Introducción a la computación

Trabajo Práctico

Fecha de entrega: Miércoles 28 de Noviembre de 2018.

1. Ejercicio 1: Divide & Conquer

Se tiene un archivo de texto con un par de números (enteros o de punto flotante) por línea, que representan puntos en el plano 2D. Los números pueden ser positivos o negativos. Un ejemplo de entrada válida es la siguiente:

```
1.3 4.4
-1.0 5.3
0.0 2.3
-0.5 5.5
5 5.5
7.5 10.8
```

Se necesita, a partir de un archivo en este formato, obtener el par de puntos diferentes cuya distancia euclídea sea mínima. La distancia euclídea entre los pares de puntos (x_0, y_0) y (x_1, y_1) se define de la siguiente manera:

$$\text{distancia}((x_0, y_0), (x_1, y_1)) \equiv \|(x_1 - x_0, y_1 - y_0)\| = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

Se pide:

1. Implemente una función `listaDePuntos(fn)` que tome como entrada un archivo, cuyo nombre está indicado por `fn`, con una lista de puntos en el formato especificado y devuelva una lista de *tuplas* con los valores de cada punto del archivo.

   Ayuda 1: una tupla con valores `x` e `y` se define en `Python` como `(x, y)`.

2. Implemente una función `distanciaMinima(a)` que tome como entrada una lista como la definida en el punto anterior y devuelva cuál es el par cuya distancia es mínima. En el caso del ejemplo, el par de distancia mínima es el $(-1{,}0; 5{,}3), (-0{,}5; 5{,}5)$, cuya distancia es $0{,}5385$. La estrategia a utilizar es la de *fuerza bruta*, es decir calcular las distancias entre todos los pares de puntos (resulta en un orden de complejidad de $O(n^2)$ en la cantidad de puntos).

3. El algoritmo anterior puede ser mejorado en complejidad a $O(n \log^2 n)$ si se utiliza la técnica de *divide & conquer*. Diseñe e implemente una función `distanciaMinimaDyC(a, algoritmo)` siguiendo esta técnica, teniendo en cuenta el esquema descripto a continuación:

a) Preprocesamiento: la lista de puntos debe encontrarse ordenada en su coordenada x. Implemente *merge sort* y *upsort*, y decida qué algoritmo utilizar en función del valor de la cadena de caracteres indicada por el parámetro `algoritmo`.

Los valores posibles son los siguientes:

- `merge`: utiliza *merge sort*.
- `up`: utiliza *upsort*.
- `python`: utiliza el sort de python [1].

b) Dividir los puntos en dos mitades cortando sobre el eje x.

c) Buscar recursivamente las distancias menores en ambas mitades, y quedarse con la menor de ambas.

d) Combinar dicho resultado con las distancias entre pares de puntos que se encuentren cada uno en una mitad diferente.

Ayuda 2: puede utilizar la menor de las distancias encontradas en los pasos recursivos como cota para determinar cuánto debe alejarse hacia cada lado en la coordenada x de la recta que corta ambas mitades, ya que la lista de puntos se encuentra ordenada en x[2].

4. Genere un gráfico en el cual se analice el tiempo de ejecución variando el tamaño del conjunto de puntos de entrada y midiendo los dos algoritmos implementados, incluyendo una curva por cada algoritmo de ordenamiento en el caso del algoritmo de *divide & conquer*. Saque conclusiones con respecto a la performance del algoritmo de *divide & conquer* con los distintos algoritmos de ordenamiento utilizados.

   Ayuda 3: Para tomar tiempos se puede usar `time.time()` (del módulo `time`) y para generar puntos de entrada se puede usar `random.random()` (del módulo `random`) que genera números aleatorios entre 0 y 1 (ajustar al rango de valores que necesite por medio de operaciones aritméticas).

2. Ejercicio 2: Árbol binario

Definir una clase en `Python` que implemente el TAD `ArbolBinario` (un árbol que tiene una raíz, una rama derecha y una izquierda). Debe tener los siguientes métodos:

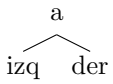
1. `__init__(self)`: devuelve un árbol vacío.
2. `vacio(self)`: indica si el árbol no posee ningún elemento.

[1] Python tiene un metodo *sort* para listas. Por ejemplo una lista l se puede ordenar usando *l.sort()*.

