

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

IIC1103 – Introducción a la Programación
2018 - 2

Examen

Instrucciones generales:

- El examen consta de **4 preguntas con igual ponderación**.
- Solo se reciben **consultas** sobre el enunciado en los primeros 30 minutos.
- No está permitido utilizar **material de apoyo adicional** al recordatorio adjunto.
- 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**, anotando tu **número de alumno** y el **número de pregunta** en todas las hojas. Si usas más de una hoja en una pregunta, **enumera las hojas** de esa pregunta.
- Puedes utilizar cualquier cosa que te hayan pedido programar en un **ítem anterior** dentro de las preguntas (incluso si no respondiste ese ítem).
- Tienes **cuatro horas** para resolver la prueba desde que se entregan los enunciados.

Pregunta 1

El laboratorio del DCC está estudiando patrones en el movimiento de canarios dentro de una jaula, con el fin de analizar su comportamiento. La jaula cuenta con tres sensores en distintos lugares. El primero, en la parte superior de la jaula; el segundo, en el columpio; y el tercero, en la parte inferior. La Figura 1 muestra la ubicación de los sensores.

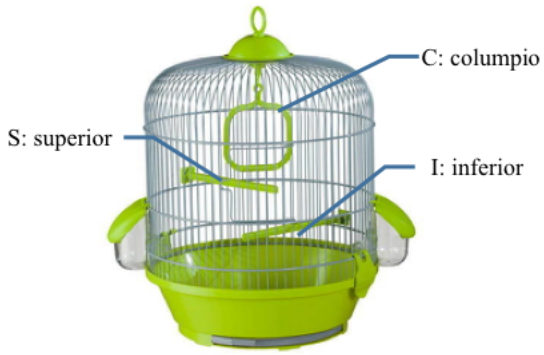


Figura 1: Jaula del canario, con sus respectivos sensores.

El registro de los movimientos se mantiene en un *string*, y cada vez que un sensor detecta al canario, se agrega un caracter a éste. El Cuadro 1 muestra el caracter que se agrega dependiendo del sensor.

sensor	caracter
Superior	`"S"`
Columpio	`"C"`
Inferior	`"I"`

Cuadro 1: Caracter por ubicación de sensor.

Un ejemplo de un registro es el siguiente: `s = "SCICISCICSICISCSICIISICICICSSISCISISICCISI"`. El laboratorio necesita ayuda procesando los registros, por lo que te ha pedido que programes las siguientes funciones:

(a) (23 puntos) `mas_frecuente(s, n)`: Recibe un *string* `s` con el registro del movimiento del canario y un *int* `n`. Retorna un *string* con el patrón más frecuente de largo `n` en el *string* `s`. Si hay dos patrones con la misma frecuencia, puedes retornar cualquiera de ellos. Puedes suponer que `n` es menor al largo de `s`.

(b) (7 puntos) `mas_frecuentes(s, inf, sup)`: Recibe un *string* `s` con el registro del movimiento del canario y dos *int* `inf` y `sup`. Retorna una *lista* con los patrones más frecuentes de largo `inf` a `sup` (uno de cada largo, incluyendo `sup`). Puedes suponer que `sup` es menor al largo de `s`.

(c) (23 puntos) `tres_veces_n(s, n)`: Recibe un *string* `s` con el registro del movimiento del canario y un *int* `n`. Retorna una *lista* con todos los patrones distintos de largo `n` en el *string* `s` que se repiten al menos tres veces consecutivas. En caso de no existir los patrones buscados, retorna una lista vacía. Notar que si el *string* es `s = "SCCCCCCCCI"` y `n = 3`, se retorna una lista vacía, puesto que el patrón `"CCC"` no alcanza a repetirse 3 veces consecutivas (faltaría una `"C"`).

(d) (7 puntos) `repetidos(s, inf, sup)`: Recibe un *string* `s` con el registro del movimiento del canario y dos *int* `inf` y `sup`. Retorna una *lista* de listas con los patrones de largo `inf` a `sup` (inclusive) que se repiten 3 veces consecutivas. En caso de que no exista un patrón para algún largo, se debe agregar una lista vacía en dicha posición.

Además, el laboratorio ya ha partido el procesamiento de los registros por lo que cuentas con la función `frecuencia(s, p)`. Esta recibe un *string* `s` con el registro del movimiento del canario y un *string* `p` con un patrón de movimiento. Retorna un *int* con el número de veces que el patrón `p` se repite en el *string* `s`. Esta función no considera el traslape entre dos patrones, esto es, que si tenemos `s = "SSS"`, que significa que se registró por tres instantes seguidos al canario en el

sensor superior de la jaula y `p = "SS"`, `frecuencia(s, p)` retorna 1 y no 2 puesto que no es válido considerar el segundo caracter del *string* `s` para obtener dos veces el patrón.

