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

IIC1103 – Introducción a la Programación
1 - 2017

Examen

Instrucciones generales:

- El examen consta de cuatro preguntas con igual ponderación.
- Solo se reciben consultas sobre el enunciado en los primeros 30 minutos.
- La última página es un recordatorio; no está permitido utilizar material de apoyo adicional.
- Las cuatro 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 RUT 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 cuatro horas para resolver la prueba desde que se entregan los enunciados.

Pregunta 1 (1/4)

Estás jugando al famoso juego del Colgado/Hangman, en el que debes adivinar una palabra que alguien ha pensado. Para ser el mejor jugador de Hangman del mundo, quieres escribir un programa computacional que te sugiera la mejor letra a "adivinar" dado lo que ya llevas adivinado.

Por ejemplo, si llevas adivinado "`__A__A`", hay muchas palabras que calzan, tales como: "CHARCA", "CHARLA", "CUARTA", "TRAMPA", "REACIA", "PLASMA", "OSADIA" (y muchas más – nota, eso sí, que "AMARRA" no calza, pues no puede haber otra "A" si ya la habías adivinado). Tendrás que hacer un programa que entregue la letra que conviene adivinar pues se repite en la mayor cantidad de palabras posibles. Por ejemplo, para el ejemplo anterior (si consideramos que solo existen las palabras posibles listadas), conviene adivinar la letra "R", dado que se encuentra en la mayor cantidad de palabras posibles (se encuentra en 5 de las 7 palabras listadas, mientras que, por ejemplo, "C" se encuentra en 4 palabras). Si hay varias letras que se repiten con igual frecuencia (y es la máxima frecuencia), puedes sugerir cualquiera de ellas.

Ejemplo de diálogo:

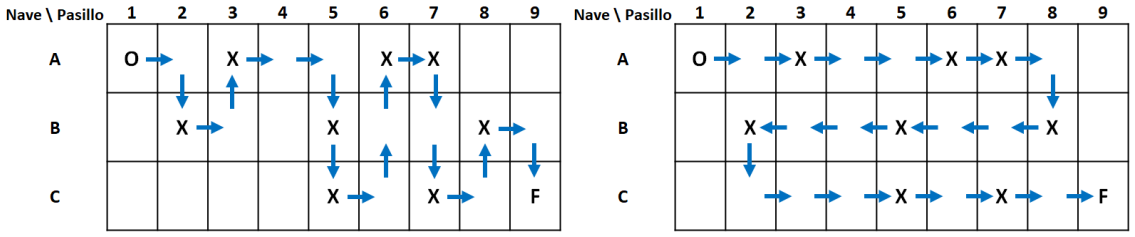
```
Ingresa lo que llevas adivinado: __A__A
Te sugiero la letra R
```

Puedes considerar que tendrás a tu disposición un archivo "diccionario.txt" con todas las palabras del idioma español (todas en mayúsculas y sin tildes), una por línea.

Nota: Si lo deseas, puedes importar el módulo string (`import string`), para usar la variable `string.ascii_uppercase`, que contiene el string "`ABCDEFGHIJKLMNOPQRSTUVWXYZ`". No consideres la letra Ñ, letras con tildes, ni otras letras especiales.

Pregunta 2 (1/4)

Tom y Jerry están encargados de hacer las compras de supermercado. Cada uno tiene una forma diferente de recorrer el supermercado (ver Figura 1a y 1b). Tom recorre cada pasillo (del 1 al 9; de arriba hacia abajo, luego de abajo hacia arriba, y así sucesivamente), cambiándose de pasillo cuando no quedan más productos que recoger en el pasillo que está recorriendo. Por su parte, Jerry recorre cada nave (de la 'A' a la 'C'; de izquierda a derecha, luego de derecha a izquierda, y finamente de izquierda a derecha), cambiándose de nave cuando no quedan más productos que recoger en la nave que está recorriendo. Ambos empiezan en la esquina superior izquierda, es decir en la posición ['A', 1] (nave 'A', pasillo 1) y terminan en la esquina inferior derecha, es decir en la posición ['C',9] (nave 'C', pasillo 9). Cada uno piensa que su método de recorrer el supermercado es mejor, por lo que harán una competencia para ver quien termina de hacer la compra más rápido. El supermercado siempre tendrá el mismo tamaño.



