

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC1103 – Introducción a la Programación
2 - 2017

Examen

Instrucciones generales:

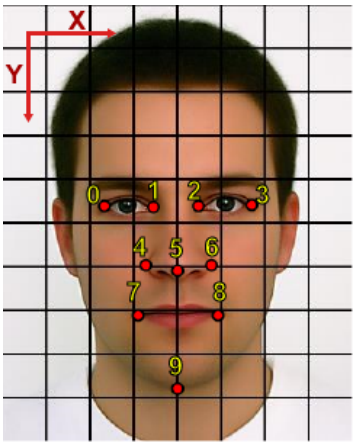
- El examen consta de 4 preguntas con igual ponderación.
- Solo se reciben consultas sobre el enunciado en los primeros 30 minutos.
- Todas las preguntas están pensadas para ser resueltas en forma sencilla con lo visto en clases. No es necesario utilizar otros mecanismos más avanzados.
- Si el enunciado de la pregunta no lo indica explícitamente, no es necesario validar los ingresos del usuario. Puedes suponer que no hay errores por ingreso de datos.
- Responde cada pregunta en una hoja separada, marcando tu número de alumno y el número de pregunta en todas las hojas.
- Puedes utilizar cualquier cosa que te hayan pedido programar en un ítem anterior dentro de las preguntas (incluso si no respondiste ese ítem).
- Tienen 4 horas para resolver la prueba desde que se entregan los enunciados.

Pregunta 1

Dado el incremento de robos en Chile, decides crear tu propio emprendimiento biométrico de Sistemas de Reconocimiento Facial. Tu idea es que una cámara analice la posición de 10 puntos de la cara de una persona (posición de los ojos, la nariz, ...), y lo compare con los puntos de las personas previamente autorizadas. Debes diseñar tu sistema usando Programación Orientada a Objetos.

Cada punto se define como una lista $[x, y]$ donde x representa la posición en el eje horizontal y donde y representa la posición en el eje vertical.

A continuación se muestra un ejemplo con los puntos enumerados del 0 al 9:



Donde los puntos según el orden anterior son:

```
puntos = [ [2.3, 4.6], [3.5, 4.6], [4.4, 4.5], [5.8, 4.5], [3.2, 6],
           [4, 6.1], [4.9, 6], [3.1, 7.1], [4.9, 7.1], [4, 9.8] ]
```

Para cumplir el objetivo debes definir las siguientes clases y métodos:

Cara (4 puntos)

- *__init__*: Inicializa el objeto. Tiene como atributos el nombre de la persona y *puntos_cara* que es una lista de puntos que representan su cara.
- *distancia_punto*: Recibe como entrada un índice i que va entre 0 y 9; y un punto p. Este método retorna la distancia euclidiana entre el $i - esimo$ punto de *puntos_cara* y el punto p . Recuerda que la distancia euclidiana entre dos puntos $[x_0, y_0]$ y $[x_1, y_1]$ se calcula como: $\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$.
- *coinciden*: Recibe como entrada una lista de puntos y compara si es que ésta se asemeja con la lista *puntos_cara* guardada al momento de inicializar el objeto. Para definir la similitud entre dos listas l_0, l_1 se debe hacer lo siguiente:
  - Se suman todas las distancias euclidianas entre los puntos del mismo índice de l_0 y l_1. Es decir, se suma la distancia entre $l_0[0]$ y $l_1[0]$. Luego de $l_0[1]$ y $l_1[1]$. Hasta finalmente llegar a $l_0[9]$ y $l_1[9]$
  - Se resta la distancia más grande del cálculo anterior, ya que Se considera un caso aislado y se debería ignorar. Por ejemplo si tenemos los siguientes puntos junto a sus distancias:

l_0	[2.3, 4.6]	[3.5, 4.6]	[4.4, 4.5]	[5.8, 4.5]	[3.2, 6]	[4, 6.1]	[4.9, 6]	[3.1, 7.1]	[4.9, 7.1]	[4, 9.8]
l_1	[2.7, 4.8]	[3.4, 4.8]	[4.3, 5]	[5.5, 4.9]	[3.3, 6.5]	[3.9, 6.7]	[4.6, 6.5]	[3.1, 7.5]	[4.9, 7.5]	[4, 9.5]
dist	0.45	0.22	0.51	0.5	0.51	0.61	0.58	0.4	0.4	0.3

    La suma sería 0.45 + 0.22 + ... + 0.3 = 4.48, luego se resta 0.61 que es la distancia más grande entre los pares de puntos. Y, por lo tanto, el cálculo final es 4.48 - 0.61 = 3.87

  Si el cálculo anterior fue menor o igual a 2, entonces el método *coinciden* debe retornar *True*. En caso contrario debe retornar *False*. En el ejemplo anterior se debería retornar *False*.

