PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN
IIC1103- INTRODUCCIÓN A LA PROGRAMACIÓN

IIC1103 2015-2
Rúbrica Examen

Entregar corrección hasta las 12:00 del jueves 26 de noviembre de 2015

Criterios de corrección generales

- No se descuenta el uso de contenidos no vistos en clases, mientras se resuelva lo pedido en la pregunta.
- Se espera que apliques un buen criterio en la corrección de cada pregunta, enfocándote en que la solución del alumno tenga los elementos necesarios para resolver la pregunta, y sólo quitar una cantidad mínima de puntos por pequeños errores de sintaxis. En ningún caso debes evaluar una pregunta o parte de ella como no lograda sólo por pequeños errores de sintaxis.
- Cada pregunta tiene 60 puntos y tres criterios de evaluación: "No logrado" (0 pts), "Logrado con observaciones" (entre 1 a 59 pts), y "Logrado" (60 pts).
- Las preguntas se corrigen tomando en cuenta el grado de cumplimiento de las respuestas con los **objetivos del enunciado**, y no la mayor o menor acomodación a los ejemplos de solución de las preguntas, que deben ser usados **sólo como referencia**. Corregir las respuestas como erróneas porque no calzan con las respuestas tipo no se debe hacer. En caso de que las respuestas de un alumno sean distintas a la pauta, es tu labor como ayudante verificar si la respuesta cumple con los objetivos de la pregunta, y en caso afirmativo, evaluar acorde a la pauta. Se debe descontar por errores de sintaxis en forma criteriosa (un error en una pregunta no implicará necesariamente que toda la pregunta es no lograda).
- En aquellos casos donde la respuesta calza en "Logrado con observaciones", y el puntaje total está desglosado por ítem, el puntaje que acompaña a cada ítem corresponde al logro del ítem en particular. Los puntajes en esta columna son referenciales, esperamos que apliques un **buen criterio**, reiteramos enfáticamente que no debes corregir una pregunta o sección de pregunta como no lograda si sólo tiene pequeños errores de sintaxis.
- Si el código realiza una labor no relacionada con la pregunta, será un error mayor inmediatamente.

Problema 1 (60 puntos)

Resolver una *ecuación diofántica* de la forma:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = C$$

consiste en encontrar valores dentro de un conjunto D de números enteros para las variables $x_1, \ldots, x_n$, dados los valores de $a_1, \ldots, a_n$ y el valor de C.

Escribe una función para resolver este tipo de ecuaciones, la que debe ser de tipo recursiva (funciones sólo iterativas no serán evaluadas). De haber una solución, tu función deberá imprimir en pantalla una lista con los valores de x_1 hasta x_n. En caso contrario, deberá imprimir "no hay solución".

Tu función deberá recibir una lista no vacía de números enteros para los coeficientes $a_1, \ldots, a_n$, y un número entero para la constante C. Además, deberá suponer que los valores de x_i son números enteros entre -10 y 10, es decir, $D = \{-10, -9, \ldots, 9, 10\}$.

Por ejemplo, si quieres resolver la siguiente ecuación diofántica

$$2x_1 + 3x_2 = 5,$$

deberás llamar a tu función usando como argumentos la lista `[2, 3]` y el número `5`, y debería imprimir en pantalla, por ejemplo, `[-8, 7]`, ya que $2 \times (-8) + 3 \times (7) = 5$. Nota que puede haber más de una solución; basta con que el programa imprima una solución válida (si la hay), no todas las soluciones.

Ejemplo solución problema 1

La solución 3 aplica especialmente a las respuestas de alumnos de la sección 2.

Solución 1

```
def resolver(sol,i,L,C):
    if L==[]:
        return C==0
    d=-10
    while d<=10:
        sol[i]=d
        if resolver(sol,i+1,L[1:],C-L[0]*d):
            return True
        d = d + 1
    return False

def diofantica(L,C):
    sol=[0]*len(L)
    if resolver(sol,0,L,C):
        print(sol)
    else:
        print("no hay solucion")
```

Solución 2

```
def resolver(listaA, C, listaX):
    #CASO BASE: valores para todas las X
    if len(listaA) == len(listaX):
        suma = 0
        for i in range(0,len(listaA)):
            suma += listaA[i]*listaX[i]
        if suma == C:
            return True
        return False

    # Pongo un valor a la 'x'
    else:
        for i in range(-10,10+1):
            listaX.append(i)
            if resolver(listaA, C, listaX):
                return True
            listaX.pop()
        return False

def diofantica(listaA, C):
    listaX = []
    if resolver(listaA, C, listaX):
```

```
        print(listaX)
    else:
        print("No tiene solucion")
```

