Universidad de los Andes
Estructuras de Datos y Algoritmos – Sección 6

José Nicolás Cárdenas Trujillo
201922006 - j.cardenast

Reto 3: Documento de Análisis

Implementación requerimiento individual:

- ➔ Requerimiento 2: José Nicolás Cárdenas.

Nota: opté por entregar este reto individualmente debido a que mi compañero no apareció y me vi obligado a hacer todos los requerimientos grupales por mi cuenta.

Análisis De Complejidad

❖ Requerimiento 1

El cuerpo de la función de este requerimiento es el siguiente:

```
583     mp_city = catalog['city']                           # Guardar mapa 'city'.
584     lt_sight = mp.get(mp_city, param_city)['value']     # Guardar arreglo avistamientos de la ciudad.
585
586     # Determinar tamaño y ordenar lista avistamientos, empaquetarlos y retornar.
587     size_lt_sight = lt.size(lt_sight)
588     ord_lt_sight = mg.sort(lt_sight, cmp_by_datetime)
589     return (size_lt_sight, ord_lt_sight)
```

Todas las operaciones que se realizan tienen orden constante (salvo una), ya que acceder a un elemento de un mapa desordenado (una Tabla de Hash implementada mediante *Separate Chaining*) se hace en tiempo constante; la operación de la línea 588 tiene complejidad $O(n \log(n))$, ya que se está usando el algoritmo de ordenamiento *mergesort.*

Suponiendo que hay m avistamientos registrados en la ciudad que se pasa por parámetro, la complejidad de este algoritmo será de:

$$O(m \log(m)).$$

❖ Requerimiento 2

El cuerpo de la función de este requerimiento es el siguiente:

```
609     om_duration = catalog['duration (seconds)']        # Guardar mapa 'duration (seconds)'.
610     lt_sight_return = lt.newList('ARRAY_LIST')         # Crear lista de retorno.
611
612     # Crear variables que guardan el ceiling y el floor de las duraciones inferior y superior.
613     min_duration = om.ceiling(om_duration, param_min_duration)
614     max_duration = om.floor(om_duration, param_max_duration)
615
616     # Recorrer todas las llaves del mapa 'duration (seconds)' que se encuentran dentro del rango.
617     for duration in lt.iterator(om.keys(om_duration, min_duration, max_duration)):
618
619         # Guardar lista de avistamientos de duration, ordenarla, iterarla y añadir todos sus elementos a lt_sight_return.
620         lt_sightings = om.get(om_duration, duration)['value']
621         ord_lt_sightings = qui.sort(lt_sightings, cmp_by_city_country)
622         for sighting in lt.iterator(ord_lt_sightings):
623             lt.addLast(lt_sight_return, sighting)
624
625     # Ordenar la lista de retorno cronológicamente, determianr su tamaño, empaquetar y retornar.
626     ordered_lt_sight_return = mg.sort(lt_sight_return, cmp_by_datetime)
627     size_lt_sight_return = lt.size(ordered_lt_sight_return)
628     return (size_lt_sight_return, ordered_lt_sight_return)
```

Asumiendo que dentro del rango del requerimiento hay m datos y que el mapa *duration (seconds)* del catálogo tiene n datos, el orden de complejidad de esta función es

$$O(m\log(m) + 2\log(n)).$$

Esto debido a que:

1- Si el rango tiene m elementos, entonces se deberá iterar sobre este, lo cual tendría complejidad lineal (m); aun así, debido a que la operación de la línea 626 tendría complejidad $O(m\log(m))$ ya que se está usando el algoritmo de ordenamiento *mergesort,* entonces este último es el orden de crecimiento que se debe considerar en el análisis.

2- Las funciones $om.ceiling()$ y $om.floor()$, en el peor de los casos, tendrán complejidad $2\log(n)$, debido a que sería necesario hacer una búsqueda hasta las hojas del árbol *duration (seconds)*.

3- Las listas de que se guardarán en cada iteración en la línea 620 tendrá muy pocos elementos, por lo que se puede considerar que ordenarlas e iterarlas se hará en tiempo constante.

4- Todas las demás operaciones son constantes.

