

# Obligatorio 2: Procesamiento de imágenes

26 de abril de 2022

## 1. Generalidades

La aprobación de este curso requiere la correcta implementación de dos pequeños proyectos de programación (que llamaremos obligatorios). Estos son propuestos en dos momentos del curso, y que forman parte de un mismo paquete. Cada obligatorio será entregado a través de una página web habilitada para tales fines, con fecha límite de entrega señalada en la misma página. Estas entregas se complementan con la prueba parcial escrita cuyo objetivo es evaluar aspectos más teóricos relacionados con los obligatorios.

Es importante recalcar que **tanto la prueba escrita como los proyectos entregados son individuales**. El sistema de recepción de entregas, además de almacenar los archivos enviados por los estudiantes, realiza un control de copias contra las entregas de otros estudiantes, así como de programas similares que se encuentran en la web. Ese programa es capaz de detectar copias “maquilladas”, es decir, donde se cambiaron nombres de variables u otras formas de ocultar una copia. Este asunto debe ser bien entendido. No tenemos ningún inconveniente en que discutan soluciones, miren en la web, etc. pero el **trabajo entregado debe ser realmente el producto de vuestro trabajo y si el programa de control de copia detecta que hubo copia, ello implica una sanción que puede implicar la pérdida del curso e incluso sanciones mayores, tal como está especificado en el reglamento de la Facultad.**<sup>1</sup>

La nota de aprobación de la unidad curricular esta definida por una evaluación global por parte de los docentes que incluye los obligatorios y el parcial (no hay examen).

Para aprobar la asignatura se debe tener:

1. Alcanzar un mínimo en el total (60 %)
2. Alcanzar un mínimo en cada obligatorio (25 %)
3. Alcanzar un mínimo en el parcial (25 %)

Los puntajes de cada instancia son:

1. Obligatorio 1: 24 puntos (6 puntos mínimo)
2. Obligatorio 2: 40 puntos (10 puntos mínimo)
3. Parcial final: 36 puntos (9 puntos mínimo)
- 4. Puntos totales: 100 puntos (60 puntos mínimo)**

---

<sup>1</sup><https://www.fing.edu.uy/es/gestion/normas-y-reglamentos>

## 1.1. Archivos a entregar

El archivo entregado debe ser un archivo comprimido en formato **zip**, con el siguiente nombre: **nombres\_separados\_por\_infraguiones.apellidos\_separados\_por\_infraguiones.zip**. Por ejemplo, supongamos que su nombre es Juan Pablo Perez Fernandez, entonces debe subir un archivo de nombre **Juan\_Pablo.Perez\_Fernandez.zip**.

El contenido del archivo zip debe incluir los archivos fuentes y estos deben estar en la raíz del zip (**NO** se deben crear directorios dentro del zip). En particular, en este obligatorio tendrán que implementar y entregar 6 archivos:

```
bits.c  
bits.h  
imagen.c  
imagen.h  
obligatorio2.c  
Makefile
```

Pueden crear un zip desde un sistema Unix con el comando **zip**. Desde una carpeta que contenga los archivos anteriores, deben introducir el siguiente comando en la terminal :

```
zip Juan_Pablo.Perez_Fernandez.zip bits.c bits.h imagen.h imagen.c obligatorio2.c Makefile
```

### 1.1.1. bits.h

El archivo de encabezado pedido en el obligatorio 1. Puede contener modificaciones respecto a la entrega anterior.

### 1.1.2. bits.c

Debe contener las definiciones de las funciones (implementaciones) pedidas en el obligatorio 1. Puede contener modificaciones respecto a la entrega anterior.

### 1.1.3. imagen.h

Debe contener las definiciones de tipo y declaraciones de todas las funciones pedidas.

### 1.1.4. imagen.c

Debe contener las definiciones de las funciones (implementaciones) pedidas. Además, puede contener otras funciones auxiliares, macros, tipos auxiliares, etc. que le resulte útil para sus implementaciones.

### 1.1.5. obligatorio2.c

Debe contener la función **main**, que implemente la interfaz de línea de comandos pedida. Este archivo, a diferencia de lo que ocurrió con el del obligatorio 1, será parte de la evaluación en todos los casos.

### 1.1.6. Makefile

Es el archivo que contiene todo lo necesario para compilar, generar bibliotecas y el ejecutable. El **Makefile** para este obligatorio podría ser así (pero no es la única forma de escribirlo):

```

# -----
# Makefile de ejemplo del obligatorio 2.
# Programación Para Ingeniería Eléctrica
# -----

# Define una constante que tiene las banderas de compilación
COMOPT=-Wall -std=c99 -ggdb -Werror
##### -Wall : muestra todos los warnings
##### -std=c99 : utiliza el estandar c99
##### -ggdb : permite utilizar los ejecutables con el debugger
##### -Werror : todos los warnings son errores

# Genera el ejecutable
obligatorio2: obligatorio2.o libbits.a libimagen.a
    gcc $(COMOPT) -o obligatorio2 obligatorio2.o -L./ -limagen -lbits
        -lm

# Compila bits.c a código objeto
bits.o: bits.c
    gcc $(COMOPT) -o bits.o -c bits.c

# Compila imagen.c a código objeto
imagen.o: imagen.c
    gcc $(COMOPT) -o imagen.o -c imagen.c

# Compila obligatorio2.c a código objeto
obligatorio2.o: obligatorio2.c
    gcc $(COMOPT) -o obligatorio2.o -c obligatorio2.c

# Crea la biblioteca libbits
libbits.a: bits.o
    ar rcs libbits.a bits.o

# Crea la biblioteca libimagen
libimagen.a: imagen.o
    ar rcs libimagen.a imagen.o

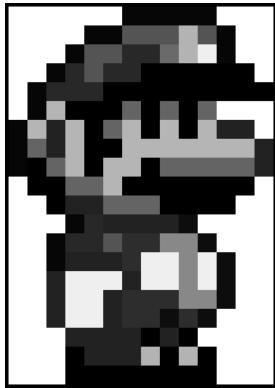
# Borra los archivos (si existen) que son producto de la compilación
clean:
    rm -f *.a *.o obligatorio2

