IIC2115 – Programación como Herramienta para la Ingeniería (I/2019)

Ejercicios Capítulo 3

1. ¿Es posible que A^* expanda todos los nodos de un grafo para llegar al destino?

   a) Sólo si la heurística no es admisible.

   b) Depende de la estructura del grafo.

   c) Sólo si la heurística no es admisible y el grafo tiene conexiones entre todos los nodos.

   d) No, nunca expandirá todos los nodos.

2. ¿Cuál es la salida del siguiente código?

```
def test(i,j):
    if(i==0):
        return j
    else:
        return test(i-1,i+j)
print(test(4,7))
```

   a) 7

   b) 13

   c) 17

   d) El algoritmo no termina.

3. Considere el algoritmo de ordenamiento **min-max sort**, que opera de la siguiente manera: dado un arreglo de números enteros, busca el máximo y el mínimo, y lo intercambia por los elementos ubicados en los extremos. Luego, se llama a si mismo de manera recursiva, con el fin de ordenar el subarreglo que no contiene los valores de los extremos. ¿Cuál es la complejidad de ejecución de min-max sort?

   a) $\mathcal{O}(\log n)$

   b) $\mathcal{O}(n)$

   c) $\mathcal{O}(n \log n)$

   d) $\mathcal{O}(n^2)$

4. ¿Cuál es la complejidad de ejecución del siguiente algoritmo?

```
def sort(array):
    for slot in range(1, len(array)):
        value = array[slot]
        test_slot = slot - 1
        while test_slot > -1 and array[test_slot] > value:
            array[test_slot + 1] = array[test_slot]
            test_slot = test_slot - 1
        array[test_slot + 1] = value
    return array
```

a) $\mathcal{O}(n^2)$

b) $\mathcal{O}(n \log n)$

c) $\mathcal{O}(n)$

d) $\mathcal{O}(\log n)$

5. ¿Cuántas llamadas a la función `fib` se realizan al ejecutar el siguiente código?

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)

fib(5)
```

a) 13

b) 14

c) 15

d) 16

6. ¿Puede usarse *backtracking* para calcular todas las permutaciones posibles de un arreglo de caracteres?

a) No, no es posible hacer en ningún caso.

b) Sólo si el arreglo (lista) está previamente ordenado.

c) Sólo si la cantidad de caracteres es impar.

d) Sí, siempre. El código está incluso disponible en el Syllabus.

7. ¿Se puede usar BFS y/o DFS para decidir si hay un camino entre dos vértices de un grafo?

   a) Ninguno

   b) Sólo BFS

   c) Sólo DFS

   d) Ambos

8. ¿Cuál de las siguientes complejidades **no** es $\mathcal{O}(n^2)$?

   a) $n^{1.98}$

   b) $(2^{20}) \times n$

   c) $\frac{n^3}{\sqrt{n}}$

   d) $(15^{10}) \times n + 12099$

9. ¿Qué algoritmo de ordenamiento está implementado en el siguiente código?

```
def sort(arr,n):
    if n<=1:
        return
    sort(arr,n-1)
    last = arr[n-1]
    j = n-2
    while (j>=0 and arr[j]>last):
        arr[j+1] = arr[j]
        j = j-1
    arr[j+1]=last
```

   a) Mergesort

   b) Insertion sort

   c) Quicksort

d) Bubble sort

10. Sea h_1 una heurística admisible para A*, y h_2 una no admisible. ¿Cuál de las siguientes es una heurística admisible?

    a) $avg\{h_1, h_2\}$

    b) $max\{h_1, h_2\}$

    c) $min\{h_1, h_2\}$

    d) Ninguna de las anteriores

11. ¿Cuál es la complejidad del siguiente algoritmo?

