Análisis de Complejidad

Nombre: Nicolas Merchan Cuestas

Código: 202112109

Correo: n.merchan@uniandes.edu.co

Notas:

- Los resultados de las pruebas de tiempos de ejecución y las correspondientes gráficas de los mismos se encuentran en '*Datos - Análisis de Complejidad.xslx*'.
- El programa 'Test_Function.py' realiza las pruebas de tiempos de ejecución de manera automática e ingresa los resultados en un archivo EXCEL llamado 'Test_Data.xlsx'.

Requerimiento 1

```
275  def Requirement1(catalog, city):
276      cities_map = catalog['cities_map']
277      cities_list = catalog['cities_list']
278      num_cities = lt.size(cities_list)
279
280      if mp.contains(cities_map, city):
281          city_events_list = trv.inorder(me.getValue(mp.get(cities_map, city)))
282          city_events_info = getFirstandLastElements(city_events_list, 3, '>')
283          first_events_list = city_events_info[0]
284          last_events_list = city_events_info[1]
285          num_events_city = city_events_info[2]
286
287          num_events_most_city = -1
288          for city in lt.iterator(cities_list):
289              num_events_particular_city = bst.size(me.getValue(mp.get(cities_map, city)))
290              if num_events_particular_city > num_events_most_city:
291                  num_events_most_city = num_events_particular_city
292                  most_events_city = city
293      else:
294          first_events_list = lt.iterator(lt.newList())
295          last_events_list = first_events_list
296          most_events_city = 'None'
297          num_events_city = 0
298
299      return first_events_list, last_events_list, num_cities, most_events_city, num_events_city
```

La complejidad del requerimiento 1 está dada por la función **Requirement1()**. El proceso de búsqueda de los avistamientos de la ciudad ingresada por parámetro se lleva cabo por medio de una búsqueda en el la tabla de Hash **catalog['cities_map']**. La tabla de Hash **catalog['cities_map']** tiene como llaves los nombres de las ciudades y como valores asociados a las llaves, BST's de los avistamientos ocurridos en las ciudades respectivas ordenados cronológicamente por sus fechas y horas. La complejidad asociada al proceso anteriormente descrito es **O(1)**. Paralelamente, la búsqueda de la ciudad con más avistamientos se basa en un recorrido por los BST's referentes a las ciudades haciendo uso de la lista ordenada **catalog['cities_list']**, la cual contiene como valores los nombres de las ciudades. Así, es posible comparar todas las ciudades ingresando los valores en

las posiciones individuales de **catalog['cities_list']** en **catalog['cities_map']** como llaves. La complejidad asociada al proceso anteriormente descrito es **O(log n)**. Es propicio resaltar que el recorrido anteriormente descrito se realiza sobre las ciudades registradas, no sobre la totalidad de avistamientos registrados. La cantidad de ciudades crece, aproximadamente, de manera logarítmica con el incremento de los avistamientos registrados. Finalmente, la complejidad del requerimiento 1 es **O(log n)**¸ dado que el proceso más complejo tiene dicha complejidad asociada.

Requerimiento 2

```
303  def Requirement2(catalog, initial_duration, end_duration):
304      durations_map = catalog['durations_map']
305      durations_BST = catalog['durations_BST']
306      durations_list = trv.inorder(durations_BST)
307
308      longest_duration_info = getMostElement(durations_list, durations_map, '>')
309      longest_duration = longest_duration_info[0]
310      num_events_longest_duration = longest_duration_info[1]
311      num_durations = longest_duration_info[2]
312
313      previous_duration = -1
314      duration_interval_events_list = getElementsIntervalValues(durations_map, durations_list,
315                                                                 previous_duration, num_durations, initial_duration, end_duration)
316
317      duration_interval_events_info = getFirstandLastElements(duration_interval_events_list, 3, '>')
318      first_events_list = duration_interval_events_info[0]
319      last_events_list = duration_interval_events_info[1]
320      num_events_duration_interval = duration_interval_events_info[2]
321
322      return first_events_list, last_events_list, num_durations, longest_duration, num_events_longest_duration, num_events_duration_interval
```

