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

IIC2115 – Programación como Herramienta para la Ingeniería (I/2020)

Laboratorio 3 - Técnicas de programación

Objetivos

- Aplicar variadas técnicas de programación para la resolución eficiente de problemas de programación

Entrega

- **Lenguaje a utilizar:** Python 3.6
- **Lugar:** repositorio privado en GitHub. Recuerde incluir todo en una carpeta de nombre **L03**.
- **Entrega: domingo 3 de mayo a las 23:59 hrs.**
- **Formato de entrega (ES IMPORTANTE EL NOMBRE DEL ARCHIVO):**
  - Archivo python notebook (**L03.ipynb**) con la solución de los problemas y ejemplos de ejecución. Utilice múltiples celdas de texto y código para facilitar la revisión de su laboratorio.
  - Archivo python (**L03.py**) con la solución de los problemas. Este archivo sólo debe incluir la definición de las funciones, utilizando exactamente los mismos nombres y parámetros que se indican en el enunciado, y la importación de los módulos necesarios. Puede crear y utilizar nuevas funciones si lo cree necesario. No debe incluir en este archivo ejemplos de ejecución ni la ejecución de dichas funciones.
  - Todos los archivos deben estar ubicados en la carpeta **L03**. No se debe subir ningún otro archivo a la carpeta. Los archivos **.ipynb** y **.py** deben contener la misma solución.
- **Descuentos:** El descuento por atraso se realizará de acuerdo a lo definido en el programa del curso. Además de esto, tareas que no cumplan el formato de entrega tendrán un descuento de 0.5 ptos.

- **Laboratorios con errores de sintaxis y/o que generen excepciones serán calificadas con nota 1.0.**
- Si su laboratorio es entregado fuera de plazo, tiene hasta el **lunes 4 de mayo a las 23:59 hrs** para responder el formulario de **entregas fuera de plazo** disponible en el Syllabus.
- Las discusiones en las *issues* del Syllabus en GitHub son parte de este enunciado.
- El uso de librerías externas que sean estructurales en la solución del los problemas no podrán ser utilizadas. Solo se podrán utilizar las que han sido aprobadas en las *issues* de GitHub.

Introducción

En este laboratorio deberán solucionar 6 problemas de programación (un problema bonus), que pueden ser resueltos de manera eficiente si utiliza técnicas de programación, en vez de un enfoque de fuerza bruta. Para cada problema, se indicará la técnica que deberán obligatoriamente utilizar en su solución, o se indicará que el problema es de solución libre, y puede utilizar las técnicas que prefiera.

Para obtener el máximo de puntaje por problema, estos deben ser solucionados usando los tópicos indicados en cada uno de ellos. De lo contrario, sólo obtendrán una parte pequeña del puntaje.

Con el fin de facilitar el desarrollo, los problemas han sido agrupados de acuerdo a su dificultad (baja, media y alta). Cada uno de estos grupos tiene el mismo puntaje (2 ptos.). En el apartado de dificultad baja encontrarán 3 problemas, deben seleccionar **SOLO** dos de ellos para resolver como parte de este laboratorio. El tercero puede resolverlo igualmente y podrán obtener hasta 0.5 puntos de BONUS en la escala de puntaje. Para los apartados medio y alto deben resolver todos los problemas.

Problemas

I. Dificultad baja (2.0 ptos.)

a) **Maximización de amplificación (Backtracking):** Considere un sistema óptico formado por una secuencia de L lentes, que tiene como objetivo amplificar un haz de luz. Cada lente tiene un factor de amplificación $f \in \mathcal{N}$, independiente de los del resto de los lentes. Dada la geometría del sistema, la potencia final de amplificación de este se maximiza poniendo los lentes con mayor factor más cerca del lugar donde primero incide el haz en el sistema.

En base a lo anterior, dado un ordenamiento inicial de lentes en el sistema y asumiendo que el haz de luz incide inicialmente en el lente de más a la derecha, escriba un programa que encuentre un

nuevo ordenamiento de los lentes que maximice la amplificación, si sólo es posible realizar como máximo k intercambios de posición entre los lentes.