(a) Ruta Tom

(b) Ruta Jerry

Figura 1: Recorridos

Cree las siguientes funciones para describir la situación planteada:

a) **(20 puntos)** `crear_lista_super(cantidad)`: recibe como parámetro la *cantidad* de productos que se va a comprar, decide aleatoriamente en qué posiciones del supermercado estarán los productos, y retorna una lista con las coordenadas en que están todos los productos a comprar en el supermercado. Considera que no puede haber más de un producto en una coordenada. Por ejemplo, si se llama con cantidad igual a 8, podría retornar la siguiente lista: `[['B',2], ['C',7], ['A',3], ['B',5], ['C',5], ['B',8], ['A',7], ['A',6]]`. Nota que, en este ejemplo, son las coordenadas marcadas en la figura con una **X**.

b) **(20 puntos)** `tiempo_ruta(ruta)`: recibe una *ruta*, que es una lista de coordenadas en el orden en que se desean recorrer, y calcula el tiempo requerido para recorrerla. Suponga que el tiempo en pasar de una nave a otra es 2 minutos, el tiempo en pasar de un pasillo a otro es 1 minuto, y que los demás tiempos son despreciables. Por ejemplo, considerando las rutas de la figura, `tiempo_ruta([['A', 1], ['B', 2], ['A', 3], ['B', 5], ['C', 5], ['A', 6], ['A', 7], ['C', 7], ['B', 8], ['C', 9]])` (la ruta de Tom) retornará 28; y `tiempo_ruta([['A', 1], ['A', 3], ['A', 6], ['A', 7], ['B', 8], ['B', 5], ['B', 2], ['C', 5], ['C', 7], ['C', 9]])` (la de Jerry) retornará 24.

c) **(20 puntos)** Escribe un programa que haga lo siguiente:

- Crear aleatoriamente una lista de 8 coordenadas a visitar en la tienda.
- Calcular la ruta de Tom, mostrarla en pantalla (imprimir la ruta), y luego mostrar cuánto se demoraría Tom.
- Calcular la ruta de Jerry, mostrarla en pantalla (imprimir la ruta), y luego mostrar cuánto se demoraría Jerry.
- Mostrar en pantalla quién de los dos lo haría en menor tiempo, o indicar si ambos se demorarían lo mismo.

Puedes asumir que tienes las siguientes funciones:

`recorrido_tom(lista)`: recibe una *lista* de coordenadas en el supermercado, y retorna la ruta que haría Tom. Para el ejemplo de la figura, retornará la ruta: `[['A', 1], ['B', 2], ['A', 3], ['B', 5], ['C', 5], ['A', 6], ['A', 7], ['C', 7], ['B', 8], ['C', 9]]`

`recorrido_jerry(lista)`: recibe una *lista* de coordenadas en el supermercado, y retorna la ruta que haría Jerry. Para el ejemplo de la figura, retornará la ruta: `[['A', 1], ['A', 3], ['A', 6], ['A', 7], ['B', 8], ['B', 5], ['B', 2], ['C', 5], ['C', 7], ['C', 9]]`
Nota que ambas funciones le agregan que el primer elemento será ['A',1] y el último ['C',9], punto de partida y punto de término de cualquier ruta.