❖ Requerimiento 4

El cuerpo de la función de este requerimiento es el siguiente:

```
648     om_date = catalog['date']                       # Guardar mapa 'date'.
649     lt_sight_return = lt.newList('ARRAY_LIST')      # Crear lista de retorno.
650
651     # Volver las fechas dadas por parámetro comparables.
652     mod_min_date = dt.datetime.strptime(param_min_date, '%Y-%m-%d')
653     mod_max_date = dt.datetime.strptime(param_max_date, '%Y-%m-%d')
654
655     # Crear variables que guardan el ceiling y el floor de las fechas inferior y superior.
656     min_date = om.ceiling(om_date, mod_min_date)
657     max_date = om.floor(om_date, mod_max_date)
658
659     # Recorrer todas las llaves del mapa 'date' que se encuentran dentro del rango.
660     for date in lt.iterator(om.keys(om_date, min_date, max_date)):
661
662         # Guardar lista de avistamientos en date, iterarla y añadir todos sus elementos a lt_sight_return.
663         lt_sightings = om.get(om_date, date)['value']
664         for sighting in lt.iterator(lt_sightings):
665             lt.addLast(lt_sight_return, sighting)
666
667     # Determianr tamaño lt_sight_return, empaquetar y retornar.
668     size_lt_sight_return = lt.size(lt_sight_return)
669     return (size_lt_sight_return, lt_sight_return)
```

Asumiendo que dentro del rango del requerimiento hay m datos y que el mapa *date* del catálogo tiene n datos, el orden de complejidad de esta función es

$$O(m + 2\log(n)).$$

Esto debido a que:

1- Si el rango tiene m elementos, entonces se deberá iterar sobre este, lo cual tendría complejidad lineal (m).
2- Las funciones $om.ceiling()$ y $om.floor()$, en el peor de los casos, tendrán complejidad $2\log(n)$, debido a que sería necesario hacer una búsqueda hasta las hojas del árbol *date*.
3- Las listas de que se guardarán en cada iteración en la línea 663 tendrá muy pocos elementos, por lo que se puede considerar que iterarlas se hará en tiempo constante.
4- Todas las demás operaciones son constantes.

❖ Requerimiento 5

El cuerpo de la función de este requerimiento es el siguiente:

```
693     om_longitude = catalog['longitude']         # Guardar mapa 'longitude'.
694     return_list = lt.newList('ARRAY_LIST')      # Crear lista de retorno.
695
696     # Crear variables que guardan el ceiling y el floor de las longitudes inferior y superior.
697     floor_long = om.ceiling(om_longitude, min_long)
698     ceil_long = om.floor(om_longitude, max_long)
699
700
701     # Crear iterador del rango [floor_long, ceil_long] e iterar sobre este.
702     iterator_long = lt.iterator(om.keys(om_longitude, floor_long, ceil_long))
703     for longitude in iterator_long:
704
705         # Determinar si existe la llave longitude en om_longitude.
706         exists = om.get(om_longitude, longitude)
707
708
709         # Si existe la pareja longitude-om_latitude en om_longitude.
710         if (exists):
711
712             # Guardar el om_latitude de longitude, crear su iterador e iterar sobre este.
713             om_latitude = om.get(om_longitude, longitude)['value']
714             iterator_om_latitude = lt.iterator(om.keys(om_latitude, om.minKey(om_latitude), om.maxKey(om_latitude)))
715             for latitude in iterator_om_latitude:
716
717                 # Determinar si latitud está entre el rango [min_lat, max_lat].
718                 valid = (latitude >= min_lat and latitude <= max_lat)
719
720                 # Si es válido.
721                 if (valid):
722                     # Guardar lt_sightings, recorrerla y añadir cada avistamiento a return_list.
723                     lt_sightings = om.get(om_latitude, latitude)['value']
724                     for element in (lt.iterator(lt_sightings)):
725                         lt.addLast(return_list, element)
726
727
728     # Determinar tamaño, empaquetar variables y retornar.
729     size_return_list = lt.size(return_list)
730     return(size_return_list, return_list)
```

Asumiendo que dentro del rango de longitud del requerimiento hay m datos, y que el mapa *longitude* del catálogo tiene n datos, el orden de complejidad de esta función es

$$O(m + 2\log(n)).$$

Esto debido a que:

1- Si el rango tiene m elementos, entonces se deberá iterar sobre este, lo cual tendría complejidad lineal (m).
2- Las funciones $om.ceiling()$ y $om.floor()$, en el peor de los casos, tendrán complejidad $2\log(n)$, debido a que sería necesario hacer una búsqueda hasta las hojas del árbol *longitude*.
3- Cada uno de los Árboles Rojo-Negro que se guardarán en cada iteración en la línea 713 tendrán un tamaño muy pequeño, ya que es muy improbable que varios avistamientos hayan sido registrados en exactamente la misma longitud. Por esta misma razón, se podría considerar que iterar la lista que se crea en cada iteración en la línea 723 tendría complejidad constante; en términos generales, se podría considerar que las operaciones realizadas de la línea 713 a la 725 tendrían complejidad constante.
4- Todas las demás operaciones son constantes.