c);
      int estimacionParcial  = in.sort(vector);
      estimacion            += estimacionParcial;
    }
    return (estimacion/num);
  }
```

6. Codificar una clase llamada GraficaTiempos con un único método principal que pinta las gráficas de los tiempos medios de ejecución de los dos métodos de ordenación considerados en un rango [1,n] suministrado como parámetro. Prueba con rangos pequeños primero (por ej. $n = 25$) y luego con rangos más grandes. A llegar a 1000 el algoritmo de comparación de tiempos ya tarda en ejecutarse un número de segundos apreciable.

7. Dibujar en la misma gráfica las funciones n^2 y $n \log n$. Buscar por el método de "prueba" constantes k_1 y k_2 de tal manera que los gráficos obtenidos en el apartado anterior se ajusten a los de $k_1 n^2$ y $k_2 n \log n$.