Tecnología Digital 1: Introducción a la Programación - TP2

Autores: Pardo, Ignacio
Silvestri, Juan Ignacio
Schneider, Santiago

Justificación

Método arboles_de_la_especie:

El método *arboles_de_la_especie* contiene la siguiente comprensión de lista*:*

```
return [a for a in self._arboles if a.especie == especie]
```

Para el analisis de los ordenes de complejidad, nos es conveniente refactorizar el código con un ciclo *for:*

```
1.    vr:List[Arbol] = []
2.    for a in self._arboles:
3.          if a.especie == especie:
4.                vr.append(a)
5.    return vr
```

La línea 1 crea una lista vacía, lo cual tiene O(1). En la línea 2, comienza la ejecución de un ciclo, que se ejecuta len(self._arboles) veces. Es decir, tiene O(n) donde n = len(self._arboles). Dentro del ciclo, tenemos las líneas 3 y 4 que forman parte de un condicional. Ambas líneas poseen O(1), como se puede leer en la documentación de python. Por último, tenemos la línea 5 con O(1) también. De esta manera, podemos proceder a calcular el orden de complejidad de la función como sigue.

El máximo entre líneas 1, 2 y 5, ya que son ejecuciones secuenciales. La línea 2 se calcula como n veces el orden de las líneas dentro del bloque de código del ciclo. El condicional, a su vez, también se calcula como el máximo de las dos líneas que lo componen. En términos más formales, tenemos:

O(1) + [O(n) * (O(1) + O(1))] + O(1)
= Max(O(1), [O(n) * Max(O(1),O(1))], O(1))
= Max(O(1), [O(n*1)], O(1))
= Max(O(1), O(n), O(1))
= O(n)

Luego, podemos concluir que el orden de la función es O(n) / lineal, donde n = len(self._arboles), es decir, la cantidad de árboles del dataset.

Método cantidad_por_especie:

El método cantidad_por_especie contiene las siguientes líneas de código:

```
1.        vr: Dict[str, int] = dict()
2.
3.        for e in self.especies():
4.            c: int = len(self.arboles_de_la_especie(e))
5.            if c > minimo:
6.                vr[e] = c
7.
8.        return vr
```

La línea 1 tiene O(1) porque crea un diccionario vacío. La línea 3 tiene O(m) donde m = len(self.especies()). La línea 4, tiene O(n) donde n = cantidad de árboles del dataset, como vimos en el análisis anterior. Luego, en las líneas 4 y 5 tenemos un condicional con O(m). Esto se debe a que vr[e] = c tiene orden de complejidad O(len(vr)). Notemos que vr puede contener como máximo (cuando el parámetro mínimo de la función es 1), todas las especies del dataset. Luego, en el peor caso, es O(m). Finalmente, en la línea 8 tenemos un return con O(1)

Calculamos el orden de complejidad de la siguiente manera:

O(1) + O(m) * (O(n) + Max(O(1) + O(m)) + O(1)
= Max(O(1), O(m) * (O(n) + Max(O(1) + O(m)), O(1)
= Max(O(1), O(m * Max(O(m), O(n)) , O(1)
= Max(O(1), O(m * n) , O(1)
= O(m * n)

Observemos que Max(O(m), O(n)) es O(n) porque en el peor caso, m = n, ya que no puede haber más especies que árboles en el dataset. Así, es seguro afirmar que el máximo siempre es n.
De esta manera, podemos concluir que el orden de la función es O(m * n) donde m = len(self.especies) y n = len(self._arboles).

Método arbol_mas_cercano:

Para el analisis del metodo arbol_mas_cercano, necesitamos analizar el método distancia de la clase Arbol, y considerar la complejidad del método arboles_de_la_especie.

El método distancia de la clase Arbol, contiene el siguiente código:

```
1.        vr: float = 0
```

```
2.        dif_x: float = (self.lat - lat) ** 2
3.        dif_y: float = (self.long - long) ** 2
4.
5.        vr = (dif_x + dif_y) ** 0.5
6.
7.        return vr
```

La complejidad de todas las líneas del código es O(1): son asignaciones, llamados de atributos, cálculos matemáticos básicos y un return. De esta forma, tenemos que el codigo del metodo tiene el siguiente orden de complejidad:

O(1) + O(1) + O(1) + O(1) + O(1)
= Max(O(1), O(1), O(1), O(1), O(1),)
= O(1)

Así, tenemos que el orden del método distancia() es O(1).