```
i = n
while i > 0:
    i = int(i/2)
```

    a) $\mathcal{O}(n)$

    b) $\mathcal{O}(\log n)$

    c) $\mathcal{O}(\sqrt{n})$

    d) $\mathcal{O}(\frac{n}{2})$

12. Suponga que ordena una arreglo de ocho enteros con Quicksort y luego de la primera partición, este tiene el siguiente aspecto: 2 5 1 7 9 12 11 10 ¿Cuál de las siguientes aseveraciones es correcta?

    a) Ni 7 ni 9 pueden ser pivotes.

    b) El pivote puede ser 7, pero no 9

    c) El pivote puede ser 9, pero no 7

    d) El pivote puede ser 7 ó 9

13. ¿Qué calcula el siguiente código?

```
def f(x, y):
    if y == 0:
        return 0
    return x + f(x, y-1)
```

a) $x \times y$

b) $x + y$

c) x^y

d) $x + x \times y$

14. Considere un mapa de casillas hexagonales construido sobre una esfera, donde algunas casillas no pueden visitarse por tener obstáculos. Si se busca el camino más corto entre dos casillas arbitrarias usando A*, ¿cuál de las siguientes heurísticas dará en promedio mejor rendimiento?

    a) Distancia euclidiana (a través del cuerpo de la esfera)

    b) Distancia geodésica (línea recta sobre la esfera)

    c) Constante igual al radio de la esfera

    d) Constante igual a cero

15. Describa un procedimiento recursivo que permita obtener la profundidad de un árbol binario, *i.e.*, el camino más largo desde el nodo raíz hasta un nodo hoja. Puede entregar su respuesta como una descripción textual, código en Python o pseudocódigo.

16. Programe un algoritmo recursivo que calcule el máximo común divisor entre el factorial de dos números. El input serán dos números y el output debe ser el máximo común divisor entre el factorial de cada uno. Para este ejercicio **no se pueden** usar ciclos `for` ni `while`.

    Ejemplo:

    Input: 4 3

    Output: 6

    Consideremos que 4! es 24 y 3! es 6. El máximo común divisor entre ambos es 6, por lo que este es nuestro output.

17. Un número perfecto es un número natural que es igual a la suma de sus divisores propios positivos. Así, 6 es un número perfecto porque sus divisores propios son 1, 2 y 3; y $6 = 1 + 2 + 3$. Escriba un programa que reciba como input un número y que entregue de output todos los números perfectos hasta ese número. En caso de no haber, retornar 0. Para este ejercicio **no se pueden** usar ciclos `for` ni `while`.

    Ejemplo:

    Input: 10

Output: 6

Opcional: Intentar mejorar el algoritmo guardando estados, es decir, en vez de calcular divisores constantemente, usar los ya calculados previamente para no volver a hacer el cálculo.

18. En este problema deberán resolver un laberinto. El input del problema es una lista de listas (matriz) con la representación del laberinto en donde las paredes se mostrarán como "X", los caminos se mostrarán como "0" y la salida estará marcada con una S. El laberinto siempre comenzará en la esquina superior izquierda (posición (0,0)). El output del problema debe ser una lista de tuplas en donde se muestre el camino a seguir para resolver el laberinto.
    Ejemplo:

    Input: `[[0,X,0,X],[0,0,0,X],[0,X,0,X],[X,X,S,X]]`
    Output: `[(0,0),(1,0),(1,1),(1,2),(2,2),(3,2)]`

19. En este ejercicio deberá asignar a cada letra un dígito de 0 a 9, para que la suma de dos palabras tenga sentido numérico. Las únicas reglas son que cada asignación de una misma letra debe tener el mismo dígito asignado, y un dígito no puede estar asignado a más de una letra. El input serán tres palabras, las tercera debe ser la suma de las dos primeras.
    Ejemplo:
    Input: SEND MORE FOOD
    Output: 1034 7520 8554
    Explicación:

    $$1034 + 7520 = 8554$$
    $$\text{SEND} + \text{MORE} = \text{FOOD}$$
    $$\text{S}=1,\ \text{E}=0,\ \text{N}=3,\ \text{D}=4,\ \text{M}=7,\ \text{O}=5,\ \text{R}=2,\ \text{F}=8.$$

20. Bastián acaba crear una nueva lista llamada `BastianList`, que no soporta los métodos `len` o `sum`. Por eso te pide que crees dos funciones recursivas (`largo` y `suma`) que calculan el largo y suma, respectivamente, de esta nueva lista. A continuación se muestra el código que debes copiar y completar:

```
class BastianList(list):
    def __init__(self, *args):
        for elem in args:
            self.append(elem)
```

```
    def __len__(self):
        raise ValueError("Metodo no permitido")

    def __iter__(self):
        raise ValueError("Metodo no permitido")

def largo(lista):
    # Completar
    pass

def suma(lista):
    # Completar
    pass

lista = BastianList(1,2,34,4,5,6,7,8)
print(suma(lista))
print(largo(lista))
```

21. Actualmente, dispones de un computador con un particular problema, **no puede multiplicar números de más de 2 dígitos que sean diferentes a potencias de 10**. Por ejemplo, no es capaz de calcular $213 \cdot 213$, $101 \cdot 2$, $3 \cdot 367$, pero sí $2 \cdot 23$, $99 \cdot 25$ o $23 \cdot 1000$ (el segundo número es potencia de 10).

    En este ejercicio deberán implementar un algoritmo que permita multiplicar números grandes usando esta desventaja. Para esto deben basarse en el *algoritmo de Karatsuba* con una base B = 10 para descomponer de forma recursiva el número y lograr su multiplicación. Para comprobar que se cumplen los requisitos, está **prohibido ocupar * para multiplicar**, por lo qeu se debe utilizar la siguiente función:

```
def mul(a, b):
    if (b % 10 != 0 and b > 100) or (a % 10 != 0 and a > 100):
         raise ValueError("No puedo procesar esto")
    return a*b

print(mul(123123123,1231233312))
```

22. Bastián tiene una caja de pasteles cuadrada donde caben $n = m \times m$ pasteles. El estado de cada uno de estos puede ser podrido (representado con una P) o comestible (representado por una C). En

un día, los pasteles podridos hacen que todos sus vecinos sanos se pudran, y esto sigue hasta que eventualmente todos los pasteles de la caja se pudran. Dado un conjunto de $n = m \times m$ estados ingresados por el usuario (uno por cada pastel de la caja), encuentra el número de días en que todos los pasteles de la caja se pudren.

Ejemplo

Input: C C C C P C C C C #Caja con 3×3 = 9 pasteles, donde el elemento central está podrido

Output: 1 # En un día todos se pudren

Restricción: Considere que en el estado inicial, más de un pastel puede estar podrido. Su solución debe tener complejidad menor a $\mathcal{O}(n^2)$, en donde n representa el número total de pasteles del problema. No está permitido usar una simulación para solucionar el problema, e ir calculando día por día iterativamente.

23. Actualmente Antonio (un supuesto amigo de Bastián) se encuentra trabajando como cajero en su parque de diversiones y tiene un gran dilema: le cuesta **calcular la cantidad mínimas de monedas** que debe dar cuando entrega vuelto a los clientes. Es por esto que le solicitó a su supuesto amigo que te pidiera una función de backtracking que resuelva su problema.

    Para resolver lo anterior, tu función recibirá un diccionario y un entero. El diccionario tendrá de *key* las diferentes monedas existentes en la caja del parque, de *value* la cantidad que se dispone de esa moneda, mientras que el entero corresponde al vuelto que se quiere dar. Como resultado, deberás entregar un diccionario donde las *keys* son las diferentes monedas y el *value* es la cantidad que se debe entregar de esa moneda. Debes tener precuación de no dar más o menos vuelto de lo solicitado y que la cantidad de monedas entregadas sea mínima[1]. Suponiendo que tu función se llama `dar_vuelto`, un posible código de esta actividad será:

```
monedas = {1000: 1, 2000: 2, 500: 3, 100: 8, 50: 2, 10:5}
vueltos = [3610, 2610, 3610, 4610, 5610]
for vuelto in vueltos:
    print(dar_vuelto(monedas, vuelto))
```

24. Un hospital en HansLandia tiene un curioso sistema para decidir las prioridades en su servicio de urgencia: no importa cuán grave está el sujeto en cuestión, sino solo su edad; es decir, las personas

[1] Porque a nadie le gusta andar con un millón de monedas de 10, ¿cierto?

más jovenes serán atendidas primero. Lo curioso del hospital no termina en este sistema, pues su lista de espera está en forma de una lista ligada, donde la información contenida en cada nodo corresponde al nombre y la edad de las personas.

Se te pide entonces que ordenes esta lista ligada para que, dada una lista de espera cualquiera, se retorne una lista ligada de pacientes cuyas edades estén en orden creciente. La lista ligada que recibirás de input corresponde a la estructura que aparece en el material de clases. **Importante**: está prohibido cambiar la estructura de datos a ordenar.

25. Decides ir con un grupo de amigos a una fonda para celebrar las Fiestas Patrias. En la fonda, hay abundancia de choripanes, anticuchos, empanadas y terremotos por toda la zona. El grupo se pone de acuerdo para poder consumir lo más posible ese día.

    Instrucciones

    Todo el grupo se encontraba en la fonda buscando que comer, y encontraron un mapa de todo el recinto con instrucciones que detallaban como era aquella zona.

    (a) Espacio transitable. Se puede caminar sobre él. Se representa con un "_".

    (b) Obstáculo. No se puede pasar sobre el. Se representa con una "X".

    (c) Choripán: Se representa con la letra "C" y tiene 500 calorías.

    (d) Anticucho: Un anticucho tiene 300 calorías. Se representa con la letra "A".

    (e) Empanada: Una empanada tiene 600 calorías. Se representa con la letra "E".

    (f) Terremoto: Esta delicia tiene 200 calorías. Se representa con la letra "T"

    Al estar todos bastante cansados, pero con las ganas intactas de comer, deciden que lo mejor es separarse para explorar la fonda de manera más efectiva y eficiente. Como eres el único del grupo que conoce sobre técnicas de programación, tu misión es proponer un algoritmo que permita, dado un mapa[2], generar rutas para cada uno de tus compañeros, de manera que entre ellos recolecten la totalidad de comida en la fonda.

    Es importante notar que, por limitaciones humanas (y criterio moral), tus compañeros no pueden consumir más de 4.000 calorías. Por suerte, el mapa les informa sobre el valor total de calorías que están repartidas en la fonda y este resulta ser menor o igual a la capacidad de consumo de los compañeros,

[2]Representado por una lista de listas.

por la cantidad de compañeros (*i.e.* si hay m compañeros, el mapa tendrá $4000m$ calorías).

Concretamente, debes crear un programa que reciba una cantidad de compañeros (desde ahora llamados "amigos") y un mapa (como una lista de listas con los símbolos descritos previamente) y que haga lo siguiente:

- **Asigne a cada amigo una posición aleatoria** (pero válida) en el mapa, desde la cual comenzar a explorar. Cabe aclarar que dos amigos pueden estar en la misma posición. Los compañeros se representan con la letra G mayúscula[3] y un número natural positivo (*e.g.* "G1" para el primer amigo). Luego de asignar las posiciones iniciales, se debe imprimir el estado actual del mapa en pantalla. En relación a esto último, cuando hay más de un amigo en una casilla, sus nombres se separan por comas (*e.g.* "G1,G6,G10" para los amigos 1, 6 y 10 en una misma casilla). Es importante que el formato visual del mapa sea claro, entendible y mantenga correctamente el alineamiento de filas y/o columnas.
