Trabajo Práctico 2

Organización del Computador II

Segundo Cuatrimestre de 2014

1. Introducción

El objetivo de este trabajo práctico es explorar el modelo de programación **SIMD**. Una aplicación popular del modelo SIMD es el procesamiento de imágenes y video.

En este trabajo práctico se implementarán filtros para estas aplicaciones, utilizando lenguaje ensamblador (ASM) e instrucciones **SSE**, y se analizará la performance del procesador al hacer uso de **SIMD** para estas aplicaciones, comparándolas con sus implementaciones respectivas en lenguaje **C**.

Para cada filtro se entrega una descripción matemática y algorítmica exacta. Notar que para las optimizaciones no es necesario seguir el procedimiento al pie de la letra. Lo importante es que el resultado final sea el mismo que el dado por la descripción matemática.

2. Filtros

Los filtros a implementar se describen a continuación. Aquí una imagen de cada uno a modo de ejemplo.

Imagen original

Cropflip

Sierpinski

Bandas

Motion blur

2.1. Filtro de Cropflip

El filtro *cropflip* es una unión de dos filtros: crop y vertical-flip. Recorta una parte de la imagen original y la voltea verticalmente. Este filtro se aplica píxel a píxel en una imagen en color. Se reciben cuatro argumentos que representan un rectángulo dentro de la imagen fuente:

tamx: la cantidad de columnas, en píxeles, a recortar. Este número es múltiplo de 4.

tamy: la cantidad de filas, en píxeles, a recortar.

offsetx: columna, en píxeles, a partir de la cual debe comenzar a recortarse. Este número es múltiplo de 4.

offsety: fila, en píxeles, a partir de la cual debe comenzar a recortarse.

El filtro recorta un rectángulo de **tamx** píxeles de ancho por **tamy** píxeles de alto a partir de la columna **offsetx** fila **offsety** y lo pega en la imagen destino, **volteado verticalmente**.

El tamaño de la imagen destino es tamx píxeles de ancho por tamy píxeles de alto. Vale $tamx + offsetx < cols \wedge tamy + offsety < rows$, es decir que el rectángulo siempre va a entrar (donde rows y cols son las filas y columnas, en píxeles, de la imagen fuente).

Por ejemplo, en la siguiente imagen de 128 x 128 píxeles, llamar a cropflip con $tamx = 64$, $tamy = 60$, $offsetx = 56$, $offsety = 40$, nos devuelve una imagen volteada verticalmente de 60 píxeles de alto por 64 píxeles de ancho, que se corresponde con la imagen fuente a partir de la columna 56 fila 40:

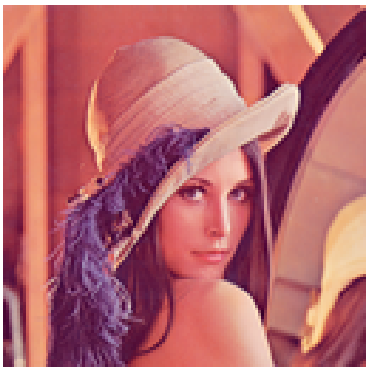

Imagen original

Cropflip

La descripción matemática de Cropflip está dada por la siguiente función:

$$\mathsf{dst}_{(i,\,j)} = \mathsf{src}_{(tamy + offsety - i - 1,\, offsetx + j)}$$

donde i y j representan fila y columna respectivamente, $i \in [0, tamy)$, $j \in [0, tamx)$.

2.2. Sierpinki

El filtro sierpinski toma una imagen fuente y genera un efecto fractálico encima, simulando los triangulos de sierpinski pero con cuadrados. El filtro toma la posición actual que esta recorriendo en la imagen y, dada cierta cuenta, genera un coeficiente entre 0 y 1, que se multiplica sobre cada componente de cada pixel.

$$\mathsf{dst}_{(\mathrm{i,j})} = \mathsf{src}_{(\mathrm{i,j})} * \mathsf{coef}_{(\mathrm{i,j})}$$

$$\mathsf{coef}_{(\mathrm{i,j})} = \frac{1}{255{,}0}(\lfloor \frac{i}{cant_{filas}} * 255{,}0 \rfloor \oplus \lfloor \frac{j}{cant_{cols}} * 255{,}0 \rfloor)$$

Las fracciones denotan que la operación se hace en punto flotante, mientras que las partes enteras denotan que se deben pasar a enteros para poder hacer el $\oplus$, el XOR de los bits. Todas las conversiones de doble a entero se deben hacer truncando, no redondeando.