```

Con el archivo **Makefile** anterior se puede generar las bibliotecas y también compilar un pequeño código de prueba **obligatorio2.c** que simplemente llama a las funciones implementadas con ciertos valores como parámetros de entrada e imprime el resultado. Esto les debe servir a ustedes para ver si dan los resultados correctos las funciones que están en la biblioteca. El archivo genera la biblioteca que se utiliza en el código de **obligatorio2.c** y genera el ejecutable **obligatorio2** que es posible invocar en una terminal como se hizo en el obligatorio 1.

## 2. Introducción al problema

El problema que se plantea en este obligatorio, el procesamiento de imágenes, es una de las subáreas más populares e importantes del procesamiento de señales. Si bien desde el punto de vista teórico y formal, las herramientas para trabajar con este tipo de problemas se ven recién en los últimos dos años de la carrera de Ingeniería Eléctrica, es posible trabajar con, y comprender informalmente, muchos algoritmos importantes de procesamiento de imágenes. En este caso vamos a trabajar con aspectos muy elementales de ese mundo, como lo son la lectura, escritura y encriptado de imágenes.



```

255 255 255 255 255 255 6   6   6   6   6   255 255 255
255 255 255 255 6   6   84  84  84  179 84  6   255 255
255 255 255 6   84  84  40  40  106 179 239 6   255 255
255 255 6   40  84  40  40  0   0   0   0   0   0   0   255
255 6   40  40  40  0   0   0   0   0   0   0   0   0   0
255 6   181 0   0   0   100 0   100 0   100 255 255 255
6   181 34  181 0   100 181 0   181 0   181 34  34  255
6   100 34  181 0   0   181 181 181 181 181 181 34
6   0   100 181 0   181 181 0   100 100 100 100 100 34
255 0   0   100 100 181 0   0   0   0   0   0   0   0   255
255 255 0   34  34  100 100 100 0   0   0   0   255 255
255 255 255 6   40  34  34  34  34  13  255 255 255 255
255 255 6   40  40  84  45  45  136 136 13  255 255 255
255 255 6   34  34  34  45  239 239 136 239 13  255 255
255 255 34  239 239 239 34  239 239 136 239 13  255 255
255 255 34  239 239 34  45  45  45  136 136 13  255 255
255 255 34  239 239 34  45  45  13  45  13  255 255 255
255 255 34  34  34  34  0   34  0   255 255 255 255
255 255 255 0   34  34  34  34  179 0   179 0   255 255 255
255 255 255 0   0   0   0   0   0   0   0   255 255 255

```

Figura 1: Izq.:Imagen en escala de grises de  $20 \times 20$  píxeles. Der.: píxeles de la imagen

## 2.1. Representación digital de imágenes

Lo primero que tenemos que definir es cómo se representa una imagen digitalmente, de modo de poder almacenarla en memoria y trabajar con ella. Consideraremos en primer lugar las imágenes en escala de grises, como la que se ve en la Figura 1. La representación usual de este tipo de imágenes es la de una matriz de tamaño  $m \times n$ , donde  $m$  es la cantidad de filas y  $n$  es la cantidad de columnas, en píxeles, de la imagen. Si denominamos como  $I$  a tal matriz, y medimos la posición de los píxeles de la imagen a partir de su esquina superior izquierda  $(0, 0)$ , el valor de la posición  $I(i, j)$  de la matriz correspondiente a esa imagen nos indicará la intensidad del pixel ubicado en la coordenada  $(i, j)$ ; mientras más alto el valor de  $I(i, j)$ , más brillante será el pixel de la coordenada  $(i, j)$  en la imagen. Para imágenes en niveles de gris es usual limitarse a 8 bits por pixel, que se traduce en que  $I(i, j)$  puede tomar valores entre 0 y 255, siendo 0 el color *negro* y 255 el *blanco*. La Figura 1 da un ejemplo de lo anterior.

## 2.2. Representación de imágenes a color

Existen muchas formas de representar una imagen a color bidimensional. En este caso, el valor de cada pixel, en lugar de representar una intensidad de gris, representa un color. En general, esto se logra representando a cada color como un vector de *canales* que definen al color. Ejemplos de estos son la descomposición RGB<sup>2</sup> (Red, Green, Blue) o la HSV<sup>3</sup> (Hue, Saturation, Value).

En nuestro caso utilizaremos la representación RGB, que es una representación *aditiva* del color, en el sentido de que el color resulta de sumar la intensidad lumínica de los tres canales; ésta es la representación de colores utilizada en todos los monitores y televisores. En esta representación, cada pixel es representado por una terna de valores enteros  $(r, g, b)$ , cada uno en el rango  $[0, 255]$  que, de la misma manera que en las imágenes de tono de gris, indican la intensidad del canal correspondiente. El valor  $(0, 0, 0)$  se corresponde con el negro, y el  $(255, 255, 255)$  con el blanco.