- **Otorgue un turno a cada fiestero**, de manera tal que en un determinado momento, no pueda haber más de un amigo moviéndose. A la secuencia de turnos de todos los amigos se le denomina *episodios*. El movimiento funciona igual para todos los amigos: en su turno se pueden mover solo una casilla hacia arriba, hacia abajo, hacia la derecha o hacia la izquierda (solo cuando no hayan obstáculos). Además, los que no pueden seguir comiendo, se quedan en la última posición utilizada. Luego de cada episodio, **se debe preguntar si se desea mostrar en pantalla el estado actual del mapa o sólo mostrar el estado final, imprimiéndolo de forma ordenada**[4] (si se solicita la segunda opción, sólo debe imprimir el estado final una vez).
- Genere las rutas necesarias para conseguir la comida de cada amigo, en base a las posiciones iniciales definidas anteriormente. Al finalizar todo el proceso (todos los episodios), se espera que se imprima en pantalla, por cada amigo, una lista de tuplas (x, y), donde la K-ésima tupla representa la posición del mapa en que estuvo el amigo en el episodio K. Además, se deben guardar en una lista la comida obtenida (choripanes, anticuchos, empanadas, y terremotos), manteniendo el orden en que estos fueron encontradas.

Búsqueda

El procedimiento de búsqueda la comida de cada compañero debe ser óptimo. Para ello, se te solicita

[3] Guatón Loyola.

[4] Al final de este enunciado se encuentran ejemplos de mapa

que utilices el algortimo de A* con cada uno de tus amigos, los que se moverán una casilla por turno (siguiendo el orden de turnos otorgado anteriormente). Debes priorizar las comidas que se encuentren más cercanas, ya que si se demoran mucho en llegar a una comida, otro de tus amigos podría quitársela (camarón que se duerme, se lo lleva la corriente). En caso de que ocurra esto, el amigo que se quedó sin comida debe volver a trazar una nueva ruta.

Con el fin de comparar rendimiento, deberá utilizar también los algoritmos de BFS y DFS para encontrar la comida, así que antes de comenzar el programa, deberá preguntar al usuario que lo utiliza qué algoritmo desea utilizar (es decir, el proceso de búsqueda se realizará de 3 formas distintas dependiendo del modo que ingrese el usuario)

Ordenamiento

Una vez resuelto el problema de búsqueda de comida, deberán ejecutar un algoritmo que ordene la lista de comidas encontradas **de manera estable**, en orden ascendente (en base al valor calórico de cada comida), mediante el siguiente procedimiento recursivo, popularmente conocido como **HuaSort**:

(a) Dividir la lista en sublistas, de la forma más equilibrada posible (tamaño de cada sublista).

(b) Ordenar las sublistas usando el mismo algoritmo.

(c) Juntar todas las sublistas ordenadas de forma tal de obtener los elementos de la lista inicial ordenados.

Ejemplos de mapa

Si quieres puedes utilizar estos mapas para probar tu código, pero para usarlos debes remover los compañeros y poner los que necesites.

Ejemplo 1

Nótese que el compañero G1 está en la misma celda que un Choripán en la esquina inferior derecha. Cuando un compañero llega a una celda con comida, se separan también por comas.

```
[['E', '_', 'G2', '_', '_', 'X', 'C'   ],
 ['_', '_', '_' , '_', 'E', 'X', '_'   ],
 ['X', 'X', 'X' , 'X', '_', 'X', '_'   ],
 ['_', 'T', '_' , 'X', '_', 'X', '_'   ],
 ['_', '_', '_' , 'X', '_', 'X', '_'   ],
 ['_', 'X', '_' , '_', '_', '_', '_'   ],
 ['_', 'X', '_' , '_', '_', '_', 'C,G1']]
```

Ejemplo 2

```
[['C','_', 'E', 'X' , '_', '_','C', 'X', '_', 'C', 'T' , 'X', '_'],
 ['_','X', '_', 'X' , 'A', 'X','_', 'X', '_', 'X', '_' , 'X', 'C'],
 ['_','X', 'C', 'X' , '_', 'X','C', 'X', '_', 'X', 'G2', 'X', '_'],
 ['C','X', '_', 'G1', '_', 'X','_', '_', 'T', 'X', '_' , '_', 'C']]
```