Universidad de los Andes
Estr. de Datos y Algoritmos – Sección 6

José Nicolás Cárdenas (201922006 - j.cardenast)
Andrés Leonardo Beltrán (202014143 - al.beltran)

Reto 2: Documento de Análisis

Implementación requerimientos individuales (3 y 4):

- ➔ Requerimiento 3: José Nicolás Cárdenas.
- ➔ Requerimiento 4: Andrés Leonardo Beltrán.

Análisis De Complejidad

❖ Requerimiento 1

La signatura de la función de este requerimiento se muestra a continuación:

```
# Función del requerimiento 1.
def req_1 (catalog: dict, first_year: int, last_year: int) -> dict:
    """
        Esta función retorna una listas cronológicamente ordenada que contiene auqellos artistas
        que nacieron dentro de un rango de años indicado por el usuario.


        Parámetros:
            -> catalog (dict): catálogo.
            -> first_year (int): año inicial.
            -> last_year (int): año final.

        Retorno:
            -> (dict): diccionario que representa la lista que contiene la respuesta.

    """
```

El primer bloque de código de la función es el siguiente:

```
863     # Crear rango que representa un intervalo cerrado que empieza en first_year
864     # y termina en last_year.
865     interval = range(first_year, last_year + 1)
866
867     # Crear lista que contendrá a los artistas que nacieron dentro de interval.
868     # Guardar el map "BeginDate" del catálogo.
869     return_list = lt.newList("ARRAY_LIST")
870     map_BeginDate = catalog["BeginDate"]
```

1- (865) No depende del tamaño de los datos, entonces tiene complejidad constante.
2- (869-870) La creación de una nueva lista y la asignación de una variable tienen complejidad constante.

El segundo bloque de la función es el siguiente:

```
873     # Iterar sobre cada año que se encuentra dentro de interval.
874     for year in interval:
875
876         # Si el año se encuentra en el mapa map_BeginDate.
877         if (mp.get(map_BeginDate, year) != None):
878
879             # Guardar la lista de los artistas que nacieron durante year.
880             artists_list = mp.get(map_BeginDate, year)["value"]
881
882             # Añadir cada artista a return_list.
883             for artist in (lt.iterator(artists_list)):
884                 lt.addLast(return_list, artist)
```

3- (874) Supóngase que el tamaño del intervalo es igual a m. En este caso, como se debe iterar sobre cada valor (año) que se encuentra dentro de este, esta iteración tiene una complejidad $O(m)$.
4- (877-880) Entrar (o no) a un condicional tiene complejidad constante, y la función $get()$ de un mapa tiene complejidad constante también.
5- (883-884) Se debe iterar sobre la lista de *ConstituentID* que tiene la obra de la iteración actual; aun así, el tamaño de dicha lista es bastante pequeño (menor que 20), y debido a que la función $addLast()$ de una lista tiene complejidad constante, entonces puede considerar que la iteración completa tiene complejidad constante.

El último bloque de código de la función es el siguiente:

```
887     # Ordenar lista cronológicamente según los BeginDates y retornarla.
888     ordered_list = qui.sort(return_list, cmp_BeginDates)
889     return ordered_list
```

6- (888-889) Si el tamaño del intervalo es igual a m, entonces ordenar una lista que contiene dicha cantidad de elementos mediante el algoritmo *Quicksort* tiene una complejidad $O(m^2)$. El retorno de una variable en una función tiene complejidad constante.

Considerando todo esto, la complejidad temporal del requerimiento 1 es de:

$$O(m^2)$$

Hay que tener en cuenta que el valor de m no puede ser un valor mayor que 2021, por lo que la complejidad temporal de este requerimiento es cosiderablemente baja.