[2] Puede pensar que las soluciones posibles se categorizan en tres grupos: 1) los pares de puntos que se encuentran en el lado izquierdo, 2) los pares de puntos que se encuentran en el lado derecho y 3) los pares de puntos tales que van de un punto del lado izquierdo a un punto del lado derecho. El paso *combine* involucra encontrar potenciales soluciones que se encuentren en la categoría 3. La restricción de distancia se da por una propiedad matemática de las distancias euclídeas, ya que es imposible encontrar un par de puntos en esa categoría que se encuentre más cerca que los pares de puntos en las categorías 1 y 2 si alguno de los dos puntos que constituyen el par se encuentra más alejado de la mitad que la menor de las distancias mínimas encontradas recursivamente.

3. `raiz(self)`: devuelve el elemento correspondiente a la raíz del árbol (¿Qué pasa si el árbol está vacío?).

4. `bin(self, a, izq, der)`: modifica el parámetro implícito con el árbol resultado de poner al número entero `a` como raíz, `izq` como rama izquierda y `der` como derecha:



5. `izquierda(self)`: devuelve un árbol correspondiente a la rama izquierda del parámetro implícito.

6. `derecha(self)`: devuelve un árbol correspondiente a la rama derecha del parámetro implícito.

7. `find(self, a)`: indica si el elemento `a` está en el árbol.

8. `espejo(self)`: devuelve un nuevo árbol que resulta de espejar el parámetro implícito, es decir, que todo elemento que se encuentre a la izquierda de un nodo pasa a estar a la derecha y viceversa.

9. `preorder(self)`: retorna una lista de enteros que resulta de colocar la raíz, seguido del preorder de las ramas izquierda y derecha.

10. `posorder(self)`: retorna una lista de enteros que resulta de colocar los posorder de las ramas derecha e izquierda y luego la raíz.

11. `inorder(self)`: retorna una lista de enteros que resulta ser el recorrido de los elementos del parámetro implícito de izquierda a derecha como si se lo aplastara (la raíz queda entre las ramas izquierda y derecha).

Se debe implementar la clase y mostrar algunos casos de prueba para las funciones de la 4 a la 11.

3. Ejercicio 3: Utilizando C++ desde Python

Una imagen digital en mapa de bits o imagen ráster es un archivo compuesto esencialmente por una matriz bidimensional donde cada elemento es un punto de color llamado píxel. Cada píxel está codificado por uno o más valores numéricos. En las imágenes en escala de grises cada píxel está representado mediante un byte, por tanto cada punto puede albergar una de 256 posibles intensidades de gris. En las imágenes en color RGB, cada píxel se compone de tres valores numéricos, cada uno de un byte y representando la intensidad de un color base: rojo (Red), verde (Green) y azul (Blue). Las imágenes se despliegan en el monitor de la computadora también a través de píxeles, si amplificamos un sector de nuestra pantalla (podemos hacerlo mediante una lupa) se puede apreciar cada píxel como la composición de los tres colores nombrados.

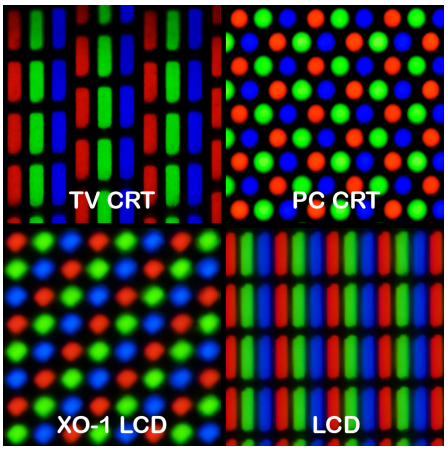


Figura 1: Visualización de la composición de los colores en diferentes tipos de pantallas (fuente: `https://en.wikipedia.org/wiki/Pixel_geometry`)

Para almacenar imágenes digitales en disco se utilizan distintos formatos gráficos, de los cuales el más conocido es el JPEG. Cada formato tiene diferentes características que indican cómo y dónde se debe guardar cada píxel de la imagen en un archivo. Estos formatos son complicados de interpretar, pero al ser problemas muy comunes existen bibliotecas en cada lenguaje para solucionarlos.

