UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Práctica 2: Abstracción. TDA Imagen

(Práctica puntuable)

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Estructuras de Datos

Grado en Ingeniería Informática
Doble Grado en Ingeniería Informática y Matemáticas
Doble Grado en Ingeniería Informática y ADE

1.- Introducción

1.1 Imágenes de niveles de gris

Una **imagen digital** de niveles de gris puede verse como una matriz bidimensional de puntos (píxeles, en este contexto) en la que cada uno tiene asociado un nivel de luminosidad cuyos valores están en el conjunto {0,1, ..., 255} de forma que el 0 indica mínima luminosidad (negro) y el 255 la máxima luminosidad (blanco). Los restantes valores indican niveles intermedios de luminosidad (grises), siendo más oscuros cuanto menor sea su valor. Con esta representación, cada píxel requiere únicamente un byte (**unsigned char**). En la figura siguiente mostramos un esquema que ilustra el concepto de una imagen con **alto** filas y **ancho** columnas. Cada una de las alto x ancho celdas representa un píxel o punto de la imagen y puede guardar un valor del conjunto {0,1, ..., 255}.

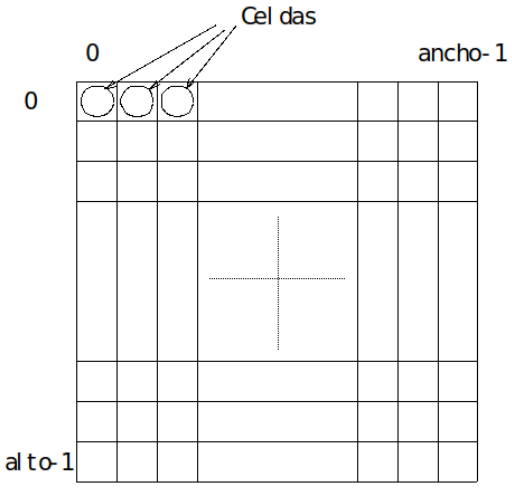


Figura 1.- Esquema de una imagen

Cada casilla de esta matriz representa un punto de la imagen y el valor guardado en ésta indica:
- En imágenes de niveles de gris: su nivel de luminosidad.
- En imágenes en color: su código de color (representación por tabla de color) o la representación del color en el esquema usado (RGB, IHV, etc.).

En esta práctica se trabajará con imágenes de niveles de gris, por lo que cada casilla contiene niveles de luminosidad que se representan con valores del conjunto {0,1, ..., 255} con la convención explicada anteriormente.

A modo de ejemplo, en la figura 2.A mostramos una imagen digital de niveles de gris que tiene 256 filas y 256 columnas. Cada uno de los 256 x 256 = 65.536 puntos contiene un valor entre 0 (visualizado en negro) y 255 (visualizado en blanco). En la figura 2.B mostramos una parte de la imagen anterior (10 filas y 10 columnas) en la que se pueden apreciar, con más detalle, los valores de luminosidad en esa subimagen.

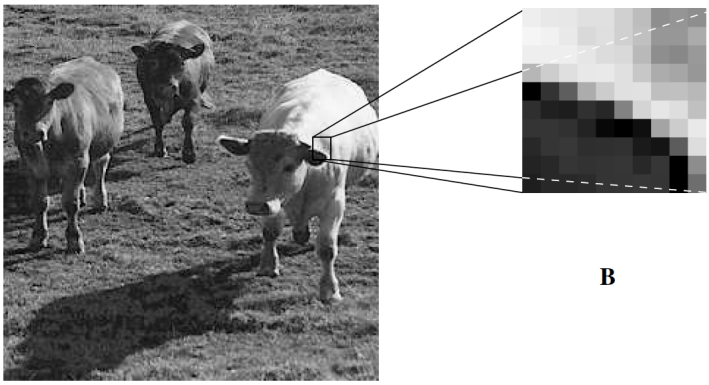


Figura 2.- A. Imagen real de 256 filas y 256 columnas. B) Una parte de esa figura, con 10 filas y 10 columnas

1.2 Procesamiento de imágenes digitales

El término **Procesamiento de Imágenes Digitales** incluye un vasto conjunto de técnicas cuyo objetivo final es la extracción y representación de información a partir de imágenes digitales. Constituye uno de los campos más atractivos de la Inteligencia Artificial, y en el ámbito científico se conoce con diferentes nombres: Computer Vision, Image Processing, etc., dependiendo del campo y ámbito de aplicación. Una definición informal, y bastante restringida, sería la siguiente: mediante *Procesamiento de Imágenes Digitales* nos referimos a un conjunto de operaciones que modifican la imagen original, dando como resultado una nueva imagen en la que se resaltan ciertos aspectos de la imagen original.

2. Formatos de imágenes

Las imágenes se almacenan en ficheros con un determinado formato. Existen numerosos formatos de imágenes y su descripción detallada puede encontrarse en bibliografía específica de Procesamiento de Imágenes o en Internet. Así, para dar versatilidad a nuestra aplicación se requiere un conjunto adicional de funciones para que ``rellenen'' una imagen tomando la información de un fichero o para que guarden una imagen en un fichero. Entre los formatos de imágenes, vamos a trabajar con el formato PGM. Las funciones de lectura y escritura de imágenes van a constituir una biblioteca adicional, que se describe en esta sección.