A continuación se muestran distintos ejemplos del retorno de las distintas funciones:

- `s = "SSSCSSCSSCSSCSSCISSCISSCISSCISSI"`
  - `frecuencia(s, "SS")`: 9
  - `mas_frecuente(s, 2)`: "SS"
  - `mas_frecuente(s, 5)`: "SSCSS"
  - `mas_frecuentes(s, 3, 5)`: ["SSC", "SSCI", "SSCSS"]
  - `mas_frecuentes(s, 2, 6)`: ["SS", "SSC", "SSCI", "SSCSS", "SSCSSC"]
  - `tres_veces_n(s, 3)`: ["SSC", "SCS", "CSS"]
  - `tres_veces_n(s, 4)`: ["SSCI", "SCIS", "CISS", "ISSC"]
  - `tres_veces_n(s, 1)`: ["S"]
  - `repetidos(s, 3, 5)`: [["SSC", "SCS", "CSS"], ["SSCI", "SCIS", "CISS", "ISSC"], []]
- `s = "CCCCCCCCCCCCCCCCCCCC"`
  - `frecuencia(s, "CCCCC")`: 4
  - `mas_frecuente(s, 2)`: "CC"
  - `mas_frecuente(s, 5)`: "CCCCC"
  - `mas_frecuentes(s, 3, 5)`: ["CCC", "CCCC", "CCCCC"]
  - `mas_frecuentes(s, 2, 6)`: ["CC", "CCC", "CCCC", "CCCCC", "CCCCCC"]
  - `tres_veces_n(s, 3)`: ["CCC"]
  - `tres_veces_n(s, 4)`: ["CCCC"]
  - `tres_veces_n(s, 1)`: ["C"]
  - `repetidos(s, 3, 5)`: [["CCC"], ["CCCC"], ["CCCCC"]]
- `s = "SSCIICCCSSIISCSICSIICCCSIICCCSSS"`
  - `frecuencia(s, "SII")`: 3
  - `mas_frecuente(s, 2)`: "CS"
  - `mas_frecuente(s, 5)`: "IICCC"
  - `mas_frecuentes(s, 3, 5)`: ["IIC", "IICC", "IICCC"]
  - `mas_frecuentes(s, 2, 6)`: ["CS", "IIC", "IICC", "IICCC", "IICCCS"]
  - `tres_veces_n(s, 3)`: []
  - `tres_veces_n(s, 4)`: []
  - `tres_veces_n(s, 1)`: ["C", "S"]
  - `repetidos(s, 3, 5)`: [[], [], []]
- `s = "SCICISCICSICISCSICISICICICSISCISISICISICSISICSISISI"`
  - `frecuencia(s, "SCI")`: 3
  - `mas_frecuente(s, 2)`: "SI"
  - `mas_frecuente(s, 5)`: "SICIS"
  - `mas_frecuentes(s, 3, 5)`: ["SIC", "ICIS", "SICIS"]
  - `mas_frecuentes(s, 2, 6)`: ["SI", "SIC", "ICIS", "SICIS", "CSICIS"]
  - `tres_veces_n(s, 3)`: []
  - `tres_veces_n(s, 4)`: []
  - `tres_veces_n(s, 1)`: []
  - `repetidos(s, 3, 5)`: [[], [], []]

Pregunta 2

Aplicando tus conocimientos de programación, has decidido crear una nueva red social que llamarás **Introgram**. Lo primero, y más importante, a implementar es justamente la parte social. Por lo tanto, en esta pregunta deberás crear a los usuarios de la red y permitirles seguir a otros usuarios.

Deberás crear las siguientes clases y métodos:

(a) (10 puntos) Clase `Usuario`. Cada usuario tiene un nombre de usuario que lo identifica, y puede **seguir** a otros usuarios. Deberás implementar sus métodos `__init__` (5 puntos) y `seguir` (5 puntos). El método seguir registra el hecho que un usuario **sigue** a otro (puedes suponer que el usuario que que se va a seguir existe en el sistema).

