Fundamentos de la programación

Sesión de Laboratorio 10

1. Queremos mantener un listado ordenado de alumnos con información básica sobre ellos. El criterio de ordenación será el habitual por apellidos y nombre. Para ello definimos el tipo `Alumno` como un registro del siguiente modo:

```
struct Alumno {
  public string Nombre, Apellido1, Apellido2;
  public int Telefono;
}
```

   y el tipo `Listado` de alumnos como:

```
struct Listado {
  public int tam;      // numero de alumnos del listado
  public Alumno [] v;  // array de alumnos
}
```

   Implementar los siguientes métodos:

   - `crea(lst,n)`: crea un listado vacío de alumnos de tamaño `n`.
   - `compara(al1,al2)`: compara apellidos y nombre de los alumnos `al1` y `al2` con respecto al orden (lexicográfico) habitual, y devuelve -1 si `al1` es menor que `al2`, 0 si son iguales y 1 si `al2` es menor que `al1`. Utilizando el método `String.Compare` de C# se puede implementar fácilmente el método pedido considerando los caracteres especiales del castellano (acentos, ñ, ü,...) así como mayúsculas/minúsculas (véase la guía de programación de C#).
   - `inserta(lst,al)`: inserta ordenadamente el alumno `al` en el listado `lst`. Si el listado está lleno o el alumno ya está presente en el listado producirá el error correspondiente.
   - `elimina(lst,al)`: elimina el alumno `al` del listado `lst`.
   - `cambiaTel(lst,al,tel)`: busca el alumno `al` en el listado `lst` y cambia su número de teléfono por el nuevo número `tel`. Produce un mensaje de error si no existe dicho alumno en el listado.
   - `vuelca(lst)`: vuelca el listado de alumnos `lst` a un archivo de texto `salida.txt`, escribiendo un dato en cada línea (para facilitar luego la lectura).
   - `recupera(lst)`: lee el listado de alumnos `lst` de un archivo de texto `salida.txt` generado por el método anterior.

2. Los algoritmos de ordenación de arrays vistos en clase son generales en sentido de que sirven para ordenar arrays de cualquier tipo ordenado, en particular de enteros. Sin embargo, para algunos problemas concretos es posible sacar partido de las particularidades de los datos. Por ejemplo, si sabemos que el array a ordenar $v[0..n-1]$ contiene enteros en un rango acotado (y relativamente pequeño). Por ejemplo, si $0 \leq v[i] \leq max$ para todo $i \in \{0,\ldots,n-1\}$ (con max relativamente pequeño), podemos utilizar el siguiente algoritmo:

   - construir la tabla de frecuencias t asociada a v. Esta tabla no es más que un vector $t[0..max]$ tal que $t[i]$ contiene el número de apariciones del elemento i en el array inicial v.
   - a partir de la tabla de frecuencias obtener la ordenación del vector v. La idea es: colocar al principio de v $t[0]$ ceros, a continuación colocar $t[1]$ unos y así sucesivamente.

   Por ejemplo, consideremos

$$v = [0, 2, 1, 3, 4, 2, 3, 1, 2, 0, 3, 4]$$

Todos los elementos están en el rango [0.,4] ($max = 4$). La tabla de frecuencias asociada es:

	0	1	2	3	4
t	2	2	3	3	2

Ahora, colocamos en v en este orden 2 ceros, 2 unos, 3 doses, 3 treses y 2 cuatros, y obtenemos el array ordenado:

$$v = [0, 0, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4]$$

Implementar este algoritmo en C# para vectores de enteros en el rango $[0, 999]$. ¿Es este algoritmo mejor que los que hemos visto en clase?

Ampliar la funcionalidad de modo que en tiempo de ejecución se determine el mínimo (min) y máximo (max) de los elementos del vector y el algoritmo trabaje con el rango correspondiente $[min, max]$.