Notar que cada canal requiere solo un byte para representar su valor. De esta manera, en un `unsigned int` de 32 bits se puede almacenar los valores de los tres canales: los 8 bits más significativos (bits 24 al 31) se ignoran, luego los siguientes 8 bits (del 16 al 23) contienen la intensidad del rojo, los siguientes 8 (del 8 al 15) la del verde, y finalmente los 8 menos significativos (del 0 al 7) la del azul. La Figura 2 muestra esto gráficamente. La Figura 3 muestra un ejemplo muy sencillo de imagen RGB.

no usado	rojo	verde	azul
31 ... 24	23 ... 16	15 ... 8	7 ... 0

Figura 2: Representación de un pixel RGB en un entero

<sup>2</sup><https://es.wikipedia.org/wiki/RGB>

<sup>3</sup>[https://es.wikipedia.org/wiki/Modelo\\_de\\_color\\_HSV](https://es.wikipedia.org/wiki/Modelo_de_color_HSV)

	255, 0, 0	255, 255, 0	0, 255, 0	0, 255, 255
	255, 0, 0	255, 255, 0	0, 255, 0	0, 255, 255
	0, 0, 255	0, 0, 0	255, 255, 255	255, 0, 255
	0, 0, 255	0, 0, 0	255, 255, 255	255, 0, 255

Figura 3: Imagen RGB de tamaño  $4 \times 4$ . Izq.: imagen, Der.: pixeles de la imagen. Cada terna representa un pixel con sus valores de rojo (R), verde (G) y azul (B). Noten que 255 quiere decir que ese canal tiene el valor máximo posible



Figura 4: Imagen a color  $I$  y sus tres componentes  $I_R$ ,  $I_G$  y  $I_B$ . Notar que el blanco se forma con el máximo de los tres canales, por lo que todos aparecen también con la máxima intensidad en el fondo. Sin embargo, el canal azul (más a la derecha) aparece más oscuro, debido a la poca presencia del azul en la imagen mostrada.

Acorde con lo arriba descrito, diremos que una imagen color  $I$  es una matriz de tamaño  $m \times n$ , donde cada elemento es una terna  $(r, g, b)$ , de modo que

$$I(i, j) = (r_{ij}, g_{ij}, b_{ij}), 0 \leq i < m, 0 \leq j < n.$$

### 2.3. Representación en memoria

Hemos visto que una imagen se representa como una matriz de pixeles. En este obligatorio utilizaremos punteros dobles como forma de representar y almacenar estas matrices de pixeles RGB dentro de un programa C. En la sección de especificación de requerimientos se explicará con más detalles esta representación.

## 3. Objetivo de la tarea

Habiendo definido cómo se representan las imágenes en dentro de un programa, vayamos a lo que es el objetivo de este proyecto, es decir, poder manipular de manera muy elemental imágenes mediante la implementación de una serie de funciones. En general tomaremos una imagen de entrada, operaremos sobre ella y produciremos una imagen de salida, de manera que esta última es una versión alterada de la primera. Vamos a leer y escribir archivos en formato *.ppm*.

En este obligatorio abordaremos fundamentalmente cuatro puntos:

1. Lectura de archivos de imagen PPM (color).
2. Escritura de archivos de imagen PPM.
3. Conversión de imágenes de color a sepia.
4. Encriptado y desencriptado de imágenes con codificación Vigènere.

## 4. Especificación de requerimientos

Dentro de `imagen.h` deberán declarar los siguientes tipos:

- Tipo `pixel_t`, que es un alias del tipo `unsigned int`.
- Tipo `Imagen_t`, en base a una estructura de nombre `imagen` con los siguientes campos:
  - `filas`: de tipo `int`
  - `columnas`: de tipo `int`
  - `pixeles`: de tipo puntero doble a `pixel_t` (es decir `**pixel_t`), que apuntará a una zona de memoria suficiente para almacenar los píxeles de la imagen.
- Tipo `FormatoPPM_t`, que es un `enum` de nombre `formato_ppm` con dos valores posibles:
  - `PLANO=0`
  - `NO_PLANO=1`
- Tipo `CodigoError_t`, basado en un `enum` de nombre `codigo_error` con los siguientes valores posibles:
  - `OK=0`: se devuelve si todo salió bien.
  - `PPM_ARCHIVO_INEXISTENTE=1`: si el archivo a abrir no se encuentra (`fopen` devuelve `NULL`).
  - `PPM_ERROR_LECTURA=2`: si ocurre un error al leer datos del archivo (por ejemplo con `fclose`, o `fscanf`).
  - `PPM_ENCABEZADO_INVALIDO=3`: se devuelve al leer el encabezado; indica que el formato del encabezado no es correcto.
  - `PPM_DATOS_INVALIDOS=4`: se devuelve al leer los datos si, por ejemplo, se termina el archivo antes de leer todos los píxeles que se esperaba leer.
  - `PPM_ERROR_ESCRITURA=5`: se devuelve ante cualquier error que ocurra durante la escritura de la imagen en la función `escribir_imagen`.
  - `PPM_CRIPTO_NO_VALIDA=6`: se devuelve ante la presencia de una cripto imagen no válida.
  - `ERROR=7`: otro error no listado anteriormente.

Asimismo, debe declarar las siguientes funciones en `imagen.h`, y definirlas en `imagen.c`:

- `CodigoError_t inicializar_imagen(Imagen_t* pin, int filas, int columnas);`

Inicializa los datos de la imagen apuntada por `pin` y reserva memoria para sus nuevos píxeles. NO se debe inicializar los píxeles con ningún valor en particular, basta con dejar el valor que por defecto asigne `malloc`.