(b) (50 puntos) Clase `Introgram`. La clase Introgram representa a la red social completa. Deberás implementar sus métodos `__init__` (5 puntos), `agregar_usuario` (5 puntos), `seguidores` (10 puntos), `perfiles_falsos` (15 puntos), y `top_users` (15 puntos). Los últimos tres métodos se describen a continuación:

- `seguidores`: Recibe un *string* `n` con el nombre de un usuario. Retorna una *lista* con los nombres de los seguidores de un usuario (quiénes lo siguen).
- `perfiles_falsos`: Recibe dos *int*, `i` y `j`. Retorna una *lista* con los nombres de aquellos usuarios que siguen a más de `i` personas y tienen menos de `j` seguidores. En caso que ninguno de los usuarios cumpla con dicha condición, se debe retornar una lista vacía.
- `top_users`: Recibe tres *int*, `i`, `j` y `k`. Retorna una *lista* con los nombres de los `k` mejores usuarios de la red social. Los `k` mejores usuarios serán quienes **no** sean perfiles falsos (aquellos usuarios que siguen a más de `i` personas y tienen menos de `j` seguidores), y tengan el mayor número de seguidores. Puedes asumir que siempre habrán al menos `k` usuarios que no sean perfiles falsos. En caso de que se produzcan empates en el el k-esimo mejor usuario, puedes retornar cualquiera.

Tu código debe ser compatible con el siguiente código principal:

```
u1 = Usuario("emilia_143")
u2 = Usuario("felipe_ck")
u3 = Usuario("spammer23232523")
u4 = Usuario("denisse.ramirez")
u5 = Usuario("megustaprogramar")

u1.seguir("felipe_ck")
u1.seguir("megustaprogramar")
u1.seguir("denisse.ramirez")

u2.seguir("emilia_143")
u2.seguir("denisse.ramirez")

u3.seguir("felipe_ck")
u3.seguir("denisse.ramirez")
u3.seguir("megustaprogramar")

u4.seguir("megustaprogramar")

u5.seguir("emilia_143")
u5.seguir("felipe_ck")
u5.seguir("denisse.ramirez")

i = Introgram()
i.agregar_usuario(u1)
i.agregar_usuario(u2)
i.agregar_usuario(u3)
i.agregar_usuario(u4)
i.agregar_usuario(u5)
```

```
print("Seguidores de *megustaprogramar* : ",i.seguidores("megustaprogramar"))
#Seguidores de *megustaprogramar* :  ['emilia_143', 'spammer23232523', 'denisse.ramirez']

print("Seguidores de *spammer23232523* : ",i.seguidores("spammer23232523"))
#Seguidores de *spammer23232523* :  []

print("\n***PERFILES FALSOS***")
#***PERFILES FALSOS***
print(i.perfiles_falsos(3,1))
#['spammer23232523']

print("\n***TOP 2 USERS***")
#***TOP 2 USERS***
print(i.top_users(3,1,2))
#['denisse.ramirez', 'megustaprogramar']
```

Pregunta 3

Como programador experto, se te ha pedido crear un programa para poder codificar oraciones. El proceso consta de dos partes:

Parte 1

En esta parte, debes poder codificar una letra. Para esto, cuentas con una lista en donde cada valor es una lista que contiene `[letraoriginal, nuevaletra]`. Un ejemplo es el siguiente:

```
letras = [["a", "z"], ["b", "t"], ["c", "w"]]
```

Debes programar la siguiente función:

- (10 puntos) `codificar_letra(l, t)`: Recibe un *string* `l` con una letra y una *lista* `t` como la descrita anteriormente. Retorna un *string* con la nueva codificación de la letra. Por ejemplo, para la lista anterior, `codificar_letra("a", letras)` retorna `"z"`. Puedes asumir que la letra `l` siempre estará en la *lista* `t` de valores (en la posición 0 de los elementos). Notar que siempre deberás tomar la `letraoriginal` (posición 0) y transformarla a la `nuevaletra` (posición 1).

Parte 2

En esta parte deberás programar la codificación de una oración, que está representada por una `lista` de palabras. Esta consta de dos partes: la primera, transformar cada palabra de la oración según la codificación por letras de la Parte 1 y la segunda, separar aleatoriamente las palabras en dos listas. A continuación se explica cada función a programar.

- (22 puntos) `codificar_oracion(p, t)`: Recibe una *lista* `p` de palabras y una *lista* `t` de valores `[letraoriginal, nuevaletra]`. Retorna una lista con cada palabra codificada según la codificación por letras de la Parte 1. Por ejemplo, si `p = ["abc", "bc", "ac"]`, utilizando la lista `letras` para hacer la codificación, `codificar_oracion(p,letras)` debe retornar `["ztw", "tw", "zw"]`.
- (28 puntos) `separar_oracion(p)`: Recibe una *lista* `p` de palabras codificadas y retorna una lista `[p1, p2]`, donde `p1` y `p2` son listas que contienen pares `[palabra_codificada, posicion_original]`. Lo que debes hacer en esta función es que cada palabra de la lista `p` sea asignada aleatoriamente a `p1` o `p2` y se guarde en ella el par `[palabra_codificada, posicion_original]`. Notar que `posicion_original` es un *int* entre 0 y largo de la lista `p` – 1 y representa la posición de esa palabra en la lista original `p`. Por ejemplo, si `p = ["ztw", "tw", "zw"]`, un posible retorno sería `[[["ztw", 0], ["zw", 2]], [["tw",1]]]`, donde `p1 = [["ztw", 0], ["zw", 2]]` y `p2 = [["tw",1]]`. Notar que dado que la separación de la oración es aleatoria, otro retorno válido podría ser `[[["ztw", 0], ["tw",1], ["zw", 2]], []]`. No es necesario que `p1` y `p2` estén ordenadas según `posicion_original`.

