Sort

Práctica 3

- Para esta actividad debes entregar un documento de tipo pdf y toda la carpeta de solución de Visual Studio. Antes de comprimir la carpeta debes borrar los ficheros innecesarios. Para ello debes ir al menú Build, y elegir "**Clean Solution**" (limpiar solución). En el caso contrario se perderá puntos.
- Se hace la practica en grupos de dos. Solo uno de los alumnos componentes del grupo debe entregar la práctica.
- Es necesario escribir como un comentario los nombres de los autores al principio del fichero que contiene el programa principal y también en el documento. En el caso contrario se perderá puntos.
- Si en algún ejercicio se ha indicado el orden o formato de entradas, debes respetar este orden/formato. En el caso contrario se perderá puntos.

El objetivo de este ejercicio es medir el tiempo de ejecución de diferentes algoritmos de ordenación y comparar el resultado con lo de teoría. Los algoritmos para evaluar son: **ordenación por inserción, por selección, por burbuja, Merge Sort, Quick Sort, Ordenación con Rango**. La estructura de datos que se usa para esta actividad es una **lista contigua**. Se pide implementar los 6 algoritmos y hacer experimentos (ver parte 1), y escribir un documento analizando el resultado (ver parte 2).

Parte 1 – Experimento (6 puntos):

Para implementar los seis algoritmos de ordenación, crear una clase llamada Orden. Esta clase no tiene ningún atributo y tiene como mínimo 6 métodos de ordenación y un método llamado esOrdenada que comprueba si la lista está ordenada. Cada método recibe un puntero a un objeto de la clase ListaContigua de la actividad 4.2 y un entero indicando el orden ascendente o descendente. Por ejemplo:

```
#define ASC  0   // de menor a mayor
#define DESC  1 // de mayor a menor
// ... //
void selectionSort(ListaContigua *lista, int orden);
```

Los métodos de ordenación no devuelven nada. El método esOrdenada devuelve un booleano.

Si es necesario, cada método puede llamar a un método privado con más parámetros.

A continuación, escribir un programa main con los siguientes pasos:

1. El programa pregunta al usuario el tamaño de la lista, por ejemplo 20,000, y genera seis listas contiguas iguales de 20,000 números enteros aleatorios entre 0 y 99 ambos incluidos. Cada algoritmo de ordenación usara una de estas listas para ordenar. Las listas deben tener mismos valores para que la comparación entre algoritmos sea justa (ver como ejemplo, el main de la actividad 5.3, donde se crea listaOrdenadaPorMergeSort y listaOrdenadaPorQuickSort).

2. El programa mide el tiempo de ejecutar cada algoritmo usando las listas generadas en paso anterior. Se pide medir el tiempo en caso general, mejor caso, y peor caso.
    a. El caso general es cuando el algoritmo recibe la lista aleatoria desordenada (la lista original).
    b. El mejor caso es cuando recibe la lista ordenada. Es decir, después de ordenar la lista, volver a ordenarlo otra vez.
    c. El peor caso es cuando recibe la lista ordenada al revés. Es decir, después de ordenar la lista volver a ordenarlo en orden al revés.
3. Para asegurarse de que cada algoritmo ha ordenado bien la lista, después de medir el tiempo de cada algoritmo, llamar al método esOrdenada.

    En ejemplo de código en main para medir el tiempo de ordenación por selección sería:

```
        // ordenar ascendente
        numeroClicksInicio = clock();
        orden.SelectionSort(&listaOrdenadaPorSelection, ASC);
        numeroClicksFin = clock();
        segundosTranscurridos = ((float)numeroClicksFin - numeroClicksInicio) / CLOCKS_PER_SEC;
        printf("Con ordenacion por Seleccion en caso general he tardado %.3f segundos.\n", segundosTranscurridos);
        if (orden.esOrdenada(listaOrdenadaPorSelection, ASC) == false) printf(">>>> EL ALGORITMO NO HA FUNCIONADA
CORRECTAMENTE!!! <<<<\n");

        // volver a ordenar ascendente
        numeroClicksInicio = clock();
        orden.SelectionSort(&listaOrdenadaPorSelection, ASC);
        numeroClicksFin = clock();
        segundosTranscurridos = ((float)numeroClicksFin - numeroClicksInicio) / CLOCKS_PER_SEC;
        printf("Con ordenacion por Seleccion en mejor caso he tardado %.3f segundos.\n", segundosTranscurridos);
        if (orden.esOrdenada(listaOrdenadaPorSelection, ASC) == false) printf(">>>> EL ALGORITMO NO HA FUNCIONADA
CORRECTAMENTE!!! <<<<\n");

        // ordenar descendente
        numeroClicksInicio = clock();
        orden.SelectionSort(&listaOrdenadaPorSelection, DESC);
        numeroClicksFin = clock();
        segundosTranscurridos = ((float)numeroClicksFin - numeroClicksInicio) / CLOCKS_PER_SEC;
        printf("Con ordenacion por Seleccion en peor caso he tardado %.3f segundos.\n", segundosTranscurridos);
        if (orden.esOrdenada(listaOrdenadaPorSelection, DESC) == false) printf(">>>> EL ALGORITMO NO HA FUNCIONADA
CORRECTAMENTE!!! <<<<\n");
```

Nota importante: para que funcione todo bien es necesario implementar un constructor de copia para la clase ListaContigua.

OJO: Considerar la eficiencia del código. Hay que recordar que la complejidad del código puede afectar al tiempo de ejecución de cada algoritmo.

Parte 2 – Análisis de resultado (4 puntos):

Se pide escribir un documento en formato pdf para analizar detalladamente el resultado. Probar el programa con diferentes tamaños de la lista, entre 12,000 a 22,000 con pasos de 2,000 (si en tu ordenador el programa con estos tamaños va muy lento o muy rápido, cambia el rango a valores adecuados).

El documento debe contener el tiempo que ha tardado cada algoritmo en caso general, peor, y mejor caso. Para cada caso dibujar un diagrama que muestre el tiempo de todos los algoritmos para diferentes tamaños de datos.

Después, el documento debe tener un análisis y comparación de los tiempos y justificar los resultados.