Un ejemplo de ejecución del algoritmo es el siguiente:

Código

```
def maximizar_amplificación(lentes, k):
    #solucion al problema


lentes = [20, 7, 89, 3, 2, 27]
k = 2
nuevo_ordenamiento = maximizar_amplificación(lentes, k)
print(nuevo_ordenamiento)
```

Salida

```
[20, 7, 2, 3, 27, 89]
```

b) **Planta de revisión técnica (Recursión):** Considere una planta de revisión técnica, cuyo sistema de atención no obedece a una cola, sino a una priorización en base a la ganancia que le genera cada revisión a la planta, y al tiempo que estas tomarán. En este esquema, los vehículos son inscritos en un registro, donde para cada uno se estima el tiempo que tomará su revisión y el valor a cobrar por esta. Luego, el administrador de la planta decide que vehículos serán revisados, buscando maximizar la ganancia, con la restricción que la suma de los tiempos de atención no supere un umbral definido en base a la cantidad de mecánicos disponibles.

En base a lo descrito anteriormente, dada una lista de pares (valor de la revisión, tiempo de atención estimado) para k de vehículos, y la cantidad máxima de tiempo de atención por parte de los mecánicos, escriba un programa que encuentre aquellos vehículos que serán seleccionados por el administrador de la planta, indicando los índices de estos en la lista original

Un ejemplo de ejecución del algoritmo es el siguiente:

Código

```
def seleccionar_vehiculos(valores_y_tiempos, tiempo_total_maximo):
```

```
        #solucion al problema


    valores_y_tiempos = [(20, 1), (5, 2), (10, 3), (40, 8), (15, 7), (25, 4)]
tiempo_total_maximo = 10
    vehiculos = seleccionar_vehiculos(valores_y_tiempos, tiempo_total_maximo)
    print(vehiculos)
```

Salida

```
[0, 3]
```

c) **Colas en salud (Ordenamiento):** En el colapsado sistema de salud de un pueblo, es necesario ingresar a una cola para ser atendido. En tal cola, entre más edad tiene una persona, entonces mayor es la prioridad que tiene esta de ser atendida. Por lo tanto, para saber el real tiempo de atención de una persona, es necesario saber cuantas personas mayores han llegado posteriormente a la cola.

En base a lo descrito anteriormente, se pide determinar, para cada persona, cuantas personas mayores han llegado posterior a ellas en la cola. Considere que recibe una lista con las edades de las personas en el orden que estas han llegado a la cola. Debe retornar una lista con el número de personas mayores que tiene cada una en el resto de la cola. Por ejemplo, si las edades en la cola fueran [20,31,12] la respuesta sería: [1,0,0].

Un ejemplo de ejecución del algoritmo es el siguiente:

Código

```
def mas_adultos(cola_actual):
    #solucion al problema


cola_actual = [5, 7, 4, 10, 8, 11]
mayores = mas_adultos(cola_actual)
print(mayores)
```

Salida

```
[4,3,3,1,1,0]
```

II. **Dificultad media (2.0 ptos.)**

a) **Colisiones aéreas (Dividir y conquistar):** La Agencia Aeronáutica Internacional (AAI) se encarga de velar por la correcta operación del trafico aéreo. Actualmente desea construir una herramienta que alerte cuando se produzca un código *6D2*. Un código *6D2* se activa cuando dos aviones se encuentren próximos a colisionar. Para ello, AAI posee la capacidad de sacar una foto aérea con la posición (x, y) de todos los aviones en el aire. Su misión será identificar a los dos aviones que estén más próximos a chocar entre si (los que tengan la menor distancia euclideana entre ellos). La foto aérea le entrega una lista de tuplas con las posiciones de todos los aviones. Usted debe indicar qué aviones son los que se encuentran más cercanos.

Un ejemplo de ejecución del algoritmo es el siguiente:

Código

```
def codigo_6d2(lista_de_aviones):
    #solucion al problema


lista_de_aviones = [(3, 4), (13, 31), (41, 51), (6, 2), (13, 11), (4, 5)]
colision = codigo_6d2(lista_de_aviones)
print(colision)
```

Salida

```
[(3,4), (4,5)]
```

b) **El muro de los Menominee (Libre):** Ya es conocido que la aldea Ojibwe ha estado en guerra con el pueblo Menominee por muchos años. Los Ojibwe construyeron un muro entre los pueblos para cesar las diferencias. Lamentablemente después de incesantes batallas, el muro fue destruido completamente. Ahora, los Menominee han decidido construir un nuevo muro para separarse de sus enemigos. En esta oportunidad posicionaron una serie bloques de piedra entre los pueblos (no necesariamente unidos). En concreto, utilizaron bloques de distinto ancho y altura, los que fueron dispuestos en una planicie, sobre una línea recta que la divide entre ambas ciudades.