❖ Requerimiento 2

La signatura de la función de este requerimiento se muestra a continuación:

```
# Función del requerimiento 2.
def req_2 (catalog: dict, first_date: str, last_date: str) -> tuple:
    """
        Dado un rango de fechas, esta función retorna una tupla que contiene lista cronológicamente ordenada
        que almacena las obras que fueron compradas por el museo durante dicho rango de tiempo y la cantidad
        de obras que fueron adquiridas por compra.

        Parámetros:
            -> catalog (dict): catálogo.
            -> first_date (str): fecha inicial.
            -> last_date (str): fecha final.

        Retorno:
            -> (tuple): tupla cuyo primer elemento es la lista que contiene la respuesta
                        y cuyo segundo elemento representa la cantidad de obras que fueron
                        adquiridas por compra.

    """
```

El primer bloque de código de la función es el siguiente:

```
912     # Crear lista y entero de retorno.
913     ordered_list = lt.newList("SINGLE_LINKED")
914     num_purch = 0
915
916     # Crear variable que guarda el mapa 'DateAcquired' del catálogo.
917     map_DateAcquired = catalog['DateAcquired']
918
919     # Crear variables con las fechas de adquisición modificadas.
920     mod_first_date = date.datetime.strptime(first_date, '%Y-%m-%d')
921     mod_last_date = date.datetime.strptime(last_date, '%Y-%m-%d')
922
923     # Crear lista que contiene las llaves de map_DateAcquired.
924     keys_map_DateAcquired = lt.iterator(mp.keySet(map_DateAcquired))
```

1- (913-917) Crear una lista y realizar una sentencia de asignación son operaciones que tiene complejidad constante.
2- (919-921) La innovación función *strptime*() del módulo *datetime* no dependerá del tamaño de los datos, por lo que estas líneas de código tienen complejidad constante.
3- (923-924) Crear el iterador de una lista tiene complejidad constante.

El segundo bloque de la función es el siguiente:

```
927     # Iterar sobre todas las llaves de keys_map_DateAcquired.
928     for DateAcquired in keys_map_DateAcquired:
929
930         # Crear variables con el DateAcquired de la iteración actual modificada.
931         mod_DateAcquired = date.datetime.strptime(DateAcquired, '%Y-%m-%d')
932
933         # Crear variable que determina si mod_DateAcquired se encuentra dentro del rango.
934         in_range = (mod_DateAcquired >= mod_first_date and mod_DateAcquired <= mod_last_date)
```

4- (927-928) Se desea iterar sobre el mapa 'DateAcquired' del catálogo. Dicho tiene un tamaño cercano a la cantidad de obras que hay en el catálogo. Por ende, esta operación tiene una complejidad temporal $O(n)$.
5- (930-931) Como se explicó anteriormente, esta operación tiene complejidad constante.
6- (933-934) Determinar el valor booleano de una comparación tiene complejidad constante.

El tercer bloque de la función es el siguiente:

```
937         # Si está en el rango.
938         if (in_range):
939
940             # Crear variable que guarda la lista de obras (el valor) de la llave DateAcquired.
941             list_DateAcquired = lt.iterator(mp.get(map_DateAcquired, DateAcquired)['value'])
942
943             # Iterar sobre las obras de list_DateAcquired.
944             for artwork in list_DateAcquired:
945
946                 # Añadir la obra a la lista de retorno.
947                 lt.addLast(ordered_list, artwork)
948
949                 # Guardar línea de crédito de la obra, determinar si es igual a 'Purchase' y, si lo es,
950                 # aumentar num_purch en 1.
951                 artwork_CreditLine = artwork['CreditLine'].lower()
952                 is_purchase = (artwork_CreditLine == 'purchase' or artwork_CreditLine == 'purchased')
953                 if (is_purchase):
954                     num_purch += 1
```

7- (937-941) Entrar (o no) a un condicional tiene complejidad constante, así como crear el iterador de una lista.
8- (944) Se desea iterar sobre una lista que contiene la cantidad de obras que fueron adquiridas por el museo en una misma fecha. Como es muy improbable que dos o más obras hayan sido adquiridas en un mismo día (y de ser así, es improbable que sea una gran cantidad), entonces se puede considerar que esta iteración tiene complejidad constante.
9- (946-954) Añadir un elemento a la última posición de una lista tiene complejidad constante, así como acceder al elemento de un diccionario de Python y realizar una operación de comparación. También tiene complejidad constante entrar (o no) a un condicional tiene complejidad constante, así como crear el iterador de una lista, así como aumentar en uno el valor de una variable.

El último bloque de la función es el siguiente:

```
956     # Ordenar lista.
957     ordered_ordered_list = mer.sort(ordered_list, cmp_by_DateAcquired)
958
959     # Empaquetar respuestas y retornalas.
960     return_tuple = (ordered_ordered_list, num_purch)
961     return return_tuple
```

10- (956-961) Se desea ordenar una lista que contiene todas las obras que fueron compradas dentro de las fechas indicadas por el usuario mediante *Mergesort*. La complejidad de esta operación dependerá del tamaño del intervalo (m); potencialmente, este podría ser muy cercano a n. Por ende, la complejidad de esta operación es de $O(m\ log(m))$. Por otro lado, empaquetar variables y retornar una variable de una función tienen complejidad constante.

