# Proyecto 1: Algoritmos de Ordenación

Algoritmos y Estructuras de Datos II - Laboratorio

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

## 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.

## 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
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
```

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, guardados en el archivo `input/example.in`:

```
5
8 5 3 1 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.in
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'` (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, unsigned 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, 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".

### 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, 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)
```

**Punto ⋆ 1** *Otro algoritmo de ordenación simple es el algoritmo de la burbuja, o* bubble sort[1]*. Implementarlo y agregarlo como opción al menú inicial.*

**Punto ⋆ 2** *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.*

*Notar que* **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).*

*Otra cosa importante es que no deben empezar a repetir código dentro de grandes bloques de* `if`*. Si la solución cae en este caso, no cuenta como resolución del punto estrella.*

**Punto ⋆ 3** *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.*

*Notar que, para lograr este punto, la implementación final del algoritmo debe permitir al llamador poder elegir si usar el pivote aleatorio o no. 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.*

## 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:

```
Número de comparaciones: 10
Número de swaps: 4
5
0 1 3 5 8
```

Para cada algoritmo, recolectar datos de cantidad de comparaciones y swaps para arreglos de 3 largos distintos:

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

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

- ordenado ascendentemente

[1] Ver en wikipedia

- ordenado descendentemente
- desordenado

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

Luego, 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 |
| 100 | Ordenado asc | | | | | | | | |
| | Ordenado desc | | | | | | | | |
| | Desordenado | | | | | | | | |
| 1000 | Ordenado asc | | | | | | | | |
| | Ordenado desc | | | | | | | | |
| | Desordenado | | | | | | | | |
| 10000 | Ordenado asc | | | | | | | | |
| | Ordenado desc | | | | | | | | |
| | Desordenado | | | | | | | | |

**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? 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.*

## Recordar

- Fecha de entrega y evaluación: Martes 9 de abril.
- Los grupos son de dos personas, pero se rinde individual.
- 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.