Sistema (2 puntos)

- *__init__*: Inicializa el objeto. No recibe parámetros de entrada, pero debe inicializar una lista que guarde las caras que van a ser registradas.
- *registrar*: Recibe como entrada un objeto de la clase *Cara* y lo agrega a la lista de caras registradas.
- *nombres_registrados*: No recibe parámetros de entrada. Entrega todos los nombres de las personas registradas como una lista de strings.
- *validar*: Recibe una lista de puntos como entrada. Retorna *True* si es que esta lista de puntos coincide (según el método *coinciden* de la clase *Cara*, antes descrito) con algunas de las caras registradas en el sistema. En caso contrario, retorna *False*.

Las listas de puntos siempre serán de tamaño 10. El código debe ser compatible con el siguiente programa:

```
puntos0 = [ [2.3, 4.6], [3.5, 4.6], [4.4, 4.5], [5.8, 4.5], [3.2, 6],
            [4, 6.1], [4.9, 6], [3.1, 7.1], [4.9, 7.1], [4, 9.8] ]
puntos1 = [ [2.7, 4.8], [3.4, 4.8], [4.3, 5], [5.5, 4.9], [3.3,6.5],
            [3.9, 6.7], [4.6, 6.5], [3.1, 7.5], [4.9, 7.5], [4, 9.5] ]
pedro = Cara("Pedro", puntos0)
sistema = Sistema()
sistema.registrar(pedro)
print(sistema.nombres_registrados())
print(sistema.validar(puntos1))
print(sistema.validar(puntos0))
```

Donde el resultado de la ejecución sería:

```
["Pedro"]
False
True
```

Pregunta 2

La gran compañía *Abobe* te ha contactado para ayudarlos con la creación de su próxima aplicación para manejo de imágenes: *Introshop*.

Las imágenes se representan como una lista de listas rectangular de *pixeles*, donde cada pixel es una lista de 3 colores: rojo, verde y azul (Esto se traduce en inglés como RGB = R: red, G: green, B: blue). Cada componente del pixel toma un valor entero de 0 a 255 que indica la intensidad de cada uno de los colores. Un ejemplo de imagen es el siguiente:

```
imagen = [
    [[58,23,125], [150,255,0], [0,0,0]   , [14,0,99]    ],
    [[0,0,200]  , [34,25,0]  , [0,65,0]  , [8,0,45]     ],
    [[58,0,1]   , [70,25,0]  , [0,80,165], [255,255,255]]
]
```

Tu trabajo es escribir las siguientes funciones que reciben una imagen con el formato anterior, y retornen una **nueva** imagen luego de haber aplicado ciertas operaciones sobre la imagen original. Ten en considereación que **no debes modificar** la imagen original.

invertir_imagen (3 puntos)

Dada una imagen, su inversa se calcula de la siguiente manera: para cada pixel de la imagen original, la imagen inversa será 255 - *valor_de_cada_color*. Usando la imagen de ejemplo anterior, su inversa sería:

```
>>> inversa = invertir_imagen(imagen)
inversa = [
    [[197, 232, 130], [105, 0, 255]  , [255, 255, 255], [241, 255, 156]],
    [[255, 255, 55] , [221, 230, 255], [255, 190, 255], [247, 255, 210]],
    [[197, 255, 254], [185, 230, 255], [255, 175, 90] , [0, 0, 0]      ]
]
```

rotar_derecha (3 puntos)

Esta función debe hacer girar la imagen en 90 grados hacia la derecha: es decir, la última fila de la imagen original queda como la primera columna de la nueva imagen, la penúltima fila queda como la segunda columna, hasta llegar a la primera fila que queda como la última columna. Usando la imagen de ejemplo, el resultado de la función sería:

```
>>> rotada = rotar_derecha(imagen)
rotada = [
    [[58, 0, 1]     , [0, 0, 200], [58, 23, 125]],
    [[70, 25, 0]    , [34, 25, 0], [150, 255, 0]],
    [[0, 80, 165]   , [0, 65, 0] , [0, 0, 0]    ],
    [[255, 255, 255], [8, 0, 45] , [14, 0, 99]  ]
]
```

Nota: Esta función debe rotar la imagen, no trasponerla.