Pregunta 3 (1/4)

La empresa EcoSushi te ha encargado que la ayudes a organizar su proceso de despacho de pedidos, para lo cual utiliza repartidores en bicicleta. Cada pedido tiene el nombre del cliente, una dirección de despacho, una descripción (que indica lo que el cliente pidió) y una coordenada, x e y, asociadas a la dirección del cliente. Existe un despachador que organiza el reparto, el cual recibe los pedidos una vez que están terminados y los distribuye a los repartidores respetando el orden en que estuvieron listos. Para decidir a qué repartidor asignará un pedido, el despachador calcula la distancia entre el pedido que quiere asignar y el último pedido en la lista de cada repartidor, asignándoselo a aquel repartidor con el que el pedido tiene una menor distancia (si hay varios con la misma distancia mínima, se lo puede asignar a cualquiera de ellos). La función de distancia que usa el despachador es:

$$\sqrt{(x_{pedido} - x_{ultimo\_lista\_repartidor})^2 + (y_{pedido} - y_{ultimo\_lista\_repartidor})^2} \quad (1)$$

Para ayudar a la empresa EcoSushi, deberás crear lo siguiente:

a) **(4 puntos)** La clase `Pedido`, con el método `__init__(self, nombre, dirección, x, y, descripción)`.

b) La clase `Repartidor`, con los siguientes métodos:

- **(4 puntos)** `__init__(self, nombre)`, pero que además tiene como atributo la lista de pedidos que debe despachar.
- **(2 puntos)** `agregar_pedido(self,pedido)`, que agrega un pedido al repartidor.
- **(12 puntos)** `distancia_ultimo_pedido(self,pedido)`, que retorna la distancia entre el último pedido de la lista del repartidor y el pedido recibido como parámetro. Cuando no hay ningún pedido en la lista del repartidor este método debe retornar cero.
- **(6 puntos)** `mostrar_ruta(self)`, que muestra la ruta del repartidor imprimiendo una línea por cada pedido, mostrando el nombre del repartidor, el nombre del cliente y la dirección del cliente.

c) La clase `Despachador`, con los siguientes métodos:

- **(4 puntos)** `__init__(self)`, que tiene como atributos la lista de pedidos que quiere despachar y la lista de repartidores.
- **(2 puntos)** `agregar_pedido(self,pedido)`. Este método solamente agrega un pedido al despachador, aún no lo asigna a los repartidores.
- **(2 puntos)** `agregar_repartidor(self,repartidor)`, que agrega un repartidor.
- **(24 puntos)** `despachar(self)`, que distribuye todos los pedidos de la lista entre los repartidores utilizando las reglas indicadas anteriormente, utilizando el método `distancia_ultimo_pedido` de la clase Repartidor para tomar la decisión de a quién asignar el pedido. Cuando los pedidos han sido asignados a los repartidores, la lista de pedidos debe quedar vacía (sin pedidos por asignar).

Los métodos que programes deberán ser compatibles con el siguiente programa principal:

```
r1=Repartidor("José")
r2=Repartidor("Pedro")
r3=Repartidor("Florencia")
d=Despachador()
p1=Pedido("Juan","Dirección 1",10,10,"Ramen")
p2=Pedido("Gabriela","Dirección 2",20,20,"Sashimi")
p3=Pedido("Josefa","Dirección 3",30,40,"Tempura")
p4=Pedido("Rodrigo","Dirección 4",25,33,"California Roll")
p5=Pedido("Mario","Dirección 5",12,21,"Onigiri")
p6=Pedido("Paula","Dirección 6",11,18,"Ramen")

d.agregar_pedido(p1)
d.agregar_pedido(p2)
d.agregar_pedido(p3)
d.agregar_pedido(p4)
```

```
d.agregar_pedido(p5)
d.agregar_pedido(p6)
d.agregar_repartidor(r1)
d.agregar_repartidor(r2)
d.agregar_repartidor(r3)
d.despachar()

r1.mostrar_ruta()
r2.mostrar_ruta()
r3.mostrar_ruta()
```

Al final el programa puede mostrar, por ejemplo, las siguientes rutas:

```
José Juan Dirección 1
Pedro Gabriela Dirección 2
Pedro Mario Dirección 5
Pedro Paula Dirección 6
Florencia Josefa Dirección 3
Florencia Rodrigo Dirección 4
```

Pregunta 4 (1/4)

Una agrónoma te pide ayuda para resolver su dilema sobre qué vegetales plantar en su parcela. Quiere no repetir vegetales, y que se ocupe bien el terreno sin pasarse en el gasto de agua.