El formato PGM constituye un ejemplo de los llamados formatos con cabecera. Estos formatos incorporan una cabecera en la que se especifican diversos parámetros acerca de la imagen, como el número de filas y columnas, número de niveles de gris, comentarios, formato de color, parámetros de compresión, etc.

2.1 Descripción del formato PGM

PGM es el acrónimo de **P**ortable **G**ray **M**ap File Format. Como hemos indicado anteriormente, el formato PGM es uno de los formatos que incorporan una cabecera. Un fichero PGM tiene, desde el punto de vista del programador, un formato mixto texto-binario: la cabecera se almacena en formato texto y la imagen en sí en formato binario. Con más detalle, la descripción del formato PGM es la siguiente:

1. Cabecera.

La cabecera está en formato texto y consta de:

- **Un “número mágico” para identificar el tipo de fichero**. Un fichero PGM que contiene una imagen digital de niveles de gris tiene asignado como identificador los dos caracteres P5.
- **Un número indeterminado de comentarios**.
- **Número de columnas (c)**.
- **Número de filas (f)**.
- **Valor del mayor nivel de gris que puede tener la imagen (m)**.

Cada uno de estos datos está terminado por un carácter separador (normalmente un salto de línea).

2. Contenido de la imagen.

Una secuencia binaria de f x c bytes, con valores entre 0 y m. Cada uno de estos valores representa el nivel de gris de un píxel. El primero hace referencia al píxel de la esquina superior izquierda, el segundo al que está a su derecha, etc.

Algunas aclaraciones respecto a este formato:

- El número de filas, columnas y mayor nivel de gris se especifican en modo texto, esto es, cada dígito viene en forma de carácter.

- Los comentarios son de línea completa y están precedidos por el carácter #+. La longitud máxima es de 70 caracteres. Los programas que manipulan imágenes PGM ignoran estas líneas.

- Aunque el mayor nivel de gris sea m, no tiene porqué haber ningún pixel con este valor. Este es un valor que usan los programas de visualización de imágenes PGM.

En la Figura 4 se muestra un ejemplo de cómo se almacena una imagen de 100 filas y 50 columnas en el formato PGM.

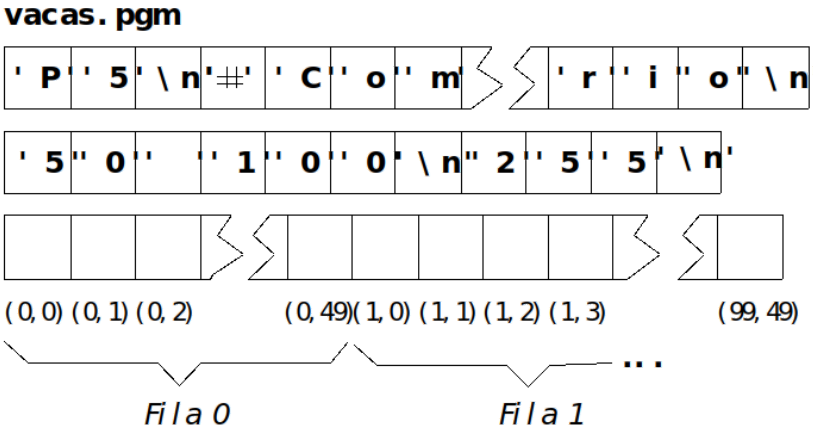


Figura 3. Almacenamiento de una imagen de niveles de gris con 100 filas y 50 columnas en un fichero con formato PGM (vacas.pgm).

2.2 Librería ImageIO

Tenemos a nuestra disposición ya implementada una librería para trabajar con imágenes PGM. Conocemos la especificación y no debemos preocuparnos por su implementación. Dispone entre otros de las siguientes funciones:

◆ ReadPGMImage()

```
unsigned char * ReadPGMImage ( const char *  path,
                               int &         rows,
                               int &         cols
                             )
```

Lee una imagen de tipo PGM.

Parameters

path archivo a leer

rows Parámetro de salida con las filas de la imagen.

cols Parámetro de salida con las columnas de la imagen.

Returns

puntero a una nueva zona de memoria que contiene *filas* x *columnas* bytes que corresponden a los grises de todos los píxeles (desde la esquina superior izqda a la inferior drcha). En caso de que no no se pueda leer, se devuelve cero. (0).

Postcondition

En caso de éxito, el puntero apunta a una zona de memoria reservada en memoria dinámica. Será el usuario el responsable de liberarla.

◆ WritePGMImage()

```
bool WritePGMImage ( const char *          path,
                     const unsigned char * datos,
                     const int             rows,
                     const int             cols
                   )
```

Escribe una imagen de tipo PGM.

Parameters

path archivo a escribir

datos punteros a los *f* x *c* bytes que corresponden a los valores de los píxeles de la imagen de grises.

rows filas de la imagen

cols columnas de la imagen

Returns

si ha tenido éxito en la escritura.

