Filtrado de imágenes - Parte 2

25 de mayo de 2017

1 Generalidades

La aprobación de este curso se consigue mediante la correcta implementación de dos pequeños proyectos de programación. estos son propuestos aproximadamente un mes antes de cada parcial, y entregados a través de una página web habilitada para tales fines, con fecha límite de entrega fijada poco antes de cada período de parcial. Cada entrega es complementada con una pequeña prueba escrita cuyo objetivo es evaluar aspectos más teóricos relacionados con el propio obligatorio.

Es importante recalcar que **tanto la prueba escrita como el proyecto entregado 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.

En casos de ser posible, el sistema intentará además compilar y ejecutar la entrega de cada estudiante, de modo de dar un mínimo de información al respecto de qué tan bien funciona la entrega. Dependiendo del proyecto, esta evaluación preliminar estará o no disponible.

En todo caso, la evaluación preliminar mencionada anteriormente **no** determina la nota obtenida en la prueba, siendo esta definida por una evaluación manual por parte de los docentes.

1.1 Formato del archivo a entregar

El archivo entregado debe ser un archivo comprimido en formato **zip** (NO se pueden subir rar), de nombre **1234567.zip** donde **1234567** es su número de cédula, sin digito de verificacion, sin espacios ni puntos, ni guión. El contenido del archivo debe incluir los siguientes elementos bajo el subdirectorio **1234567** (no pueden haber subdirectorios):

- Todos los archivos fuente creados por el estudiante (**.h** y **.c**)
- La biblioteca precompilada entregada por los docentes, si es que existe.
- Un archivo **Makefile** para compilar el o los programas requeridos en el trabajo.

Por ejemplo, supongamos el obligatorio consiste en la generación de un ejecutable de nombre `oblig`, su cédula es 1234567-8, y usted implementó dicho ejecutable en tres módulos **a.c** (y su correspondiente encabezado **a.h**), **b.c** (encabezado **b.h**) y **main.c**. Además, los docentes le entregaron una biblioteca auxiliar con un encabezado **aux.h** y código objeto **aux.o**. Entonces debe subir un archivo de nombre `1234567.zip` con el siguiente contenido:

```
1234567/
  a.c
  a.h
  b.c
  b.h
  aux.h
  aux.o
```

```
  main.c
  Makefile
```

El **Makefile** podría ser así:

```
oblig: main.o a.o b.o aux.o
  cc -o oblig main.o a.o b.o aux.o -lm

main.o: main.c
  cc -c main.c

a.o: a.c
  cc -c a.c

b.o: b.c
  cc -c b.c
```

Nota: Pueden crear un zip desde la máquina virtual con el comando `zip`; la sintaxis es

```
$zip -r nombre_archivo.zip carpeta_a_comprimir
```

en el ejemplo anterior, sería `zip -r 1234567.zip 1234567`.

1.2 Metodología de trabajo

Algunas recomendaciones generales sobre cómo trabajar con proyectos como los que se proponen aquí:

- Simplicidad (KISS - Keep It Simple, Stupid). No complicar el código más allá de lo requerido.
- Prolijidad. No importa cuánto aburra, documentar bien lo que se hace es fundamental; es muy fácil olvidarse lo que uno mismo hizo.
- Incrementalidad. Implementar y probar de a pequeños pasos. "No construir un castillo de entrada". Es muy difícil encontrar las causas de un problema si se prueba todo simultáneamente.

2 Introducción al problema

En esta parte del obligatorio agregaremos fundamentalmente cuatro puntos:

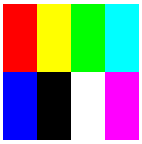
1. Manejo de imágenes a color
2. Trabajar con imágenes de tamaño variable.
3. Implementación de una biblioteca de lectura y escritura de archivos de imagen PGM (grises) y PPM (color)
4. Implementar algunos filtros para imágenes de color.

2.1 Representación de imágenes a color

Existen muchas formas de representar una imagen a color bidimensional. En esta segunda parte del proyecto trabajaremos con una extensión sencilla de lo visto en la primera parte, donde el valor binario de cada píxel, en lugar de representar una intensidad de gris, representa un color. En general, esto se logra representando a cada píxel como un vector de *canales* que definen al color. Ejemplos de representación de color son el llamado RGB (Red, Green, Blue) y el conocido como HSV (Hue,

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

Figura 1: Representación de un píxel RGB en un entero



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 2: Imagen RGB de tamaño 4×4. Izq.: imagen, Der.: su representación numérica.

