Proyecto 1: Algoritmos de Ordenación

Algoritmos y Estructuras de Datos II - Laboratorio

Docentes: Leonardo Rodríguez, Gonzalo Peralta, Diego Dubois, Jorge Rafael.

Objetivos

- La implementación en C de los algoritmos de ordenación por selección (*Selection sort*), por inserción (*Insertion sort*) y *Quick sort*.
- Reusar código dado por la cátedra, entendiendo las utilidades provistas, y siendo capaz de integrar código propio con el dado.

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

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. Es uno de los requerimientos para la evaluación usar todos los flags y saber modificar y re-compilar el proyecto.

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

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

El archivo de datos (que describe un array a ordenar) 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 (archivo `input/example-unsorted.in`):

```
5
2 -1 3 8 0
```

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-unsorted.array
Choose the sorting algorithm. Options are:
    s - selection sort
    i - insertion sort
    q - quick sort
    r - rand quick sort
    e - exit this program
Please enter your choice:
```

Si elegimos la opción `'s'`, 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
-1 0 2 3 8
```

El objetivo principal de este proyecto es que implementen los algoritmos de ordenación por inserción, por selección, quick sort y random quick sort. Para implementar todos los algoritmos, seguir el detalle del pseudocódigo dado en el teórico.

Algoritmos

Los algoritmos a implementar tienen la siguiente signatura:

```
unsigned int selection_sort(int a[], unsigned int length);
unsigned int insertion_sort(int a[], unsigned int length);
unsigned int quick_sort(int a[], unsigned int length);
unsigned int rand_quick_sort(int a[], unsigned int length);
```

Cada función toma como argumentos un arreglo de enteros a ordenar y la longitud del mismo. Devuelve la cantidad de comparaciones *entre elementos del arreglo* que se hicieron para ordenarlo. Es decir cada vez que se realice una operación <=, >=, <, >, == o != entre dos elementos del arreglo (por ejemplo, `a[i] < a[j]`) se considera una comparación.

A tener en cuenta en el conteo de comparaciones:

- Se cuentan las comparaciones realizadas por el algoritmo, independientemente del valor booleano que adquieren.
- Tener cuidado con el orden de evaluación. Por ejemplo, en la siguiente expresión de C, el operador `&&` devuelve directamente `false` y la comparación no se realiza: `2 < 1 && a[i] < a[j]`.
- En `quick_sort`, la función `pivot` devuelve el pivote nuevo, pero además tiene que devolver la cantidad de comparaciones, ya que forman parte del algoritmo. Para devolver más de un valor se suelen utilizar estructuras `struct`. Esa estructura debe ser auxiliar y no se debe publicar en `sort.h`.
- Cuando hay llamadas recursivas, se cuenta la cantidad de comparaciones de cada llamada recursiva y luego se suman para obtener la cantidad total.
- No se pueden usar variables globales.

Comparador

La cátedra provee el programa `compare.c` que ejecuta todos los algoritmos de ordenación e imprime la cantidad de comparaciones que se realizan por cada arreglo.

Para compilarlo escribir:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c array_helpers.c sort.c
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -o compare *.o compare.c
```

Y para ejecutarlo escribir:

```
$./compare input/
```

(Aquí `input/` hace referencia a la carpeta que contiene los archivos con arreglos).

Pivote aleatorio

La versión de `quick_sort` que se dio en el teórico elige siempre la primera posición del arreglo como pivote. Sin embargo, el algoritmo da mejores resultados en la práctica cuando el pivote se elige (pseudo) aleatoriamente. La función `rand_quick_sort` debe implementar ese algoritmo.

Dentro de *pivot*:

- Generar una posición aleatoria en el arreglo e intercambiarla por la primera posición (usar la función `rand`).
- La función continúa normalmente usando la primera posición como pivote.

Dentro de `rand_quick_sort`: Generar la semilla (seed) de la secuencia aleatoria, (usar la función `srand()`).

Leer la documentación de las funciones `rand` y `srand`.

Punto estrella (opcional)

Implementar el algoritmo `bubble_sort` o bien `cocktail_sort`, agregarlo al menú de `sort.c` y actualizar `compare.c`.

Entrega y evaluación

- Fecha de entrega del código:
  - Martes 29 de Marzo, el mismo día se toma el parcialito individual y se presenta el proyecto siguiente.
  - Llegado el momento se indicará dónde y cómo enviar el proyecto.

Recomendaciones

- Leer y entender los algoritmos **antes** de implementarlos. Para ganar intuición se recomienda correr a mano algún ejemplo y buscar animaciones que muestren el funcionamiento del algoritmo.
- Implementar los algoritmos primero ignorando el conteo de comparaciones (que devuelvan siempre 0). Luego de comprobar que los algoritmos funcionan bien, pensar cómo agregar el conteo de comparaciones.

Recordar

- Los grupos son de dos personas, pero se rinde individual.
- La evaluación consiste de un parcialito, individual, en donde habrá que escribir código **nuevo**, modificando cualquier parte del código.
  - Deben saber compilar a mano usando todos los flags requeridos.
  - Deben poder entender los errores del compilador para corregirlos.
- Comentar e indentar el código apropiadamente, siguiendo el estilo de código ya provisto por la cátedra.
- Todo el código tiene que usar la librería estándar de C, y no se puede usar extensiones GNU de la misma.
- Con la función `assert` se pueden chequear pre y post condiciones (ver manpages).
- Recordar que la traducción de pseudocódigo al lenguaje C **no** es directa. En particular, tener en cuenta que la indexación en los arreglos de C comienzan en la posición 0.
- Cualquier modificación/agregado/borrado a la interfaz de línea de comando tiene que ser hecho en `main.c`. La biblioteca `sort` nunca debería pedir nada al usuario, ni imprimir nada por pantalla. Es decir, end `sort.c` **nunca** debería haber una llamada a `printf` (ni a `array_dump`).