Solución 3 (con plantilla)

```
import sys

class Diofantica:
    coef_a = []
    C = 0
    min = 0
    max = 9

    def __init__(self, coef_a, C, min, max):
        self.coef_a = coef_a
        self.C = C
        self.min = min
        self.max = max

    # Evalua una posible solucion
    def Evaluar(self, solucion):
        suma = 0
        for i in range(len(self.coef_a)):
            suma += self.coef_a[i] * solucion[i]
        return suma == self.C

    # Retorna True si vale la pena explorar la solución candidata
    def SirveComoSolucionParcial(self, solucion_candidata):
        if len(solucion_candidata) > len(self.coef_a):
            return False
        else:
            return True

    # Retorna True si la solución candidata resuelve el problema
    def SirveComoSolucionFinal(self, solucion_candidata):
        if len(solucion_candidata) < len(self.coef_a):
            return False
        else:
            return self.Evaluar(solucion_candidata)

    # Usa la solución candidata como una solución valida al problema
    def MostrarSolucionFinal(self, solucion_candidata):
        print("Solucion:", solucion_candidata)

    # Genera una lista de soluciones extendidas, derivadas de la solución candidata
    def ObtenerSolucionesCandidatas(self, solucion_candidata):
        listaCandidatas = []
        for v in range(self.min, self.max+1):
            listaCandidatas.append(solucion_candidata + [v])
        return listaCandidatas
```

```
# Retorna una solución candidata inicial
def SolucionInicial(self):
    return []

# Backtracking genérico
def Backtracking(self, solucion_candidata):
    if not self.SirveComoSolucionParcial(solucion_candidata):
        return False
    if self.SirveComoSolucionFinal(solucion_candidata):
        self.MostrarSolucionFinal(solucion_candidata)
        return True
    lista_candidatas = self.ObtenerSolucionesCandidatas(solucion_candidata)
    for solucion in lista_candidatas:
        if self.Backtracking(solucion):
            return True
```

Criterios de evaluación problema 1

Nota: esta pregunta tiene dos criterios de evaluación. El descrito en la tabla es general, excepto para la sección 2, donde es opcional usar una plantilla para solucionar problemas de backtracking. En el caso de un alumno que usó la plantilla (un set de funciones), usar el criterio de solución con plantilla descrito después de la tabla general.

Aspecto	No logrado	Logrado con observaciones	Logrado
Función recursiva para resolver la ecuación diofántica (60 pts)	Sin código, código tiene errores mayores, o código no corresponde a un algoritmo recursivo (0 pts)	▪ Definición de caso base (5 pts) ▪ Recursión sobre elementos de x (10 pts) ▪ Iteración o recursión por valores de D por cada x (5 pts) ▪ Criterio de término de recursión (5 pts) ▪ Impresión en pantalla de solución o mensaje de "no hay soluciónçon errores menores, o retorna la solucion en vez de imprimir en pantalla. (5 pts)	▪ Definición de caso base (10 pts) ▪ Recursión para generar la lista de x (20 pts) ▪ Iteración o recursión por valores de D por cada x (10 pts) ▪ Criterio de término de recursión (10 pts) ▪ Impresión en pantalla de solución o mensaje de "no hay solución"(10 pts)

Criterios de evaluación problema 1 con plantilla

1. Correcta implementación de función/método **SirveComoSolucionParcial(solucion_candidata)** (5 pts)
    - 0 a 5 pts – toda solución parcial es válida, pero no puede pasarse del número de elementos de la lista de x.
2. Correcta implementación de función/método **SirveComoSolucionFinal(solucion_candidata)** (20 pts)
    - 0 a 5 pts – chequear que se requiere tener la misma cantidad de elementos que coeficientes a_i.
    - 0 a 15 pts – chequear que se cumpla la ecuación.
3. Correcta implementación de función/método **MostrarSolucionFinal(solucion_candidata)** (5 pts)

- 0 a 5 pts - Impresión en pantalla de solución.