Saturation, Value). La primera representación es más sencilla, mientras que la segunda está más alineada con la forma en que el ojo humano percibe el color.

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; esta es la representación de colores utilizada en todos los monitores y televisores. En esta representación, cada píxel es representado por una terna de valores enteros (r, g, b), cada uno en el rango $[0, M]$ 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 (M, M, M) con el blanco. En nuestro caso, trabajaremos con 8 bits por canal, por lo que $M = 255$. De esta manera, un `int` de 32 bits como los que utilizamos en la primera parte es capaz de acomodar 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 1 muestra esto gráficamente. La figura 2 muestra un ejemplo muy sencillo de imagen RGB.

NOTA: *Se recomienda fuertemente comenzar esta tarea por escribir (y probar) dos funciones (o macros) para extraer la terna de valores r, g, b de un entero y viceversa, es decir, dada una terna de valores (r, g, b) (por ejemplo, de tipo `unsigned char`), generar el `int` correspondiente que los almacene como se muestra en la figura 1.*

Así podemos representar una imagen con el mismo tipo de datos que en la parte anterior del obligatorio, es decir como un arreglo de enteros de tamaño $m \times n$ donde m es el ancho y n el alto de la imagen. Para obtener los distintos canales por separado podemos usar operaciones de bits.

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

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

Asimismo, defimos los canales de una imagen color RGB como las tres imágenes de escalas de grises $\mathbf{I}_R$, $\mathbf{I}_G$ e $\mathbf{I}_B$ cuyos valores son las intensidades de cada uno de los canales por separado, es decir

$$\mathbf{I}_R(i, j) = r_{ij}, \quad \mathbf{I}_G(i, j) = g_{ij}, \quad \mathbf{I}_B(i, j) = b_{ij}, \quad 0 \leq i < m, 0 \leq j < n$$

Esto se muestra en la figura 3.

2.2 De color a blanco y negro

En ocasiones (en particular, para aplicar el filtro de detección bordes, ver figura 4), es necesario obtener una versión en grises de una imagen a color. Idealmente, la versión en blanco y negro de una imagen $\mathbf{I}$, a la que denominaremos $\mathbf{I}_M$, corresponde a la intensidad de luz percibida por el ojo humano para cada uno de sus píxels. Existen varias fórmulas para estimar esta intensidad a partir de los colores de un píxel. Nosotros utilizaremos la siguiente. Si (r, g, b) son los tres canales de un píxel color, su versión blanco y negro de acuerdo a esta fórmula será $w = 0.2 \times r + 0.7 \times g + 0.1 \times b$.

Figura 3: Imagen a color $\mathbf{I}$ y sus tres componentes $\mathbf{I}_R, \mathbf{I}_G$ y $\mathbf{I}_B$. 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.

3 Descripción de la tarea

La implementación de esta tarea consiste a grandes razgos en dos partes. La primera es una biblioteca de lectura/escritura de imágenes, tal como la suministrada en la primera parte del obligatorio. La segunda consiste nuevamente en un ejecutable que permita efectuar un número de filtros a una imagen de entrada, sea color o blanco y negro.

3.1 Biblioteca de lectura/escritura de imágenes

La biblioteca deberá ser capaz de leer y escribir imágenes de tamaño *arbitrario*, tanto a color como blanco y negro. Consistirá en dos archivos: un encabezado `imagen.h` y una implementación `imagen.o`, los cuales serán utilizados dentro del ejecutable a entregar. En `imagen.h` se deberán declarar una serie de constantes, tipos de datos, y funciones, de acuerdo a requerimientos a ser especificados más adelante en la sección 3 de este documento, y luego implementar las funciones en `imagen.o`.

Los formatos de imagen soportados por la biblioteca serán el "PGM plano" utilizado en la primera parte del entregable para imágenes blanco y negro, y el "PPM rawbits", que se utiliza para representar imágenes color. Ambos formatos serán detallados en un apéndice al final de este documento.

La biblioteca será utilizada como parte de la implementación del ejecutable que aplica los filtros, nuevamente llamado `obligatorio`, pero también deberá poder ser invocada por un ejecutable no especificado, que será utilizado para probar la biblioteca en la prueba final de la entrega.