El procesamiento de imágenes digitales es un problema computacionalmente costoso para el que implementar su solución completamente en `Python` puede resultar ineficiente. Para no perder tanta eficiencia, pero de todas formas aprovechar las características que tiene `Python`, el lenguaje ofrece mecanismos para invocar código `C` ó `C++`.

Esto funciona mediante la generación de un código intermedio (llamado *wrapper*) en `C++` que se encarga de convertir los parámetros recibidos desde `Python` en algo que entienda `C++` y viceversa. Dicho procedimiento es complicado de hacer manualmente, aunque existen múltiples bibliotecas para facilitar la generación de los mismos. En el caso de este ejercicio, el código intermedio ya está resuelto sin utilizar librerías adicionales.

En nuestro problema trabajaremos leyendo las imágenes a través de las bibliotecas `PIL` (Python Image Library) que se incluyen en `SciPy` y resolveremos la implementación de dos filtros gráficos en `Python` *puro* y en `C++`. La matriz bidimensional que representa a la imagen estará implementada sobre matrices de `NumPy` en el caso de `Python` y en arreglos unidimensionales en el caso de `C++`[3]. En el caso de las imágenes RGB, cada elemento de la lista contiene los tres valores numéricos correspondientes.

Utilizaremos como base el código que se encuentra en el directorio `src_ej3` de los archivos adicionales del TP (ver apéndice A).

1. Implementar una función en `Python gray_filter(img)` que tome una imagen en colores `img` y devuelva una nueva imagen con la misma cantidad de filas y columnas pero con una sola componente de color, con el resultado de aplicar la siguiente transformación al espacio de color YUV (quedándose

[3]El tipo de datos en ambos casos es una matriz de `NumPy`, pero en el caso de `C++` accederemos a los datos *"directamente"* como si fuera un arreglo unidimensional.

únicamente con la componente de luma Y):

$$res_{i,j} = 0{,}3 \cdot img_{i,j}.\mathbf{r} + 0{,}6 \cdot img_{i,j}.\mathbf{g} + 0{,}11 \cdot img_{i,j}.\mathbf{b}$$

Ayuda 4: Para acceder a cada componente de color **r**, **g** o **b**, utilizar el índice de la tercera dimensión de la matriz `img` ($0 = \mathbf{r}$, $1 = \mathbf{g}$, $2 = \mathbf{b}$). Por ejemplo, para acceder al valor de la componente verde del pixel $(2, 2)$, utilizar `img[2,2,1]`.

2. Implementar una función en `Python` `blur_filter(img)` que tome una imagen en escala de grises `img` y devuelva una nueva imagen del mismo tamaño con el resultado de aplicar un filtro de difuminado sobre la imagen original. Dicho filtro se corresponde a aplicar la siguiente matriz de convolución de 3×3 a cada elemento de `img`:

$$\begin{pmatrix} 0 & 0{,}25 & 0 \\ 0{,}25 & 0 & 0{,}25 \\ 0 & 0{,}25 & 0 \end{pmatrix}$$

   El cálculo anterior equivale a calcular la siguiente expresión para cada elemento i, j tal que $0 < i < filas(img) - 1 \wedge 0 < j < columnas(img) - 1$:

$$res_{i,j} = \frac{img_{i-1,j} + img_{i+1,j} + img_{i,j-1} + img_{i,j+1}}{4}$$

   Por simplicidad, los elementos no alcanzados por la condición anterior deben estar en negro (valor 0).

3. Implementar una función en `C++` con el prototipo:
   `void blur_filter(float *im_res, float *im, int ii, int jj)`,
   que realice la misma operación que la función homónima en `Python`.

   Los argumentos de esta versión son:

   im_res Imagen resultante (escala de grises).

   im Imagen origen (escala de grises).

   ii Cantidad de filas de `im` y de `im_res`.

   jj Cantidad de columnas de `im` y de `im_res`.

4. Implementar una función en `C++` con el prototipo:
   `void gray_filter(float *im_res, pixel_t *im, int ii, int jj)`,
   que realice la misma operación que la función homónima en `Python`.

   Los argumentos de esta versión son:

   im_res Imagen resultante (escala de grises).

   im Imagen origen (color, en 24 bits RGB).

   ii Cantidad de filas de `im` y de `im_res`.

   jj Cantidad de columnas de `im` y de `im_res`.