3.- El TDA imagen

A la hora de implementar programas de manipulación de imágenes digitales debemos plantearnos la conveniencia de utilizar un tipo de dato abstracto (TDA en lo sucesivo) para este propósito. Antes de nada, debemos considerar que previamente a su manipulación se requiere que la imagen resida en memoria y que, una vez que finalice el procesamiento, deberemos liberar la memoria que ocupaba. Hay dos operaciones básicas necesarias que son comunes a prácticamente la totalidad de procesos sobre una imagen: consultar el valor de un punto de la imagen y asignar un valor a un punto de la imagen. Todas las operaciones de procesamiento se pueden hacer en base a estas dos simples operaciones. Realmente no se requiere más para construir un programa de procesamiento de imágenes, aunque es útil disponer de dos operaciones complementarias: consultar el número de filas y consultar el número de columnas de una imagen.

Así, tras este breve análisis podemos plantear ya el TDA imagen. En primer lugar describiremos de forma abstracta el TDA (sección 2.1) a nivel de dominio y operaciones asociadas, para, posteriormente, proponer una representación de los objetos de tipo imagen y una implementación de las operaciones basadas en esa representación.

3.1 Descripción abstracta del TDA imagen

3.1.2 Dominio

Una imagen digital de niveles de gris, i, puede verse como una matriz bidimensional de **nf** filas y **nc** columnas, en la que en cada elemento, i (f, c), se guarda el nivel de luminosidad de ese punto. El tipo asociado a un objeto de estas características será el tipo **imagen**. Los valores de luminosidad están en el conjunto {0,1, ..., 255} de forma que el 0 indica mínima luminosidad (y se visualiza con el color negro) y el 255 la máxima luminosidad (y se visualiza con el color blanco). Los restantes valores indican niveles intermedios de luminosidad y se visualizan en distintos tonos de grises, siendo más oscuros cuanto menor sea su valor. El tipo asociado a cada una de las casillas de un objeto de tipo **imagen** será el tipo **byte**.

3.1.2 Especificación

Definimos una imagen digital i como

```
Image i;
```

Las operaciones básicas asociadas son las siguientes:

1. Creación de una imagen.
2. Destrucción de una imagen.
3. Consultar el número de filas de una imagen.
4. Consultar el número de columnas de una imagen.
5. Asignar un valor a un punto de la imagen.
6. Consultar el valor de un punto de la imagen.

Algunas operaciones más complejas pueden ser:

1. Cargar una imagen desde un archivo.
2. Guardar una imagen en un archivo.

En el archivo `Image.h` puedes encontrar la **especificación** de estos y otros métodos, sus precondiciones y postcondiciones.

3.1.3 Ejemplos de uso del TDA imagen

Una vez definido de forma abstracta el TDA imagen, mostraremos algunos ejemplos de uso. Obsérvese que en ningún momento hemos hablado de su representación exacta en memoria ni de la implementación de las operaciones. De hecho, este es nuestro propósito, mostrar que puede trabajarse correctamente sobre objetos de tipo imagen independientemente de su implementación concreta.

Ejemplo: invertir una imagen

Se trata de invertir una imagen plana. Queremos invertir una imagen existente alojada en la ruta `origen` y guardarla en la ruta `destino`.

```
Image image;
image.Load(origen);
for (int i=0; i < image.get_rows(); i++)
    for (int j=0; j < image.get_cols(); j++)
        image.set_pixel( i, j, MAX_BYTE_VALUE - image.get_pixel(i, j));
image.Save(destino);
```

Así, podemos invertir cualquier imagen, abstrayéndonos de su representación interna. De hecho, si entendemos que invertir una imagen es una operación habitual en el contexto del procesamiento de imágenes (parece que lo es), podemos encapsular su funcionamiento en un método de la propia clase.

```
Image image;
image.Load(origen);
image.Invert();
image.Save(destino);
```

De esta forma nos abstraemos de los detalles de la representación y la implementación. A través de la especificación de los métodos nos abstraemos del **cómo** y nos centramos en el **qué**.

3.2 Representación del tipo imagen

Aunque una imagen se vea o se contemple como una matriz bidimensional estática, no es ésta la representación más adecuada para los objetos de tipo imagen. La razón es que el compilador debe conocer las dimensiones de la matriz en tiempo de compilación y ésto no responde a nuestros propósitos al ser demasiado restrictivo. Así, debe escogerse una representación que permita reservar memoria para la imagen en tiempo de ejecución.

Resulta evidente que la naturaleza bidimensional de la imagen hace que una buena representación sea la de una matriz bidimensional dinámica. Esto nos permite acceder a cada punto de la imagen a través de la notación mediante índices y simplifica la programación de las operaciones.

Entre las distintas alternativas de implementación de matrices bidimensionales mediante vectores de punteros, la elección de una u otra dependerá de la especificación de las operaciones que vayamos a realizar sobre las imágenes y del conjunto de operaciones primitivas que proporcionemos al usuario del TDA.