Considerando todo esto, la complejidad temporal del requerimiento 1 es de:

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

Como se mencionó anteriormente, debido a que el valor de m podría potencialmente ser igual (o muy cercano) a n, entonces las complejidades asociadas a este deben considerarse. Se puede evidenciar que la compeljidad de este requerimiento es bastante alta, y de realizarse para los datos de los archivos *large* se podría llegar a incurrir en una demora considerable.

❖ Requerimiento 3

La signatura de la función de este requerimiento se muestra a continuación:

```
# Función del requerimiento 3.
def req_3 (catalog: dict, param_DisplayName: str) -> tuple:
    """
        Dado el nombre de un/una artista, esta función retorna

        Parámetros:
            -> catalog (dict): catálogo.
            -> param_DisplayName (str): nombre del artista.

        Retorno:
            -> (tuple): tupla cuyo primer elemento es la lista que contiene la respuesta
                        y cuyo segundo elemento representa la cantidad de obras que fueron
                        adquiridas por compra.

    """
```

El primer bloque de código de la función es el siguiente:

```
982     # Guardar el mapa de las técnicas del artista con nombre param_DisplayName.
983     Mediums_map = mp.get(catalog['DisplayName'], param_DisplayName)['value']
984
985     # Crear variables de retorno.
986     total_artworks = 0
987     total_mediums = mp.size(Mediums_map)
988     most_used_medium = ''
989     list_most_used_medium = None
990     list_medium_sizes = lt.newList('SINGLE_LINKED')
```

1- (982-990) Acceder a un elemento de un mapa, determinar su tamaño, realizar una sentencia de asignación y crear una nueva lista son todas operaciones de complejidad constante.

El segundo bloque de código de la función es el siguiente:

```
 993     # Recorrer todas las técnicas de Mediums_map.
 994     for medium in lt.iterator(mp.keySet(Mediums_map)):
 995
 996         # Guardar tamaño de lista que contiene las obras que fueron creadas usando la técnica medium.
 997         # y aumentar valor total_artworks.
 998         size_artwork_list = lt.size(mp.get(Mediums_map, medium)['value'])
 999         total_artworks += size_artwork_list
1000
1001         # Añadir tupla (medium, size_artwork_list) a list_medium_sizes.
1002         lt.addLast(list_medium_sizes, (medium, size_artwork_list))
```

2- (993-994) Se desea recorrer una lista que contiene las técnicas que ha usado un artista para crear sus obras. Teniendo en cuenta que este número es pequeño, se puede considerar que toda esta iteración tiene complejidad constante.
3- (996-1002) Determinar el tamaño de un mapa, realizar una sentencia de asignación y añadir un elemento a una lista en la última posición tienen todas complejidad constante.

El último bloque de código de la función es el siguiente:

```
1004        # Organizar list_medium_sizes.
1005        ordered_list_medium_sizes = qui.sort(list_medium_sizes, cmp_Mediums)
1006
1007        # Guardar técnica más usada.
1008        most_used_medium = lt.getElement(ordered_list_medium_sizes, 1)[0]
1009
1010        # Guardar en list_most_used_medium la lista de las obras que fueron creadas con most_used_medium.
1011        # Crear varable que guarda dicha lista ordenada.
1012        list_most_used_medium = mp.get(Mediums_map, most_used_medium)['value']
1013        ordered_list_most_used_medium = qui.sort(list_most_used_medium, cmp_artworks_by_Date)
1014
1015
1016        # Armar tupla de retorno y retornarla.
1017        answer = (total_artworks, total_mediums, most_used_medium, ordered_list_most_used_medium, ordered_list_medium_sizes)
1018        return answer
```

4- (1004-1005) Asumiendo que el artista ha usado m técnicas para crear sus obras, la complejidad de ordenar una lista que las contiene usando *Quicksort* sería $O(m^2)$.
5- (1007-1012) Acceder al primer elemento de un arreglo tiene complejidad constante, así como acceder al elemento de un mapa.
6- (1013-1018) Asumiendo que la cantidad de obras del catálogo que fueron creadas usando la técnica más usada del artista es igual a s, entonces la complejidad de ordenar una lista que las contiene sería $O(s^2)$ usando *Quicksort*. Empaquetar variables y retornar una variable en una función tiene complejidad constante.