4. Correcta implementación de función/método **ObtenerSolucionesCandidatas(solucion_candidata)** (10 pts)
   - 0 a 5 pts – Iteración por valores de D por cada x_i
   - 0 a 5 pts – Construcción de lista de soluciones candidatas para siguiente nivel de recursión.
5. Correcta implementación de función/método **SolucionInicial()** (5 pts)
   - 0 a 5 pts – Solución inicial es simplemente una lista vacía.
6. Correcta implementación de función/método **Backtracking(solucion_candidata)**: (10 pts)
   - 0 a 5 pts – Buen uso de la plantilla
   - 0 a 5 pts – Identificar que la solución candidata corresponde a asignar valores a un x_i más en cada llamado en profundidad a la función Backtracking
7. Impresión en pantalla de mensaje de "no hay solución"(5 pts)
   - 0 a 5 pts – se reconoce que no se encontró solución y se despliega mensaje “no hay solución”

Problema 2 (60 puntos)

La función `insertar`, mostrada abajo, utiliza la idea de búsqueda binaria para insertar un número dentro de una una lista de números, suponiendo que dicha lista se encuentra ordenada ascendentemente. Luego de retornar, la lista sigue estando ordenada ascendentemente.

La función recibe en el primer argumento a la lista y en el segundo al elemento a insertar. Así, para insertar el elemento `4` en la lista `lista`, se hace el siguiente llamado:

```
insertar(lista,4)
```

Abajo se muestra el código de la función con algunas partes reemplazadas con recuadros que debes rellenar, marcados como [1?], [2?] y [3?].

```
def comparacion(x,y):
    return [ 1? ]

def insertar(lista, elemento):
    i = 0
    j = len(lista)-1
    while i <= j:
        medio = [ 2? ] // 2
        if comparacion(elemento,lista[medio]):
            j = medio - 1
        else:
            i = medio + 1
    lista.insert([ 3? ] ,elemento)
```

1. (40 puntos) Di cómo completar en código en cada uno de estos recuadros para que la función haga lo que se supone debe hacer. (En tu hoja de respuesta, escribe a qué corresponde [1?], [2?] y [3?])

2. (20 puntos) Escribe la función más simple que puedas para ordenar una lista de números y que use la función `insertar`.

Ejemplo solución problema 2

```
def comparacion(x,y):
    return x < y

def insertar(lista, elemento):
    i=0
    j=len(lista)-1
    while i <= j:
        medio = (i + j) // 2
        if comparacion(elemento,lista[medio]):
            j = medio - 1
        else:
            i = medio + 1
    lista.insert(i,elemento)

def ordenar(l):
    ordenada = []
    for x in l:
        insertar(ordenada,x)
    return ordenada
```

Criterios de evaluación problema 2

Nota: en la parte a), no hay puntajes intermedios, los códigos en los recuadros deben ser correctos, por eso se desglosa en los recuadros 1, 2 y 3.

Aspecto	**No logrado**	**Logrado con observaciones**	**Logrado**
Completar código faltante. No hay puntajes intermedios, el código dentro de los recuadros debe ser exacto (40 puntos)	Sin código (0 pts)	No hay evaluación con observaciones, código debe ser exacto.	▪ Primer recuadro (`x<y`) o (`x<=y`) (10 pts) ▪ Segundo recuadro (`i+j`) (10 pts) ▪ Tercer recuadro (`i`) o (`j+1`) (20 pts)
Función que llame a `insertar` para insertar un elemento en una lista en forma ordenada (20 pts)	Sin código, o código tiene errores mayores (0 pts)	Código con errores menores (10 pts)	Código correcto (20 pts)

Problema 3 (60 puntos)

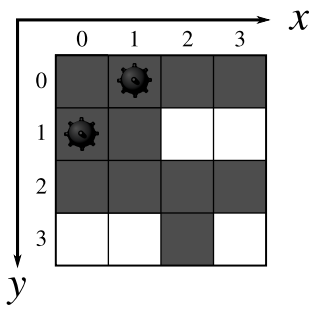
Escribe un programa Python que permita realizar acciones sobre un tablero de *buscaminas*. Para esto, considera que tienes un archivo llamado `buscaminas.txt` con el estado de un tablero. En éste, cada celda es un string de dos caracteres. El primero (de izquierda a derecha) dice si la celda está descubierta o no (`'1'` o `'0'`), mientras que el segundo caracter indica si en esa casilla hay una bomba o no (`'1'` o `'0'`). Con esto, existen 4 posibilidades para cada celda:

- `'00'`: La celda no está descubierta y no tiene bomba.
- `'01'`: La celda no está descubierta, pero tiene bomba.
- `'10'`: La celda está descubierta y no tiene bomba.
- `'11'`: La celda está descubierta y además tiene una bomba.