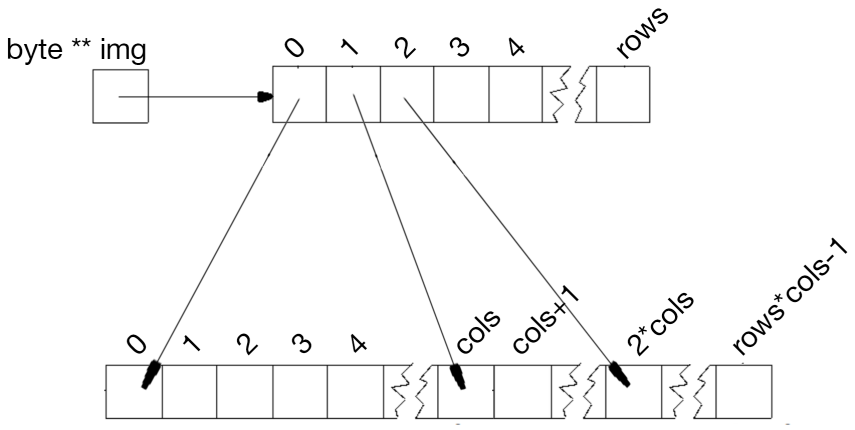


Figura 4. Representación sugerida

Ya que la librería ImageIO espera y devuelve un puntero a **`byte`**s podemos plantearnos esta como una posible representación (ver Figura 3). Los pasos a seguir para obtener la matriz dinámica `img` serían:

1. Reservamos `rows` punteros a listas de `bytes`.
   ```
   img = new byte * [rows]
   ```
2. Reservamos rows*cols bytes
   ```
   byte * total = new byte [rows * cols]
   ```
3. Asignamos a cada puntero a fila la posición del primer byte correspondiente.
   ```
   for (int i=0; i < rows; i++)
               img[i] = total[i*cols];
   ```

La clase Image, en la parte privada, además de la matriz bidimensional, el tamaño de la matriz:

```
      int rows;      // Numero de filas de la imagen
      int cols;       // Numero de columnas de la imagen
      unsigned char **img;    // La imagen en si: una matriz dinamica 2D
de bytes
```

Podemos agrupar los tipos presentados y los prototipos de las funciones en un fichero de cabecera, al que llamaremos `imagen.h`. La idea es tener una cabecera con las especificaciones por un lado y la implementación abstraída y contenida en la librería `libimage.a` por otro, generada a partir de los archivos fuente `image.cpp` (constructores y operaciones primitivas) y `imageop.cpp` (operaciones más complejas). Así, el usuario del TDA dispondrá de la biblioteca (con las funciones compiladas) y del fichero de cabecera asociado (con los tipos descritos anteriormente y los prototipos de las funciones públicas de la biblioteca).

Práctica a realizar

En esta práctica se propone la creación y documentación de la **clase Image**, que debe incluir además de los operadores básicos ya descritos, métodos para las operaciones descritas en los siguientes apartados. También debe implementarse un programa que gestione la lectura, transformación y guardado de la imágenes para cada tarea. Separamos cada tarea por lo tanto en:

1. *Especificación del método*: Describe **qué** hace el **método de la clase** Image que implementa un proceso concreto. (Ver Image::Invert() como ejemplo)

No entiendo bien a que se refiere con esto

2. *Especificación del programa*: Describe qué hace el programa encargado de la lógica completa de la operación, e.i. lectura, transformación y guardado de datos. (Ver negativo.cpp como ejemplo)

1.- Extracción de subimágenes

Especificación del método

◆ Crop()

```
Image Image::Crop ( int  nrow,
                    int  ncol,
                    int  height,
                    int  width
                  )      const
```

Genera una subimagen.

Parameters

nrow	Fila inicial para recortar
ncol	Columna inicial para recortar
height	Número de filas
width	Número de columnas

Returns

Imagen con el recorte.

Postcondition

El objeto que llama a la función no se modifica.

Especificación del programa `subimagen.cpp`

Se trata de realizar un programa que genere, por copia, una imagen PGM a partir de otra imagen PGM. La imagen generada será, necesariamente, de un tamaño menor o igual que la original. Debe apoyarse en el método Image::Crop(). Los parámetros de la función deberían ser:

<fich_orig> es el nombre del fichero que contiene la imagen original y
<fich_rdo> es el nombre del fichero que contendrá la subimagen resultado.
<fila> y <col> especifican la coordenada (sobre la imagen original) de la esquina superior izquierda de la subimagen que se va a extraer.
<filas_sub> y <cols_sub> indican el número de filas y columnas, respectivamente, de la subimagen que se va a extraer.

Por ejemplo, en la imagen siguiente mostramos una subimagen extraída con los parámetros:

```
vacas.pgm 85 145 60 80 cabeza.pgm
```

2.- Zoom de una porción de la imagen

Consiste en realizar un zoom 2X de una porción de la imagen mediante un simple procedimiento de interpolación consistente en construir a partir de una subimagen N x N, una imagen de dimensión (2N-1) x (2N-1), poniendo entre cada 2 filas (resp columnas) de la imagen original otra fila (resp. columna) cuyos valores de gris son la media de los que tiene a la izquierda y a la derecha (resp. arriba y abajo). P.ej. a partir de un trozo 3 x 3 se genera una imagen 5 x 5 de la siguiente forma:

```
10   6  10
 6  10   6
10   4  10
```

Se interpola sobre las columnas:

```
10   8   6   8  10
 6   8  10   8   6
10   7   4   7  10
```

Finalmente se interpola sobre las filas:

```
10   8   6   8  10
 8   8   8   8   8
 6   8  10   8   6
 8 7.5   7 7.5   8
10   7   4   7  10
```

donde los valores reales se redondean al entero más próximo por exceso (p.ej. 7.5 pasa a ser 8)

