# Examen final 2021-03-22

95.11/75.02 - 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** Escribir la función `bool octal_decodificar(const char s[], uint32_t *n)` que recibe una cadena con un número en notación octal, y guarda en `n` el número decodificado. Devuelve `true` si la decodificación fue exitosa.

En la implementación **no** se permite utilizar ninguna función de la biblioteca estándar.

**Ejercicio 2** Sea un archivo binario que almacena un listado de evaluaciones y calificaciones. Cada registro tiene la siguiente información:

- Fecha (año, mes, día)
- Padron o legajo (número entero)
- Tipo (puede ser `PARCIAL` o `FINAL`)
- Calificación (número entero)

Se pide:

- Declarar un tipo de dato `examen_t` (y cualquier tipo de dato auxiliar necesario) para representar un examen.
- Implementar la función `bool guardar_examenes(examen_t examenes[], size_t cantidad, const char *nombre_archivo)` que recibe un arreglo de registros y los guarda en un archivo binario con el nombre indicado.
- Implementar la función `examenes_t *leer_examenes(const char *nombre_archivo, size_t *cantidad)` que lee del archivo la información guardada, y devuelve el arreglo reservado en memoria dinámica, y guarda en `cantidad` la cantidad de exámenes leídos.

Nota: No es necesario implementar `examen_t` como un TDA con funciones y encapsulamiento; es suficiente con declarar el tipo de dato. La elección de los tipos de datos y el formato del archivo queda a libre criterio.

**Ejercicio 3** Sea el TDA Conjunto, que permite almacenar un conjunto de números enteros entre `0` y `k - 1` (inclusive). En un conjunto, un número cualquiera puede pertenecer o no, pero no puede estar incluido más de una vez.

Se pide declarar el tipo de dato `conjunto_t` y las funciones:

- `conjunto_t *conjunto_crear(size_t k)` que crea un conjunto vacío con capacidad `k`.
- `void conjunto_agregar(conjunto_t *c, int n)` que agrega el número `n` al conjunto (no hace nada si el número ya pertenece). Asumir que `0 <= n < k`.
- `void conjunto_quitar(conjunto_t *c, int n)` que quita el número del conjunto (no hace nada si el número no pertenece).
- `bool conjunto_pertenece(conjunto_t *c, int n)` que determina si el número pertenece o no al conjunto.
- `void conjunto_eliminar(conjunto_t *c)`

Todas las funciones (salvo `conjunto_crear`) deben ser de **tiempo constante**.

Ayuda: implementar el TDA como un vector de booleanos.

**Ejercicio 4** Dado el TDA lista enlazada en el que los nodos guardan un dato de tipo `int`, implementar la función `void lista_mapear(lista_t *lista, int (*f)(int dato))`, que, para cada entero en la lista, llama a la función `f` pasándole como parámetro el entero, y luego reemplaza el mismo por el resultado de `f`.

**Ejercicio 5** Escribir una función que recibe un arreglo de `n` caracteres únicos y un número `k` ($k \leq n$), e imprime todas las posibles combinaciones sin repetición de `k` caracteres tomados del arreglo.

Por ejemplo:

```
char caracteres[] = {'a', 'b', 'c', 'd'};
size_t n = 4;
size_t k = 2;
```

```
combinaciones(caracteres, n, k);
```

Imprime:

```
ab
ac
ad
bc
bd
cd
```

Ayuda: pensar el problema en forma recursiva. Es posible que sea necesario crear funciones auxiliares.