El archivo `buscaminas.txt` tiene cada una de las celdas separadas por comas "`,`". Por ejemplo, se muestra el contenido de un ejemplo del archivo `buscaminas.txt`, y al lado derecho el tablero que representa.

buscaminas.txt

```
00,01,00,00
01,00,10,10
00,00,00,00
10,10,00,10
```



Con esta información se te pide:

1. (10 puntos) Crear la función `esta_descubierta(x,y)` que reciba las coordenadas x e y de una celda y retorne `True` si la celda está descubierta o `False` si no está descubierta (está oculta) en el tablero. Por ejemplo, `esta_descubierta(2,3)` retorna `False`, pues la celda es `'00'`, por lo tanto no está descubierta.
2. (20 puntos) Crear la función `descubrir(x,y)` que reciba las coordenadas x e y de una celda y le cambie el estado a la celda (el primer dígito) siempre y cuando sea posible (es decir, no está descubierta) y actualiza el estado de todo el tablero en el archivo `buscaminas.txt`.
3. (30 puntos) Crear la función `dar_pista(x,y)` que recibe las coordenadas x e y de una celda y retorne la cantidad de bombas adyacentes a dicha celda (considerando 8 direcciones). Por ejemplo, `dar_pista(2,1)` retorna `1`, dado que hay una bomba en la coordenada (1,0).

Nota que no puedes agregar más parámetros a las funciones.

Ejemplo solución problema 3

Solución 1

```
def leer_tablero():
    lector = open('buscaminas.txt')
    tablero = []
    for linea in lector:
        fila = linea.strip().split(",")
        tablero.append(fila)
    lector.close()
    return tablero

def escribir_tablero(tablero):
    escritor = open('buscaminas.txt','w')
    for fila in tablero:
        linea = ",".join(fila)
        print(linea,file=escritor)
    escritor.close()

def esta_descubierta(col, fila):
    tablero = leer_tablero('buscaminas.txt')
    celda = tablero[fila][col]
    if celda[0] == "1":
        descubierta = True
    else:
        descubierta = False
    return descubierta

def descubrir(col, fila):
    tablero = leer_tablero('buscaminas.txt')
    celda = tablero[fila][col]
    if celda[0] == "0":
        nueva_celda = "1" + celda[1]
        tablero[fila][col] = nueva_celda
        escribir_tablero(tablero, "tablero_mod.txt")

def dar_pista(col, fila):
    tablero = leer_tablero('buscaminas.txt')
    bombas = 0

    if fila > 0 and col > 0 and tablero[fila-1][col-1][1] == "1":
        bombas += 1
    if fila > 0 and tablero[fila-1][col][1] == "1":
        bombas += 1
    if fila > 0 and col+1 < len(tablero[0]) and
              tablero[fila-1][col+1][1] == "1":
        bombas += 1
    if col > 0 and tablero[fila][col-1][1] == "1":
        bombas += 1
    if col+1 < len(tablero[0]) and tablero[fila][col+1][1] == "1":
        bombas += 1
```

```
    if col > 0 and fila+1 < len(tablero) and  \
            tablero[fila+1][col-1][1] == "1":
        bombas += 1
    if fila+1 < len(tablero) and tablero[fila+1][col][1] == "1":
        bombas += 1
    if col+1 < len(tablero[0]) and fila+1 < len(tablero) and \
            tablero[fila+1][col+1][1] == "1":
        bombas += 1

    return bombas
```

Solución 2

```
def leer_tablero(nombre_archivo):
    ar = open(nombre_archivo)
    tablero = []
    for l in ar:
        tablero.append(l.strip().split(',’))
    ar.close()
    return tablero

def escribir_tablero(nombre_archivo, tablero):
    ar = open(nombre_archivo, "w")
    for f in tablero:
        for i in range(len(f)):
            ar.write(f[i])
            if i < len(f)-1:
                ar.write(',’)
            else:
                ar.write('\n’)
    ar.close()

def esta_descubierta(x,y):
    tablero = leer_tablero("buscaminas.txt")
    celda = tablero[x][y]
    return celda[0] == '1’

def descubrir(x,y):
    tablero = leer_tablero("buscaminas.txt")
    if not esta_descubierta(x,y):
        tablero[x][y] = '1’ + tablero[x][y][1]
    escribir_tablero("buscaminas.txt", tablero)

def dar_pista(x,y):
    tablero = leer_tablero("buscaminas.txt")
    n_bombas = 0
    alto = len(tablero)
    ancho = len(tablero[0])
    if x-1 >= 0 and y-1 >= 0 and tablero[x-1][y-1][1] == '1’:
        n_bombas += 1
    if x-1 >= 0 and tablero[x-1][y][1] == '1’:
```

```
    n_bombas += 1
if x-1 >= 0 and y+1 < ancho and tablero[x-1][y+1][1] == '1':
    n_bombas += 1
if y+1 < ancho and tablero[x][y+1][1] == '1':
    n_bombas += 1
if x+1 < alto and y+1 < ancho and tablero[x+1][y+1][1] == '1':
    n_bombas += 1
if x+1 < alto and tablero[x+1][y][1] == '1':
    n_bombas += 1
if x+1 < alto and y-1 >= 0 and tablero[x+1][y-1][1] == '1':
    n_bombas += 1
if y-1 >= 0 and tablero[x][y-1][1] == '1':
    n_bombas += 1
return n_bombas
```

