Tecnología Digital 1: Introducción a la Programación - TP2

Autores: Leis, Guadalupe

Fainburg, Lara

Finck, Violeta

Justificación de por qué los programas tienen la complejidad requerida

Método d.tamano()**:**

```python
def tamano(self) -> int: #O(1)
    '''Devuelve la cantidad de campanas verdes'''
    cantidad:int= len(self.campanas) #O(1)
    return cantidad #O(1)
```

Este método debe tener complejidad O(1) en el peor caso. En nuestra función utilizamos len(self.campanas) la cual calcula la longitud de una lista, como vimos en clase, Python mantiene internamente un contador del número de elementos en la lista, por lo que simplemente devolver ese valor es una operación de tiempo constante. Luego realizamos una asignación de variables lo cual también tiene O(1). Debajo de esa línea utilizamos un return, que devuelve la variable cantidad, lo cual también tiene O(1). Como es una secuencialización toma el máximo de esas dos líneas: O(1) + O(1) = O(max(1,1) = O(1)

Método barrio():

```python
def barrios(self) -> set: #O(N*B)
    '''Devuelve el conjunto de todos los barrios existentes'''
    barrios:set[str]=set() #O(1)
    for campana in self.campanas: #O(N)
        barrios.add(campana.barrio) #O(B)
    return barrios #O(1)
```

Este método debe tener complejidad O(N*B) en el peor caso, donde N es la cantidad de campanas verdes
y B la cantidad de barrios.
Primero creamos un set, como es un set vacío, tiene O(1), además de asignarlo a la variable O(1). Luego la condición del ciclo tiene O(N) ya que va iterando campana en el self.campanas, siendo N la cantidad de campanas. Dentro del ciclo a el set vacío le appendeamos el barrio de cada campana, lo cual tiene un orden O(B) siendo B la cantidad de barrios que se extraen de cada campana. Por último el return de la variable barrios tiene O(1). Cómo utilizamos un ciclo debemos calcular los órdenes: O(N*B)= O(N*B). Por último debemos sacar el máximo con secuencialización: O(1) + O(N*B) + O(1) = O(max(1,N*B,1))= O(N*B).

Método **campanas_del_barrio()**:

```python
def campanas_del_barrio(self,barrio) -> list: #O(N)
    '''Devuelve una lista con las campanas verdes que
    están en el barrio indicado'''
    campanas:list[CampanaVerde]=[] #O(1)
    for campana in self.campanas: #O(N)
        if campana.barrio == barrio: # O(1)
            campanas.append(campana) # O(1)
    return campanas # O(1)
```

Este método debe tener complejidad temporal O(N) en el peor caso, donde N es la cantidad de
campanas verdes. Primero creamos una lista vacía que tiene O(1), y la asignamos a una variable que tiene O(1). Luego la condición del ciclo, itera sobre self.campanas, que el iterar sobre elementos tiene O(N). El condicional compara dos str, pero como los str están acotados en su longitud (y no crecen arbitrariamente lo cual lo haría O(N)), tiene una complejidad constante, O(1). Luego se le appendea la campana a la lista de campanas, lo cual tiene O(1). Por último retornamos la variable campanas. En el condicional debemos calcular la complejidad,: O(1)+O(1)= O(max(1,1)=O(1). Calculamos el ciclo: O(N) * O(1) = O(N*1) = O(N). Por último calculamos la secuencialización: O(1) + O(N) + O(1) =O(max(1,N,1))=O(N)

Metodo **cantidad_por_barrio ()**

```python
def cantidad_por_barrio(self, material) -> dict[str,int]: #O(N*B)
    '''Devuelve un diccionario que indica la cantidad
    de campanas verdes en cada barrio en las que se
    puede depositar el material indicado'''
    cant_campanas_verdes:dict[str,int]={} #O(1)
    for campana in self.campanas: #O(N)
        if material in campana.materiales: #O(B)
            barrio:str= campana.barrio #O(1)
            if barrio in cant_campanas_verdes: #O(1)
                cant_campanas_verdes[barrio]=cant_campanas_verdes [barrio] + 1 #O(1)
            else:
                cant_campanas_verdes[barrio]=1 #O(1)
    return cant_campanas_verdes #O(1)
```

Este método debe ser O(N*B) en el peor caso. Primero inicializamos un diccionario vacío, el cual tiene O(1) y asignar la variable tiene O(1). La condición del ciclo tiene O(N), ya que itera en todos los elementos de self.campanas. Dentro del bucle, if material in campana.materiales: verifica si el material está en la lista de materiales (campana.materiales) de la campana actual, lo cual depende de los elementos de la lista materiales y tiene O(B). Luego asignación de variable barrio tiene O(1). Luego el nuevo condicional, verifica si el barrio está en el diccionario, tiene O(1) ya que barrio no crece con cada iteración. Después, cant_campanas_verdes[barrio] += 1 es una operación O(1) que incrementa el contador de campanas en el barrio especificado. Lo mismo sucede con inicializar el contador en la línea siguiente tiene O(1). Por último en el return devolver la variable tiene O(1). En los condicionales debemos calcular la complejidad O(1)+O(1) +O(1) = O(max(1,1,1))=O(1). Estos condicionales están dentro de otra condición: O(B)+O(1) = O(max(B,1)=O(B). Luego el ciclo que engloba el código: O(N) *O(B)*=O(N*B)= O(N*B). Por último realizamos la secuencialización: O(1)+O(N*B)+O(1)= O(max(1,N*B,1))= O(N*B)

