Laboratorio 1: Algoritmos de Ordenación.

En este laboratorio se implementarán los siguientes algoritmos de ordenación

- *Selection_sort:* Ordenación por selección.
- *Insertion_sort:* Ordenación por inserción.
- *Quick_sort:* Ordenación rápida.

adaptando el pseudocódigo que se provee en el material teórico.

Duración del proyecto: 2 clases.

Primera clase: Completar ejercicios 1 al 3.

Segunda clase: Completar ejercicio 4 y puntos extra.

Ejercicio 1 – Verificar si un arreglo está ordenado.

En el archivo *sort.c*, implementar la función

bool array_is_sorted(int array[], unsigned int length)

que devuelve verdadero si el arreglo está ordenado de menor a mayor, y devuelve falso en caso contrario.

Compilar:
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c main.c sort.c array_helpers.c arguments_parser.c
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -o sorter main.o sort.o array_helpers.o arguments_parser.o

La carpeta *input* contiene arreglos con los cuáles se puede probar la función, por ejemplo:

Ejecutar:
$ ./sorter -c input/example-sorted.in

También es posible probar con varios archivos a la vez:

$ ./sorter -c input/.in*

El argumento **"-c"** del comando anterior habilita dos chequeos al arreglo:
- *array_is_sorted:* Verifica si el arreglo resultante está ordenado.
- *array_is_permutation_of (ya implementada):* Verificar si el arreglo es una permutación del original.

Ejercicio 2 – Ordenación por selección.

Implementar el procedimiento

void selection_sort(int array[], unsigned int length)

que ordena un arreglo de manera ascendente usando el algoritmo de ordenación por selección.

Para implementar el algoritmo se necesitarán dos funciones auxiliares:
- *swap(array, i, j):* Intercambiar las posiciones *i*, *j* en el arreglo *array*.
- *min_pos_from(array, length, i)*: Calcular la posición del elemento mínimo de *array* a partir de la *i*-ésima posición.

Compilar el código igual que en el ejercicio anterior, y luego probar la función con el siguiente comando.

Ejecutar:
$ ./sorter -spc input/example-sorted.in

El argumento **"-s"** habilita el algoritmo de ordenación por selección.
El argumento **"-p"** imprime el arreglo resultante al finalizar.

Ejercicio 3 – Ordenación por inserción.

Implementar el procedimiento

void insertion_sort(int array[], unsigned int length)

que ordena un arreglo de manera ascendente usando el algoritmo de ordenación por inserción. Implementar además las funciones auxiliares que sean necesarias.

Ejecutar:
$ ./sorter -ipc input/example-sorted.in

El argumento **"-i"** habilita el algoritmo de ordenación por inserción.
El argumento **"-p"** imprime el arreglo resultante al finalizar.

Nota: Tener en cuenta que en el material teórico se usan arreglos indexados a partir de 1, pero
en C los arreglos comienzan en la posición 0, por lo tanto será necesario adaptar el pseudocódigo antes de traducirlo.

Ejercicio 4 – Ordenación rápida.

Implementar el procedimiento

void quick_sort(int array[], unsigned int length)

que ordena un arreglo de manera ascendente usando el algoritmo de ordenación rápida.

Será necesario implementar la función *partition* para el cálculo del pivote, y la función auxiliar recursiva *quick_sort_rec* que se mencionan en el material teórico.

Nota: Será necesario adaptar la función *partition* del pseudocódigo, ya que la anotación "out" del parámetro *pivot* no existe en C.

Ejecutar:
$ ./sorter -qpc input/example-sorted.in

El argumento **"-q"** habilita el algoritmo de ordenación rápida.

Observar los tiempos de ejecución que devuelve el comando *sorter*. Notar se pueden ejecutar varios algoritmos a la vez para compararlos, por ejemplo:

Ejecutar:
$ ./sorter -isq input/unsorted-100000.in

¿Cuál de los tres algoritmos es más rápido para este caso?

Otro ejemplo:
$./sorter -isq input/sorted-asc-10000.in

El algoritmo de ordenación rápida no es eficiente cuando el arreglo original ya se encuentra ordenado ¿por qué?.

Puntos extra (opcionales)

- Modificar la función *partition* para que se elija el pivote de manera aleatoria. Hint: Generar una posición aleatoria del arreglo e intercambiar su elemento con la posición izquierda del mismo. ¿Cómo mejora esto la eficiencia de *quick_sort* sobre arreglos ordenados?

- Modificar *sorter*. Agregar un nuevo algoritmo de ordenación (ejemplo, Bubble Sort) y agregar una opción **"-b"** para poder ejecutarlo usando *sorter*.

- Modificar *sorter*. Contar y mostrar en pantalla la cantidad de comparaciones y de llamadas a *swap* utilizadas por los algoritmos. Se permite excepcionalmente utilizar variables globales para su implementación.