- Parámetros:
  - `pin`: Puntero a imagen a inicializar.
  - `filas`: Cantidad de filas de la imagen.
  - `columnas`: Cantidad de columnas de la imagen.
- Retorno: Devuelve un valor de tipo `CodigoError_t` según el caso.

Nota 1: La variable `pin` es pasada como referencia (recordar que en C la forma de hacer esto es mediante punteros). Es decir, fuera de la función deben haber creado una variable de tipo `Imagen_t` y pasar su dirección de memoria como parámetro `pin`.

Nota 2: La Figura 5 muestra como debe ser la representación en memoria del puntero doble `pin->pixeles`, por más detalles se sugiere revisar el material <https://articleworld.com/dynamically-allocate-2d-array-c/>.

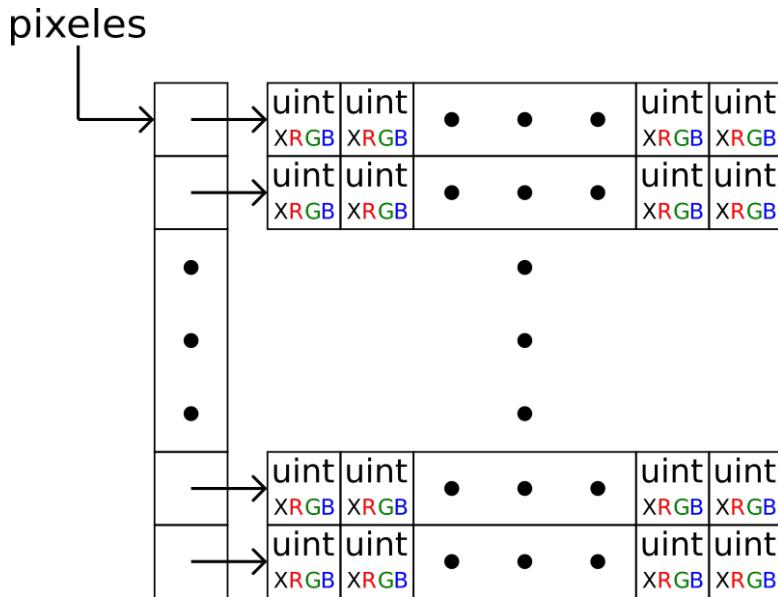


Figura 5: Representación en memoria de los pixeles de la imagen.

- `CodigoError_t destruir_imagen(Imagen_t* pin);`

Libera la memoria asociada a los pixeles de la imagen apuntada por `pin` y pone todos sus atributos en `0`.

Se asume que `pin` se encuentra inicializada al pasarse a la función.

- Parámetros:

`pin`: Puntero a imagen a destruir.

- Retorno: Devuelve un valor de tipo `CodigoError_t` según el caso.

Nota: para liberar una memoria reservada con dos niveles de referencia, se deberá liberar en sentido inverso a cómo fue reservada y la totalidad de la memoria reservada.

- `CodigoError_t duplicar_imagen(const Imagen_t* pin, Imagen_t* pout);`

Copia los atributos `filas` y `columnas` de la imagen apuntada por `pin` en la imagen apuntada por `pout`. Además reserva la memoria para los pixeles de `pout`. NO se debe inicializar los pixeles con ningún valor en particular.

Se asume que previo a la llamada de esta función, la imagen apuntada por `pout` no tiene reservada memoria para sus pixeles, es decir, no se debe llamar a `destruir_imagen`.

- Parámetros:

`pin`: Puntero a imagen fuente que se desea duplicar.

`pout`: Puntero a imagen que quedará inicializada.

- Retorno: Devuelve un valor de tipo `CodigoError_t` según el caso.

**Sugerencia:** Implemente en una sola linea de código utilizando `inicializar_imagen`.

■ `CodigoError_t leer_imagen(const char* ruta_imagen, Imagen_t* pin);`

Lee el contenido del archivo ubicado en `ruta_imagen` y lo guarda en la imagen apuntada por `pin`. Se asume que no se ha inicializado `pin` (no se ha reservado memoria para `pixeles`) y la misma se debe realizar dentro de la función. Deben implementar la lectura de imágenes en formatos *PPM* y *PPM plano* (ver Anexo A).

- Parámetros:

`ruta_imagen`: Ruta desde donde leer la imagen.

`pin`: Puntero a tipo `Imagen_t`, en el que cargar el contenido leído.

- Retorno: Devuelve un valor de tipo `CodigoError_t` según el caso.

**Sugerencia:** Utilizar la función `fscanf` para implementar la lectura de números en formato plano y `fgetc` para implementar la lectura de números en formato no plano.

■ `CodigoError_t escribir_imagen(const Imagen_t* pin, const char* ruta_imagen, FormatoPPM_t formato);`

Guarda el contenido de la imagen `pin` en el archivo especificado por `ruta`. Deben implementar la escritura de imágenes en formatos *PPM* y *PPM plano*.

Se asume que `pin` está inicializada previo a la llamada de la función.

- Parámetros:

`pin`: Puntero a imagen que se desea escribir a archivo.

`ruta_imagen`: Ruta en la que se desea escribir el archivo.

`formato`: Formato de escritura, determinando si se escribirá PPM binario o PPM plano.

- Retorno: Devuelve un valor de tipo `CodigoError_t` según el caso.

