Integrantes Grupo 39: Victoria Stefania Schenone Fernández, Nadia Guadalupe Molina y Jazmín Agustina Sneider

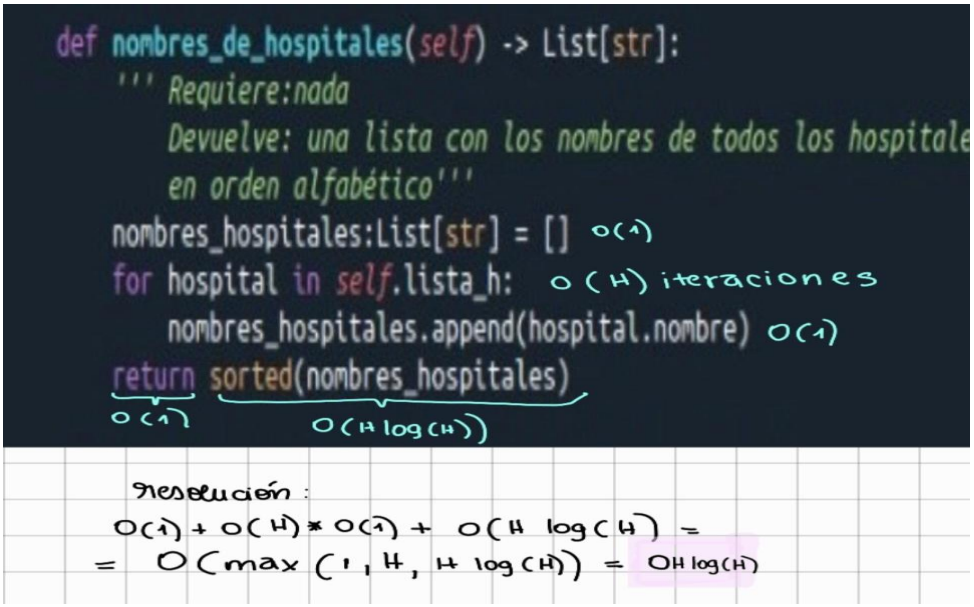
Tecnología Digital 1: Introducción a la Programación

Trabajo Práctico 2
Escuela de Negocios, UTDT
Primer semestre 2022

Justificación de por qué los programas tienen la complejidad requerida:

1) **Método nombres_de_hospitales:**

Consideremos primero la siguiente función nombre_de_hospitales:



La primera línea define la lista llamada nombres_hospitales de tipo string que está vacía. Como es una escritura de una variable la operación es simple y tiene complejidad constante O(1)

Luego comienza con ciclo utilizando for. El for ejecuta su bloque de código H veces, según la cantidad de hospitales que tenga lista_h. Entonces el algoritmo tiene O(H) hasta este momento. A esto hay que multiplicarlo por la complejidad del bloque de código que le sigue al for:

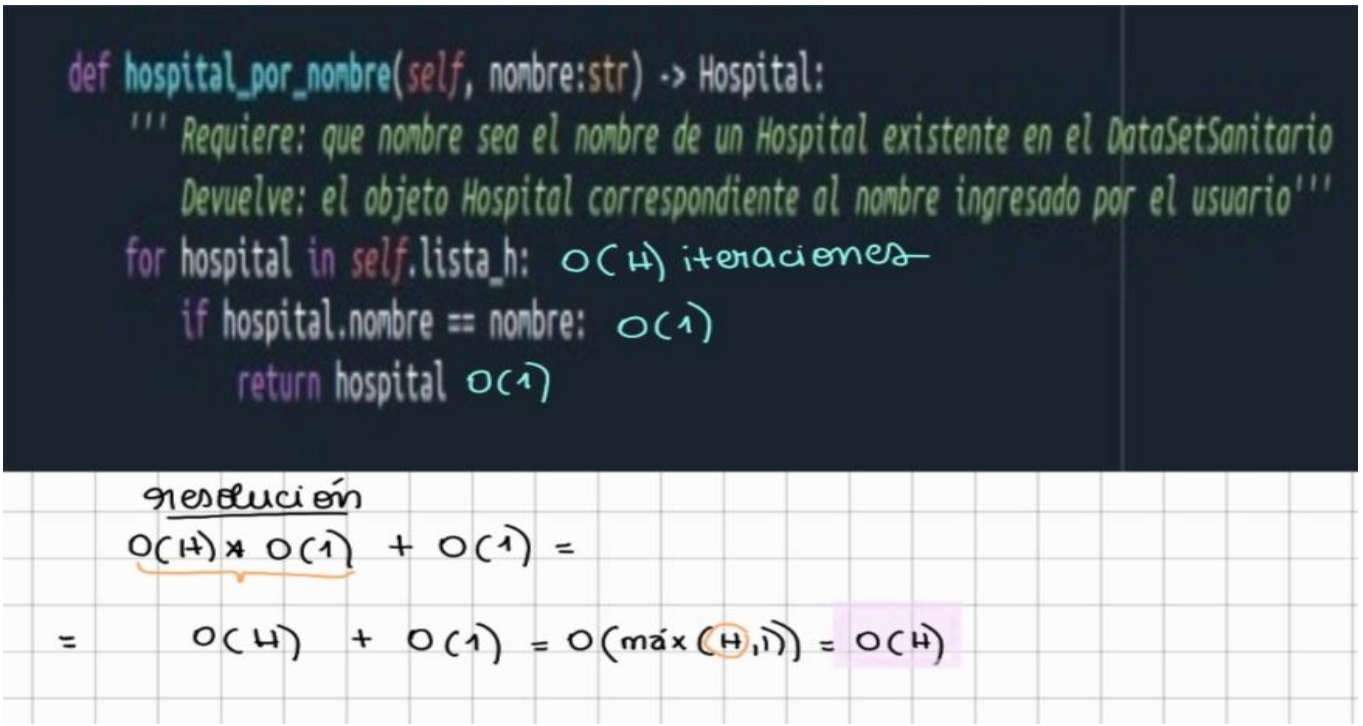
- La siguiente línea agrega a nuestra lista inicial el nombre del hospital, esto es una operación sencilla, por lo tanto es de orden constante, es decir O(1).

La última línea tiene más de una operación, por un lado el retorno de un valor que es de orden 1 y la segunda operación es usar sorted, para poder organizar de forma ordenada (alfabéticamente) los nombres de los hospitales y por convención sabemos que tiene un orden logarítmico O(H log(H))

Para sacar la complejidad final tenemos que sacar el máximo entre las cuentas que hicimos antes. Así, podemos deducir haciendo las cuentas que: por un lado, el máximo entre H y H log H es H log H. Esto es porque log H > 1 (con H suficientemente grande) y además log H siempre crece. **Entonces como H log H > H, la complejidad algorítmica del programa es H log H**

2) **Método hospital_por_nombre:**

Consideremos como segundo punto la función hospital_por_nombre:



En la primera línea nos encontramos con el ciclo que se repetirá durante H iteraciones, según la cantidad de hospitales que self.lista_h tenga. La complejidad del ciclo se resuelve multiplicando su complejidad con las del bloque de código debajo:

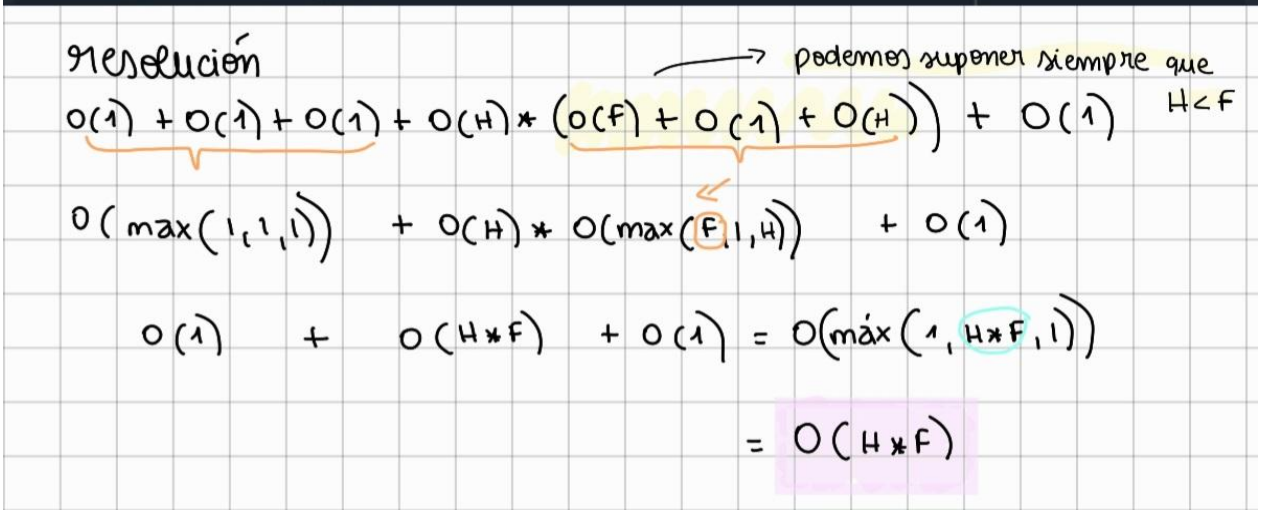
- Tenemos un if cuya condición tiene O(1) por ser comparación de strings y su código de abajo (el return) tiene complejidad O(1) también. Por lo que haciendo un máximo entre ambos resultamos con que tiene O(1) de complejidad.

Finalmente multiplicamos la complejidad del ciclo con la de su bloque de abajo, resultando O(H*1), hacemos máximo entre O(H,1) y finalmente deducimos que **la función tiene orden de complejidad O(H)**.

3) **Método farmacia_por_hospital:**

Consideremos como tercer punto la función farmacia_por_hospital:

```python
def farmacia_por_hospital(self) -> Dict[Hospital, Tuple[Farmacia, float]]:
    ''' Requiere: nada
        Devuelve: un diccionario que asocia a cada hospital del DataSetSanitario un par
        (es decir, una tupla) con la farmacia más cercana y la distancia a la misma (en metros)
        '''
    farmacia:Farmacia = () O(1)
    farmacia_distancia:Tuple[Farmacia,float] O(1)
    farmacia_hospital:Dict[Hospital, Tuple[Farmacia,float]] = dict() O(1)
    for hospital in self.lista_h: O(H) iteraciones
        farmacia = self.farmacia_mas_cercana(hospital) O(F)
O(1)    farmacia_distancia = (farmacia, farmacia.distancia(hospital.latitud, hospital.longitud))
        farmacia_hospital[hospital] = farmacia_distancia O(H)
    return farmacia_hospital O(1)
```



Las primeras 3 líneas son de O(1) pues son operaciones simples, entre las que se encuentran la creación de una farmacia, de una tupla y de un diccionario, por el momento los 3 vacíos.
Luego llegamos al ciclo donde el código del for se va a repetir H iteraciones, es decir tantas veces como hospitales tenga self.lista_h, por lo que tiene O(H) multiplicado por lo que le cuesta realizar las operaciones que siguen:

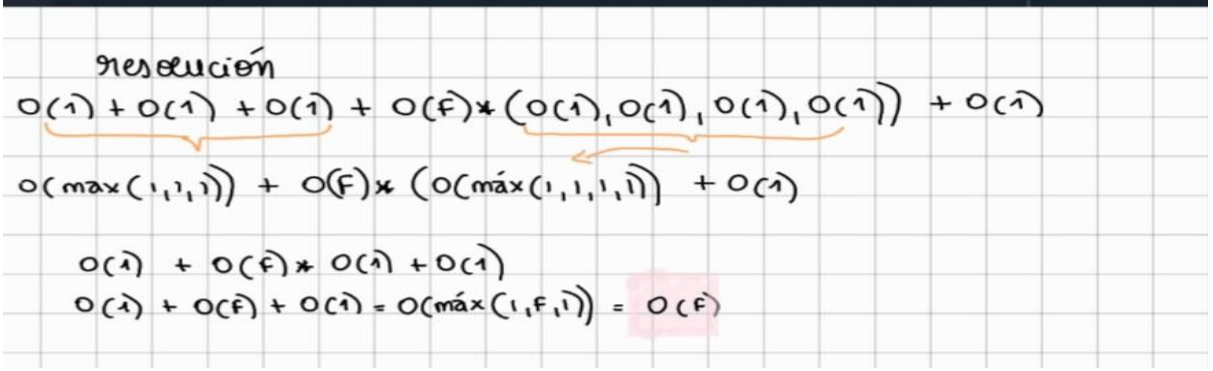
- Asignarle a farmacia la Farmacia más cercana a cada hospital le cuesta O(F) porque llama a la función farmacia_mas_cercana que tiene complejidad O(F)
- Asignarle a la tupla la farmacia y la distancia de esa farmacia a las coordenadas del hospital le cuesta O(1) porque el cálculo de distancia utilizando haversine tiene esa complejidad.
- Asignarle un valor a una clave en el diccionario de hospitales le cuesta O(H)

El ciclo se resuelve multiplicando O(H) * el máximo de O(F,1,H). Como podemos suponer que siempre H<F, el maximo de esa cuenta es F y quedaría O(H*F).
En la última línea el return tiene orden de complejidad O(1)
Finalmente para resolver toda la complejidad algorítmica sacamos el máximo entre O(1, H*F, 1) y **nos queda O(H*F)**.

4) **Método farmacia_mas_cercana:**

Consideremos como cuarto punto la función farmacia_mas_cercana:

```
def farmacia_mas_cercana(self, hosp:Hospital)  -> Farmacia:
    ''' Requiere: que hosp sea un Hospital existente en el DataSetSanitario
        Devuelve: la farmacia del DataSetSanitario que está más cercana al hospital hosp
    '''
    distancia_res:float = 0.0  O(1)
    distancia:float = 0.0  O(1)
    res:Farmacia  O(1)
    for farmacia in self.lista_f: O(F) iteraciones
        distancia = farmacia.distancia(hosp.latitud, hosp.longitud)  O(1)
        if distancia_res == 0.0 or distancia_res > distancia:  O(1)
            distancia_res = distancia  O(1)
            res = farmacia  O(1)
    return res  O(1)
```



Las primeras tres líneas tienen complejidad 0(1) pues son operaciones simples, entre las que encontramos operaciones con floats y además la creación de una Farmacia llamada res.
Luego nos encontramos con el ciclo quien repite F veces(por eso O(F)) el cuerpo del código que tiene debajo. Decimos que es O(F)* todo lo que le cuesta llevar a cabo la ejecución del programa:

- Establecer el valor de distancia le cuesta O(1) porque la complejidad temporal de la función haversine con la que trabaja distancia es O(1), más precisamente lo que hace f.haversine es involucrar una cantidad constante de operaciones aritméticas simples.
- El if se resuelve haciendo un máximo entre la condición (de O(1))y el bloque de código que le sigue debajo:
  *las dos líneas debajo del if tienen O(1) porque son asignación de valores

El ciclo se resuelve multiplicando O(F) por el máximo de su bloque de abajo, que como todos tienen O(1), es O(F*1)=O(F)
En la última línea el return tiene orden de complejidad O(1)
Para resolver la complejidad final hacemos el máximo entre todos los órdenes que calculamos previamente, es decir entre O(1, F, 1) y nos queda que esta función tiene **órden de complejidad O(F)**