2.3. Bandas

El filtro *bandas* toma una imagen fuente y genera bandas en varios tonos de gris. Se suman los componentes R, G y B de cada píxel, dando como resultado un número b entre 0 y 765.

$$\mathsf{b}_{(i,j)} = \mathsf{src}.r_{(i,j)} + \mathsf{src}.g_{(i,j)} + \mathsf{src}.b_{(i,j)}$$

En función de b, se determina el color del pixel en la imagen destino.

$$\mathsf{dst}_{(i,j)} < r, g, b >= \begin{cases} < 0, 0, 0 > & \text{si } b < 96 \\ < 64, 64, 64 > & \text{si } 96 \leq b < 288 \\ < 128, 128, 128 > & \text{si } 288 \leq b < 480 \\ < 192, 192, 192 > & \text{si } 480 \leq b < 672 \\ < 255, 255, 255 > & \text{si no} \end{cases}$$

2.4. Motion blur

El filtro *motion blur* toma una imagen fuente y aplica un efecto de desenfoque de movimiento. Esto se logra tomando parte del valor del píxel original y añadiendo partes del valor de sus vecinos. En este trabajo, el desenfoque será de un movimiento de 45 grados.

Para cada componente independiente del píxel (R, G y B) la fórmula matemática será

$$\mathsf{dst}_{(i,j)} = 0{,}2 \cdot \mathsf{src}_{(i-2,j-2)} + 0{,}2 \cdot \mathsf{src}_{(i-1,j-1)} + 0{,}2 \cdot \mathsf{src}_{(i,j)} + 0{,}2 \cdot \mathsf{src}_{(i+1,j+1)} + 0{,}2 \cdot \mathsf{src}_{(i+2,j+2)}$$

El resultado debe saturarse en 255. A la hora de implementar este filtro, dado que se realizan cálculos en punto flotante, se permitirá para $\mathsf{dst}_{(i,j)}$ una tolerancia de ± 5.

Además, dado que en los bordes no es posible calcular *mblur* por la ausencia de vecinos, deberá escribirse en esos casos el valor 0. Es decir, que vale que

$$\mathsf{dst}_{(i,j)} = 0 \text{ si } i < 2 \vee j < 2 \vee i + 2 \geq \mathsf{tamy} \vee j + 2 \geq \mathsf{tamx}$$

donde i está i indexado a partir de 0.

A la hora de implementar este filtro se puede asumir que la imagen tiene un ancho mayor a 32 pixeles y multiplo de 4.

3. Enunciado

Este trabajo tiene dos objetivos principales

- Explorar el modelo de programación **SIMD**
- Realizar un análisis **riguroso** de los resultados de performance del procesador al hacer uso de las intrucciones **SSE**.

Para esto, cada uno de los filtros deberá ser implementado en dos versiones: una en lenguaje **C**, y una en **ASM** haciendo uso de las instrucciones **SSE**.

Los ejercicios que se enumeran a continuación sirven como guía mínima para que realicen optimizaciones y analicen los resultados de las mismas. También pueden plantear otras optimizaciones que surjan del desarrollo de cada filtro y acompañarlas de un respectivo análisis.

Nota: No intentar realizar el código ASM directamente, implementar primero el código C para asegurarse haber comprendido los detalles de implementación de cada filtro.

Preámbulo

A la hora de analizar la velocidad con la que se ejecuta un programa es importante tener en cuenta cuales son las limitaciones inherentes a nuestro modelo de cómputo. En una arquitectura de Von Neumann (como la de Intel), las limitaciones de performance suelen dividirse en dos grandes áreas: capacidad de cómputo y ancho de banda de la memoria.

Es decir, dado un hardware fijo, lo que hace que un programa no termine más rápidamente puede ser,

- o bien que tarda mucho tiempo en procesar cada dato
- o bien que tarda mucho tiempo en recibir y enviar datos desde y hacia la memoria.

