Proyecto 1: Algoritmos de Ordenación

Algoritmos y Estructuras de Datos II - Laboratorio

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

El proyecto consta de tres partes (A, B, C), que den deben realizarse en orden. La parte C es opcional ya que contiene ejercicios estrella.

Parte A: Implementación

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.

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/
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.

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 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.

Chequeo de ordenación

La primera función a implementar se usará para verificar si los algoritmos de ordenación funcionan adecuadamente. Debe devolver `true` si el arreglo está ordenado y `false` en otro caso.

```
bool array_is_sorted(int *a, unsigned int length)
```

Ordenación por selección

A continuación 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 necesaria 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".

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`.

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, 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)
```

Parte B: Comparación de eficiencia

Objetivos

- 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).
- Familiarizarse con el uso de estructuras en C.

Contar comparaciones y swaps

Una vez finalizada la parte A, la siguiente tarea es modificar la implementación de tal forma que podamos llevar cuenta de la cantidad de comparaciones y swaps que los algoritmos realizan. Con esta información podremos comparar la eficiencia de los algoritmos.

Cambios en la implementación

En el `sort.h` agregar la siguiente estructura:

```
struct sorting_stats {
    unsigned long int comps;
    unsigned long int swaps;
};
```

(no olvidarse el último punto y coma).

Luego modificar la signatura de los algoritmos para que el valor de retorno sea una estructura de tipo `sorting_stats`. Por ejemplo, la función `insertion_sort` ahora tendrá la siguiente signatura:

```
struct sorting_stats insertion_sort(int *a, unsigned int length)
```

y el código de la función tendrá la siguiente forma:

```
struct sorting_stats insertion_sort(int *a, unsigned int length) {
       ...
      struct sorting_stats result;
      result.comps = 0;
      result.swaps = 0;
       ...
           result.comps = result.comps + 1;
       ...
           result.swaps = result.swaps + 1;
       ...
      return (result);
}
```

Cada vez que el algoritmo hace una comparación entre dos elementos del arreglo, incrementamos el contador `comps`. De manera similar, cada vez que el algoritmo realiza una llamada a `swap` (intercambio entre dos elementos del arreglo) incrementamos el contador `swaps`. Al finalizar el algoritmo, los contadores guardaran la cantidad total de operaciones realizadas durante toda la ejecución.

Solamente se tienen en cuenta comparaciones entre elementos del arreglo, **no** así las comparaciones entre índices de un bucle, o posiciones del arreglo.

En las funciones auxiliares, por ejemplo `min_pos_from`, también necesitarán devolver una estructura puesto que allí también se realizan comparaciones y swaps. Esta estructura puede ser distinta a `sorting_stats`, y con diferentes miembros. Pero es **importante** que esas nuevas estructuras auxiliares no estén en `sort.h` sino en `sort.c`, puesto que son estructuras *privadas*. Dicho de otra forma, un usuario de los algortimos sólo necesita conocer la estructura `sorting_stats` pero no las estructuras auxiliares.

Tener cuidado en el caso `quick_sort`, ya que el algoritmo es recursivo. Para obtener la cantidad de operaciones total hay que sumar las operaciones realizadas por cada llamada recursiva.

Por último, modificar la interfaz para que ahora muestre el valor de los contadores:

```
$ ./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
Comparisons: 7
Swaps: 4
```

Llenar cuadro comparativo

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

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 hace la parte C):

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

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?

Parte C: Ejercicios estrella

Algoritmo de la burbuja

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

```
struct sorting_stats bubble_sort(int *a, unsigned int length)
```

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

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.

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 esta nueva función debe ser:

```
struct sorting_stats rand_quick_sort(int *a, unsigned int length)
```

Agregarlo también como nueva opción al menú inicial, utilizando la letra `r`.

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

Orden alternativo

Hasta ahora hemos trabajado con el orden natural de los números enteros. Pero se podría dar un orden diferente, una definición alternativa que indique cuándo un número "es menor o igual" que otro. Por ejemplo,

[1] Ver en wikipedia

definamos el orden $\leq'$ de la siguiente forma:

$$n \leq' m \equiv \begin{cases} par(m) \vee n \leq m & impar(n) \\ par(m) \wedge n \leq m & par(n) \ . \end{cases}$$

Se puede ver que si ordenamos el arreglo

```
1 5 8 9 2 4 0
```

usando el orden $\leq'$ obtendríamos

```
1 5 9 0 2 4 8
```

Notar que el efecto buscado es que todos los números impares queden ordenados en la parte izquierda del arreglo, y los pares a la derecha. Implementar la función

```
struct sorting_stats odd_even_sort(int *a, unsigned int length)
```

que ordena el arreglo usando ese orden particular. Agregarlo al menú con la opción 'o'.

Ayuda: Lo único que hay que hacer es tomar el código de cualquier algoritmo de ordenación de la parte B y cambiar todas las comparaciones para que usen $\leq'$ en vez de usar el orden $\leq$.

Entrega y evaluación

- Fecha de entrega del código:
  - Jueves 26 de marzo, el mismo día se toma el parcialito individual. Excepcionalmente se tomará esta evaluación un Jueves por ser el Martes anterior feriado.
  - Entregar el proyecto por mail al profesor a cargo del grupo.

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.
- 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`).