Ayuda 5: Para acceder a cada componente de color de `im`, utilizar la sintaxis `im[offset].r`, `im[offset].g` ó `im[offset].b`, donde `offset` es el desplazamiento necesario con respecto al comienzo de la imagen para acceder al pixel deseado (la imagen se encuentra almacenada en memoria en filas consecutivas).

5. Modificar ambas funciones en `C++` para que utilicen `OpenMP` para que los ciclos implementados se ejecuten en paralelo[4].

6. Realizar el siguiente experimento desde el código en `Python`:

   a) Elegir una imagen y cargarla[5].

   b) Medir el tiempo que demora convertir la imagen cargada a escala de grises, y luego aplicar el filtro de difuminado, utilizando la implementación en `Python`.

   c) Medir el tiempo que demora realizar la misma operación indicada por el ítem anterior, pero invocando a su implementación en `C++` a través del módulo `imfilters` provisto por la cátedra.

   d) Almacenar en disco ambas imágenes difuminadas[6]. Verificar que el resultado obtenido coincida.

7. Comparar los resultados obtenidos en el experimento anterior en ambas implementaciones, repitiéndolo para imágenes de distintos tamaños.

4. Ejercicio 4: Backtracking

Se tiene un laberinto de tamaño $N \times M$ (N filas por M columnas) y se desea simular el comportamiento que tendría una rata al intentar recorrerlo, hasta alcanzar un trozo de queso. Originalmente existía un programa completo que lo modelaba, con una interfaz gráfica que nos permitía visualizar la simulación, pero se perdió la implementación del TAD Laberinto en una actualización fallida de Ubuntu, por lo que es necesario reconstruirlo.

Después de arduas horas de analizar el código que sobrevivió, se logró reconstruir el esqueleto del TAD, pero sigue faltando el código que permite que todo funcione. Su objetivo será encargarse de esta tarea.

Para que todo vuelva a funcionar, se pide completar los siguientes métodos de la clase Laberinto (en el archivo `laberinto.py`) para que cumplan con la funcionalidad requerida en cada caso.

Utilizaremos como base el código que se encuentra en el directorio `src_ej4` de los archivos adicionales del TP (ver apéndice B).

1. `cargar(self, fn)`: abre, lee, procesa el archivo de texto cuyo nombre se indica en la variable `fn`, cargando el laberinto allí descripto. El archivo debe respetar el formato indicado en el apéndice C. Al finalizar la carga, el laberinto debe encontrarse reseteado (ver método `resetear(self)`), y debe mover la rata a la posición $(0, 0)$ (borde superior izquierdo) y el queso a la posición $(N - 1, M - 1)$ (borde inferior derecho).

[4] Se explicará en clase.
[5] Utilizar `matriz = misc.imread(nombre)`.
[6] Utilizar `misc.imsave(nombre, matriz)`.

2. `tamano(self)`: devuelve una tupla de dos elementos, donde el primero representa la cantidad de filas y el segundo es la cantidad de columnas del laberinto.

3. `resetear(self)`: limpia el laberinto, desmarcando las posiciones que aparecen como visitadas o como parte de un camino.

4. `getPosicionRata(self)`: devuelve una tupla de dos elementos, donde el primero indica la fila y el segundo la columna donde se encuentra la rata.

5. `getPosicionQueso(self)`: devuelve una tupla de dos elementos, donde el primero indica la fila y el segundo la columna donde se encuentra el queso.

6. `setPosicionRata(self, i, j)`: cambia la posición de la rata a la fila i, columna j. Devuelve `True` si el cambio fue posible (es decir, si la posición indicada se encontraba dentro de los límites del laberinto), `False` en otro caso.

7. `setPosicionQueso(self, i, j)`: cambia la posición del queso a la fila i, columna j. Devuelve `True` si el cambio fue posible (es decir, si la posición indicada se encontraba dentro de los límites del laberinto), `False` en otro caso.

8. `esPosicionRata(self, i, j)`: devuelve `True` si el par ordenado determinado por (i, j) corresponde a la posición de la rata, `False` en caso contrario.

9. `esPosicionQueso(self, i, j)`: devuelve `True` si el par ordenado determinado por (i, j) corresponde a la posición del queso, `False` en caso contrario.