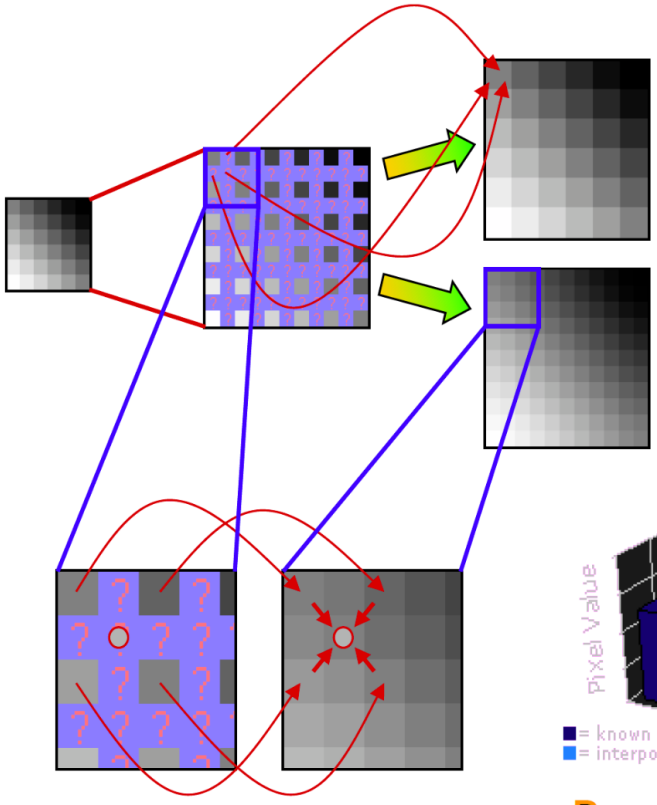
Especificación del método
No hay requerimientos especiales sobre los métodos que la implementen. Se debe procurar no duplicar funcionalidades ya implementadas.

Especificación del programa `zoom.cpp`

Los parámetros de la función deberían ser:

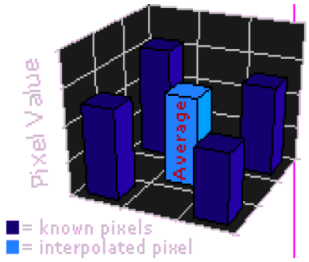
<fich_orig> es el nombre del fichero que contiene la imagen original y
<fich_rdo> es el nombre del fichero que contendrá el resultado del zoom.
<fila> y <col> especifican la coordenada (sobre la imagen original) de la esquina superior izquierda de la subimagen que se va a extraer.
<lado> tamaño del lado del cuadrado. El cuadrado descrito debe estar totalmente incluído en la imagen.

Interpolación bilineal (media) para aumentar el zoom.



Replicando

Interpolación bilineal



Replicando píxeles

Interpolación

Zoom x3

Interpolación bilineal

3.- Crear un icono a partir de una imagen.

Consiste en crear una imagen de un tamaño muy reducido a partir de una imagen original.
El algoritmo de reducción consiste básicamente en tomar cada píxel de la salida como la media de los n x n píxeles de la entrada si se quiere hacer una reducción de nx. P.ej si la imagen de entrada es:

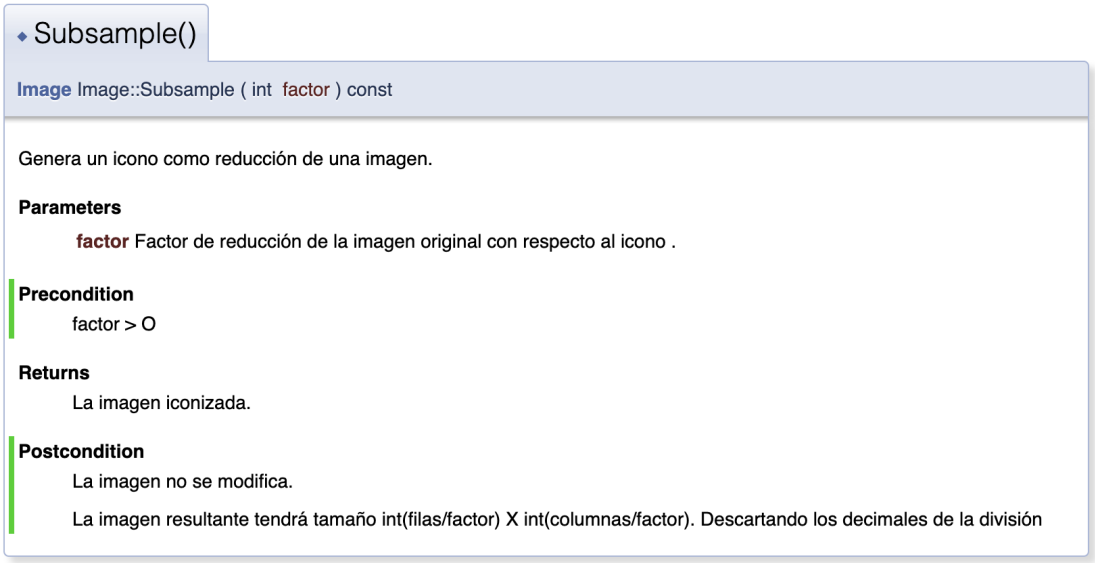
```
10  6 10 12
 6 10  6  8
10  4 10 10
12 10  6  8
```

y se hace una reducción 2x quedaría la imagen:

```
8 9
9 9
```

donde los valores reales se redondean al entero más próximo (p.ej., 8.4 pasa a ser 8 y 8.5 pasa a ser 9).