■ `CodigoError_t filtrar_sepia(const Imagen_t* pin, Imagen_t* pout);`

El filtrado sepia es un tipo de filtro fotográfico que da un aspecto de nostalgia y calidez. Aplicarle un filtro sepia a una imagen RGB es muy sencillo, ya que consiste en hacer la misma operación para cada pixel. Siendo  $(r_{in}, g_{in}, b_{in})$  los valores de brillo de un pixel genérico de la imagen de entrada, los pixels de salida para la imagen filtrada serán:

$$r_{out} = \min(\text{round}(0,393 r_{in} + 0,769 g_{in} + 0,189 b_{in}), 255) \quad (1)$$

$$g_{out} = \min(\text{round}(0,349 r_{in} + 0,686 g_{in} + 0,168 b_{in}), 255) \quad (2)$$

$$b_{out} = \min(\text{round}(0,272 r_{in} + 0,534 g_{in} + 0,131 b_{in}), 255) \quad (3)$$

Donde la operación `round` es el redondeo de la biblioteca `math.h`, y el mínimo se toma para garantizar que ningún pixel supere el valor máximo admitido de 255.

Se asume que `pin` está inicializada al pasarse a la función, mientras que `pout` no lo está.

Se pide desarrollar una función que aplique un filtro sepia sobre la imagen de entrada apuntada por `pin` y guarda el resultado en la imagen apuntada por `pout`. La reserva de memoria para los pixeles de `pout` se debe realizar **dentro** de la función.

- Parámetros:

`pin`: Puntero conteniendo la imagen a filtrar.

`pout`: Puntero conteniendo el resultado de aplicar el filtro sepia.

- Retorno: Devuelve un valor de tipo `CodigoError_t` según el caso.

- `CodigoError_t generar_cripto_imagen(Imagen_t* pcriptoim, int filas, int columnas, int min_largo_clave);`

Inicializa la imagen apuntada por `pcriptoim` reservando memoria para sus pixeles, y los rellena con claves de Vigènere `aleatorias` cuyo largo mínimo es el especificado por `min_largo_clave` y cuyo largo máximo es 19, como se explica en el Anexo B.2.

Se asume que `pcriptoim` no está inicializada previo a la llamada de la función. También se asume que  $1 \leq \text{min\_largo\_clave} \leq 19$ .

- Parámetros:

`pcriptoim`: Puntero a imagen en el que generar la cripto-imagen.

`filas`: Número de filas de la cripto-imagen deseada.

`columnas`: Número de columnas de la cripto-imagen deseada.

`min_largo_clave`: Largo mínimo de cada clave de Vigènere.

- Retorno: Devuelve un valor de tipo `CodigoError_t` según el caso.

**Sugerencia:** Para generar las cripto-imágenes sugerimos utilizar la función `rand()`<sup>4</sup> de la biblioteca `stdlib.h`, y recordar que el comando `rand() % n` genera un entero aleatorio entre `0` y `n - 1`

- `CodigoError_t validar_cripto_imagen(const Imagen_t* pcriptoim);`

Evalúa si un puntero a imagen es una cripto-imagen válida para el algoritmo de encriptación-decriptación descrito en el anexo, es decir todos sus largos son mayores o iguales a 1 y menores o iguales a 19. En caso de serlo retorna `OK`, y en caso contrario retorna `PPM_CRIPTO_NO_VALIDA`.

Se asume que `pcriptoim` está inicializada previo a la llamada de la función.

- Parámetros:

`pcriptoim`: Puntero a imagen con la candidata a cripto-imagen a evaluar.

- Retorno: Devuelve `OK` si es valida, en caso contrario retorna `PPM_CRIPTO_NO_VALIDA`.

- `CodigoError_t encriptar_imagen(const Imagen_t* pin, const Imagen_t* pcriptoim, Imagen_t* pout);`

Aplica una encriptación de Vigènere con una clave distinta para cada pixel de la imagen apuntada por `pin` de acuerdo al contenido de cada pixel de `pcriptoim`, y de la explicación en el Anexo B.

El resultado se guarda en la imagen apuntada por `pout`. La reserva de memoria para los pixeles de `pout` se debe realizar `dentro` de la función.

Se asume que `pin` y `pcriptoim` están inicializadas y que `pout` no lo está al llamar la función. También se asume que `pcriptoim` siempre seguirá el formato de cripto-imagen indicado (largo de clave menor o igual a 19).

- Parámetros:

`pin`: Puntero conteniendo la imagen a encriptar.

`pcriptoim`: Puntero a la cripto-imagen utilizada para encriptar.

`pout`: Puntero conteniendo el resultado de encriptar.

- Retorno: Devuelve un valor de tipo `CodigoError_t` según el caso.

---

<sup>4</sup>[https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_rand.htm](https://www.tutorialspoint.com/c_standard_library/c_function_rand.htm)

## 4.1. Interfaz de linea de comandos

La sintaxis será por línea de comando y estará compuesta por el nombre del ejecutable seguido del nombre de un subcomando, cuyo comportamientos se describen a continuación.

### 4.1.1. Subcomando: convertir\_formato

Subcomando para convertir de formato plano a no plano y viceversa.

```
./obligatorio2 convertir_formato RUTA_IM_ENTRADA RUTA_IM_SALIDA FORMATO_PPM
```

- **RUTA\_IM\_ENTRADA**: nombre de la imagen PPM a leer.
- **RUTA\_IM\_SALIDA**: nombre de la imagen PPM a guardar.
- **FORMATO\_PPM**: formato en que se quiere salvar el archivo de salida. Deberá ser:
  - **plano**: si se quiere guardar formato PPM binario.
  - **no\_plano**: si se quiere guardar formato PPM plano.

Notar que si **FORMATO\_PPM** es igual al formato de **RUTA\_IM\_ENTRADA** no es más que una copia del archivo.