Considerando todo esto, la complejidad temporal del requerimiento 1 es de:

$$O(m^2 + s^2)$$

Aun así, es necesario tener en cuenta que los valores de m y s son considerablemente pequeños (potencialmente cercanos a 1). Por ende, la complejidad temporal de este requerimiento es cosiderablemente baja.

❖ Requerimiento 4

La signatura de la función de este requerimiento se muestra a continuación:

```
# Función del requerimiento 4.
def req_4 (catalog: dict) -> tuple:
    """
        Esta función retorna una tupla con los siguientes elementos:
            1- Una lista cuyos elementos son mapas; las parejas llave-valor de estos son:
                -> Llaves: 'Nationality' y 'Size'.
                -> Valores: cadena referente a una nacionalidad y cantidad de obras
                            cuyo/s autor/es pertenece/n a dicha nacionalidad.
            2- Lista con las obras pertenecientes a la nacionalidad que más obras tiene.

        Parámetro:
            -> catalog (dict): catálogo.

        Retorno:
            -> (dict): tupla con los elementos descritos anteriormente.

    """
```

El primer bloque de código de la función es el siguiente:

```
1040    map_natio = catalog['Nationality']          # Mapa nacionalidades.
1041    lt_natio = lt.newList('ARRAY_LIST')         # Mapa de retorno.
1042
1043    # Recorrer cada pareja llave-valor de map_natio.
1044    for natio in lt.iterator(mp.keySet(map_natio)):
```

1- (1040-1041) Realizar una sentencia de asignación y crear una nueva lista tienen complejidad constante.
2- (1043-1044) Se desea iterar sobre las llaves del mapa 'Nationality' del catálogo. El tamaño de este es un número cercano a 200, denótese con m; por ende, la complejidad de esta iteración será de $O(m)$.

El segundo bloque de código de la función es el siguiente:

```
1046        # Guardar lista de obras la nacionalidad y determinar su tamaño.
1047        lt_artw_natio = mp.get(map_natio, natio)['value']
1048        size_lt_artw_nacion = lt.size(lt_artw_natio)
1049
1050        # Crear mapa con pareja nacio-size_lt_artw_nacion, añadir parejas y añadir mapa a lt_natio.
1051        map_nacio_size = mp.newMap(numelements = 10, maptype = 'CHAINING')
1052        mp.put(map_nacio_size, 'Nationality', natio)
1053        mp.put(map_nacio_size, 'Size', size_lt_artw_nacion)
1054        lt.addLast(lt_natio, map_nacio_size)
```

3- (1046-1048) Obtener el elemento de un mapa y determinar el tamaño de una lista son operaciones de complejidad constante.
4- (1050-1054) Crear un nuevo mapa, añadir parejas llave-valor a este y añadir un elemento a la última posición de una lista son todas operaciones de complejidad constante.

El último bloque de código de la función es el siguiente:

```
1057    # Ordenar lt_natio, determinar nacionalidad con más obras y guardar sus obras.
1058    ordered_lt_natio = qui.sort(lt_natio, cmp_nacion)
1059    nacio_most_artw = mp.get(lt.getElement(ordered_lt_natio, 1), 'Nationality')['value']
1060    lt_nacio_most_artw = mp.get(map_natio, nacio_most_artw)['value']
1061
1062    # Empaquetar variables y retornarlas.
1063    return_tuple = (lt_natio, lt_nacio_most_artw)
1064    return return_tuple
```

5- (1057-1058) Ordenar una lista que contiene una cantidad de elementos igual al tamaño del mapa 'Nationality' del catálogo (es decir, cercano a 200) denotado con m mediante *Quicksort* tiene complejidad $O(m^2)$.
6- (1059-1064) Obtener un elemento de un mapa, empaquetar variables y retornar a variable es una función tienen todas complejidad constante.

Considerando todo esto, la complejidad temporal del requerimiento 1 es de:

$$O(m^2)$$

Aun así, es necesario tener en cuenta que el valor de m es considerablemente pequeño (cercano a 200). Por ende, la complejidad temporal de este requerimiento es cosiderablemente baja.