En concreto, se te pide que implementes – usando backtracking – la función `resolver(vegetales,T,A)`, donde:

- La función recibe como parámetro *vegetales* con los posibles vegetales que se pueden plantar. Los vegetales están representados como una lista de listas de la forma `[n,t,a]`, donde el string `n` es el nombre del vegetal, el entero `t` es el terreno que se ocuparía para plantar ese vegetal, y el entero `a` es el agua que se consumiría.
  Por ejemplo:
  `vegetales = [['Tomates',2,20], ['Lechugas',1,80], ['Paltas',2,40], ['Quinoa',2,20], ['Melones',4,80]]`
  Es una lista con 5 posibles vegetales, donde, por ejemplo, los tomates necesitan 2 de terreno y gastarán 20 de agua.
- La función recibe también como parámetro los enteros `T` y `A`, tal que, la solución propuesta debe usar igual o más de `T` de terreno, y menos o igual `A` agua.
- La función retorna una lista de strings con UNA posible solución. Es decir, una lista con los nombres de los vegetales que le propones plantar tal que ocupen más o igual T terreno y menos o igual A agua.
  Por ejemplo, dados los vegetales:
  `vegetales = [['Tomates',2,20], ['Lechugas',1,80], ['Paltas',2,40], ['Quinoa',2,20], ['Melones',4,80]]`
  y los límites `T = 7` y `A = 150`.
  Una posible solución sería `['Tomates', 'Paltas', 'Melones']`.

Nota: este ejemplo solo tiene 3 soluciones posibles, dado que el resto de posibilidades se pasan en el agua o no ocupan suficiente terreno:

- `['Tomates', 'Paltas', 'Melones']` con 8 de terreno y 140 de agua
- `['Tomates', 'Quinoa', 'Melones']` con 8 de terreno y 120 de agua
- `['Paltas', 'Quinoa', 'Melones']` con 8 de terreno y 140 de agua.

Algunas consideraciones:

- No importa el orden de los vegetales en la solución.
- La solución no puede contener vegetales repetidos.
- La función debe recibir y retornar los parámetros indicados, pero puedes definir todas las funciones adicionales que quieras (puedes implementar el backtracking en otra función).

Recordatorio contenidos examen

En los ejemplos, lo marcado como `<texto>` se interpreta como lo que debes rellenar con tu código según cada caso.

1. Tipos de datos y operadores

Tipo de dato	Clase	Ejemplo
Números enteros	`int`	`2`
Números reales	`float`	`2.5`
Números complejos	`complex`	`2 + 3j`
Valores booleanos	`bool`	`True/False`
Cadenas de texto	`str`	`"hola"`

Operación	Descripción	Ejemplo
`+`	Suma	`2.3+5.4`
`-`	Resta	`45.45-10.02`
`-`	Negación	`-5.4`
`*`	Multiplicación	`(2.3+4.2j)*3`
`**`	Potenciación	`2**8`
`/`	División	`100/99`
`//`	División entera	`100//99`
`%`	Módulo	`10%3`

Prioridad (de mayor a menor): `()`; `**`; `*`, `/`, `//` o `%`; `+` o `-`.

Operación	Descripción	Ejemplo
`== (!=)`	Igual (distinto) a	`2==2`
`<(<=)`	Menor (o igual)	`1<1.1`
`>(>=)`	Mayor (o igual)	`3>=1`
`and`	Ambos `True`	`2>1 and 2<3`
`or`	Algún `True`	`2!=2 or 2==2`
`not`	Negación	`not True`

Prioridad (de mayor a menor): `()`; `or`; `and`; `not`; comparadores.