La complejidad del requerimiento 2 está dada por la función **Requirement2()**. La tabla de Hash **catalog['durations_map']** tiene como llaves las duraciones y como valores asociados a las llaves, BST's de los avistamientos ocurridos con las duraciones respectivas ordenados ascendentemente por sus fechas y horas. El BST **catalog['durations_BST']** tiene como llaves y valores asociados a las llaves las duraciones de los avistamientos. La lista ordenada **durations_list** contiene todas las duraciones de los avistamientos ordenados de menor a mayor en función de la duración. El orden de **durations_list** se da porque es producto del recorrido inorder del BST **catalog['durations_BST']**. El proceso de búsqueda de la mayor duración se da por medio de la función **getMostElement()**, la cual busca el último elemento de **durations_list**. La complejidad asociada al proceso anteriormente descrito es **O(1)**. El proceso de búsqueda de los avistamientos con una duración dentro del intervalo ingresado por parámetro se lleva cabo por medio de la función **getElementsIntervalValues()**, la cual realiza una búsqueda lineal en **durations_list** que se detiene cuando encuentra la duración superior del intervalo o una duración mayor a la misma. La complejidad asociada al proceso anteriormente descrito es **O(lon n)**. Es propicio resaltar que el recorrido anteriormente descrito se realiza sobre las duraciones registradas, no sobre la totalidad de avistamientos registrados. La cantidad de duraciones crece, aproximadamente, de manera logarítmica con el incremento de los avistamientos registrados. Finalmente, la complejidad del requerimiento 2 es **O(log n)**¸ dado que el proceso más complejo tiene dicha complejidad asociada.

Requerimiento 3

```
326 def Requirement3(catalog, initial_time, end_time):
327     times_map = catalog['times_map']
328     times_BST = catalog['times_BST']
329     times_list = trv.inorder(times_BST)
330     initial_time = datetime.strptime(initial_time, "%H:%M:%S")
331     end_time = datetime.strptime(end_time, "%H:%M:%S")
332 
333     latest_time_info = getMostElement(times_list, times_map, '>')
334     latest_time = latest_time_info[0]
335     num_events_latest_time = latest_time_info[1]
336     num_times = latest_time_info[2]
337 
338     previous_time = datetime.strptime('00:00:00', "%H:%M:%S")
339     time_interval_events_list = getElementsIntervalValues(times_map, times_list, previous_time, num_times,
340                                                            initial_time, end_time)
341 
342     time_interval_events_info = getFirstandLastElements(time_interval_events_list, 3, '>')
343     first_events_list = time_interval_events_info[0]
344     last_events_list = time_interval_events_info[1]
345     num_events_time_interval = time_interval_events_info[2]
346 
347     return first_events_list, last_events_list, num_times, latest_time, num_events_latest_time, num_events_time_interval
```

La complejidad del requerimiento 3 está dada por la función **Requirement3()**. La tabla de Hash **catalog['times_map']** tiene como llaves los tiempos y como valores asociados a las llaves, BST's de los avistamientos ocurridos en los tiempos respectivos ordenados ascendentemente por sus fechas. El BST **catalog['times_BST']** tiene como llaves y valores asociados a las llaves los tiempos de los avistamientos. La lista ordenada **times_list** contiene todos los tiempos de los avistamientos ordenados de menor a mayor en función del tiempo. El orden de **times_list** se da porque es producto del recorrido inorder del BST **catalog['times_BST']**. El proceso de búsqueda del mayor tiempo se da por medio de la función **getMostElement()**, la cual busca el último elemento de **times_list**. La complejidad asociada al proceso anteriormente descrito es **O (1)**. El proceso de búsqueda de los avistamientos ocurridos en un tiempo dentro del intervalo ingresado por parámetro se lleva cabo por medio de la función **getElementsIntervalValues()**, la cual realiza una búsqueda lineal en **times_list** que se detiene cuando encuentra el tiempo superior del intervalo o un tiempo mayor al mismo. La complejidad asociada a el proceso anteriormente descrito es **O(lon n)**. Es propicio resaltar que el recorrido anteriormente descrito se realiza sobre los tiempos registrados, no sobre la totalidad de avistamientos registrados. La cantidad de tiempos crece, aproximadamente, de manera logarítmica con el incremento de los avistamientos registrados. Finalmente, la complejidad del requerimiento 3 es **O(log n)**, dado que el proceso más complejo tiene dicha complejidad asociada.