Especificación del método



◆ Subsample()

Image Image::Subsample (int factor) const

Genera un icono como reducción de una imagen.

Parameters

factor Factor de reducción de la imagen original con respecto al icono .

Precondition

factor > O

Returns

La imagen iconizada.

Postcondition

La imagen no se modifica.

La imagen resultante tendrá tamaño int(filas/factor) X int(columnas/factor). Descartando los decimales de la división

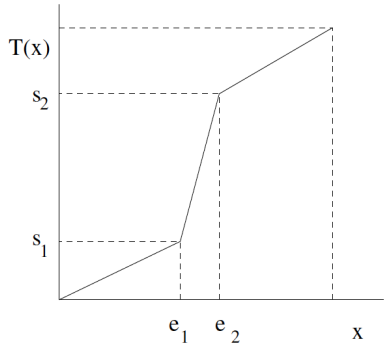
Especificación del programa `icono.cpp`
Los parámetros de la función deberían ser:

<fich_orig> nombre del fichero que contiene la imagen original.
<fich_rdo> nombre del fichero donde se guardará el icono.
<factor> factor de reducción. La imagen resultante tendrá floor(filas/factor) x floor(cols/factor)

4.- Aumento de contraste de una imagen mediante un estiramiento del histograma.

Consiste en generar una imagen de niveles de gris con más contraste que la original. Por ejemplo, en la imagen siguiente, a la izquierda, mostramos una imagen con poco contraste y en la derecha, la misma imagen una vez aumentado el contraste.


Esta operación se conoce también como “estirar” el histograma o **histogram stretching**. En realidad se trata de aplicar una transformación lineal a trozos sobre los valores de los píxeles. La transformación a aplicar depende del valor del píxel y se puede deducir a partir de la siguiente gráfica:



En la gráfica, **x** es el nivel de gris original y T(x) es el nuevo nivel de gris resultante de la transformación.Se deduce que los nuevos valores de los píxeles estarán en el rango:

- $[0,s_1 - 1]$ si sus valores iniciales están en $[0,e_1 - 1]$,
- $[s_1, s_2]$ si sus valores iniciales están en $[e_1,e_2]$, y
- $[s_2 + 1,255]$ si sus valores iniciales están en $[e_2 + 1,255]$.

A la hora de diseñar la transformación lineal a trozos, debe considerarse lo siguiente: supongamos que el rango de niveles de gris en el eje x es [a,b], o sea, el mínimo nivel de gris en ese trozo es a y el máximo es b. Es evidente que:

```
a <= z <= b
```

donde z es el nivel de gris de un píxel cualquiera de ese rango.

Si queremos que el nuevo rango sea [min, max], la transformación lineal a aplicar, t, a los niveles de gris de la imagen inicial, z, será la siguiente:

```
T(z) = z' = min + [((max – min) / (b – a)) * (z – a)] (E1)
```

Debemos comentar con algún detalle la manera en la que se debe implementar la función de cálculo del contraste. En primer lugar debemos considerar que la función se va a evaluar para cada píxel en ese rango de la imagen por lo que debemos evitar, en lo posible, realizar cálculos

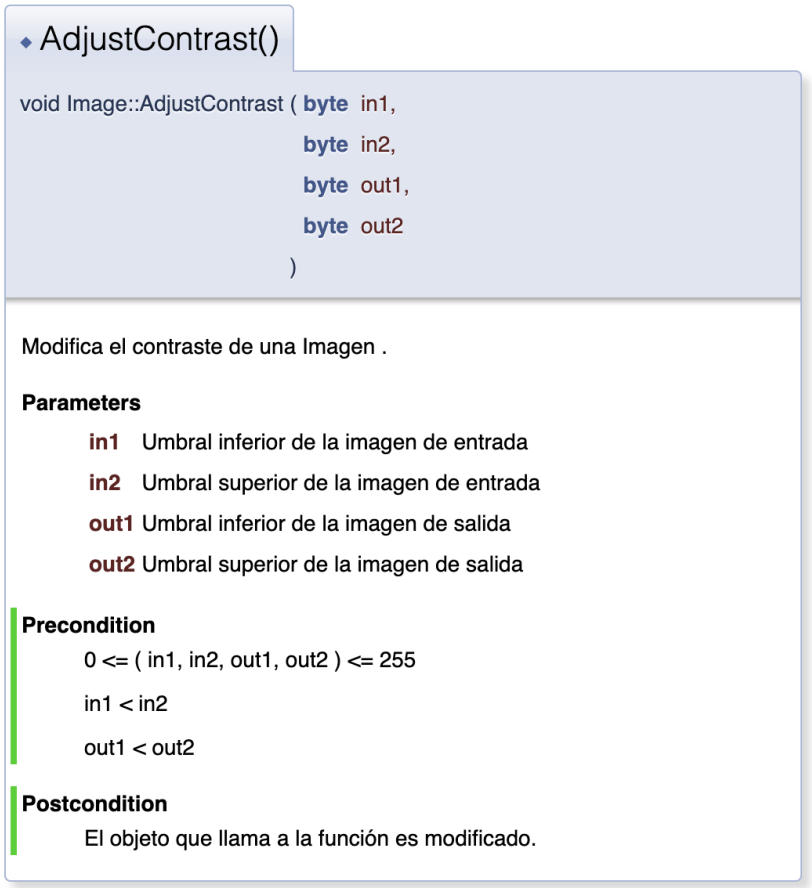
redundantes. En segundo lugar, hay que tener en cuenta que en los cálculos intervienen valores reales por lo que hay que prestar atención a los posibles errores por redondeo.