❖ Requerimiento 5

La signatura de la función de este requerimiento se muestra a continuación:

```
# Función del requerimiento 5.
def req_5 (catalog: dict, param_Dep: str) -> tuple:
    """
        Dado el nombre de un departamento del museo, esta función retorna un tupla que contiene los siguientes elementos:
            1- Precio en USD en el que se debería incurrir si se desea transportar todas las obras de
               dicho dpto.
            2- Peso estimado de todas las obras.
            3- Lista ordenada de las obras del dpto. según su fecha.
            4- Lista ordenada de las obras del dpto. según su precio de transporte.

        Parámetros:
            -> catalog (dict): catálogo.
            -> param_Dep (str): cadena referente a un dpto. del museo.

        Retorno:
            -> (tuple): tupla con los elementos descritos anteriormente.

    """
```

El primer bloque de código de la función es el siguiente:

```
1089    map_Dep = catalog['Department']                    # Mapa 'Department' del catálogo.
1090    artwork_list = mp.get(map_Dep, param_Dep)['value'] # Lista que contiene las obras pertenecientes al dpto.
1091    total_price = 0.0                                  # Variable precio total.
1092    total_weight = 0.0                                 # Variable peso total.
1093
1094    # Crear lista que contiene las obras que pertenecen al dpto. y que tiene una fecha registrada.
1095    lt_artworks_dep_date = lt.newList('ARRAY_LIST')
```

1- (1089-1095) Realizar una sentencia de asignación, obtener el elemento de un mapa y crear una lista son todas operaciones de complejidad constante.

El segundo bloque de código de la función es el siguiente:

```
1098    # Recorrer artwork_list.
1099    for artwork in lt.iterator(artwork_list):
1100
1101        # Calcular precio de la obra y sumarlo al total.
1102        price = calc_price(artwork)
1103        total_price += price
1104
1105        # Calcular peso de la obra y sumarlo al total.
1106        weight = artwork['Weight (kg)']
1107        total_weight += weight
1108
1109        # Añadir obra a lt_artworks_dep_date si tiene fecha registrada.
1110        if (artwork['Date'] != ''):
1111            lt.addLast(lt_artworks_dep_date,artwork)
```

2- (1098-1099) Asumiendo que el departamento especificado por parámetro tiene m obras, entonces realizar una iteración sobre una lista que contiene a dichas tendrá una complejidad $O(m)$.
3- (1101-1102) La función *calc_price*() no depende del tamaño de los datos entrada, por lo que su invocación tiene complejidad constante.

4- (1103-1111) Realizar una sentencia de asignación, entrar (o no) a un condicional y añadir un elemento a la última posición de una lista son todas operaciones de orden constante.

El último bloque de código de la función es el siguiente:

```
1114    # Variable que guarda la lista que contiene las obras del dpto..
1115    # Dicha está ordenada según las fechas de cada obra.
1116    lt_ord_art_date = qui.sort(lt_artworks_dep_date, cmp_artworks_by_Date)
1117
1118    # Variable que guarda la lista que contiene las obras del dpto..
1119    # Dicha está ordenada según los precios de transporte de cada obra.
1120    lt_ord_art_price = qui.sort(artwork_list, cmp_artworks_by_Price)
1121
1122    # Empaquetar variables y retornarlas.
1123    return_tuple = (round(total_price, 3), round(total_weight, 3), lt_ord_art_date, lt_ord_art_price)
1124    return return_tuple
```

5- (1114-1120) Ordenar dos listas que contiene las obras que pertenecen al departamento (m) mediante *Quicksort* tiene complejidad $O(2m^2)$.
6- Empaquetar variables y retornar una variable en una función son operaciones de complejidad constante.