3.2 Filtrado de imágenes

Interfaz de linea de comandos La sintaxis de esta parte se mantendrá intacta, es decir, el programa principal se llamará `obligatorio`, y se invocará de la misma manera:

```
./obligatorio filtro parametro entrada salida
```

En este caso, sin embargo, tanto la entrada como la salida pueden ser a colores. En particular, el programa debe actuar de manera acorde según si la entrada es a colores o blanco y negro. Un ejemplo, como se mencionó anteriormente, es el filtro de bordes, el cual deberá aplicarse a una versión blanco y negro de la imagen de entrada, y no directamente a la imagen a colores.

Mantendremos para esta tarea los filtros `copia`, `reflejo`, `negativo` y `borde`. Agregaremos luego un filtro nuevo, relativamente complejo, que describiremos en breve.

Filtros viejos Debido a que utilizaremos el mismo tipo de datos (un `int`) para almacenar tanto los píxels en grises como los píxels a color, y tanto `copia` como `reflejo` solamente copian píxels de una posición a otra, estos dos filtros funcionarán idénticamente sin necesidad de saber qué tipo de imagen están procesando.

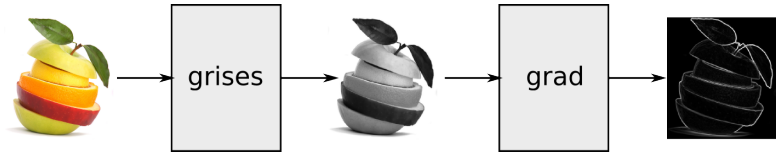


Figura 4: Detección de bordes de imagen a color, tal cual lo definimos en esta tarea. Primero se convierte a grises, y luego se aplica el detector de bordes de la primera parte (con parámetro 2) a la imagen de grises.

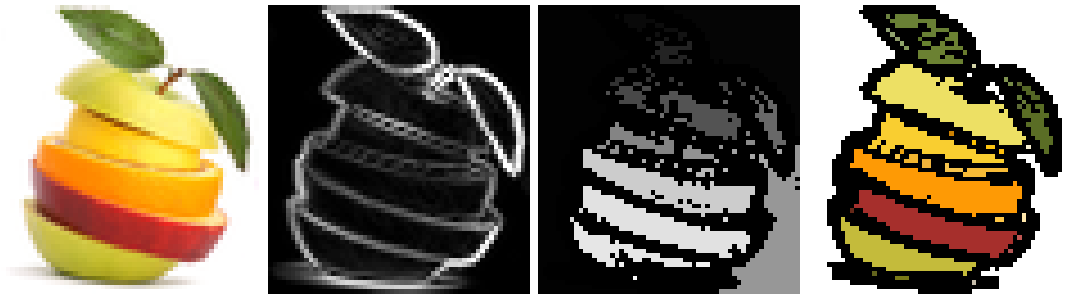


Figura 5: Proceso de caricatura. De izquierda a deracha: imagen original, detección de bordes, etiquetado de regiones, y promediado de color de regiones.

En el caso del filtro `negativo`, sin embargo, será necesario negar cada canal por separado, es decir. Si (r, g, b) es un píxel, su valor en negativo será $(M - r, M - g, M - b)$. Una posiblidad, para reutilizar el código ya disponible, es descomponer la imagen color $\mathbf{I}$ en sus tres canales $\mathbf{I}_R$, $\mathbf{I}_G$ e $\mathbf{I}_B$, y aplicar el filtro negativo a cada canal, para luego recomponer la imagen en negativo a partir de los negativos de los tres canales.

PROBE NEGADO pero no bordes, conversion a grises, deberia poder sacar los tres canales y recombinar incluyendo las opciones en el llamado

El filtro `borde` se aplicará siempre a imágenes en blanco y negro, por lo que nuevamente puede reutilizarse el código ya implementado en la primera parte. En este caso, en caso de recibirse una imagen a color, primero deberá obtenerse una versión en blanco y negro de ella, tal como se describiera arriba. Esto se muestra en la figura 4.

