Proyecto 1
Algoritmos de Ordenación

Algoritmos y Estructuras de Datos II - Laboratorio

Docentes: Natalia Bidart, Matías Bordese, Diego Dubois,
Leonardo Rodríguez, David Arch, Maximiliano Bustos.

Objetivo

El objetivo de este proyecto es aprender a implementar algoritmos de ordenación y a comparar el rendimiento de cada uno para diferentes entradas de datos.

En esta primera parte del proyecto, nos enfocaremos en:

- La implementación en C de los algoritmos de ordenación por selección y por inserción.
- El análisis de la eficiencia de estos dos algoritmos para distintos inputs, comparando la cantidad de operaciones representativas que el algoritmo requiere para llevar a cabo la ordenación.

Intrucciones generales

En la página de la materia, junto a este enunciado, podrán encontrar un link para bajar el "esqueleto" del código con el cual deberán trabajar. Los archivos que encontrarán son los siguientes:

```
array_helpers.c
array_helpers.h
input/
main.c
sort.c
sort.h
```

El archivo `sort.h` contiene la especificación de las funciones que ustedes deberán implementar. El código de esas funciones deberá estar en `sort.c`. El archivo `array_helpers.h` contiene la descripción de funciones provistas por los docentes, que podrán utilizarlas para leer datos desde archivos de texto, y construir arreglos para probar los algoritmos. En el archivo `main.c` está la función principal, que muestra un menú en pantalla y permite al usuario elegir entre los diferentes algoritmos de ordenación disponibles.

Una vez que completen el archivo `sort.c`, pueden proceder a compilar el programa en una terminal utilizando el siguiente comando:

```
>$ gcc -Wall -Werror -Wextra -ansi -pedantic -std=c99 -c array_helpers.c sort.c
>$ gcc -Wall -Werror -Wextra -ansi -pedantic -std=c99 -o sorter *.o main.c
```

o bien ejecutar el script provisto `compile`. Es muy importante compilar utilizando todos los flags anteriores (`-Wall, Werror, ...`) ya que permiten que el compilador informe sobre posibles errores y malas prácticas de programación.

Luego de compilar, pueden ejecutar el programa de la siguiente manera:

```
>$ ./sorter <ruta_al_archivo_de_datos>
```

El archivo de datos (que describe a un array a ordernar)debe tener el siguiente formato:

```
<array_length>
<array_elem_1> <array_elem_2> <array_elem_3> ... <array_elem_N>
```

La primer línea debe contener un entero que debe ser la cantidad de números que contiene el archivo (el tamaño del arreglo). La segunda línea, debe contener los valores que tendrá el arreglo que queremos ordenar, separados cada uno por uno o más espacios. En la carpeta `input/` podrán encontrar algunos archivos de ejemplo.

Supongamos que, por ejemplo, queremos ordenar los siguientes datos, guardados en el archivo `input/example.in`:

```
5
8 5 3 0 1
```

Entonces, si ejecutamos el programa con el comando ya descripto, se muestra un menú para elegir entre los diferentes algoritmos disponibles de ordenación:

```
>$ ./sorter input/example.in

Choose the sorting algorithm. Options are:
        s - selection sort
        i - insertion sort
        e - exit this program
Please enter your choice:
```

Si elegimos la opción `'s'` (por ejemplo), se ejecutará el algoritmo de ordenación por selección, implementado por ustedes en `sort.c`. Luego de correr el algoritmo, el programa muestra en pantalla el arreglo ordenado resultante (el formato de la salida es idéntico al formato del archivo de entrada):

```
5
0 1 3 5 8
```

A continuación se explica con más detalle las tareas a realizar.

Ordenación por selección

La primera parte del proyecto es implementar el algoritmo de ordenación por selección, que tendrá la siguiente signatura:

```
void selection_sort(int *a, int length)
```

El parámetro "a" es un arreglo de enteros, y "length" es la longitud del arreglo. Tanto éste como el resto de los algoritmos de este proyecto deben modificar el arreglo únicamente mediante el procedimiento:

```
void swap(int *a, int i, int j)
```

que intercambia los valores de las posiciones "i" y "j" en el arreglo "a". Será necesario también implementar la función:

```
int min_pos_from(int *a, int length, int i)
```

que retorna la posición del mínimo valor de "a" comenzando desde la posición "i". Como antes, el parámetro "length" contiene la longitud de "a".

Ordenación por inserción

El siguiente algoritmo a implementar es el de ordenación por inserción. El procedimiento deberá tener la signatura:

```
void insertion_sort(int *a, int length)
```

Y al igual que el ítem anterior, el array "a" podrá ser modificado únicamente llamando a `swap`.

Punto ⋆ *Otro algoritmo de ordenación simple es el algoritmo de la burbuja, o* bubble sort *(http://en.wikipedia.org/wiki/Bubble_sort). Implementarlo y agregarlo como opción al menú inicial.*

Punto ⋆ *Cambiar la signatura de los algoritmos (y de toda otra función que lo requira), y cambiar el menú, de manera tal que el usuario pueda optar entre ordenación ascendente o descendente.*

Comparación de eficiencia

La última tarea es comparar la eficiencia de los algoritmos. En este caso, deberán mostrar en pantalla cuántas comparaciones y cuántas operaciones `swap` usan los algoritmos para distintas entradas (un array sin ordernar, uno ya ordenado, uno ordenado a la inversa de lo que el algoritmo ordena). Para el ejemplo anterior, una forma de mostrarlo podría ser la siguiente:

```
5
0 1 3 5 8
```

```
Selection sort:
number of swap operations: 4
number of comparisons: 10
```

Ayuda: *Para el caso de ordenación por selección ¿Es necesario utilizar un contador para conocer el número de comparaciones y de swaps?*

Pregunta 1 *Se observa alguna relación entre la longitud del arreglo a ordenar y la cantidad de comparaciones? respecto de los swaps? Cambia si el arreglo de entrada ya está ordernado? Pensar las respuestas para el día de la entrega.*

Recordar

- Fecha de entrega y evaluación: Martes 27 de marzo.
- Los grupos son de dos personas, pero se rinde individual.
- Para obtener un 10 es necesario hacer el proyecto sin errores más los puntos $\star$.
- Para regularizar se debe obtener 4 y para promocionar un 7.