Proyecto 1: Algoritmos de Ordenación

Algoritmos y Estructuras de Datos II - Laboratorio

Docentes: Natalia Bidart, Matías Bordese, Diego Dubois, Leonardo Rodríguez.

1. Objetivos

- La implementación en C de los algoritmos de ordenación por selección (*Selection sort*), por inserción (*Insertion sort*) y *Quick sort*.
- El análisis de la eficiencia de estos algoritmos para distintas entradas, comparando la cantidad de operaciones representativas que el algoritmo requiere para llevar a cabo la ordenación (contando la cantidad de comparaciones y swaps que cada algoritmo requiere para ordenar un arreglo dado).
- Analizar y comparar los tres algoritmos entre sí usando como base del análisis las mediciones hechas en cada caso.
- Reusar código dado por la cátedra, entendiendo las utilidades provistas, y siendo capaz de integrar código propio con el dado.
- Utilizar regularmente un sistema simplificado de "Integración Continua" para obtener un diagnóstico temprano del proyecto.

2. 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:

```
input/
tests/
array_helpers.c
array_helpers.h
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.

Ningún archivo fuera de `sort.c` debería ser modificado, excepto si están resolviendo puntos estrellas, en cuyo caso van a necesitar modificarlos a casi todos.

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
```

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 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 (archivo `input/example-unsorted.array`):

```
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
    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 y quick sort. Para implementar todos los algoritmos, seguir el detalle del pseudocódigo dado en el teórico.

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

3. 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, unsigned int length)
```

El parámetro "a" es un arreglo de números 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, unsigned int i, unsigned 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, unsigned int length, unsigned 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".

4. 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, unsigned int length)
```

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

5. Quick sort

El último algoritmo a implementar es el Quick sort. El procedimiento deberá tener la signatura:

```
void quick_sort(int *a, unsigned int length)
```

Como arriba, el parámetro "a" es un arreglo de enteros, y "length" es la longitud del arreglo. El arreglo puede ser modificado únicamente mediante la función:

```
unsigned int pivot(int *a, unsigned int length, int left, int right)
```

Además van a necesitar implementar una función auxiliar recursiva, que surge directamente de lo estudiado en el teórico:

```
void recursive_quick_sort(int *a, unsigned int length, int left, int right)
```

6. Comparación de eficiencia

La última tarea es comparar la eficiencia de los algoritmos. En este caso, deberán hacer dos tareas principales:

1. Mostrar en pantalla cuántas comparaciones y cuántas operaciones `swap` usan los algoritmos para las distintas entradas (por ej para un array sin ordernar, uno ya ordenado, uno ordenado a la inversa de lo que el algoritmo ordena, etc).
2. Completar el cuadro comparativo de más abajo para un conjunto predefindo de arrays.

Para la tarea 1, los contadores deben ser variables globales declaradas en el archivo `sort.c`, y no deben ser accedidas directamente desde ningún otro archivo. Estos contadores deben ser del tipo:

```
unsigned long int
```

Para poder imprimir sus valores (y eventualmente resetear los contadores), habrá que proveer (desde la biblioteca `sort`), funciones de consulta y reseteo de los mismos:

- `unsigned long int get_comp_counter(void);`
- `unsigned long int get_swap_counter(void);`
- `void reset_counters(void);`

Para el ejemplo anterior, la forma de mostrar los valores de los contadores por pantalla debe ser (respetar el texto mostrado):

```
$ ./sorter input/example-unsorted.array
Choose the sorting algorithm. Options are:
    s - selection sort
    i - insertion sort
    q - quick sort
    e - exit this program
Please enter your choice: q
5
-1 0 2 3 8
Comparaciones: 9
Swaps: 4
```

Para la tarea número 2, para cada algoritmo, recolectar datos de cantidad de comparaciones y swaps para arreglos de largos distintos:

- $N = 0$
- $N = 100$
- $N = 1000$
- $N = 10000$

Para cada N, recolectar datos para arreglos con las siguientes características:

- ordenado ascendentemente
- ordenado descendentemente
- desordenado

(notar que sólo hay que ordenar los arrays de entrada de manera ascendente, la ordenación descendente no tiene relevancia en esta parte del proyecto).

Para facilitar esta tarea, se proveen archivos para todos los casos, dentro del directorio `input/`. Los archivos tienen nombres autoexplicativos:

```
$ ls -1 input/*.in
empty.in
sorted-asc-10000.in
sorted-asc-1000.in
sorted-asc-100.in
sorted-desc-10000.in
sorted-desc-1000.in
sorted-desc-100.in
unsorted-10000.in
unsorted-1000.in
unsorted-100.in
```

Los archivos de nombre `sorted-*` son array ya ordenados, y los `unsorted-*` están desordenados. Los archivos `*-asc-*` están ordenados ascendentemente y los `*-desc-*` descendentemente. Por último, los números en el nombre del archivo indican cuántos elementos hay en el array.

Se deberá presentar una tabla comparativa de la siguiente forma (la columna de *bubble sort* deberá ser llenada sólo si se hizo el punto estrella que lo pedía):

N	Array de entrada	Selection sort		Insertion sort		Quick sort		Bubble sort $\star$	
		Comps	Swaps	Comps	Swaps	Comps	Swaps	Comps	Swaps
0	Vacío								
100	Ordenado asc								
	Ordenado desc								
	Desordenado								
1000	Ordenado asc								
	Ordenado desc								
	Desordenado								
10000	Ordenado asc								
	Ordenado desc								
	Desordenado								

Para pensar para la defensa oral

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?

Tener en cuenta que el día de la evaluación se les harán preguntas respecto de los datos arriba especificados, buscando conclusiones respecto de la eficiencia de cada algoritmo, para cada caso.

7. Puntos estrella

Punto $\star$ 1 *Algoritmo de la burbuja*

Otro algoritmo de ordenación simple es el algoritmo de la burbuja, o *bubble sort*[1]. Implementarlo con la siguiente signatura:

```
void bubble_sort(int *a, unsigned int length)
```

Y agregarlo como nueva opción al menú inicial, utilizando la letra `b`.

Punto ⋆ 2 *Ordenación descendente*

Cambiar la signatura de los algoritmos (y de toda otra función que lo requiera), y cambiar el menú, de manera tal que el usuario pueda optar entre ordenación ascendente o descendente, para todos los algoritmos implementados (si implementaron *bubble sort*, también para éste).

Notar que las siguientes cosas **no** son válidas como solución de este punto estrella:

- **NO** se debe definir ninguna función ni ningún procedimiento nuevo para implementar este punto estrella. Es decir que **únicamente** pueden cambiar los prototipos de las funciones existentes para lograr el objetivo (esto aplica a todos los módulos, incluso los auxiliares dados por la cátedra). Claro está que las implementaciones de las funciones y procedimientos existentes pueden variar un poco.
- **NO** se debe repetir código dentro de grandes bloques `if`. Es decir, no deberían tener la mismas líneas de código repetidas en dos cuerpos de un `if` para resolver este estrella.
- **NO** se debe cambiar las complejidades de los algoritmos: o sea, si un algoritmo es lineal, la solución a este punto debe mantener esa característica para todo algoritmo. Ejemplos de cosas que violan este requerimiento:
  - Ordenar el array ascendentemente y luego revertir el orden de los elementos.
  - Multiplicar todos los elementos por -1, ordernar ascendentemente, y re-multiplicar todos los elementos por -1.

Punto ⋆ 3 *Quick sort*

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. Implementar esa modificación respetando lo siguiente:

- Se debe agregar un procedimiento dentro de la biblioteca `sort`, que implementará el algoritmo usando un pivote elegido de manera aleatoria para todos los casos (no solo la primera vez). La signatura de este nuevo procedimiento debe ser:

  ```
  void rand_quick_sort(int *a, unsigned int length)
  ```

  Agregarlo también como nueva opción al menú inicial, utilizando la letra `r`.
- Si se implementó el punto estrella 2 (ordenación descendente), este nuevo procedimiento tiene que soportar las dos ordenaciones también.

Además, de completar este punto, se deberá agregar a la tabla comparativa una columna extra para anotar las comparaciones y swaps requeridos al usar el algoritmo con pivote aleatorio.

Ayuda: Ver la documentación de la función de C *rand*.

[1] Ver en wikipedia

8. Jaime: ayudante para la integración continua

Nota: el contenido de esta sección está también disponible en el wiki.

A lo largo de este cuatrimestre, las entregas de los proyectos y las entregas de los "parcialitos"van a ser a través de nuestro ayudante, Jaime.

Jaime es un ejecutor de tareas muy muy simple. Por cada proyecto, Jaime va a tener disponible un listado de tareas que va a permitir correr una serie de tests sobre el código producido por cada alumno.

Para visitar Jaime, ir a `http://jaime.no-ip.info/`. Es obligatorio tener un usuario, y se va a crear un usuario automáticamente por cada miembro de cada grupo del taller. Por cualquier duda, consultar en la lista de la materia.

9. Entrega y evaluación

- Fecha de entrega del código:
  - Desde el Lunes 24 de Marzo a las 00:00 hrs hasta las 23:59 hrs del mismo día.
  - Obligatorio hacerlo a través de Jaime, habrá una tarea específica para subir el código (separada de la tarea para probar los proyectos).
  - Se permitirá una sola entrega por grupo, así que asegurarse de subirla cuando esté lo suficientemente probada.
- Fecha de evaluación (parcialito individual): Martes 25 de marzo a las 14 hrs.

10. 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. Practicar mucho, hay únicamente una hora por persona:
  - Deben saber compilar a mano usando todos los flags requeridos.
  - Deben poder entender los errores del compilador para corregirlos.
  - Deben haber trabajado con *Jaime*, el ayudante informático de la cátedra, ya que los parcialitos se entregan a través de este sistema.
- 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.
- Leer y entender los algoritmos **antes** de implementarlos. *Tip*: Para ganar intuición se recomienda correr a mano algún ejemplo y buscar animaciones que muestren el funcionamiento del algoritmo.
- 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`).