# Examen final 2022-02-07

## 95.11 - Algoritmos y Programación I - Curso Essaya

### Objetivo

Se dispone de los siguientes archivos:

- Archivos provistos con código:
  - `Makefile`
- Archivos a completar con código:
  - `ej1.c`, `ej2.c`, `ej3.c`, `ej4.c`, `ej5.c`: Implementación + pruebas de cada ejercicio

Al compilar con `make`, se genera un archivo ejecutable para cada ejercicio: `ej1`, `ej2`, etc. Cada uno de ellos, al ejecutarlo, corre las pruebas para verificar el correcto funcionamiento de la implementación.

El examen se aprueba con al menos 3 ejercicios correctamente resueltos. Un ejercicio se considera correctamente resuelto si:

- El programa `ej<n>` compila sin advertencias ni errores
- La implementación cumple con lo pedido en el enunciado

En algunos ejercicios se incluye un ejemplo de uno o dos casos de prueba y queda a cargo del alumno agregar más casos de prueba, para los que se provee sugerencias. En otros ejercicios se provee únicamente sugerencias. La implementación de las pruebas adicionales es **opcional**, pero se recomienda hacerlo ya que permite asegurar que la resolución del ejercicio es correcta.

### Makefile

Para compilar el ejercicio `<n>`: `make ej<n>`. Por ejemplo, para compilar y ejecutar el ejercicio 1:

```
$ make ej1
gcc -Wall -pedantic -std=c99 ej1.c -o ej1
$ ./ej1
ej1.c: OK
```

### Salida del programa

Al ejecutar `./ej<n>` se imprime el resultado de las pruebas. Si todas las pruebas pasan correctamente, se imprime `OK`. En caso contrario, cuando una de las verificaciones falla, se imprime un mensaje de error y el programa termina su ejecución. Por ejemplo:

```
$ ./ej1
ej1: ej1.c:45: main: Assertion `p != NULL' failed.
sh: “./ej1” terminated by signal SIGABRT (Abort)
```

**Recordar:** Correr cada uno de los ejercicios con `valgrind --leak-check=full` para mayor seguridad de que la implementación es correcta.

## Pruebas

Se recomienda usar la función `assert` de la biblioteca estándar para verificar condiciones en las pruebas. Ejemplo de uso:

```c
#include <stdio.h>
#include <assert.h>

// funcion a probar
int sumar(int a, int b) {
    return a + b;
}

// pruebas
int main(void) {
    assert(sumar(0, 0) == 0);
    assert(sumar(2, 3) == 5);
    assert(sumar(2, -2) == 0);

    printf("%s: OK\n", __FILE__);
    return 0;
}
```

Nota: A veces para depurar un error en las pruebas es útil imprimir valores; se permite el uso de `printf()` para ello.

Nota: A veces para implementar las pruebas es útil utilizar números aleatorios. Se permite el uso de `rand()` para ello. En ese caso, se recomienda ejecutar `srand(0);` al inicio del programa para asegurar que la secuencia de números aleatorios sea siempre la misma, y así facilitar la depuración.

## Ejrcicios

**Ejercicio 1** Un árbol binario es una estructura enlazada en la que el primer nodo se llama raíz, y cada nodo contiene referencias a otros dos nodos, llamados *hijo izquierdo* y *derecho*.

Sea la estructura `nodo_t` que representa un nodo del árbol, cuyo dato es de tipo `char`. Se pide implementar las funciones:

- `void arbol_destruir(nodo_t *nodo)`
- `nodo_t *arbol_crear(char s[])` que devuelve un árbol binario creado a partir de su descripción `s`.

  La descripción consiste en una cadena de caracteres con las siguientes propiedades:

  - el árbol vacío se representa con un punto (`.`)
  - un árbol no vacío se representa con un caracter distinto de `.` (que corresponde al dato contenido en el nodo raíz), seguido primero de la descripción del hijo izquierdo y después de la descripción del hijo derecho.

  Ejemplo: la descripción `"abc...de..f.."` corresponde al siguiente árbol:

```
|           a   <--- raiz
|         /  \
|        b    d
|      /     / \
|     c     e   f
```

Recomendación: pensar ambas funciones en forma recursiva.

Nota: Se permite suponer que la cadena está bien formada y que `malloc` nunca falla.

**Ejercicio 2** Escribir la función `int buscar(const char *s, const char *p)`, que encuentra el primer caracter de `s` que coincide con alguno de los caracteres de `p`, y devuelve su posición, o -1 en caso de no encontrarlo.

Ejemplo: `buscar("algoritmos", "qrst") -> 4` (ya que `s[4] == 'r'`)

**Ejercicio 3** Sea la siguiente estructura que representa el resultado de un hisopado de Covid-19:

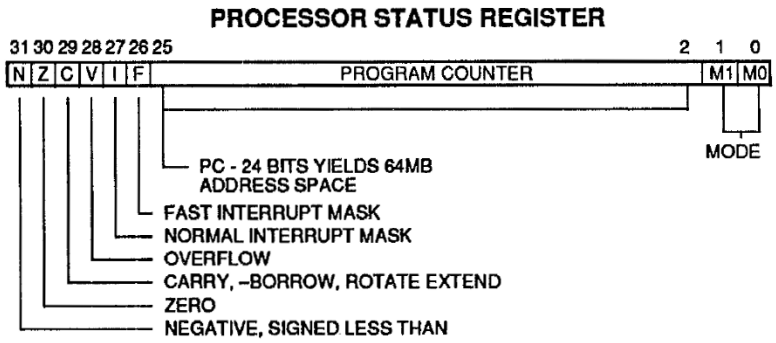
```c
typedef struct {
    uint8_t dia;
    uint8_t mes;
    uint16_t anno;

    uint32_t dni;

    bool es_positivo;
} resultado_t;
```

Escribir la función `void ordenar_resultados(resultado_t resultados[], size_t n)` que ordena los resultados en forma creciente según la fecha, y para los resultados de la misma fecha, según el DNI.

**Ejercicio 4** En el microprocesador ARM1 hay un registro de 32 bits llamado *processor status register* (PSR), que tiene la siguiente estructura:




Uno de los datos que almacena el registro es el *program counter* (PC), que es un número sin signo de 24 bits.

Escribir las funciones:

- `uint32_t psr_pc_get(uint32_t *psr)`, que extrae el valor PC del registro PSR
- `void psr_pc_set(uint32_t *psr, uint32_t pc)`, que asigna el valor PC en el registro PSR
- `bool psr_pc_sumar(uint32_t *psr, int d)`, que calcula el valor $PC' = PC + d$ (siendo $d$ un valor con signo) y si el resultado es válido, guarda el nuevo valor de $PC'$ en el registro. En caso contrario, no modifica el registro y devuelve `false`. Se considera que el resultado es válido cuando está en el rango $[0, 2^{24} - 1]$

**Ejercicio 5** Escribir un programa que recibe por argumentos de línea de comandos dos nombres de archivos binarios, y determina si sus contenidos son iguales o no. En caso de que sean distintos, debe imprimir el índice del primer byte diferente entre ambos. En caso de que sean iguales, no imprime nada.

Ejemplos: Si el archivo `a.bin` contiene `abcd` y el archivo `b.bin` contiene `ab.d`:

```
$ ./ej4 a.bin b.bin
2
```

Si el archivo `a.bin` contiene `abcd` y el archivo `c.bin` contiene `abcdef`:

```
$ ./ej4 a.bin c.bin
4
```