Método **campanas_del_barrio(barrio)**

```
def tres_campanas_cercanas(self, lat, lng) -> tuple:
    '''Devuelve una tupla con las 3 campanas verdes más cercanas al punto ingresado'''
    if len(self.campanas) < 3: # O(1)
        return tuple(self.campanas)  # O(1)

    trescampanas:list [CampanaVerde,CampanaVerde,CampanaVerde]= [self.campanas[0], self.campanas[1], self.campanas[2]] # O(1)
    if trescampanas[0].distancia(lat, lng) > trescampanas[1].distancia(lat, lng): # O(1)
        trescampanas[0], trescampanas[1] = trescampanas[1], trescampanas[0] # O(1)
    if trescampanas[1].distancia(lat, lng) > trescampanas[2].distancia(lat, lng):# O(1)
        trescampanas[1], trescampanas[2] = trescampanas[2], trescampanas[1]# O(1)

    for d in self.campanas[3:]: # O(N)
        if d.distancia(lat, lng) < trescampanas[2].distancia(lat, lng): # O(1)
            trescampanas[2] = d # O(1)
            if trescampanas[2].distancia(lat, lng) < trescampanas[0].distancia(lat, lng): # O(1)
                trescampanas[0], trescampanas[2] = trescampanas[2], trescampanas[0] # O(1)
            elif trescampanas[2].distancia(lat, lng) < trescampanas[1].distancia(lat, lng): # O(1)
                trescampanas[1], trescampanas[2] = trescampanas[2], trescampanas[1] # O(1)
    return tuple(trescampanas) # O(1)
```

Este método debe tener complejidad temporal O(N). Primero verifica si hay menos de 3 campanas en self.campanas es una operación O(1). Si hay menos de 3 campanas, return tuple(self.campanas) devuelve todas las campanas en una tupla, lo cual es una operación O(1). Después inicializamos una lista una lista con las primeras tres campanas, es O(1). Los bloques if (if trescampanas[0].distancia(lat, lng) > trescampanas[1].distancia(lat, lng): y if trescampanas[1].distancia(lat, lng) > trescampanas[2].distancia(lat, lng):) son operaciones que realizan un intercambio si es necesario para asegurar que trescampanas esté en orden de distancia creciente, ambas son O(1). Luego el ciclo (for d in self.campanas[3:]:)itera sobre las campanas restantes en self.campanas, comenzando desde la cuarta campana hasta el final, realizar esto dependerá de la cantidad de elementos, por lo que iterará aproximadamente N-3 veces, por lo tanto, tiene una complejidad O(N). Dentro del ciclo, las comparaciones if d.distancia(lat, lng) < trescampanas[2].distancia(lat, lng): y las asignaciones trescampanas[2] = d son operaciones O(1). Las operaciones dentro de los bloques if y elif que reordenan trescampanas también son operaciones O(1) cada una. Fuera del ciclo, return tuple(trescampanas) es una operación O(1). Para todos los condicionales debemos calcular la complejidad: O(1)+O(1)+O(1)+O(1)= O(max(1,1,1,1)=O(1). Sin embargo, el último condicional está dentro de un ciclo: O(N)*O(1)=O(N*1)=O(N). Por último calculamos la secuencialización: O(1)+O(1)+O(1)+O(N)+O(1)=O(max(1,1,1,N,1)=O(N).

Metodo cantidad_por_barrio(material):

```
def exportar_por_materiales(self, materiales,archivo_csv) -> None:
    '''genera un nuevo archivo con
    nombre archivo_csv que contiene las campanas verdes en el dataset d en las que se pueda
    depositar todos los materiales del conjunto materiales, conjunto indicado como input del
    método. El archivo generado contiene únicamente las columnas DIRECCION y BARRIO .
    '''
    f:TextIO= open(archivo_csv, 'w', encoding= "utf-8") #O(1)
    columnas:str ='DIRECCION,BARRIO\n' #O(1)
    f.write(columnas) #O(1)
    for campana in self.campanas: #O(N)
        if materiales & campana.materiales == materiales: #O(B)
            campanas:str=str({campana.direccion}) +',' + str({campana.barrio}) +'\n' #O(1)
            f.write(campanas) #O(1)
    f.close #O(1)
```

Este método no es requerido con ningún orden específico. Primero, abrimos un archivo para escribir, esta operación es O(1). Después, realizamos una asignación de variable, que es O(1). Luego, f.write(columnas) es una operación O(1), que escribe una línea de texto al archivo. Dentro del ciclo, (if materiales & campana.materiales == materiales: verifica si el conjunto materiales está contenido en campana.materiales. La operación “and” (&) entre dos conjuntos tiene una complejidad O(B), donde B es el número de materiales en campana.materiales. La comparación de conjuntos es una operación O(1). La línea campanas: str = str({campana.direccion}) + ',' + str({campana.barrio}) + '\n' es una operación O(1), ya que convierte en str a la dirección y al barrio. Después, f.write(campanas), escribe una línea al archivo, es O(1). La condición del ciclo, itera sobre todas las campanas en self.campanas, como self.campanas tiene N elementos, este bucle

tiene una complejidad O(N). Por último, f.close(), cierra el archivo, es O(1). Primero debemos calcular la complejidad de el condicional: O(B)+O(1)+O(1)= O(max(B,1,1)=O(1). Luego calculamos la complejidad del ciclo: O(N)*O(B)=O(N*B)=O(N*B). Por último calculamos la secuenciación: O(1)+O(1)+O(1)+O(N*B)+O(1)=O(max(1,1,1,N*B,1)=O(N*B)

Aclaraciones:

- Nos pareció necesario crear más de un archivo extra csv, así que en este caso les adjuntamos cuatro que hemos utilizado con distintas cantidades de campanas, aunque extraídas del archivo proporcionado por ustedes.