Pregunta 3

¡Nunca más `flojeda@uc.cl`, `elbalazo@uc.cl`!, hagamos un programa para emails dignos en la UC. Tu trabajo es hacer un programa que lea los archivos `emails_creados.txt` que en cada línea contiene un email, y `palabras_feas.txt`, que en cada línea contiene una palabra no deseada. Tu programa debe generar el archivo `emails_rechazados.txt` con todos los emails que contengan **dentro de su username** un *substring* con al menos los primeros 4 caracteres de una palabra fea. Ten en cuenta que el *username* de un email es el *substring* hasta antes del caracter `@`. Puedes asumir que todas las palabras feas tienen 4 caracteres o más.

Por ejemplo, si los archivos son:

```
emails_creados.txt
------------------
flojeda@uc.cl
marcos.sepulveda@uc.cl
idioteiza@uc.cl
elbalazo@uc.cl
vherskovic@uc.cl
mun@idiota.com
```

```
palabras_feas.txt
------------------
flojo
bala
idiota
```

Entonces, el archivo de salida debería ser:

```
emails_rechazados.txt
------------------
flojeda@uc.cl
idioteiza@uc.cl
elbalazo@uc.cl
```

Pregunta 4

Como muchos otros estudiantes de la UC, estás involucrado en los trabajos sociales de verano del centro de estudiantes de tu facultad. Dados tus conocimientos de programación, te piden que los ayudes a asignar las labores a los diferentes grupos de trabajo.

Los estudiantes están divididos en tres grupos de trabajo: Grupo 0, 1, y 2. Además, existen n trabajos que deben ser realizados. Por otra parte, cuentas con la siguiente información:

- $horas_disponibles = [h_0, h_1, h_2]$ que es la lista de horas de trabajo disponibles por cada grupo.
- $horas_trabajos = [t_0, t_1, ..., t_{n-1}]$ que es una lista de las horas necesarias para realizar con éxito cada uno de los n trabajos.

Una asignación es válida si para cada grupo de trabajo, sus horas asignadas son menores o iguales a sus horas disponibles; y mayores o iguales al 50 % de sus horas disponibles (no tiene sentido tener un equipo haciendo menos de la mitad de lo que puede hacer).

En concreto, se te pide que implementes (**usando recursión**) la función *resolver(horas_disponibles, horas_trabajos)*, que **retorna** una lista de n enteros $[e_0, e_1, ..., e_{n-1}]$ con **UNA** asignación válida, donde e_i ($e_i \in \{0, 1, 2\}$) representa qué grupo tiene asignado el trabajo t_i, con $0 \leq i < n$. En caso de no existir ninguna solución, debe retornar una lista vacía. Puedes agregar parámetros a la función *resolver*, si lo necesitas. Además, puedes definir otras funciones auxiliares que implementen el *backtracking*.

A continuación, se muestra un ejemplo:

```
horas_disponibles = [9, 22, 15]
horas_trabajos    = [6, 3, 7, 3, 5, 7, 9, 2, 3]
```

Donde una solución posible es:

```
solucion_posible = [0, 2, 1, 1, 1, 1, 2, 0, 2]
```

Esta solución es correcta ya que:

- El Grupo 0, tiene asignadas $6 + 2 = 8$ horas, lo cual es menor o igual a 9, y mayor o igual a 4.5.
- El Grupo 1, tiene asignadas $7 + 3 + 5 + 7 = 22$ horas, la totalidad de sus horas disponibles.
- El Grupo 2, tiene asignadas $3 + 9 + 3 = 15$ horas, la totalidad de sus horas disponibles.

Notar que la solución no es única. En este ejemplo, hay 133 soluciones posibles.

Nota: **NO SE DARÁ PUNTAJE POR SOLUCIONES ITERATIVAS**.