Ahora podemos proceder a calcular el orden de la función cantidad_por_especie. El método contiene el siguiente código:

```
1.        vr: Arbol = self.arboles_de_la_especie(especie)[0]
2.        min_d: int = vr.distancia(lat, lng)
3.        for a in self._arboles:
4.            if a.especie == especie:
5.                d: int = a.distancia(lat, lng)
6.                if not min_d:
7.                    min_d = d
8.                    vr = a
9.                else:
10.                    if d < min_d:
11.                        min_d = d
12.                        vr = a
13.       return vr
```

La línea 1 tiene complejidad O(N) respecto a la longitud del DataSet como es indicado en la justificación de la complejidad del método arboles_de_la_especie. La línea 2 como establecimos es de complejidad O(1). Luego, en la línea 3 tenemos un ciclo que se ejecuta n veces donde n = len(self._arboles). Dentro del ciclo, tenemos un condicional. El condicional se calcula como el máximo entre las dos opciones, ya sea si el condicional se ejecuta por if o por else. Sin embargo, notemos que todas las líneas del condicional son O(1), ya que son asignaciones y comparaciones. En la línea 5, donde se llama al método distancia(), también tenemos O(1) como lo calculamos anteriormente.

De esta manera, podemos calcular el orden como sigue:

O(n) + O(1) + O(n) * [Max(O(1), O(1), O(1), O(1), O(1), O(1), O(1),)] + O(1)
= Max(O(n), O(1), O(n) * [O(1)], O(1))
= Max(O(n), O(1), O(n * 1), O(1))
= O(n)

De esta manera, podemos concluir que el orden del método es O(n) donde n = len(self._arboles), es decir, la cantidad de árboles del dataset.

Metodos tamaño, barrios y especies

Para los métodos tamano, barrios y especies de la clase DataSetArboreo se tomó la decisión de diseño de contabilizar los 3 valores a la hora de la instanciación de un nuevo objeto DataSetArboreo. De esta forma, los llamados a dichos métodos resultan de orden constante(1), a diferencia del orden lineal. El método para obtener el tamaño del DataSet podría reemplazarse con la evaluación de len(self._arboles) y mantendrá la misma complejidad, sin embargo la complejidad de los métodos barrios y especies se ve afectada en cada llamado.

Para ejemplificar otra manera de implementar el método. especies() de la clase DataSetArboreo, podría ser con una comprensión de conjuntos de la forma

```
{a.especie for a in self._arboles}
```

O refactorizado a un ciclo for:

```
1.    vr: Set[str] = set()
2.    for a in self._ arboles:
3.          vr.add(a.especie)
4.    return vr
```

El cual resulta de complejidad:

O(1) + (O(N) * (O(M))
con N = len(self._arboles) y M = len(vr) por cada vuelta del for.
= Max[O(1) + O(N*M)]
= O(N*M)

Algo a tener en cuenta de definir atributos en la clase DataSetArboreo para disminuir la complejidad del llamado de estas funciones, es la complejidad a la hora de modificarlos de ser mutable los árboles del DataSet. El tamaño del DataSet el cual se obtiene del atributo ._size, del cual su incremento o decremento es orden 1.

Sin embargo, a la hora de añadir un nuevo Arbol al DataSet, agregar su especie y barrio a los conjuntos ._especies y ._barrios resulta de complejidad lineal respecto a la longitud de dichos conjuntos. De ser removido un Arbol del DataSet, habria que considerar si su barrio y especie deben ser removidos también de los conjuntos del DataSet. Esto se podría resolver contabilizando también árboles por

especie y árboles por barrio en la instanciación del DataSet, resultando entonces en una complejidad O(1).

Estas son sólo consideraciones de tener que modificar el DataSet, pero, como establecimos previamente, el diseño del DataSet nos condiciona el método **cantidad_por_especie** a una complejidad O(N), la cual hasta se podría haber reducido a orden lineal de ser contabilizada en un principio con este fin indicado.