Si un algoritmo realiza muchos operaciones aritméticas y accede poco a memoria, tardará mas en procesar cada dato que en recibirlo de memoria, y por lo tanto estará limitado por la capacidad de cómputo. Si en cambio, realiza poco procesamiento pero con grandes cantidades de datos, tardará más en la transmisión de los datos que en su procesamiento, y por lo tanto estará limitado por el ancho de banda de la conexión con la memoria.

A lo largo de este trabajo experimentaremos cómo esto afecta la performance de las diferentes implementaciónes de los filtros.

3.1. Filtro cropflip

Programar el filtro *cropflip* en lenguaje C y luego en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 1.1 - análisis el código generado

En este experimento vamos a utilizar la herramienta `objdump` para verificar como el compilador de C deja ensamblado el código C.

Ejecutar

```
objdump -Mintel -D cropflip_c.o
```

¿Cómo es el código generado? Indicar *a*) Por qué cree que hay otras funciones además de `cropflip_c` *b*) Cómo se manipulan las variables locales *c*) Si le parece que ese código generado podría optimizarse

Experimento 1.2 - optimizaciones del compilador

Compile el código de C con flags de optimización. Por ejemplo, pasando el flag `-O1`[1]. Indicar a) Qué optimizaciones observa que realizó el compilador b) Qué otros flags de optimización brinda el compilador c) Los nombres de tres optimizaciones que realizan los compiladores.

3.2. Mediciones

Realizar una medición de performance *rigurosa* es más difícil de lo que parece. En este experimento deberá realizar distintas mediciones de performance para verificar que sean buenas mediciones.

En un sistema "ideal" el proceso medido corre solo, sin ninguna interferencia de agentes externos. Sin embargo, una PC no es un sistema ideal. Nuestro proceso corre junto con decenas de otros, tanto de usuarios como del sistema operativo que compiten por el uso de la CPU. Esto implica que al realizar mediciones aparezcan "ruidos" o "interferencias" que distorsionen los resultados.

El primer paso para tener una idea de si la medición es buena o no, es tomar varias muestras. Es decir, repetir la misma medición varias veces. Luego de eso, es conveniente descartar los outliers [2], que son los valores que más se alejan del promedio. Con los valores de las mediciones resultantes se puede calcular el promedio y también la varianza, que es algo similar el promedio de las distancias al promedio[3].

Las fórmulas para calcular el promedio μ y la varianza σ^2 son

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

[1] agregando este flag a `CCFLAGS64` en el makefile

[2] en español, valor atípico: `http://es.wikipedia.org/wiki/Valor_atpico`

[3] en realidad, elevadas al cuadrado en vez de tomar el módulo

Experimento 1.3 - calidad de las mediciones

a) Medir el tiempo de ejecución de cropflip 10 veces.

b) Implementar un programa en C que no haga más que ciclar infinitamente sumando 1 a una variable. Lanzar este programa tantas veces como *cores lógicos* tenga su procesador. Medir otras 10 veces mientras estos programas corren de fondo.

c) Calcular el promedio y la varianza en ambos casos.

d) Consideraremos outliers a los 2 mayores tiempos de ejecución de la medicion a) y también a los 2 menores, por lo que los descartaremos. Recalcular el promedio y la varianza después de hacer este descarte.

e) Realizar un gráfico que presente estos dos últimos items.

A partir de aquí todos los experimentos de mediciones deberán hacerse igual que en el presente ejercicio: tomando 10 mediciones, luego descartando outliers y finalmente calculando promedio y varianza.

Experimento 1.4 - secuencial vs. vectorial

En este experimento deberá realizar una medición de las diferencias de performance entre las versiones de C y ASM (el primero con -O0, -O1, -O2 y -O3) y graficar los resultados.

Experimento 1.5 - cpu vs. bus de memoria

Se desea conocer cual es el mayor limitante a la performance de este filtro en su versión ASM.

¿Cuál es el factor que limita la performance en este caso? En caso de que el limitante fuera la intensidad de cómputo, entonces podrían agregarse instrucciones que realicen accesos a memoria extra y la performance casi no debería sufrir. La inversa puede aplicarse, si el limitante es la cantidad de accesos a memoria. [4]

Realizar un experimento, agregando 4, 8 y 16 instrucciones aritméticas (por ej `add rax, rbx`) analizando como varía el tiempo de ejecución. Hacer lo mismo ahora con instrucciones de acceso a memoria, haciendo mitad lecturas y mitad escrituras (por ejemplo, agregando dos `mov rax, [rsp]` y dos `mov [rsp+8], rax`).[5]

Realizar un único gráfico que compare: a) La versión original b) Las versiones con más instrucciones aritméticas c) Las versiones com más accesos a memoria

Acompañar al gráfico con una tabla que indique los valores graficados.

