

# Algoritmos y Estructuras de Datos II

TALLER - 29 de marzo 2022

## Laboratorio 2: Ordenación

- Revisión 2018: Sergio Canchi
- Revisión 2021: Marco Rocchietti
- Revisión 2022: Marco Rocchietti

### Objetivos

1. Implementar el algoritmo de ordenación por inserción (*insertion-sort*)
2. Implementar el algoritmo de ordenación rápida (*quick-sort*)
3. Comparar desempeño de los algoritmos *selection-sort*, *insertion-sort* y *quick-sort* en distintos ejemplos
4. Lectura y comprensión del código entregado por la cátedra
5. Trabajar con implementaciones opacas de funciones leyendo su documentación
6. Abstraer la noción de orden
7. Usar procedimientos en C
8. Uso de funciones locales en módulos en C

### Ejercicio 1: Insertion Sort

Dentro de la carpeta **ej1** se encuentran los siguientes archivos

Archivo	Descripción
<b>array_helpers.h</b>	Prototipos y descripciones de las funciones auxiliares para manipular arreglos.
<b>array_helpers.c</b>	Implementaciones de las funciones de la librería <code>array_helpers</code>
<b>sort_helpers.h</b>	Prototipos y descripciones de las funciones <code>goes_before()</code> , <code>swap()</code> y <code>array_is_sorted()</code>
<b>sort_helpers.o</b>	Archivo binario con las implementaciones de las funciones declaradas en <code>sort_helpers.h</code> (código compilado para la arquitectura <b>x86-64</b> )
<b>sort.h</b>	Prototipo de la función <code>insertion_sort()</code> y su descripción
<b>sort.c</b>	Contiene una implementación incompleta de <code>insertion_sort()</code> , falta implementar <code>insert()</code>
<b>main.c</b>	Programa principal que carga un <i>array</i> de números, luego lo ordena con la función <code>insertion_sort()</code> y finalmente comprueba que el arreglo sea permutación ordenada del que se cargó inicialmente.



Si se trabaja en una computadora con arquitectura distinta a **x86-64**, entonces seleccionar y renombrar uno de los siguientes archivos, `sort_helpers.o_32` o `sort_helpers.o_macos` según la arquitectura de su máquina.

## Parte A: Ordenación por Inserción

Se debe realizar una implementación del algoritmo de ordenación por inserción (alias *insertion-sort*). Para ello es necesario completar la implementación del “procedimiento” `insert()` en el módulo `sort.c`. Como guía se puede examinar el resto del archivo `sort.c` y la definición del [algoritmo de ordenación por inserción vista en el teórico](#). El algoritmo debe ordenar con respecto a la relación `goes_before()` declarada en `sort_helpers.h` cuya implementación está oculta puesto que viene ya compilada en `sort_helpers.o`.

## Parte B: Chequeo de Invariante

Se debe modificar el “procedimiento” `insertion_sort()` agregando la verificación de cumplimiento de la invariante del ciclo `for` que se vio en el teórico. Por simplicidad sólo se debe verificar la siguiente parte de la Invariante:

- el segmento inicial `a[0,i)` del arreglo está ordenado.

Para ello usar las funciones `assert()` y `array_is_sorted()`.

## Compilación

Una vez implementados los incisos *a)* y *b)*, se puede compilar ejecutando:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c array_helpers.c sort.c main.c
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -no-pie array_helpers.o sort.o sort_helpers.o main.o -o sorter
```

la opción `-no-pie` tiene que ver con que se están “linkeando” los objetos `array_helpers.o`, `sort.o` y `main.o` compilados en nuestra computadora con el objeto precompilado `sort_helpers.o`, cuya compilación fue realizada en una computadora distinta. En consecuencia esta opción puede ser necesaria para lograr compatibilidad entre los archivos binarios durante el “linkeo” y así poder generar el ejecutable. El programa puede ejecutarse de la siguiente manera:

```
$ ./sorter ../input/example-unsorted.in
```

Si el programa funciona bien en ese ejemplo (es decir, si no reporta error), probar con otros archivos de la carpeta `../input`, sin olvidar realizar una prueba con el archivo `../input/empty.in`

Analizar los resultados del programa y responder: ¿**Qué relación implementa la función** `goes_before()`?

## Ejercicio 2: Quick Sort I

En este ejercicio se realizará una implementación *top-down* del algoritmo de ordenación rápida vista en el teórico. En la carpeta `ej2` se encuentran los siguientes archivos:

Archivo	Descripción
<code>array_helpers.h</code>	Es el mismo que en el ejercicio anterior.
<code>array_helpers.c</code>	Es el mismo que en el ejercicio anterior.
<code>sort_helpers.h</code>	Contiene además la declaración y descripción de <code>partition()</code>
<code>sort_helpers.o</code>	Contiene implementaciones ilegibles de esas funciones (código compilado para la arquitectura <b>x86-64</b> )
<code>sort.h</code>	Contiene descripción de la función <code>quick_sort()</code>
<code>sort.c</code>	Contiene una implementación muy incompleta de <code>quick_sort()</code> , además falta implementar <code>quick_sort_rec()</code>
<code>main.c</code>	Contiene el programa principal que carga un arreglo de números, luego lo ordena con la función <code>quick_sort()</code> y finalmente comprueba que el arreglo sea una permutación ordenada del que se cargó inicialmente.



*Si se trabaja en una computadora con arquitectura distinta a **x86-64**, entonces seleccionar y renombrar uno de los siguientes archivos, `sort_helpers.o_32` o `sort_helpers.o_macos` según la arquitectura de su máquina.*

### Parte A: Implementación de `quick_sort_rec()`

Implementar el “procedimiento” `quick_sort_rec()` en el archivo `sort.c`. Tener en cuenta que **no es necesario** implementar la función `partition()` puesto que la misma ya está implementada (aunque no puede leerse su código por estar compilada en `sort_helpers.o`). Para saber cómo utilizarla, examinar su descripción en `sort_helpers.h`.

A modo de guía se puede revisar la presentación del algoritmo de ordenación rápida realizada en la [clase del teórico](#).

### Parte B: Función `main()`

Se debe abrir el archivo `main.c` y completar la función `main()` con una llamada al “procedimiento” `quick_sort()`. Para entender cómo utilizar este “procedimiento”, examinar el archivo `sort.h`.

## Compilación

Una vez completadas las partes A y B, compilar el código con `gcc` siguiendo el mismo método del ejercicio 1.

## Ejercicio 3: Quick Sort II

En la carpeta `ej3` se encuentran los siguientes archivos

Archivo	Descripción
<code>sort_helpers.h</code>	Contiene descripciones de las funciones <code>goes_before()</code> , <code>swap()</code> y <code>array_is_sorted()</code>
<code>sort_helpers.o</code>	Contiene implementaciones ilegibles de todo lo descrito en <code>sort_helpers.h</code> (código compilado para la arquitectura <b>x86-64</b> ). Notar que la función <code>partition()</code> no está más aquí.
<code>sort.h</code>	Contiene descripción de la función <code>quick_sort()</code>
<code>sort.c</code>	contiene una implementación incompleta de <code>quick_sort()</code> , falta implementar <code>quick_sort_rec()</code> y <code>partition()</code> .



*Si se trabaja en una computadora con arquitectura distinta a **x86-64**, entonces seleccionar y renombrar uno de los siguientes archivos, `sort_helpers.o_32` o `sort_helpers.o_macos` según la arquitectura de su máquina.*

Copiar los archivos `array_helpers.h`, `array_helpers.c` y `main.c` del *ejercicio 2*. Luego copiar el “procedimiento” `quick_sort_rec()` (también del *ejercicio 2*) en el archivo `sort.c` y **definir** allí la función `partition()` usando como guía la presentación que se dio del algoritmo de ordenación rápida en la [clase del teórico](#).

### Compilación

Una vez completada la definición de `partition()`, compilar el código con `gcc` siguiendo el mismo método del *ejercicio 1*.

## Ejercicio 4: Versus

Realizar una comparación de todos los algoritmos de ordenación implementados en este laboratorio. En la carpeta **ej4** se encuentran los siguientes archivos:

Archivo	Descripción
<b>sort_helpers.h</b>	Se agregan nuevas declaraciones de funciones para manejo de contadores
<b>sort_helpers.o</b>	Contiene implementaciones ilegibles de todo lo descrito en <code>sort_helpers.h</code> (código compilado para la arquitectura <b>x86-64</b> )
<b>sort.h</b>	Contiene las declaraciones y descripciones de las implementaciones de los métodos de ordenación <i>selection-sort</i> , <i>insertion-sort</i> y <i>quick-sort</i>
<b>sort.c</b>	Contiene las definiciones incompletas de las funciones declaradas en <code>sort.h</code> . Deben completarse con el código de los ejercicios anteriores.
<b>main.c</b>	Contiene el programa principal que carga un arreglo de números, luego lo ordena usando alguno de los algoritmos de ordenación implementados y muestra: <ul style="list-style-type: none"><li>• Tiempo de ejecución</li><li>• Número de comparaciones</li><li>• Intercambios realizados.</li></ul>



Si se trabaja en una computadora con arquitectura distinta a **x86-64**, entonces seleccionar y renombrar uno de los siguientes archivos, `sort_helpers.o_32` o `sort_helpers.o_macos` según la arquitectura de su máquina.

Copiar los archivos **array\_helpers.h** y **array\_helpers.c** del ejercicio anterior y luego:

1. Abrir el archivo **sort.c** y copiar el código de cada uno de los algoritmos de ordenación resueltos en los ejercicios anteriores.
2. Abrir el archivo **main.c** y completar la función `main()` siguiendo los pasos indicados en los comentarios.

## Compilación y Ejecución

Una vez completados los ítems 1 y 2, compilar el código con **gcc** siguiendo el mismo método del ejercicio 1.

Analizar los resultados de la ejecución del programa para distintos ejemplos y sacar conclusiones sobre el desempeño de cada algoritmo de ordenación.