Otras consideraciones

Función auxiliar list_equals

Para testear el método arboles_de_la_especie de la clase DataSetArboreo, el cual retorna un valor del tipo List[Arbol], surgió la necesidad de poder comparar el valor de retorno contra uno establecido a mano. Sin embargo, al estas listas no respetan un orden, el método assertListEqual de unittest solo podríamos aprovecharlo de ordenar las listas y asegurarnos que no haya valores repetidos.

Para solucionar este problema evaluamos la posibilidad de crear dos conjuntos a partir de las listas para comparar su igualdad, sin embargo, para crear objetos del tipo Set[Arbol], la clase Arbol debía de tener implementado el método __hash__(), lo cual luego de consultar se encontraba fuera del contenido de la materia. Recién ahí se podría testear con el método assertEqual la igualdad entre el conjunto de árboles resultante del método arboles_de_la_especie y el conjunto seleccionado manualmente.

A la hora de comparar ambas listas obtuvimos una solución de complejidad cuadrática, la cual encuadramos en una función auxiliar list_equals.

```
1.    if (len(a) != len(b)):
2.          return False
3.    le = True
4.    for e in a:
5.          if e not in b:
6.                le = False
7.    for e in b:
8.          if e not in a:
9.                le = False
10.   return le
```

La línea 1 es de complejidad O(1), dado que len(a) es O(1), len(b) O(1), ambos resultantes en valores enteros, y la comparación int != int es O(1). La línea 3 es una asignación de complejidad constante(1).

En la línea 4 comienza un ciclo de orden lineal O(N) donde N es el len(a). En la línea 5 la evaluación `e not in b` es de complejidad O(N) también, como

establecimos previamente len(a) == len(b). La línea 6 contiene otra asignación de orden 1. El ciclo for de la línea 4 entonces resulta de orden cuadrático $O(N^2)$. En la línea 7 se realiza un ciclo idéntico en complejidad al de la línea 4 dado que la orden depende de un mismo N. La complejidad de la función list_equals resulta entonces

$O(1) + O(1) + O(1) + O(N) * (O(N) * O(1)) + O(N) * (O(N)*O(1))$
$= O(1) + O(1) + O(1) + O(N) * O(N) + O(N) * O(N)$
$= O(1) + O(1) + O(1) + O(N^2) + O(N^2)$
$= Max[O(1) + O(1) + O(1) + O(N^2) + O(N^2)]$
$= O(N^2)$

Al igual que en la evaluación de igualdad de dos sets, donde primero se compara la longitud de ambas, previo a evaluar la igualdad a fondo de dos listas, se fuerza un assertEqual entre la longitud de la lista devuelta y el valor esperado. Esto se tiene en cuenta en la solución propuesta list_equals para su uso fuera del de testing, pero decidimos tener en cuenta otra assertion con el valor de la longitud implícito.

Como ejemplo, en `test_dt1_arboles_de_la_especie_Fraxinus_pennsylvanica`, donde `ae` es la lista de árboles de la especie *Fraxinus Pennsylvanica* y `gt` la lista ground truth.

```
self.assertEqual(len(ae), 11) #comparación entre len(ae) y 11 implícito
self.assertTrue(list_equals(ae, gt)) #compara len(ae) == len(gt) dentro
```

Otra solución posible hubiera sido posible de implementar los métodos __lt__(), __le__(), __eq__(), __ne__(), __ge__() y __gt__() en la clase Arbol, para así poder ordenar ambas listas y testear la igualdad entre ellas con el método assertListEqual. La complejidad de las operaciones en esta solución hubiera sido O(N) cada sorting, resultando en un orden lineal contra el cuadrático obtenido, aunque no contaría con una verificación de que no haya árboles repetidos en las listas.

Método arbol_mas_cercano

En el método arbol_mas_cercano de la clase DataSetArboreo, tanto el valor de retorno como la distancia mínima requeridos para nuestra primera solución, se inicializan como None para el correcto funcionamiento.

```
vr: Arbol = None
min_d: int = None
```

Para evitar esto optamos por inicializar ambos valores basandonos en el primer arbol de la especie, cuya obtención es en el peor de los casos O(N), en el cual basamos nuestra resolución.

```
vr: Arbol = self.arboles_de_la_especie(especie)[0]
min_d: int = vr.distancia(lat, lng)
```