10. `get(self, i, j)`: devuelve una lista con 4 elementos *booleanos*, que indican si hay pared o no en cada uno de los 4 bordes de la celda ubicada en la fila i, columna j. El orden en el que deben aparecer los bordes es el siguiente: *Izquierda*, *Arriba*, *Derecha*, *Abajo*. Por ejemplo, si la celda pedida tiene paredes a su derecha y arriba, este método debe devolver `[False, True, True, False]`.

11. `getInfoCelda(self, i, j)`: devuelve un diccionario con información acerca de la celda ubicada en la posición (i, j). El diccionario debe tener dos claves: `visitada` y `caminoActual`. La clave `visitada` es un *booleano* que indica si la celda fue visitada por el backtracking, mientras que `caminoActual` indica si es parte del camino actual que está siendo probado por el backtracking. Por ejemplo, si la celda ya fue visitada por el backtracking y no es parte del camino actual, debe devolver `{'visitada': True, 'caminoActual': False}`.

12. `resuelto(self)`: devuelve `True` si el laberinto está resuelto, es decir, si la posición de la rata es la misma que la del queso, y `False` en otro caso.

13. `resolver(self)`: resuelve el laberinto utilizando *backtracking*, comenzando por la posición en la que se encuentra la rata. Durante el proceso de resolución debe actualizar el estado de cada celda recorrida, de forma que

quede marcado el camino actual, y en caso de volver hacia atrás por quedarse sin movimientos posibles, las celdas por las que vuelve deben ser marcadas como visitadas (y desmarcadas como parte del camino actual). Devuelve `True` si llega a la posición del queso, `False` si esto no es posible (si no existe un camino).

Importante: Para ver actualizaciones en pantalla durante el backtracking, cada vez que cambie la posición de la rata o el estado de una celda es necesario llamar al método `self._redibujar()`.

Algunas notas adicionales que pueden ser útiles:

- Se recomienda programar los métodos pedidos en el orden en el que aparecen en el enunciado, de forma de poder ir probando de manera parcial la funcionalidad implementada.
- Para hacer pruebas se puede utilizar el modo interactivo del intérprete sobre el código de la clase laberinto (ejecutando `python3 -i laberinto.py`).
- Como guía para elegir la forma de estructurar los datos dentro del TAD, pensar en términos de hacer lo más sencillo posible devolver lo pedido en cada método.
- Ustedes deben elegir las variables para almacenar el estado del laberinto, que deberán ser marcadas como *privadas*, de manera de no utilizarse fuera de la clase.
- Pueden agregar métodos privados, pero no modificar la interface pública de la clase.

A. Carpeta `src_ej3` de archivos adicionales

Dentro de la carpeta `src_ej3` se encuentran los siguientes archivos:

- `ej3.py`: esqueleto del código completo en `Python`.
- `imfilters.cpp`: código intermedio para llamar a `C++` desde `Python`.
- `gray_filter.cpp`: esqueleto del código del filtro de escala de grises en `C++`.
- `blur_filter.cpp`: esqueleto del código del filtro de difuminado en `C++`.
- `filters.h`: *include* con los prototipos de las funciones y la estructura para almacenar un píxel.

Desde la terminal, ejecutar el comando `./setup.py build_ext --inplace` para compilar el módulo `imfilters`. Luego, se puede utilizar `import imfilters` desde `Python` y ejecutar las dos funciones implementadas en C++.

B. Cómo utilizar la interfaz gráfica del Ej. 4

La interfaz gráfica provista en la carpeta `src_ej4` del ZIP de archivos adicionales nos permite cargar, mostrar en pantalla y resolver laberintos, así como también visualizar el mecanismo de backtracking utilizado en la resolución. Para ejecutarla, en una terminal posicionada adecuadamente, ejecutar el siguiente comando: `python3 tplaberinto.py` (o bien `./tplaberinto.py`).

Esto inicializará el programa, mostrando una ventana con un recuadro negro en el lado izquierdo y varios botones y controles en el lado derecho. En el lado izquierdo se visualizará el laberinto una vez cargado.

Los botones tienen la siguiente funcionalidad:

- **Cargar**: abre un cuadro de diálogo que permite elegir un archivo con un laberinto a cargar.
- **Resolver**: resuelve el laberinto, llamando al método `resolver` de la clase Laberinto.
- **Resetear**: resetea el laberinto, llamando al método `resetear` de la clase Laberinto.
- **Salir**: sale del programa.