Pregunta 4

La empresa DCCable necesita instalar antenas en una calle residencial con el fin de que a todas las casas que están en dicha calle les llegue una potencia entre 100 y 200 (inclusive). En concreto, la calle cuenta con `n` bloques y `x` casas, en donde en cada bloque puede haber una casa (o ninguna) y $\texttt{x} \leq \texttt{n}$. Un ejemplo se muestra a continuación:

bloque	0	1	2	3	4	5	6	7	8
casa			🏠		🏠				

Cuadro 2: Ejemplo calle con 8 bloques y 2 casas.

En cada bloque se puede poner a lo más una antena, y cada una de éstas tiene un alcance de 5 bloques: en el que se pone la antena, dos a la derecha y dos a la izquierda. Además, si a un bloque le llega potencia de dos antenas distintas, las potencias se suman. Por ejemplo, si DCCable dispone de una antena de 80 de potencia y otra de 20 de potencia, y decide colocar la primera en el bloque 2 y la segunda en el bloque 4 de la calle del Cuadro 2, la potencia que recibiría cada bloque es la siguiente:

bloque	0	1	2	3	4	5	6	7	8
casa			🏠		🏠				
antena			📶		📶				
potencia	80	80	100	100	100	20	20	0	0

Cuadro 3: Ejemplo potencia de las antenas.

Tu misión es ayudar a DCCable a ubicar las antenas que dispone en la calle con el fin de que a cada casa le llegue una potencia dentro del rango definido. Para ello cuentas con las siguientes variables:

- `calle`: Lista de booleanos. Si `calle[i]` es `True`, significa que en ese bloque existe una casa. En caso contrario significa que en ese bloque no hay una casa. Ten en consideración que el largo de la lista representa el largo de la calle y la cantidad de elementos `True` de la lista representa la cantidad de casas.
- `antenas`: Lista de enteros. Cada elemento representa la potencia de cada antena. El largo de la lista representa la cantidad de antenas con las que se cuentan para distribuir por la calle.

Las antenas se representan por enteros desde el *0* hasta *cantidad_de_antenas_menos_1*. Debes implementar (**usando backtracking**) una función que encuentre una distribución para las antenas dentro de la calle en donde a todas las casas les llegue entre 100 y 200 de potencia (inclusive). Deberás retornar una ***lista*** de enteros con el largo de la calle, en que cada elemento represente la solución para el bloque correspondiente. Si en ese bloque va una antena, debe ir el índice de la antena que se debe instalar en el bloque; en caso contrario, el elemento debe ser un `-1`. Ten en cuenta lo siguiente:

- Puedes implementar todas las funciones que desees para solucionar el problema.
- No es necesario que se utilicen todas las antenas.
- Cada antena puede ser instalada en un solo bloque.
- No es necesario que las antenas se ubiquen solamente en bloques con casa.
- Si existe más de una distribución posible, puedes entregar cualquiera de ellas.
- Si **no** existe una distribución posible, debes retornar una *lista* del largo de la calle en donde todos los elementos son `-1`.

Por ejemplo, para la siguiente configuración:

```
calle = [False,False,True,False,True,False,False,False,False,False,False,False,False,False,True]
antenas = [50,60,100,120]
```

Una posible solución sería `[-1, -1, -1, -1, 2, -1, -1, -1, -1, -1, -1, -1, -1, 0, 1]`, en donde en el bloque 4 ponemos `antenas[2]` que tiene 100 de potencia, en el bloque 13 ponemos `antenas[0]` que tiene 50 de potencia y finalmente, en el bloque 14 ponemos `antenas[1]` que tiene 60 de potencia (notar que no fue necesario utilizar la cuarta antena). El siguiente cuadro muestra la solución de la distribución de antenas:

bloque	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
casa			🏠		🏠										🏠
antena					a_2									a_0	a_1
potencia			100	100	100	100	100					50	110	110	110

Cuadro 4: Ejemplo solución distribución.

Así, vemos que a todas las casas le llega una potencia entre 100 y 200 porque a las que están en el bloque 2 y 4 les llega 100 (de la antena 2), y a la que está en el bloque 14 le llega 110 (por la suma de las antenas 0 y 1).