El muro es representado por una lista de tuplas (cada tupla es un bloque) del estilo (**izq**, **alt**, **der**), donde **izq** corresponde a la posición en la horizontal de lado izquierdo del bloque; **alt** corresponde a la altura del bloque y **der** corresponde a la posición en la horizontal de lado derecho del bloque. Por ejemplo, $[(3, 7, 9), (5, 10, 12)]$ es un bloque que comienza en 3, tiene altura 7 y termina en 9; y otro que comienza en 5, tiene altura 10 y termina en 12.

Los ingenieros de los Menominee desean medir la efectividad del muro por medio del perfil de este, es decir, el contorno que se observa observa al mirar los bloques puestos de izquierda a derecha. Este perfil se representa mediante una lista de tuplas (**pos**, **alt**), que muestran los cambios de alturas de izquierda a derecha. **pos** corresponde a la posición en la horizontal donde existe un cambio de altura y **alt** a la altura de ese cambio. Para el ejemplo del párrafo anterior, el perfil sería un muro que comienza en 3 con altura 7 hasta el 5 donde comienza con altura 10 y luego de 12 en adelante tiene altura 0. Esto se representa: $[(3, 7), (5, 10), (12, 0)]$

Escriba un programa que permita calcular el perfil del muro, en base a las tuplas que describen las partes de este. Un ejemplo de ejecución del algoritmo es el siguiente:

Código

```
def perfil_de_muro(bloques):
    #solucion al problema

bloques = [(1, 11, 5), (12, 7, 16), (3, 13, 9), (2, 6, 7), (14, 3, 25),
     (19, 18, 22), (23, 13, 29), (24, 4, 28)]

perfil = perfil_de_muro(bloques)
print(perfil)
```

Salida

```
[(1, 11),  (3, 13),  (9, 0),  (12, 7),  (16, 3),  (19, 18),
(22, 3),  (23, 13),  (29, 0)]
```

III. Dificultad alta (2.0 ptos.)

a) **Reorganización de habitaciones (A^*):** Considere una habitación rectangular de N×M m^2 (met-

ros cuadrados), donde cada uno de los muebles en su interior es distinto y utiliza 1 m^2. Con el fin de utilizar de mejor manera el espacio, se busca reorganizar el posicionamiento de los muebles, sujeto a ciertas restricciones de movimiento de estos y a su posición final. Asumiendo que los muebles utilizarán en conjunto como máximo (N×M)-1 m^2, escriba un programa que dado un estado inicial y uno final de la habitación, encuentre la cantidad mínima posible de movimientos de muebles para hacer el cambio de disposición deseado. Considere que: i) los muebles solo pueden moverse de a uno, ii) únicamente a un espacio vacío, y iii) no pueden realizarse intercambios entre la posición de los muebles.

Un ejemplo de ejecución del algoritmo es el siguiente:

Código

```
def reorganizar_habitacion(estado_inicial, estado_final):
    #solucion al problema

estado_inicial = [[1,2,'_',3 ],
                  [4,5, 6 ,7 ],
                  [8,9,'_',10]]

estado_final = [[1 ,5  ,9, 4 ],
                [7 ,'_',6, 2 ],
                [10,3  ,8,'_']]

movimientos = reorganizar_habitacion(estado_inicial, estado_final)
print(movimientos)
```

Salida

```
33
```

b) **Terrenos en arriendo (Libre):** Un antiguo granjero ha decidido arrendar su propiedad rectangular para que otras personas puedan construir graneros en ella. Para maximizar sus ganancias, ha decidido subdividir su propiedad en pequeños retazos donde él espera que sus arrendatarios

construyan los graneros. Además, con fines de transporte, el granjero ha dispuesto de una serie de caminos que conectan algunos retazos. Dado que el granjero es muy mayor, el espera lidiar con la menor cantidad de arrendatarios posible, pero quiere evitar que una misma persona arriende retazos que se encuentren conectados por un camino de forma directa, para evitar que los transformen en un gran retazo. Por lo tanto, el granjero desea conocer como podría arrendar sus retazos a la menor cantidad de personas posibles, con la restricción planteada, dada una subdivisión del terreno y caminos que los conectan.