### 4.1.2. Subcomando: filtrar\_sepia

Subcomando para aplicar el filtro sepia a una imagen.

```
./obligatorio2 filtrar_sepia RUTA_IM_ENTRADA RUTA_IM_SALIDA FORMATO_PPM
```

- **RUTA\_IM\_ENTRADA**: nombre de la imagen PPM a leer.
- **RUTA\_IM\_SALIDA**: nombre de la imagen PPM a guardar.
- **FORMATO\_PPM**: formato en que se quiere salvar el archivo de salida. Deberá ser:
  - **plano**: si se quiere guardar formato PPM plano.
  - **no\_plano**: si se quiere guardar formato PPM binario.

### 4.1.3. Subcomando: generar\_cripto\_imagen

Subcomando para generar cripto imagenes aleatorias.

```
./obligatorio2 generar_cripto_imagen FILAS COLUMNAS LARGO_CLAVE_MIN RUTA_CRIPTO_IM  
FORMATO_PPM
```

- **FILAS**: cantidad de filas de la cripto imagen a generar.
- **COLUMNAS**: cantidad de columnas de la cripto imagen a generar.
- **LARGO\_CLAVE\_MIN**: Mínimo largo de la clave de cada pixel.
- **RUTA\_CRIPTO\_IM**: nombre de la cripto imagen PPM a guardar.
- **FORMATO\_PPM**: nombre de la cripto imagen PPM a guardar. Deberá ser:
  - **plano**: si se quiere guardar formato PPM plano.
  - **no\_plano**: si se quiere guardar formato PPM binario.

#### 4.1.4. Subcomando: validar\_cripto\_imagen

Subcomando para verificar que una imagen PPM sea una cripto imagen con el formato especificado. El `main` deberá imprimir un mensaje de error y retornar 1 en caso de error, mientras que si se trata de una cripto imagen valida se debe retornar un 0 e imprimir que es una cripto imagen.

```
./obligatorio2 validar_cripto_imagen RUTA_CRIPTO_IM
```

- `RUTA_CRIPTO_IM`: ruta de la imagen PPM que se quiere verificar si es una cripto imagen.

#### 4.1.5. Subcomando: encriptar\_imagen

Subcomando para encriptar una imagen al formato plano o no. Debe verificar que la cripto-imagen sea valida, en caso contrario lanza un error y el `main` retorna 1.

```
./obligatorio2 encriptar_imagen RUTA_IM_ENTRADA RUTA_CRIPTO_IM RUTA_IM_SALIDA FORMATO_PPM
```

- `RUTA_IM_ENTRADA`: nombre de la imagen PPM a leer.
- `RUTA_CRIPTO_IM`: nombre de la cripto-imagen PPM a utilizar.
- `RUTA_IM_SALIDA`: nombre de la imagen PPM encriptada a guardar.
- `FORMATO_PPM`: formato en que se quiere salvar el archivo de salida. Deberá ser:
  - `plano`: si se quiere guardar formato PPM plano.
  - `no_plano`: si se quiere guardar formato PPM binario.

### Consideraciones y sugerencias

- **Tener especial cuidado con la especificación de los formatos.** Son formatos sencillos, pero hay que prestar mucha atención. Por ejemplo, la cantidad de espacios que puede haber entre campos de los encabezados es arbitraria (siempre mayor o igual a 1), pero **en el formato no plano** entre el fin del encabezado y los datos debe haber **uno y solo un** carácter de espacio (puede ser espacio o nueva linea o tabulador).
- Recuerden que si utilizan funciones matemáticas de `math.h` deben luego linkear con la biblioteca de matemática con la opción `-lm` al final de la linea que genera el ejecutable, para que dichas funciones estén definidas.
- Es **fundamental** liberar correctamente toda la memoria que haya sido reservada por el programa. **Parte de la evaluación del funcionamiento correcto de la tarea incluirá verificar que esto se esté haciendo correctamente.**
- Como siempre, implementar y probar de a poco.
- En caso de bugs, el GDB es su mejor amigo.

## A. Formatos de archivos de imagen

En este obligatorio trabajamos con imágenes de tipo *PPM*. Estas son un caso particular de una familia de formatos de imágenes muy sencillo, llamado *PNM*. Las imágenes *PNM* en blanco y negro son las *PGM*, mientras que a las que son a color se identifican con la extensión *PPM*. Uno de los objetivos de esta parte del obligatorio es la implementación de una biblioteca de lectura/escritura de *PPM*.

Todos los formatos de esta familia se caracterizan por tener un encabezado en donde se describen los atributos de la imagen (tipo, tamaño, valor máximo de pixel), y luego siguen los datos, que son la secuencia de valores de pixel de la imagen en cuestión. A continuación se transcribe en español la descripción oficial de los formatos que usaremos, tomadas de

<http://netpbm.sourceforge.net/doc/ppm.html>.

### A.1. PPM

Son archivos que almacenan imágenes en color. El contenido de un archivo *PPM* es el siguiente:

1. Un *número mágico*<sup>5</sup> para identificar el tipo de archivo. El número mágico de una imagen *PPM* es el par de caracteres ASCII<sup>6</sup> **P6**.
2. Uno o más espacios en blanco (espacios, tabuladores, carácter de nueva linea).
3. El ancho (cantidad de columnas) *n* de la imagen, descrito como una cadena de caracteres ASCII, por ejemplo 123.
4. Uno o más espacios en blanco.
5. El alto (cantidad de filas) *m* de la imagen, de nuevo en ASCII.
6. Uno o más espacios en blanco.
7. El valor máximo de canal (*M*), de nuevo en decimal ASCII. Nosotros asumiremos que las imágenes a procesar tendrán como valor máximo a 255.
8. Un **único** carácter de espacio en blanco (por lo general una nueva linea).
9. Una secuencia de *m* filas de la imagen. Cada fila consta de una secuencia de *n* valores de pixel. Cada pixel, a su vez consta de una secuencia de 3 bytes en el rango [0, 255]: uno para rojo, otro para verde, y otro para azul, en ese orden.