Requerimiento 4

```
351  def Requirement4(catalog, initial_date, end_date):
352      dates_map = catalog['dates_map']
353      dates_BST = catalog['dates_BST']
354      dates_list = trv.inorder(dates_BST)
355      initial_date = datetime.strptime(initial_date, "%Y-%m-%d")
356      end_date = datetime.strptime(end_date, "%Y-%m-%d")
357
358      oldest_date_info = getMostElement(dates_list, dates_map, '<')
359      oldest_date = oldest_date_info[0]
360      num_events_oldest_date = oldest_date_info[1]
361      num_dates = oldest_date_info[2]
362
363      previous_date = datetime.strptime('00:00:00', "%H:%M:%S")
364      date_interval_events_list = getElementsIntervalValues(dates_map, dates_list, previous_date, num_dates,
365                                                             initial_date, end_date)
366
367      date_interval_events_info = getFirstandLastElements(date_interval_events_list, 3, '>')
368      first_events_list = date_interval_events_info[0]
369      last_events_list = date_interval_events_info[1]
370      num_events_date_interval = date_interval_events_info[2]
371
372      return first_events_list, last_events_list, num_dates, oldest_date, num_events_oldest_date, num_events_date_interval
```

La complejidad del requerimiento 4 está dada por la función **Requirement4()**. La tabla de Hash **catalog[‘dates_map’]** tiene como llaves las fechas y como valores asociados a las llaves, BST’s de los avistamientos ocurridos en las fechas respectivas ordenadas ascendentemente por sus tiempos. El BST **catalog[‘dates_BST’]** tiene como llaves y valores asociados a las llaves las fechas de los avistamientos. La lista ordenada **dates_list** contiene todas las fechas de los avistamientos ordenados de menor a mayor en función de la fecha. El orden de **dates_list** se da porque es producto del recorrido inorder del BST **catalog[‘dates_BST’]**. El proceso de búsqueda de la fecha más antigua se da por medio de la función **getMostElement()**, la cual busca el primer elemento de **dates_list**. La complejidad asociada al proceso anteriormente descrito es **O (1)**. El proceso de búsqueda de los avistamientos ocurridos en una fecha dentro del intervalo ingresado por parámetro se lleva cabo por medio de la función **getElementsIntervalValues()**, la cual realiza una búsqueda lineal en **dates_list** que se detiene cuando encuentra la fecha superior del intervalo o una fecha mayor al mismo. La complejidad asociada a el proceso anteriormente descrito es **O(lon n)**. Es propicio resaltar que el recorrido anteriormente descrito se realiza sobre las fechas registradas, no sobre la totalidad de avistamientos registrados. La cantidad de fechas crece, aproximadamente, de manera logarítmica con el incremento de los avistamientos registrados. Finalmente, la complejidad del requerimiento 4 es **O(log n)**, dado que el proceso más complejo tiene dicha complejidad asociada.

Requerimiento 5

```python
def Requirement5(catalog, initial_longitude, end_longitude, initial_latitude, end_latitude):
    latitudes_map = catalog['latitudes_map']
    latitudes_BST = catalog['latitudes_BST']
    latitudes_list = trv.inorder(latitudes_BST)
    num_latitudes = bst.size(latitudes_BST)

    previous_value_latitude = -181
    previous_value_longitude = -181

    area_events_list = getElementsDoubleIntervalValues(latitudes_map, latitudes_list,
            previous_value_latitude, previous_value_longitude, num_latitudes, initial_latitude, end_latitude,
            initial_longitude, end_longitude)

    area_interval_events_info = getFirstandLastElements(area_events_list, 5, '>')
    first_events_list = area_interval_events_info[0]
    last_events_list = area_interval_events_info[1]
    num_events_area = area_interval_events_info[2]

    return first_events_list, last_events_list, num_events_area
```