En base a lo descrito anteriormente, escriba un programa que permita conocer la asignación mínima de arrendatarios sin que estos arrienden terrenos conectados directamente. El *input* está compuesto por el numero de retazos del terreno y cómo estos se conectan. Por ejemplo, si fueran 3 retazos (1, 2 y 3), y los caminos existentes son del retazo 1 al 2 y del 1 al 3, el *input* sería: 3 y [(1,2), (1,3)]. Su respuesta debe entregarla como una lista de tuplas donde los retazos estén ordenados de menor a mayor y agrupados por arrendatario, para el ejemplo sería: [(1), (2,3)]. El retazo 1 para un arrendatario y el 2 y 3 para otro. Además ordenados de menor a mayor.

Un ejemplo de ejecución del algoritmo es el siguiente:

Código

```
def minimos_arrendatarios(retazos, caminos):
    #solucion al problema


retazos = 6
caminos = [(1, 2), (1, 5), (1, 6), (5, 6), (2, 5), (2, 4), (3, 4), (3, 5)]


asigancion = minimos_arrendatarios(retazos, caminos):


print(asigancion)
```

Salida

```
    [(1,3), (2,6), (4,5)]
```

Corrección

Para la corrección de este laboratorio, se revisarán dos ítems por problema, cada uno con igual valor en la nota (50%). El primero serán los contenidos y mecanismos utilizados para resolver cada uno de los problemas propuestos. De este modo, es importante que comente correctamente su código para que sea más sencilla la corrección. Recuerde que debe utilizar las estructuras de datos adecuadas a cada problema. Además, se evaluará el orden de su trabajo.

El segundo ítem será la correctitud de los resultados y el tiempo de ejecución de las soluciones entregadas. Por cada problema se probarán distintos valores de entrada, y para cada uno de estos se evaluará su correctitud. Además de esto, el tiempo de ejecución de sus algoritmos no debe sobrepasar el indicando en el enunciado para cada problema. Por lo tanto, para cada problema, si el resultado entregado por el algoritmo no es siempre correcto, o si el tiempo de ejecución supera el máximo indicado, su solución no obtendrá puntaje en este ítem.

Política de Integridad Académica

"Como miembro de la comunidad de la Pontificia Universidad Católica de Chile me comprometo a respetar los principios y normativas que la rigen. Asimismo, prometo actuar con rectitud y honestidad en las relaciones con los demás integrantes de la comunidad y en la realización de todo trabajo, particularmente en aquellas actividades vinculadas a la docencia, el aprendizaje y la creación, difusión y transferencia del conocimiento. Además, velaré por la integridad de las personas y cuidaré los bienes de la Universidad."

En particular, se espera que mantengan altos estándares de honestidad académica. Cualquier acto deshonesto o fraude académico está prohibido; los alumnos que incurran en este tipo de acciones se exponen a un procedimiento sumario. Ejemplos de actos deshonestos son la copia, el uso de material o equipos no permitidos en las evaluaciones, el plagio, o la falsificación de identidad, entre otros. Específicamente, para los cursos del Departamento de Ciencia de la Computación, rige obligatoriamente la siguiente política de integridad académica en relación a copia y plagio: Todo trabajo presentado por un alumno (grupo) para los efectos de la evaluación de un curso debe ser hecho individualmente por el alumno (grupo), sin apoyo en material de terceros. Si un alumno (grupo) copia un trabajo, se le calificará con nota 1.0 en dicha evaluación y dependiendo de la gravedad de sus acciones podrá tener un 1.0 en todo ese ítem de evaluaciones o un 1.1 en el curso. Además, los antecedentes serán enviados a la Dirección de Docencia de la Escuela de Ingeniería para evaluar posteriores sanciones en conjunto con la Universidad, las que pueden incluir un procedimiento sumario. Por "copia" o "plagio" se entiende incluir en el trabajo presentado como propio, partes desarrolladas por otra persona. Está permitido usar material disponible públicamente, por ejemplo, libros o contenidos tomados de Internet, siempre y cuando se incluya la cita correspondiente.