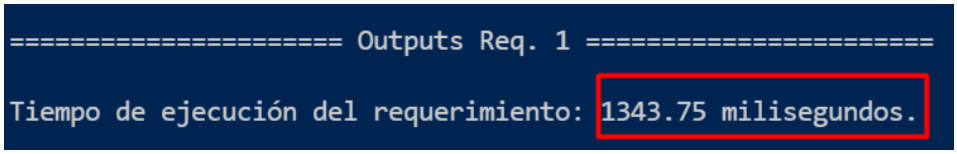
Considerando todo esto, la complejidad temporal del requerimiento 1 es de:

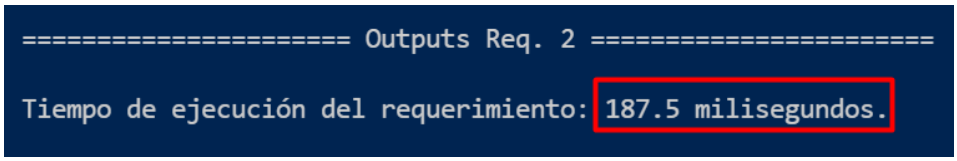
$$O(2m^2)$$

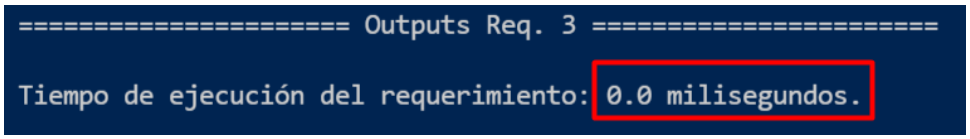
El valor de m equivaldrá a la cantidad de obras que tiene un departamento del museo. Este puede ser un número considerablemente grande.

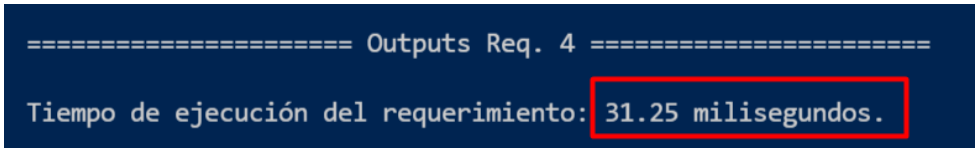
Pruebas de Tiempos de Ejecución

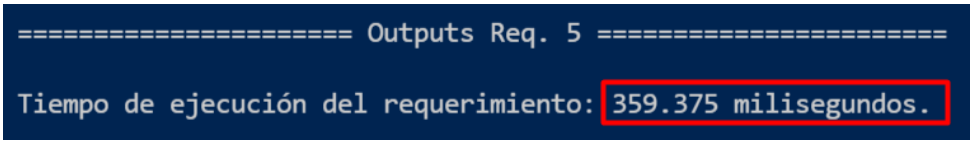
Usando los archivos de tamaño *small*, cada requerimiento arrojó los siguientes tiempos de ejecución:



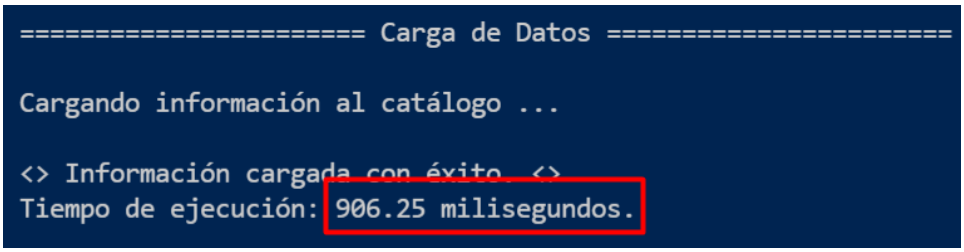








La carga de datos tuvo el siguiente tiempo de ejecución:



Comparación Complejidad Reto 1

En el reto número 1 no se realizó un análisis de los órdenes de crecimiento temporal de cada requerimiento. Aun así, se puede concluir que implementar y resolver los requerimientos mediante el uso de tablas de Hash conlleva a una complejidad temporal increíblemente menor que mediante el uso de listas. Esto se debe, principalmente, a dos razones:

1- Añadir y obtener parejas llave-valor de una tabla de Hash son operaciones que tiene complejidad constante.
2- Debido a la gran flexibilidad que tienen las llaves, y considerando el punto anterior, era posible crear mapas que tuviesen exactamente la información que se requería. Por ejemplo, para el requerimiento 3 era necesario obtener las obras que fueron creadas usando una técnica determinada. Por ende, debido a que era posible cargar los datos de tal forma que se tuviese una lista con las obras creadas con dicha técnica, entonces acceder a la información interés de estas se podía realizar en tiempo constante, y por ello este requerimiento presenta un tiempo de ejecución tan bajo.