Filtro *Sierpinski*

Programar el filtro *Sierpinski* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 2.1 - secuencial vs. vectorial

[4] también podría pasar que estén más bien balanceados y que agregar cualquier tipo de instrucción afecte sensiblemente la performance

[5] Notar que en el caso de acceder a `[rbp]` o `[rsp+8]` probablemente haya siempre hits en la cache, por lo que la medición no será de buena calidad. Si se le ocurre la manera, realizar accesos a otras direcciones alternativas.

Analizar cuales son las diferencias de performace entre las versiones de C y ASM de este filtro, de igual modo que para el experimento 1.4.

Experimento 2.1 - cpu vs. bus de memoria

¿Cuál es el factor que limita la performance en este filtro? Repetir el experimento 1.5 para este filtro.

Filtro *Bandas*

Programar el filtro *Bandas* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 3.1 - saltos condicionales

Se desea conocer que tanto impactan los saltos condicionales en el código de filtro Bandas con `-O1` (la versión en C).
Para poder medir esto de manera aproximada, remover el código que detecta a que banda pertenece cada pixel, dejando sólo una banda. Por más que la imagen resultante no sea correcta, será posible tomar una medida aproximada del impacto de los saltos condicionales. Analizar como varía la performance.

Experimento 3.2 - secuencial vs. vectorial

Repetir el experimento 1.4 para este filtro.

Filtro *Motion Blur*

Programar el filtro *mblur* en lenguaje C y en ASM haciendo uso de las instrucciones **SSE**.

Experimento 4.1

Repetir el experimento 1.4 para este filtro

3.3. Código

Para implementar los filtros descriptos anteriormente, tanto en C como en ASM se deberán implementar las siguientes funciones para imágenes en color (32 bits):

- *cropflip_c*, *cropflip_asm*,
- *sierpinski_c*, *sierpinski_asm*,
- *bandas_c*, *bandas_asm*
- *mblur_c*, *mblur_asm*

Los parámetros genéricos de las funciones son:

- *src*: Es el puntero al inicio de la matriz de elementos de 32 bits sin signo (el primer byte corresponde al canal azul de la imagen (B), el segundo el verde (G), el tercero el rojo (R)), y el cuarto el alpha (A) que representa a la imagen de entrada. Es decir, como la imagen está en color, cada píxel está compuesto por 4 bytes.
- *dst*: Es el puntero al inicio de la matriz de elementos de 32 bits sin signo que representa a la imagen de salida.
- *filas*: Representa el alto en píxeles de la imagen, es decir, la cantidad de filas de las matrices de entrada y salida.
- *cols*: Representa el ancho en píxeles de la imagen, es decir, la cantidad de columnas de las matrices de entrada y salida.
- *src_row_size*: Representa el ancho en bytes de cada fila de la imagen incluyendo el padding, es decir, la cantidad de bytes que hay que avanzar para moverse a la misma columna de fila siguiente/anterior.

Además, la función cropflip recibe los siguientes argumentos.

- *tamx*: Cantidad de píxeles de ancho del recuadro a copiar.
- *tamy*: Cantidad de píxeles de alto del recuadro a copiar.
- *offsetx*: Cantidad de píxeles de ancho a saltear de la imagen fuente.
- *offsety*: Cantidad de píxeles de alto a saltear de la imagen fuente.

3.3.1. Consideraciones

Las funciones a implementar en lenguaje ensamblador deben utilizar el set de instrucciones **SSE**, a fin de optimizar la performance de las mismas. Tener en cuenta lo siguiente:

- El ancho de las imágenes es siempre mayor a 16 píxeles.
- No se debe perder precisión en ninguno de los cálculos, a menos que se indique lo contrario.
- La implementación de cada filtro deberá estar optimizada para el filtro que se está implementando. No se puede hacer una función que aplique un filtro genérico y después usarla para implementar los que se piden.
- Para el caso de las funciones implementadas en lenguaje ensamblador, deberán trabajar con **al menos 2 píxeles simultámeamente**.