Criterios de evaluación problema 3

Aspecto	**No logrado**	**Logrado con observaciones**	**Logrado**
Función `esta_descubierta` (10 puntos)	Sin código, o código tiene errores mayores (0 pts)	Lectura del archivo y obtención de estado de celda (5 puntos)	Lectura del archivo y obtención de estado de celda (10 puntos)
Función `descubrir` (20 puntos)	Sin código, o código tiene errores mayores (0 pts)	▪ Cambio del estado cuando es posible (4 pts) ▪ Actualización estado del tablero en archivo (6 pts)	▪ Cambio del estado cuando es posible (8 pts) ▪ Actualización estado del tablero en archivo (12 pts)
Función `dar_pista` (30 puntos)	Sin código, o código tiene errores mayores (0 pts)	▪ Lectura de si hay o no bomba en una celda (2 pts) ▪ Verificación de bomba en interior de tablero (2 pts) ▪ Verificación de bomba sin traspasar límite superior (2 pts) ▪ Verificación de bomba sin traspasar límite inferior (2 pts) ▪ Verificación de bomba sin traspasar límite izquierdo (2 pts) ▪ Verificación de bomba sin traspasar límite derecho (2 pts) ▪ Contar bombas y retornar suma (2 pts)	▪ Lectura de si hay o no bomba en una celda (5 pts) ▪ Verificación de bomba en interior de tablero (4 pts) ▪ Verificación de bomba sin traspasar límite superior (4 pts) ▪ Verificación de bomba sin traspasar límite inferior (4 pts) ▪ Verificación de bomba sin traspasar límite izquierdo (4 pts) ▪ Verificación de bomba sin traspasar límite derecho (4 pts) ▪ Contar bombas y retornar suma (5 pts)

Problema 4 (60 puntos)

Debes completar el programa que se muestra más abajo. Este programa modela la interacción entre los siguientes objetos: dos **cocineros**, un **mesón**, y dos **mozos** del restorán *Don Yadran*.

Los **cocineros** ponen platos en el **mesón** para que sean retirados por los **mozos**. El mesón tiene un espacio limitado y los **cocineros** no pueden poner más platos cuando el **mesón** está lleno. *Don Yadran* es muy famoso y se especializa en solo dos tipos de platos: "*lomitos*" y "*completos*". El **cocinero1** solo hace "*lomitos*" y el **cocinero2** sólo hace "*completos*".

Los **mozos** sacan platos del **mesón** (si es que los hay) y los ponen en una bandeja distribuidora que los llevan a los hambrientos clientes. El **mozo1** sólo saca del **mesón** platos "*lomitos*" y el **mozo2** sólo saca platos "*completos*".

El siguiente programa ejecuta muchas iteraciones de la interacción entre **cocineros**, **mesón** y **mozos**. En cada iteración, cada **cocinero** prepara y pone en el **mesón** una cantidad aleatoria de (entre cero y **P**) platos. En el programa, **P** vale **6** ("*lomitos*") para **cocinero1** y **8** ("*completos*") para **cocinero2**. Si un cocinero debe poner **x** platos en el **mesón**, pero en éste sólo caben **k**, con **k < x**, sólo pone **k** platos.

En cada iteración, cada **mozo** saca del **mesón** una cantidad aleatoria de (entre cero y **S**) platos. En el programa, **S** vale **5** ("*lomitos*") para **mozo1** y **4** ("*completos*") para **mozo2**. Si un mozo debe sacar **x** platos del **mesón**, pero en éste sólo quedan **k**, con **k < x**, sólo saca **k** platos.