Sobre el primer punto a tener en cuenta, observar que en la expresión E1 anterior, hay una parte constante e independiente del valor de cada píxel a transformar, el cociente:

```
(max - min)/(b - a)
```

Esta es la razón de que se deba calcular fuera de los ciclos que recorren la imagen. El resultado de este cociente se puede guardar en una variable fija.

Sobre el segundo punto, hay que considerar que el valor z' = T(z) (calculado en la variable valor) es de tipo `double` y a la hora de asignarlo a la imagen contrastada hay que convertirlo a `byte`. El valor a guardar debe ser el entero más cercano, por lo que debe redondearse a éste.
Especificación del método



◆ AdjustContrast()

void Image::AdjustContrast (**byte** in1,
byte in2,
byte out1,
byte out2
)

Modifica el contraste de una Imagen .

Parameters

in1 Umbral inferior de la imagen de entrada
in2 Umbral superior de la imagen de entrada
out1 Umbral inferior de la imagen de salida
out2 Umbral superior de la imagen de salida

Precondition

0 <= (in1, in2, out1, out2) <= 255
in1 < in2
out1 < out2

Postcondition

El objeto que llama a la función es modificado.

Nótese que, al contrario que en los casos anteriores, se modifica la propia imagen que llama a este método. Como regla general, los métodos que modifiquen el tamaño devolverán una nueva imagen, mientras que las operaciones sobre los píxeles se harán *inplace*.

Por simplicidad las operaciones que se apliquen sobre la totalidad de los píxeles no deberían tener que llevar la lógica de filas y columnas. Para esto mismo están implementadas las funciones

```
byte get_pixel(int k);
void set_pixel(int k, byte v);
```

Donde `k < rows*cols`.

La implementación actual de estas dos funciones tratan directamente con la representación interna de la matriz, cuando podrían abstraerse y utilizar los métodos primitivos ya disponibles. Por lo general, siempre que sea posible, debemos apoyarnos en la abstracción de la representación, reduciendo al mínimo las operaciones que tratan con ella.

Especificación del programa `contraste.cpp`

Los parámetros serían, en este orden:

<fich_orig> es el nombre del fichero que contiene la imagen original y <fich_rdo> es el nombre del fichero que contendrá la imagen resultado.
<e1>, <e2>, <s1> y <s2> son los valores usados para la transformación.

(OPCIONAL) 5.- Barajado de las filas de una imagen. Consideraciones sobre la eficiencia de las operaciones.

Existen casos de uso o especificaciones que nos obligan a replantearnos la representación interna utilizada en nuestros TDA. Hasta ahora nuestras decisiones han estado condicionadas por las operaciones de lectura y escritura de imágenes, ofrecidas por una librería ya implementada con una especificación impuesta. Sin embargo, se nos presenta un nuevo caso de uso con nuevas implicaciones.

Especificación para el método `void Image::ShuffleRows()`
Parámetros de entrada: Ninguno
Precondiciones:
- rows < 9973

Salida: Ninguna
Poscondiciones:
- El objeto que llama al método contiene ahora una nueva imagen igual que la anterior pero con las filas ordenadas según el siguiente algoritmo:

$$\hat{r} = (r * p) \bmod \text{rows}$$

Donde $\hat{r}$ es el nuevo índice de la fila r, p es un **coprimo** de *rows,* y *rows* es el número de filas de la imagen. (Dos números son **coprimos** si no tienen ningún factor primo en común, por simplicidad usaremos 9973 como número primo por defecto). Una implementación posible sería la siguiente:

```
void Image::ShuffleRows() {
      const int p = 9973;
      Image temp(rows,cols);
      int newr;


      for (int r=0; r<rows; r++){
          newr = r*p;
          for (int c=0; c<cols;c++)
              temp.set_pixel(newr,c,get_pixel(r,c));
      }

      Copy(temp);
}
```

Se incluye una imagen de ejemplo `shuffle_9973.pgm` que debería resultar en una imagen legible una vez barajada con p=9973.

La complejidad computacional de esta operación es O(rows*cols). Si nuestro caso de uso necesitase realizar esta operación con mucha frecuencia es posible que nos interese cambiar la representación de nuestro tipo de dato para agilizar este proceso.

Una representación alternativa a la actual es la siguiente: `byte ** img` ahora apunta a una lista de `byte *` no necesariamente consecutivos en memoria. Podemos, o bien llamar a `row[i] = new byte[cols]` por cada fila (ver Figura 5) o bien, más sencillo, mantener la implementación como hasta ahora (Ver apartado 3.2) pero dejar de asumir que `img[i+1] == img[i]+cols` tan pronto como finalicemos la inicialización de la imagen. Si lo hemos hecho todo correctamente, el único implicado en este cambio debería ser `bool Image::Save (const char * file_path) const`.