2. Funciones predefinidas

- `int(arg)` convierte `arg` a entero.
- `float(arg)` convierte `arg` a número real.
- `str(arg)` convierte `arg` a cadena de texto (string).
- `list(arg)` genera una lista con elementos según `arg`, que debe ser *iterable* (strings, listas, tuplas, `range`).

3. Función `print`

- Un argumento:
  `print(arg)`
- Dos o más argumentos:
  `print(arg1, arg2,arg3)`
- Uso de parámetros, por ejemplo, para eliminar salto de línea y separar con guión:
  `print(arg, sep='-', end='')`

4. Función `input`

- `ret = input(texto)` guarda en `ret` un `str` ingresado.
- `ret = int(input(texto))` guarda en `ret` un `int` ingresado.
- `ret = float(input(texto))` guarda en `ret` un `float` ingresado.

5. `if/elif/else`

```
if <cond 1> :
  <codigo si se cumple cond 1>
if <cond 1.1> :
  <codigo si se cumple 1.1>
else :
  <codigo si no se cumple 1.1>
elif <cond 2> :
  <codigo si se cumple cond 2 pero no cond 1>
else :
  <codigo si no se cumple cond 1 ni cond 2>
```

6. `while`

```
while <condicion> :
  <codigo que se ejecuta repetidas
veces mientras se cumpla
condicion>
```

7. Funciones propias

```
def funcion(<argumentos>):
  <codigo de funcion>
  return <valor de retorno>
```

Variables y parámetros definidos dentro de funciones no son visibles fuera de la función (*scope* local).

8. Programación orientada a objetos

```
class <NombreClase>:
  def __init__(self, <parametros>):
    self.<atributos> = <algun parametro>
  def __str__(self):
    <codigo sobrecarga de funcion str()>
    return <string que representa
      los atributos>
  def <metodo propio>(self, <parametros>):
    <codigo de modulo propio>
```

9. Strings, clase `str`

Acceso a caracteres particulares con operador `[]`, partiendo con índice cero. Porción de string con *slice*, por ejemplo si `string='Hola'`, `string[1:3]` es `'ol'`. Algunos métodos y funciones de strings:

- Operador `+`: une (concatena) dos strings.
- Operador `in`: cuando `a in b` retorna `True`, entonces el string `a` está contenido en el string `b`.

- `string.find(a)`: determina si `a` está contenido en `string`. Retorna la posición (índice) dentro de `string` donde comienza la primera aparición del sub-string `a`. Si no está, retorna `-1`.
- `string.upper()`, `string.lower()`: retorna `string` convertido a mayúsculas y minúsculas, respectivamente.
- `string.strip()`: retorna un nuevo string en que se eliminan los espacios en blanco iniciales y finales de `string`.
- `string.split(a)`: retorna una lista con los elementos del `string` que están separados por el string `a`. Si se omite `a`, asume que el separador es uno o más espacios en blanco o el salto de línea.
- `p.join(lista)`: suponiendo que `p` es un string, retorna un nuevo string conteniendo los elementos de la lista "unidos" por el string `p`.
- Función `len(string)`: entrega el número de caracteres de `string`.

Una forma de iterar sobre los caracteres de `string`:

```
for char in string:
  <operaciones con el caracter char>
```

10. Listas

Secuencias de elementos-objetos. Se definen como `lista = [<elem1>,<elem2>,...,<elemN>]`. Los elementos pueden o no ser del mismo tipo. Para acceder al elemento `i`, se usa `lista[i]`. La sublista `lista[i:j]` incluye los elementos desde la posición `i` hasta `j-1`. Algunos métodos y funciones de listas:

- Operador `+`. concatena dos listas.
- Operador `in`: `a in b` retorna `True` cuando el elemento `a` está contenido en la lista `b`. Si no está contenido, retorna `False`.
- `lista.append(a)`: agrega `a` al final de la lista.
- `lista.insert(i,a)`: inserta el elemento `a` en la posición `i`, desplazando los elementos después de `i`.
- `lista.pop(i)`: retorna el elemento de la lista en la posición `i`, y lo elimina de `lista`.
- `lista.remove(elem)`: elimina la primera aparición de `elem` en la lista.
- Función `len(lista)`: entrega el número de elementos de `lista`.