  De no ser posible esto para algún filtro, deberá justificarse debidamente en el informe.

- El procesamiento de los píxeles se deberá hacer **exclusivamente** con instrucciones **SSE**. No está permitido procesarlos con registros de propósito general, a menos que se indique lo contrario.
- El TP se tiene que poder ejecutar en las máquinas del laboratorio.

3.4. Desarrollo

Para facilitar el desarrollo del trabajo práctico se cuenta con todo lo necesario para poder compilar y probar las funciones que vayan a implementar.

Dentro de los archivos presentados deben completar el código de las funciones pedidas. Puntualmente encontrarán el programa principal (de línea de comandos), denominado **tp2**, que se ocupa de parsear las opciones ingresadas por el usuario y ejecutar el filtro seleccionado sobre la imagen ingresada.

Para la manipulación de las imagenes/videos (cargar, grabar, etc.) el programa hace uso de la biblioteca **OpenCV**, por lo que no se requiere implementar estas funcionalidades.

Para instalar las dependencias necesarias, en las distribuciones basadas en **Debian** basta con ejecutar: *$ make installopencv*

Los archivos entregados están organizados en las siguientes carpetas:

- *bin* : Contiene el ejecutable del TP.
- *enunciado* : Contiene este enunciado.
- *src* : Contiene los fuentes del programa principal, junto con su respectivo **Makefile** que permite compilar el programa y algunas imágenes de prueba.
- *test*: Contiene scripts para realizar tests sobre los filtros y uso de la memoria, y las imágenes de prueba de la cátedra.

El uso del programa principal es el siguiente:

$ ./tp2 <opciones> <nombre_filtro> <nombre_archivo_entrada> [parámetros]

El programa soporta imagenes y videos, aunque para el TP las pruebas de la cátedra se correrán sólo con imágenes. Es posible usar el programa con la mayoría de los formatos de imágenes y videos comunes.

Los filtros que se pueden aplicar son:

- **cropflip**
  Parámetros: `tamx`, `tamy`, `offsetx`, `offsety`

  Ejemplo de uso: *cropflip -i c lena.bmp 150 150 250 300*
- **sierpinski**
  Parámetros: Ejemplo de uso: *sierpinski -i c lena.bmp*
- **bandas**
  Parámetros:
  Ejemplo de uso: *bandas -i c -w ink.avi*
- **mblur**
  Parámetros: Ejemplo de uso: *mblur -i lena.bmp*

Las opciones que acepta son las siguientes:

- *-h, –help*
  Imprime la ayuda
- *-i, –implementacion NOMBRE_MODO*
  Implementación sobre la que se ejecutará el proceso seleccionado. Los implementaciones disponibles son: c, asm
- *-t, –tiempo CANT_ITERACIONES*
  Mide el tiempo que tarda en ejecutar el filtro sobre la imagen de entrada una cantidad de veces igual a CANT_ITERACIONES
- *-f, –frames*
  Genera frames independientes en vez de armar un archivo de video. Es utilizado para testing. Sólo para archivos de video.
- *-o, –output CARPETA*
  Genera el resultado en CARPETA. De no incluirse, el resultado se guarda en la misma carpeta que el archivo fuente
- *-v, –verbose*
  Imprime información adicional
- *-w, –video*
  Interpreta el archivo de entrada como video. En caso de no estar, se interpreta la entrada como una imagen.

Por ejemplo:

$ ./tp2 -v cropflip -i asm lena.bmp 100 120 50 60

Aplica el filtro de **cropflip** al archivo lena.bmp utilizando la implementación en lenguaje asm del filtro, pasándole como parámetro 100, 120, 50 y 60 como valores de tamx, tamy, offsetx y offsety respectivamente.

3.4.1. Tests

Para verificar el correcto funcionamiento de los filtros, se provee un comparador de imágenes, el binario de la solución de la cátedra y varios scripts de test. El binario de la cátedra se encuentra en la carpeta *solucion/bin*. El comparador de imágenes se ubica en la carpeta *solucion/tools*, y debe compilarse antes de correr los scripts (correr *make* en la carpeta *solucion/tools*).