Debes completar este programa escribiendo las clases **Meson**, **Cocinero**, y **Mozo**, asegurando que funcione a cabalidad. No puedes modificar el código presentado. Debes definir las clases, atributos y métodos necesarios. El método `lleno()` retorna `True` si el **mesón** está lleno y `False` en caso contrario. El atributo `faltan` indica la cantidad de platos que no estaban disponibles en el **mesón** para ser retirados por un **mozo** (no se acumulan de una iteración a otra). El atributo `tipo` indica si es "*lomito*" o "*completo*". El atributo `lom` indica la cantidad de "*lomitos*" en el **mesón**. El atributo `comp` indica la cantidad de "*completos*" en el **mesón**.

```
### programa principal ###
meson=Meson(20)                             # crea un mesón de capacidad 20 platos
cocinero1 = Cocinero("lomito",6,meson)      # crea cocinero con máx. 6 lomitos
cocinero2 = Cocinero("completo",8,meson)    # crea cocinero con máx. 8 completos
mozo1=Mozo("lomito",5,meson)                # crea mozo que retira máx. 5 lomitos
mozo2=Mozo("completo",4,meson)              # crea mozo que retira máx. 4 completos

t=0     # ejecuta 50 iteraciones
while t < 50:
    cocinero1.agregaPlatos()
    cocinero2.agregaPlatos()
    print("  Mesón lleno : ", meson.lleno())
    mozo1.retiraPlatos()
    print("  Faltan : ", mozo1.faltan," ",mozo1.tipo)
    mozo2.retiraPlatos()
    print("  Faltan : ", mozo2.faltan," ",mozo2.tipo)
    t = t + 1
    print ("t = "+str(t) + " Meson: " + str(meson.lom) + ", " + str(meson.comp))
# fin de la iteración
```

Ejemplo solución problema 4

```
import random

class Meson:
    def __init__(self, capacidad):
        self.capacidad = capacidad
        self.lom = 0
        self.comp = 0

    def lleno(self):
        return self.capacidad - self.lom - self.comp <= 0

class Mozo:

    def __init__(self, tipo, Max, meson):
        self.tipo = tipo
        self.Max = Max
        self.meson = meson
        self.faltan = 0

    def retiraPlatos(self):
        nPlatos = random.randint(0,self.Max)
        cuenta = 0
        for i in range (0,nPlatos):
            if self.tipo == "lomito" and not self.meson.lom <= 0:
                cuenta += 1
                self.meson.lom -= 1
            elif self.tipo == "completo" and not self.meson.comp <= 0:
                cuenta += 1
                self.meson.comp -= 1
        self.faltan = nPlatos - cuenta

class Cocinero:
    def __init__(self,tipo,Max,meson):
        self.tipo = tipo
        self.Max = Max
        self.meson = meson

    def agregaPlatos(self):
        nPlatos = random.randint(0,self.Max)
        cuenta = 0
        for i in range (0,nPlatos):
            if not self.meson.lleno():
                cuenta += 1
                if self.tipo == "lomito":
                    self.meson.lom += 1
                else:   # self.tipo == "completo"
                    self.meson.comp += 1
```

Criterios de evaluación problema 4

Aspecto	No logrado	Logrado con observaciones	Logrado
Clase `Meson` (10 pts)	Sin código, o código tiene errores mayores (0 pts)	▪ Definición de clase, constructor y atributos (2 pts) ▪ Método `lleno` (2 pts)	▪ Definición de clase, constructor y atributos (5 pts) ▪ Método `lleno` (5 pts)
Clase `Mozo` (25 pts)	Sin código, o código tiene errores mayores (0 pts)	▪ Definición de clase, constructor y atributos (5 pts) ▪ Método `retiraPlatos`: generación de cantidad de platos (2 pts) ▪ Método `retiraPlatos`: actualización de platos retirados según tipo de plato (5 pts)	▪ Definición de clase, constructor y atributos (10 pts) ▪ Método `retiraPlatos`: generación de cantidad de platos (5 pts) ▪ Método `retiraPlatos`: actualización de platos retirados según tipo de plato (10 pts)
Clase `Cocinero` (25 pts)	Sin código, o código tiene errores mayores (0 pts)	▪ Definición de clase, constructor y atributos (5 pts) ▪ Método `agregaPlatos`: generación de cantidad de platos (2 pts) ▪ Método `agregaPlatos`: actualización de platos agregados según tipo de plato (5 pts)	▪ Definición de clase, constructor y atributos (10 pts) ▪ Método `agregaPlatos`: generación de cantidad de platos (5 pts) ▪ Método `agregaPlatos`: actualización de platos agregados según tipo de plato (10 pts)