En el formato descrito arriba, los primeros 8 items describen el encabezado, y el último describe la trama de datos. Notar que el formato anteriormente descrito, la trama de datos es **binaria**, ya que los bytes que describen a los pixeles pueden tener cualquier valor entre 0 y 255, por lo que si intentan abrir este archivo en un editor de texto usual, el encabezado de la imagen sera legible mientras que la trama de datos no.

<sup>5</sup>[https://en.wikipedia.org/wiki/List\\_of\\_file\\_signatures](https://en.wikipedia.org/wiki/List_of_file_signatures)

<sup>6</sup><https://www.asciitable.com/>

## A.2. PPM plano

El formato *PPM* que presentamos anteriormente es binario, es decir, los canales de los píxeles se representan como bytes. En el formato *PPM plano* los valores de pixel se dan todos como cadenas de texto ASCII, al igual que los datos del encabezado, y se separan por uno o más caracteres de espaciado (espacios, tabs, nueva línea).

1. Un *número mágico* para identificar el tipo de archivo. El número mágico de una imagen *PPM plano* es el par de caracteres ASCII **P3**.
2. Uno o más espacios en blanco (espacios, tabuladores, carácter de nueva linea).
3. El ancho (cantidad de columnas)  $n$  de la imagen, descrito como una cadena de caracteres ASCII, por ejemplo 123
4. Uno o más espacios en blanco.
5. El alto (cantidad de filas)  $m$  de la imagen, de nuevo en ASCII.
6. Uno o más espacios en blanco.
7. El valor máximo de canal ( $M$ ), de nuevo en decimal ASCII. Nosotros asumiremos que las imágenes a procesar tendrán como valor máximo a 255.
8. Uno o más espacios en blanco (a diferencia del PPM no plano, donde necesariamente debe ser uno solo).
9. Una secuencia de  $m$  filas de la imagen. Cada fila consta de una secuencia de  $n$  valores de pixel. Cada pixel, a su vez consta de una secuencia de 3 bytes en el rango [0, 255]: uno para rojo, otro para verde, y otro para azul, en ese orden. El valor de cada byte de cada pixel es representado como una cadena de texto ASCII, por ejemplo 200, seguida de uno o más caracteres de espaciado.

Notar que el encabezado es prácticamente idéntico (**P3** en lugar de **P6**), a menos del número mágico. La trama de datos es, por otro lado, muy distinta. Notar que el *PPM plano* es de hecho un **archivo de texto ASCII** de punta a punta, por lo que el archivo es legible en su completitud al abrirlo con un editor de texto usual.

## B. Algoritmo de encriptación a utilizar, basado en Vigènere binario

### B.1. Introducción

Cómo se esbozó en el obligatorio 1, la mayoría de algoritmos de encriptación constan de una clave que permite pasar de mensaje a criptograma, y otra clave que permite pasar del criptograma al mensaje, como se ilustra en la Figura 6. En el caso de que ambas claves coincidan, el algoritmo se conoce como algoritmo de clave simétrica o de clave única.

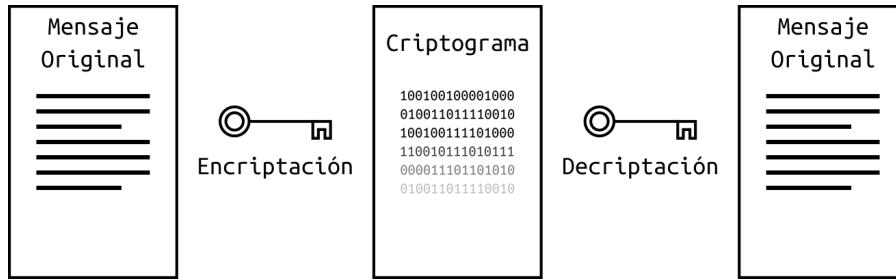


Figura 6: Esquema base de un algoritmo de encriptación. En caso de que la clave de encriptación y de des-encriptación coincidan, el algoritmo es de *clave simétrica*.

Deseamos desarrollar un algoritmo de encriptación sencillo basado en la encriptación de Vigènere descripta en el Obligatorio 1. Para que nuestro algoritmo verifique ser un algoritmo de encriptación debe cumplir:

1. Que la imagen encriptada sea irreconocible a simple vista.
2. Que solo sea posible recuperar el contenido original conociendo la clave (y el algoritmo) de encriptación.

Algunos de los algoritmos más sencillos que se vienen a la cabeza tienen riesgo de no cumplir el primer punto. En particular, si encriptáramos cada pixel con una misma clave de Vigènere utilizando el algoritmo del Obligatorio 1, regiones en la imagen de entrada de un mismo color quedarían representadas con un único color (diferente al original) en la imagen encriptada, como se ve en la Figura 8.



Figura 7: Imagen a encriptar



Figura 8: Intento de encriptación utilizando la misma clave de Vigènere en cada pixel.

Por esto optaremos por tener una clave de Vigènere diferente para cada pixel. Una forma de implementar lo anterior es que la clave sea una **cripto-imagen**, es decir, que en cada pixel se tenga codificada una clave como las del obligatorio 1. Por lo tanto, una cripto-imagen de tamaño  $m \times n$  contiene  $m \cdot n$  claves como las del obligatorio 1. La manera de convertir cada pixel de la cripto-imagen en una clave es explicada en el Anexo B.2.