Los scripts de test toman como entrada las corridas especificadas en *corridas.txt*. Para cada imagen de test, se ejecutan todas las corridas ahí indicadas. Para verificar que la implementación funciona correctamente con imágenes de distinto tamaño, *generar_imagenes.sh* genera variaciones de las imágenes fuente (que se encuentran en *solucion/tests/data*), y las deposita en *imagenes_a_testear*. Para que este script ande se requiere la utilidad *convert* que se encuentra en la biblioteca *imagemagick*, para instalar *sudo apt-get install imagemagick*.

El archivo *test_dif_cat.sh* verifica que los resultados de la catedra den igual que la implementación de C. *test_dif_c_asm.sh* verifica que los resultados de las versiones de C y Assembler sean iguales. *test_mem.sh* chequea que no haya problemas en el uso de la memoria. Finalmente, *test_all.sh* corre todos los checks anteriores uno después del otro.

3.4.2. Mediciones de tiempo

Utilizando la instrucción de assembly `rdtsc` podemos obtener el valor del Time Stamp Counter (TSC) del procesador. Este registro se incrementa en uno con cada ciclo del procesador. Restando el valor del registro antes de llamar a una función a su valor luego de la llamada, podemos obtener la duración en ciclos de esa ejecución.

Las macros para medir tiempo se encuentran en el archivo `tiempo.h`. Para usarlas, se debe determinar como y donde implementarlas para poder obtener las mediciones más exactas de tiempo con granularidad de ciclo de clock.

Esta cantidad de ciclos no es siempre igual entre invocaciones de la función, ya que este registro es global del procesador y, si el programa es interrumpido por el *scheduler* para realizar un cambio de contexto, contaremos muchos más ciclos que si la función se ejecutara sin interrupciones. Por esta razón el programa principal del TP permite especificar una cantidad de iteraciones para repetir el filtro, con el objetivo de suavizar este tipo de *outliers*.

3.5. Informe

El informe debe incluir las siguientes secciones:

a) **Carátula** Contiene

- número / nombre del grupo
- nombre y apellido de cada integrante
- número de libreta y mail de cada integrante

b) **Introducción** Describe lo realizado en el trabajo práctico.

c) **Desarrollo** Describe cada una de las funciones que implementaron, respondiendo *en profundidad* cada una de las preguntas de los *Experimentos*.

Para la descripción de cada función deberán decir cómo opera una iteración del ciclo de la función. Es decir, cómo mueven los datos a los registros, cómo los reordenan para procesarlos, las operaciones que se aplican a los datos, etc. Para esto pueden utilizar pseudocódigo, diagramas (mostrando gráficamente el contenido de los registros `XMM`) o cualquier otro recurso que le sea útil para describir la adaptación del algoritmo al procesamiento simultáneo SIMD. No se deberá incluir el código assembler de las funciones (aunque se pueden incluir extractos en donde haga falta).

Las preguntas en cada ejercicio son una guía para la confección de los resultados obtenidos. Al responder estas preguntas, se deberán analizar y comparar las implementaciones de cada funciones en su versión `C` y `ASM`, mostrando los resultados obtenidos a través de tablas y gráficos.

También se deberá comentar acerca de los resultados obtenidos. En el caso de que sucediera que la versión en C anduviese más rápidamente que su versión ASM, **justificar fuertemente** a qué se debe esto.

d) **Conclusión** Reflexión final sobre los alcances del trabajo práctico, la programación en el modelo **SIMD** a bajo nivel, problemáticas encontradas, y todo lo que consideren pertinente.

Importante: El informe se evalúa de manera independiente del código. Puede reprobarse el informe, y en tal caso deberá ser reentregado para aprobar el trabajo práctico.

4. Entrega y condiciones de aprobación

El presente trabajo es de carácter **grupal**, siendo los grupos de **3 personas**, pudiendo ser de 2 personas en casos excepcionales previa consulta y confirmación del cuerpo docente. Se deberá entregar un archivo comprimido con el mismo contenido que el dado para realizarlo, habiendo modificado solo los archivos que tienen como nombre las funciones a implementar.

Es codicion necesaria para la aprobación de este trabajo que pasen correctamente todos los tests de la catedra.

La fecha de entrega última de este trabajo es **Martes 30 de Septiembre** y deberá ser entregado a través de la página web. El sistema solo aceptará entregas de trabajos hasta las **17:00hs** del día de entrega.

Ante cualquier problema con la entrega, comunicarse por mail a la **lista de docentes**.