Los spinboxes *Posición inicial* y *Posición final* modifican la posición de la rata y del queso respectivamente. Si alguno de los dos botones **Modificar** se encuentra presionado, se activan los spinboxes correspondientes y además es posible mover la rata o el queso por el laberinto utilizando las flechas del teclado.

Finalmente, el slider *Velocidad de la animación* permite variar la velocidad con la que se redibuja el laberinto cada paso del backtracking. A efectos prácticos, se ve afectada la espera entre cada llamado a `self._redibujar()` y el siguiente paso.

C. Formato del laberinto

Un laberinto se representa por medio de un archivo de texto que tiene el siguiente formato:

1. La primer línea indica el tamaño del laberinto con la sintaxis:

   `Dim(`N`,`M`)`, donde N y M son números que indican la cantidad de filas y la cantidad de columnas respectivamente. Por ejemplo: `Dim(10,20)` indica un laberinto de 10 filas por 20 columnas.

2. Cada una de las líneas restantes indica una fila, en orden.

3. El formato de cada fila es el siguiente:

$$\overbrace{[\mathtt{0}_{\mathtt{left}}, \mathtt{0}_{\mathtt{up}}, \mathtt{0}_{\mathtt{right}}, \mathtt{0}_{\mathtt{down}}] \cdots\cdots [\mathtt{M}-\mathtt{1}_{\mathtt{left}}, \mathtt{M}-\mathtt{1}_{\mathtt{up}}, \mathtt{M}-\mathtt{1}_{\mathtt{right}}, \mathtt{M}-\mathtt{1}_{\mathtt{down}}]}^{\text{M veces}}$$

   Cada uno de los bloques entre corchetes indica la descripción de la celda ubicada en la columna i-ésima de la fila a la que corresponde esa línea del laberinto. A su vez, cada uno de esos bloques tiene 4 elementos separados por comas, sin espacios, que pueden tener dos valores posibles, 1 o 0, según la presencia o ausencia de una pared en cada dirección (izquierda, arriba, derecha y abajo, que aparecen en ese orden). Por ejemplo, si una celda tiene paredes arriba y a la derecha, su descripción en el archivo es la siguiente: `[0,1,1,0]`.

Junto al código fuente de la GUI van encontrar varios ejemplos en este formato, de extensión `*.lab`, listos para probar con su implementación.

Condiciones de entrega:

- Entregar una versión impresa **y** un archivo comprimido ZIP que contenga los archivos fuentes correspondientes a cada ejercicio y un informe (en formato PDF).
- La versión impresa debe incluir tanto el código como el informe (no incluir en la impresión el código ya provisto para la interfaz *Python–C++* o la interfaz gráfica del laberinto).
- El informe debe incluir:
  - **Ej. 1**: decisiones de diseño y cómo correr las pruebas necesarias para generar el gráfico incluido en el informe.
  - **Ej. 2**: decisiones de diseño, ejemplos que muestren funcionalidad y cómo ejecutarlos.
  - **Ej. 3**: resultados de los experimentos, qué pruebas realizaron y cómo se corren.
  - **Ej. 4**: decisiones de diseño (atributos que hayan definido en la clase `Laberinto`, cómo se guardan los datos que cargan, estrategia del backtracking, etc.)
  - Cualquier otra aclaración que consideren pertinente.
- Los archivos fuentes deberán tener comentarios.
- La entrega electrónica se realiza vía Bitbucket.
- **Nota**: se puede tomar coloquio individual a criterio de los docentes sobre cualquier tema del TP.
- Una vez terminado, deben enviar un mail de aviso a la dirección de correo electrónico de los docentes de la materia: **icb-doc@dc.uba.ar**. Deberán poner como *subject*:
  [Grupo ⟨NN⟩: ⟨Nombre grupo⟩] TP - ⟨Apellido1 LU1⟩ - ⟨Apellido2 LU2⟩. Por ejemplo, un grupo podría ser:
  [Grupo 99: "Pajarones"] TP - Gómez 334/89 - González 671/19
- El cuerpo del mail de aviso debe indicar el comando que tiene que ejecutar el docente para clonar el repositorio donde se encuentra el TP.