La complejidad del requerimiento 5 está dada por la función **Requirement5()**. La tabla de Hash **catalog[‘latitudes_map’]** tiene como llaves las latitudes registradas y como valores asociados a las llaves, BST’s de los avistamientos ocurridos en las latitudes respectivas ordenadas ascendentemente por sus longitudes. El BST **catalog[‘latitudes_BST’]** tiene como llaves y valores asociados a las llaves las latitudes de los avistamientos. La lista ordenada **latitudes_list** contiene todas las latitudes de los avistamientos ordenados de menor a mayor en función de la latitud. El orden de **latitudes_list** se da porque es producto del recorrido inorder del BST **catalog[‘latitudes_BST’]**. El proceso de búsqueda de los avistamientos ocurridos en una latitud y longitud dentro de los intervalos ingresados por parámetro se lleva cabo por medio de la función **getElementsDoubleIntervalValues()**, la cual realiza una búsqueda lineal en **latitudes_list** y las listas producto de los recorrido inorder de los BST’s referentes a cada latitud en **catalog[‘latitudes_map’]**. Dicho recorrido se detiene cuando encuentra la latitud y longitud superior de los intervalos o una latitud y longitud mayores a los mismos. La complejidad asociada a el proceso anteriormente descrito es **O(lon n)**. Es propicio resaltar que el recorrido anteriormente descrito se realiza sobre la latitudes y longitudes registradas, no sobre la totalidad de avistamientos registrados. La cantidad de latitudes y longitudes crece, aproximadamente, de manera logarítmica con el incremento de los avistamientos registrados. Finalmente, la complejidad del requerimiento 5 es **O(log n)**¸ dado que el proceso más complejo tiene dicha complejidad asociada.

Requerimiento 6

```
398 def Requirement6(catalog, initial_longitude, end_longitude, initial_latitude, end_latitude):
399     latitudes_map = catalog['latitudes_map']
400     latitudes_BST = catalog['latitudes_BST']
401     latitudes_list = trv.inorder(latitudes_BST)
402     num_latitudes = bst.size(latitudes_BST)
403 
404     previous_value_latitude = -181
405     previous_value_longitude = -181
406 
407     area_events_list = lt.iterator(getElementsDoubleIntervalValues(latitudes_map, latitudes_list,
408             previous_value_latitude, previous_value_longitude, num_latitudes, initial_latitude, end_latitude,
409             initial_longitude, end_longitude))
410 
411     return area_events_list
```

La complejidad del requerimiento 6 está dada por la función **Requirement6()**. La tabla de Hash **catalog['latitudes_map']** tiene como llaves las latitudes registradas y como valores asociados a las llaves, BST's de los avistamientos ocurridos en las latitudes respectivas ordenadas ascendentemente por sus longitudes. El BST **catalog['latitudes_BST']** tiene como llaves y valores asociados a las llaves las latitudes de los avistamientos. La lista ordenada **latitudes_list** contiene todas las latitudes de los avistamientos ordenados de menor a mayor en función de la latitud. El orden de **latitudes_list** se da porque es producto del recorrido inorder del BST **catalog['latitudes_BST']**. El proceso de búsqueda de los avistamientos ocurridos en una latitud y longitud dentro de los intervalos ingresados por parámetro se lleva cabo por medio de la función **getElementsDoubleIntervalValues()**, la cual realiza una búsqueda lineal en **latitudes_list** y las listas producto de los recorrido inorder de los BST's referentes a cada latitud en **catalog['latitudes_map']**. Dicho recorrido se detiene cuando encuentra la latitud y longitud superior de los intervalos o una latitud y longitud mayores a los mismos. La complejidad asociada a el proceso anteriormente descrito es **O(lon n)**. Es propicio resaltar que el recorrido anteriormente descrito se realiza sobre la latitudes y longitudes registradas, no sobre la totalidad de avistamientos registrados. La cantidad de latitudes y longitudes crece, aproximadamente, de manera logarítmica con el incremento de los avistamientos registrados. Finalmente, la complejidad del requerimiento 6 es **O(log n)**¸ dado que el proceso más complejo tiene dicha complejidad asociada.