## B.2. Algoritmo a utilizar

En el algoritmo de encriptación que utilizaremos, nuestra clave simétrica será una cripto-imagen de tamaño a priori distinto de la imagen mensaje. Cada pixel de la cripto-imagen (es decir, cada palabra de 24 bits) representará una clave de Vigenère mediante el siguiente protocolo: en sus primeros 5 bits contendrá el largo de clave de Vigenère a utilizar, y en sus restantes 19 bits el valor de la clave a utilizar. Un esquema del contenido de cada pixel se observa en la Figura 9.

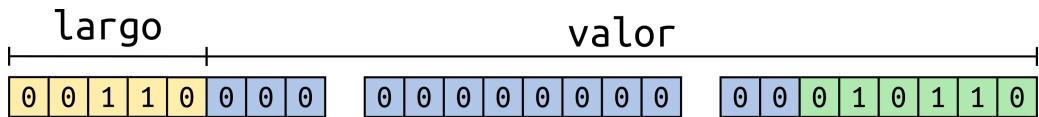


Figura 9: Cada pixel de la cripto-imagen tendrá una clave de Vigenère de la cual los primeros 5 bits serán el largo a utilizar, y los restantes 19 el valor de la clave.

A partir de la estructura anterior diremos que una **cripto-imagen es válida** si todos los píxeles de la imagen verifican que  $1 \leq \text{largo} \leq 19$ . En la Figura 10 se ve un ejemplo de cripto-imagen válida (generada de forma aleatoria con la función `generar_cripto_imagen`), notar la aleatoriedad de la misma y la relativa carencia del color rojo (¿a qué se debe?).

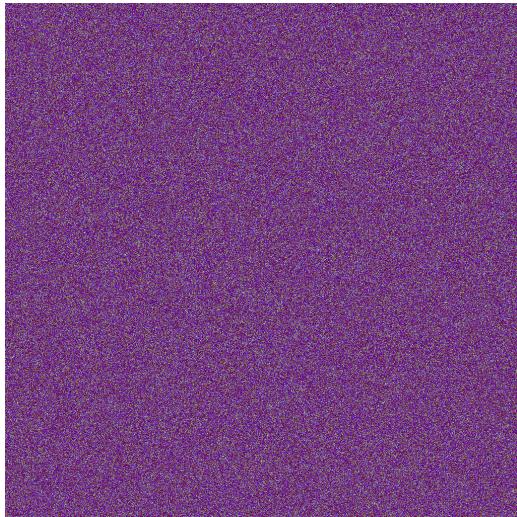


Figura 10: Cripto-imagen generada según el protocolo explicado en esta sección: cada uno de los píxeles sigue la estructura de la Figura 9.

Una vez tienen la cripto-imagen y la imagen mensaje, deben generar la imagen encriptada, cuyos píxeles surgen de aplicar la función `encriptar` del obligatorio 1, con el pixel de la imagen mensaje y la clave formada a partir del pixel que corresponda de la cripto-imagen.

### ¿Cómo encriptar una imagen de tamaño distinto a la cripto-imagen?

La forma general de tratar con imágenes de tamaño distinto es alinear la imagen y cripto-imagen en el vértice superior izquierdo y repetir la cripto-imagen tantas veces hacia la derecha y abajo como sea necesario hasta cubrir por completo toda la imagen a encriptar, la Figura 11 ilustra lo anterior. Notar que si la imagen es más pequeña que la cripto-imagen no se necesita repetir la imagen.

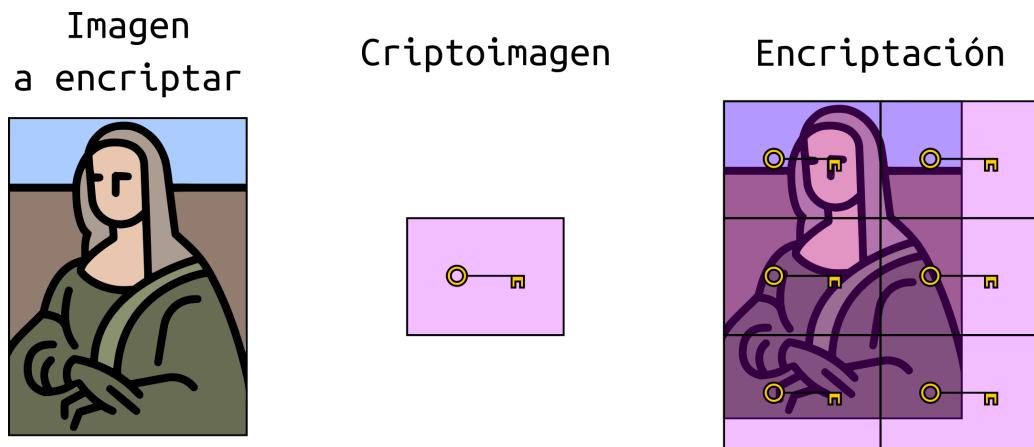


Figura 11: Debe repetirse la cripto-imagen hasta cubrir completamente la imagen a encriptar. El resultado de la encriptación debe tener el mismo tamaño que la imagen a encriptar.

Para implementar lo anterior en código **NO** es necesario crear una nueva imagen de tipo `Imagen_t` que sea la repetición de la cripto-imagen, alcanza con encriptar el pixel  $(i, j)$  de la imagen mensaje por el pixel  $(i \% m, j \% n)$  de la cripto-imagen, con  $m$  y  $n$  la cantidad de filas y columnas de la cripto-imagen.