Caricatura Finalmente, el nuevo filtro llamado `caricatura`, toma una imagen (color o blanco y negro) y un umbral escalar $u > 0$. Primero detecta los bordes de la imagen, luego utiliza esos bordes para dividir la imagen en *regiones* conexas, y finalmente pinta cada una de las regiones con su color *promedio*. Más precisamente, si $\mathcal{R} \subseteq \mathbb{Z}^2$ es el subconjunto de índices de una imagen asociado a una región, definimos el color promedio $(\bar{r}, \bar{g}, \bar{b})$ de la región comprendida por $\mathcal{R}$ como

$$\bar{r} = \frac{\sum_{(i,j)\in\mathcal{R}} r_{ij}}{|\mathcal{R}|}, \quad \hat{g} = \frac{\sum_{(i,j)\in\mathcal{R}} g_{ij}}{|\mathcal{R}|}, \quad \hat{b} = \frac{\sum_{(i,j)\in\mathcal{R}} b_{ij}}{|\mathcal{R}|},$$

donde $|\mathcal{R}|$ indica la cantidad de píxeles en la región (el tamaño del conjunto).

Para identificar cada región en la imagen utilizaremos una pseudo-imagen de escala de grises denominada *etiquetado*, a la que denotaremos como $\mathbf{E}$, donde el valor de $\mathbf{E}(i, j)$ indica a cuál región pertenece el píxel (i, j) de la imagen $\mathbf{I}$. Así, si en la imagen hay L regiones distintas, el valor máximo de píxel de $\mathbf{E}$ será L. El valor 0 lo utilizaremos para marcar bordes, los cuales no asociaremos a ninguna región. El proceso anterior, y su resultado, puede verse en la figura 5.

El valor de umbral u es utilizado para determinar si un píxel en la imagen de bordes es efectivamente borde o no. Concretamente, si $\mathbf{G}$ es la imagen de bordes de $\mathbf{I}$, diremos que el píxel (i, j) es borde si $\mathbf{G}_{ij} > u$. En resumen, el proceso de caricatura implica las siguientes etapas:

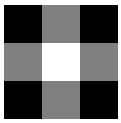


Figura 6: Adyacencia de puntos. El píxel blanco es adyacente a los píxeles grises, pero no a los negros. Los píxeles grises no son adyacentes entre sí (naturalmente, los negros tampoco).

1. Obtener los bordes de la imagen
2. Determinar las regiones definidas por esos bordes; a esto lo denominamos *etiquetado*
3. Sustituir el color de los píxels de cada región por su promedio. Los bordes se pintan de negro.

Etiquetado De estas tres etapas, la más difícil (y que no utiliza nada implementado anteriormente) es la del etiquetado de regiones. A continuación explicamos en detalle una forma de realizar este etiquetado.

Dos puntos de la imagen están en la misma región si se puede llegar de uno al otro siguiendo un camino de píxeles adyacentes que no corte ningún borde. Dos puntos se consideran adyacentes sólo si están o bien uno encima del otro, o bien uno al lado del otro. No se consideran adyacentes por la diagonal. Por ejemplo en la figura 6 el píxel en blanco es adyacente a los píxeles grises pero no a los negros, y los grises no son adyacentes entre si.

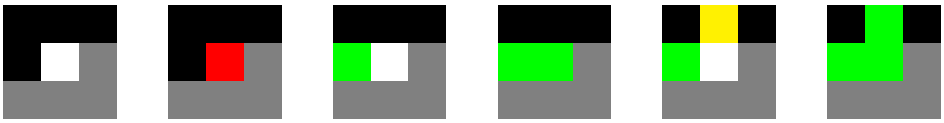
Consideraremos como borde todos aquellos píxeles (i, j) para los cuales $\mathbf{G}(i, j) > u$. También consideraremos como borde a todo píxel fuera del rango de la imagen $[0, m] \times [0, n]$.

Una forma sencilla de implementar el etiquetado es recorrer la imagen secuencialmente, de arriba a abajo y de izquierda a derecha, y a cada nuevo píxel asignarle una etiqueta en base a los píxeles etiquetados previamente. Llamemos $n = \mathbf{E}(i-1, j)$ y $w = \mathbf{E}(i, j-1)$ (recordar que los píxels fuera de rango son considerados borde, por lo que n y w están bien definidos para todo (i, j)). Supongamos además que se han marcado hasta el momento K regiones distintas. Al asignar la etiqueta del píxel (i, j), se consideran las siguientes posiblidades:

- Si w y n son ambos borde, marcamos la posición (i, j) con una nueva etiqueta de valor $K + 1$, $\mathbf{E}(i, j) = K + 1$ (luego, de manera acorde, debe incrementarse K en uno).
- Si w es borde y n está etiquetado, $\mathbf{E}(i, j) = n$.
- Si n es borde y w está etiquetado, $\mathbf{E}(i, j) = w$.
- Si ni n ni w son borde, y sus etiquetas coinciden, $\mathbf{E} = (i, j) = w = n$.
- Si ni n ni w son borde, pero sus etiquetas son diferentes, entonces debemos "pegar" ambas regiones. Para esto, elegimos una de las dos (no importa cuál), y reemplazamos todos los píxeles de $\mathbf{E}$ con el valor de una, por el valor de la otra.

El procedimento anterior se puede ver gráficamente en la figura 7. El resultado de este filtro puede verse en la penúltima imagen de la figura 5.

NOTA: *Debido a la relativa complejidad del filtro de* `caricatura`*, es muy recomendable partir su implementación en etapas: bordes (ya implementado anteriormente), etiquetado y luego "rellenado" con el color promedio. De esta manera, eventualmente, se pueden generar y almacenar imágenes intermedias (como por ejemplo la imagen de etiquetado) para facilitar la depuración.*



(a) (b) (b)

Figura 7: Proceso de etiquetado. píxels marcados con negro son borde, los píxels marcados con gris no han sido visitados aún, y el píxel blanco es el que se quiere marcar. píxels a color se corresponden a distintas etiquetas (ojo! no es que la imagen de etiquetas sea a color; los colores se muestran para marcar distintas cosas). De izquierda a derecha: (a) no hay etiquetas adyacentes, sólo bordes, por lo que se pinta con nueva etiqueta "rojo"; (b) hay un píxel adyacente marcado con etiqueta "verde", por lo que se copia; (c) hay dos píxels vecinos, pero con distintas etiquetas, por lo que se elije una y se pinta todo con esa.

4 Especificación de requerimientos

4.1 Biblioteca de lectura/escritura de imágenes

La biblioteca debe declarar los siguientes tipos en `imagen.h`:

- Tipo `Pixel` de tipo entero.
- Tipo `CodigoError`, basado en una enumeración `codigo_error` con los siguientes valores posibles:
  - `PNM_OK=0`: se devuelve si todo salió bien
  - `PNM_ARCHIVO_INEXISTENTE=1`: si el archivo a abrir no se encuentra (`fopen` devuelve `NULL`)
  - `PNM_ERROR_LECTURA=2`: si ocurre un error al leer datos del archivo (por ejemplo con `fread`, o `fscanf`)
  - `PNM_ENCABEZADO_INVALIDO=3`: se devuelve al leer el encabezado; indica que el formato del encabezado no es correcto
  - `PNM_DATOS_INVALIDOS=4`: se devuelve al leer los datos si, por ejemplo, se termina el archivo antes de leer todos los píxels que se esperaba leer
  - `PNM_ERROR_ESCRITURA=5`: se devuelve ante cualquier error que ocurra durante la función `escribir_imagen`
- Tipo `TipoImagen`, en base a una enumeración de nombre `tipo_imagen` con los valores posibles `GRISES=0` y `COLOR=1`
- Tipo `Canal`, en base a una enumeración de nombre `canal` con los valores posibles `ROJO=0`, `VERDE=1` y `AZUL=2`
- Tipo `Imagen`, en base a una estructura de nombre `imagen` con los siguientes campos:
  - `tipo`, de tipo `TipoImagen`
  - `ancho`, de tipo entero
  - `alto`, de tipo entero
  - `valor_maximo`, entero
  - `pixels`, puntero a `Pixel`