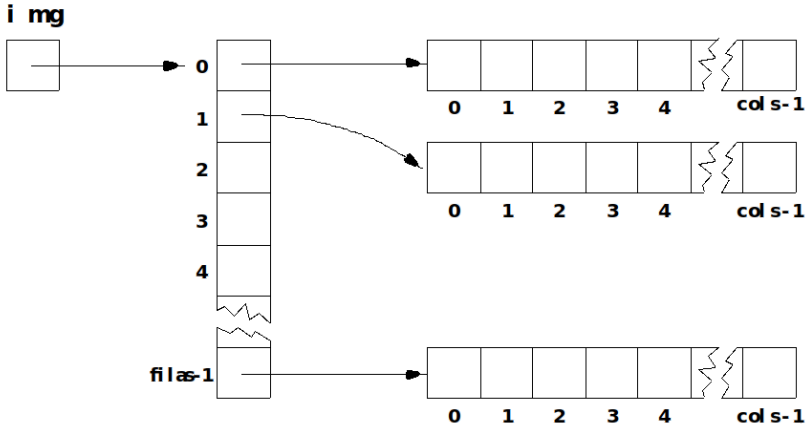


Figura 5. Representación alternativa del contenido de una imagen

Lo que se pide en este ejercicio es:

- Evaluar la complejidad experimental para:
  - Distinto número de filas
  - Distinto número de columnas
  - Distinto número de llamadas a `ShuffleRows()` entre `Load()` y `Save()`
- Cambiar la representación interna a una basada en filas no consecutivas como las explicadas antes y adaptar el código para que todo siga funcionando como hasta ahora. Es posible que alguna primitiva degrade su rendimiento.
- Modificar `ShuffleRows()` para aprovechar las ventajas de la nueva representación para su caso de uso.
- Repetir el análisis de la complejidad experimental para la nueva representación.
- Comparar y justificar los resultados.
- Se valorará positivamente no solo la generación de gráficas sino el ajuste y obtención de las constantes de la complejidad teórica estimada.

** Para el análisis y la generación de gráficas puede usarse gnuplot o cualquier otro método que el estudiante prefiera.*

4.- Documentación y entrega

Toda la documentación de la práctica se incluirá en el propio Doxygen generado, para ello se utilizará tanto las directivas Doxygen de los archivos .h y .cpp como los archivos .dox incluídos en la

carpeta doc. (Leed detenidamente la documentación de ejemplo para ver cómo desarrollarla correctamente).

1. La documentación debe incluir:
   - TODOS los métodos y clases debidamente documentados con la especificación completa. Se valorará positivamente descripciones exhaustivas y el uso de elementos adicionales como fórmulas e imágenes. (La documentación disponible de ejemplo puede y DEBE ser completada)
   - Cada tarea de la práctica tiene un ejecutable asociado que debe ser descrito en la página principal o en una página *ad hoc*. Se valorará positivamente el uso de imágenes para ilustrar el funcionamiento de los programas. (Ver doc/mainpage.dox como ejemplo)
   - En caso de hacer el **Ejercicio 5**, se deberá incluir el análisis de la complejidad teórica y experimental y la comparativa en la página principal o en una página específica.
   - Todos las páginas de la documentación generada deben ser completas y estar debidamente descritas.

2. A considerar:
   - Deben respetarse todos los conocimientos adquiridos en los temas de Abstracción y Eficiencia. En especial el **principio de ocultamiento de información** y las distintas estrategias de **abstracción**.
   - Una solución algorítmicamente correcta pero que contradiga el punto anterior será considerada como errónea.
   - El número de métodos que interactúen directamente con la representación interna del TDA deberá mantenerse al mínimo posible y deberá estar debidamente justificado su uso.

3. Código:
   - Se dispone de la siguiente estructura de ficheros:
     - /
       - estudiante/ (Aquí irá todo lo que desarrolle el alumno)
         - doc/ (Imágenes y documentos extra para Doxygen)
         - include/ (Archivos cabecera)
         - src/ (Archivos fuente)
       - img/ (Imágenes de ejemplo)
       - expected_img/ (Salida esperada en las pruebas públicas del juez)
       - CMakeList.txt (Instrucciones CMake)
       - Doxyfile.in (Archivo de configuración de Doxygen)
       - juez.sh (Script del juez online **[HAY QUE CONFIGURARLO]**)
   - Es importantísimo leer detenidamente y entender qué hace CMakeList.txt.
   - El archivo Doxyfile.in no tendríais por qué tocarlo para un uso básico, pero está a vuestra disposición por si queréis añadir alguna variable (no cambiéis las existentes)
   - juez.sh crea un archivo submission.zip que incluye ya TODO lo necesario para subir a Prado a la hora de hacer la entrega. No tenéis que añadir nada más, especialmente binarios o documentación ya compilada.

4. Elaboración y puntuación
   - La práctica se realizará POR PAREJAS. Los nombres completos se incluirán en la descripción de la entrega en Prado. Cualquiera de los integrantes de la pareja puede subir el archivo a Prado, pero SOLO UNO.
   - Desglose:
     - **(0.25)** Ejercicios 1-4
     - **(0.25)** Documentación
     - **(0.25)** Ejercicio 5 *[OPCIONAL]*
   - La fecha límite de entrega aparecerá en la subida a Prado.