Para iterar sobre los elementos de `lista`:

```
for elem in lista:
  <lo que quieran hacer con elem>
```

11. Archivos

Abrir un archivo:
`archivo = open(<nombre archivo>,<modo>)`,
p. ej. `archivo = open('archivo.txt','r')`. `<modo>` puede ser `'w'` para escribir un archivo nuevo, `'r'` para leer un archivo (predeterminado), y `'a'` para escribir en un archivo ya existente, agregando datos al final del archivo.

Algunos métodos del objeto que retorna la función `open`:

- `archivo.readline()`: retorna un string con la línea siguiente del archivo, comenzando al inicio del archivo.
- `archivo.write(string)`: escribe en el archivo el string `string`.
- `archivo.close()`: cierra el archivo.

Para leer un archivo entero puedes usar `for`, que iterará línea por línea del archivo:

```
archivo = open('archivo.txt','r')
for linea in archivo:
  <lo que quieran hacer con linea>
```

12. Búsqueda y ordenamiento

Búsqueda secuencial o lineal Busca secuencialmente un elemento dentro de una lista de tamaño n hasta encontrarlo. Peor caso: elemento no está en lista, n comparaciones. Uso: lista desordenada.

Búsqueda binaria Supone lista ordenada. Divide la lista en sublistas dependiendo del valor del elemento buscado: si el elemento es mayor que el elemento medio de la lista, se sigue por la sublista derecha; si no, por la lista izquierda. Peor caso: para una lista de n elementos, se realizan $log_2(n)$ comparaciones.

Ordenamiento por selección Busca el índice del elemento más pequeño de la lista, e intercambia los valores entre el índice encontrado y el primer elemento; luego, encuentra el segundo elemento más pequeño, y lo intercambia con el elemento de la segunda posición, realizando la misma operación para el tercero, cuarto, etc..

Ordenamiento por inserción Itera por los elementos de la lista, partiendo por el primer elemento, y generando una lista ordenada con los elementos mientras itera. Es similar a cómo una persona ordena una lista de cartas.

Métodos de Python para ordenamiento

- `lista.sort()`: ordena `lista` en forma ascendente.

13. Recursión y backtracking

Elementos de una función recursiva:

- Caso base: para terminar la recursión.
- Llamada recursiva: llamada a la misma función dentrl de la función, con parámetros distintos que hacen disminuir el problema original.

Para programar una función recursiva, descomponer la función en elementos que puedan ser llamados con subconjuntos de datos. Ejemplo: suma de los primeros N números

```
def suma(N):
  if N == 1: # caso base
    return 1:
  else: #descomponer en N y restantes N-1
    return N + suma(N-1)
```

Backtracking: cuando hay muchas formas o caminos de buscar una solución. Se puede formar un aŕbol con las soluciones, y el algoritmo comienza en el nodo raíz. Cada camino dentro del árbol es un código recursivo, y dentro del código se exploran las ramas del árbol. Ejemplo: encontrar la suma de todas las formas de combinar una lista de números:

```
def suma_conm_lista(lista, sublista=[], suma=0):
  # Caso base: sublista tiene N elementos,
  # o la lista tiene 0 elementos
  if len(lista) == 0:
    print sublista, suma
    return
  else:
    # Quito un elemento de la lista y lo
    # agrego a la sublista para llamar
    # a la funcion en forma recursiva,
    # y aumento la suma con el
    # elemento que saque
    for i in range(len(lista)):
      suma_conm_lista( lista[:i]+lista[i+1:],\
          sublista+[lista[i]], suma+lista[i])
```