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

- `int inicializar_imagen(int ancho, int alto, TipoImagen tipo, Imagen* pnew)`: inicializa los datos de la imagen apuntada por `pnew` con los parámetros especificados, y reserva memoria para sus nuevos píxeles. El valor máximo se inicializa por defecto en `255`. Devuelve un entero de valor `1` si hubo algún problema al crear la imagen, o `0` en otro caso.
- `void destruir_imagen(Imagen* pimg)` libera la memoria asociada a los píxeles de la imagen apuntada por `pimg` y pone todos sus atributos en `0`. No tiene valor de retorno.
- `void duplicar_imagen(const Imagen* pin, Imagen* pout` copia los atributos de una imagen de entrada apuntada por `pin` (inicializada previamente) en la imagen apuntada por `pout` (no inicializada aún), reservando espacio para los nuevos píxels. Notar que, a pesar del nombre, los valores de píxels de la nueva imagen *no* serán copiados de la de entrada; se dejarán sin inicializar.
- `CodigoError leer_imagen(const char* ruta, Imagen* pimg)` lee el contenido del archivo ubicado en `ruta` en la imagen apuntadad por `pimg`. Devuelve un valor de tipo `CodigoError` según el caso. (ver `CodigoError`).
- `CodigoError escribir_imagen(const Imagen* pimg, const char* ruta)` : guarda el contenido de la imagen `pimg` en el archivo especificado por `ruta`. Al igual que en el caso anterior, debe devolver un valor de tipo `CodigoError` según el caso.

4.2 Programa de filtrado

La interfaz para los filtros `copia`, `negativo`, `reflejo` y `borde` se mantiene tal cual la primera parte. No se implementarán los filtros `contraste` ni `promedio`. El nuevo filtro, `caricatura`, se invoca como:

```
./obligatorio caricatura umbral entrada salida
```

Consideraciones y sugerencias

- Se sugiere utilizar la función `fscanf` para implementar la lectura de números representados en ASCII en los archivos PNM.
- **Tener muy 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 que 1), pero entre el fin del encabezado y los datos debe haber **uno y solo un** caracter de espaciado (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, sobre todo los filtros complejos.
- En caso de bugs, el GDB es su mejor amigo.

A Formatos de archivos de imagen

En la primera parte del obligatorio trabajamos con imágenes de tipo "PGM". Estas a su vez 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 (algunos de) estos formatos.

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 píxel), y luego siguen los datos, que son la secuencia de valores de píxel de la imagen en cuestión. A continuación se transcribe en español la descripción oficial de los formatos que usaremos, tomadas respectivamente de `http://netpbm.sourceforge.net/doc/ppm.html` y `http://netpbm.sourceforge.net/doc/ppm.html`.

A.1 PPM

El contenido de un archivo PPM es el siguiente:

1. Un "número mágico" para identificar el tipo de archivo. El número mágico de una imagen ppm es el par de caracteres ascii `''P6''`.
2. Uno o más espacios en blanco (espacios , tabuladores, caracter de nueva linea).
3. El ancho n de la imagen, descrito como una cadena de caracteres ASCII, por ejemplo "123"
4. Uno o más espacios en blanco.
5. El alto 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. Debe ser mayor que 0 y menor que 256 (el estandar PPM define un máximo de 65536, pero nosotros lo acotaremos a 8 bits para este caso)
8. *Un único* carácter de espacio en blanco (por lo general una nueva línea) .
9. Una secuencia de m filas de la imagen. Cada fila consta de una secuencia de n valores de píxel. Cada píxel, a su vez consta de una secuencia de 3 bytes en el rango $[0, M]$: 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 es **binario**, ya que los bytes que describen a los píxeles pueden tener cualquier valor entre 0 y 255.

A.2 PGM "plano"

El formato PPM que presentamos anteriormente es binario, es decir, los canales de los píxels se representan como bytes. El formato PGM "común" es también binario. Nosotros, sin embargo, trabajaremos con una variante llamada "PGM plano" (que es de hecho la que utilizamos en la primera parte). En este caso, los valores de píxel 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 PGM "plana" es el par de caracteres ascii `''P2''`.

2. Uno o más espacios en blanco (espacios , tabuladores, caracter de nueva linea).
3. El ancho n de la imagen, descrito como una cadena de caracteres ASCII, por ejemplo “123”
4. Uno o más espacios en blanco.
5. El alto 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. Debe ser mayor que 0 y menor que **65536**
8. *Un único* carácter de espacio en blanco (por lo general una nueva línea) .
9. Una secuencia de m filas de la imagen. Cada fila consta de una secuencia de n valores de píxel. La intensidad de cada píxel es a su vez representada como una cadena de texto ASCII, por ejemplo “200”, seguida de uno o más caracteres de espaciado.

Notar que el encabezado es prácticamente idéntico, a menos del número mágico. La trama de datos es, por otro lado, muy distinta. Notar que el PGM plano es de hecho un **archivo de texto ASCII** de punta a punta.