Análisis de Complejidad

Nombre: Nicolas Merchan Cuestas

Código: 202112109

Correo: n.merchan@uniandes.edu.co

Notas:

- Los resultados de las pruebas de ejecución y las correspondientes gráficas de representación se encuentran en '*Datos - Análisis de Complejidad.xslx*'.
- El programa 'information.py' proporciona información sobre el número de artistas, obras de arte, años de nacimiento, años de adquisición, nacionalidades y departamentos del museo.
- 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

```
251 def getArtistsByBirthYear(catalog, data_structure, initial_birth_year, end_birth_year):
252     artists_birth_years_interval = lt.newList(data_structure)
253     birth_years_map = catalog['birth_years']
254     for year in range(initial_birth_year, end_birth_year + 1):
255         year = str(year)
256         if mp.contains(birth_years_map, year):
257             artists_birth_year = me.getValue(mp.get(birth_years_map, year))
258             for artist in lt.iterator(artists_birth_year):
259                 lt.addLast(artists_birth_years_interval, artist)
260     return artists_birth_years_interval
```

La complejidad asociada a **getArtistsByBirthYear()** en el Reto 2 es **O(1)**, dado que la función adquiere la información de los artistas nacidos en un rango de años específico por medio de la tabla de hash **catalog['birth_years']**, la cual fue creada en el cargue de datos. En dicha tabla de hash la llave es el año de nacimiento y el valor asociado a cada llave es una lista con los artistas nacidos en el año correspondiente. De ese modo, para obtener los artistas nacidos en un rango de años dado, solo es necesario consultar los respectivos valores de los años en el rango.

```
400 def SortArtistByBirthYear(sub_list, sorting_method, initial_year_birth, end_year_birth, data_structure):
401     start_time = time.process_time()
402 
403     sub_list = sub_list.copy()
404     artists_birth_year_range_list = FilteringArtistsByBirthYear(sub_list, initial_year_birth,
405                                                                  end_year_birth, data_structure)
406     sorted_list = SortingMethodExecution(sorting_method, artists_birth_year_range_list, cmpArtistByBirthDate)
407 
408     stop_time = time.process_time()
409     elapsed_time_mseg = elapsed_time_mseg = (stop_time - start_time)*1000
410 
411     return elapsed_time_mseg, sorted_list
```

La complejidad asociada a **FilteringArtistsByBirthYear()** en el Reto 1 es **O(n)**, dado que separan los artistas nacidos en el rango de años indicado por medio de la comparación individual de cada elemento. De manera similar, la función **SortingMethodExecution()** realiza un ordenamiento en función del año de nacimiento de los artistas nacidos en el rango de años especificado. La complejidad de esta operación depende del tipo de algoritmo de ordenamiento utilizado. La complejidad es **O(n^2)**, **O(n^3/2)**, **O(n^2)** y **O(nlog(n))** para los algoritmos Insertion Sort, Shell Sort, Quick Sort y Merge Sort, respectivamente. Finalmente, la complejidad es modelada en su mayor parte por la complejidad en **SortingMethodExecution()**.

La complejidad del requerimiento 1 es mejor en el Reto 2 que en el Reto 1 por una diferencia de **O(1)** a **O(n)** al momento de buscar los artistas nacidos en un rango de años dados.

Requerimiento 2

```
264 def getArtworksByAdquisitonDate(catalog, data_structure, sorting_method,
265                                                 initial_adquisiton_date, end_adquisition_date):
266     date_adquired_artworks_list = lt.newList(data_structure)
267 
268     adquisition_years_map = catalog['adquisition_years']
269     initial_adquisition_year = getAdquisitionYear(initial_adquisiton_date)
270     initial_adquisiton_date_in_days = TransformationDateToDays(initial_adquisiton_date)
271     end_adquisition_year = getAdquisitionYear(end_adquisition_date)
272     end_adquisition_date_in_days = TransformationDateToDays(end_adquisition_date)
273 
274     if mp.contains(adquisition_years_map, initial_adquisition_year):
275         first_year_artworks = me.getValue(mp.get(adquisition_years_map, initial_adquisition_year))
276         for artwork in lt.iterator(first_year_artworks):
277             date = TransformationDateToDays(artwork['DateAcquired'])
278             if date >= initial_adquisiton_date_in_days:
279                 lt.addLast(date_adquired_artworks_list, artwork)
280 
281     for year in range(initial_adquisition_year + 1, end_adquisition_year):
282         if mp.contains(adquisition_years_map, year):
283             year_artworks = me.getValue(mp.get(adquisition_years_map, year))
284             for artwork in lt.iterator(year_artworks):
285                 lt.addLast(date_adquired_artworks_list, artwork)
286 
287     if mp.contains(adquisition_years_map, end_adquisition_year):
288         last_year_interval_artworks = me.getValue(mp.get(adquisition_years_map, end_adquisition_year))
289         for artwork in lt.iterator(last_year_interval_artworks):
290             date = TransformationDateToDays(artwork['DateAcquired'])
291             if date <= end_adquisition_date_in_days:
292                 lt.addLast(date_adquired_artworks_list, artwork)
293 
294     SortingMethodExecution(sorting_method, date_adquired_artworks_list, cmpArtworksByDateAcquired)
```

La complejidad asociada a **getArtworksByAdquisitionDate()** en el Reto 2 está determinada principalmente por el tipo de algoritmo de ordenamiento utilizado. Por una parte, la función hace uso de la tabla de hash **catalog[‘adquisition_years’]** para encontrar todas las obras de arte en un rango de fechas. Las llaves de **catalog[‘adquisition_years’]** son los años de adquisición y el valor asociado a dicha llave es una lista con todas las obras de arte adquiridas en el año en cuestión. La tabla de hash **catalog[‘adquisition_years’]** fue creada en el cargue de datos. Por otra parte, la complejidad del proceso de ordenamiento depende del algoritmo utilizado. La complejidad es

O(n), **O(nlog(n)), O(nlog(n))** y **O(nlog(n))** para los algoritmos Insertion Sort, Shell Sort, Quick Sort y Merge Sort, respectivamente.

```
428 def SortArtworksAdquisitionRange(sub_list, sorting_method, initial_date_adquisition,
429                                                                 end_date_adquisition, data_structure):
430     start_time = time.process_time()
431     sub_list = sub_list.copy()
432 
433     artworks_adquisition_date_range_list = FilteringArtworksByAdquisitionDate(sub_list, initial_date_adquisition,
434                                                                               end_date_adquisition, data_structure)
435     purchase_artworks_num = lt.size(FilteringArtworksByAdquisitionDateAndCreditLine(artworks_adquisition_date_range_list,
436                                                                                     data_structure))
437     sorted_list_by_date = SortingMethodExecution(sorting_method, artworks_adquisition_date_range_list,
438                                                                                   cmpArtworkByDateAcquired)
439 
440     stop_time = time.process_time()
441     elapsed_time_mseg = elapsed_time_mseg = (stop_time - start_time)*1000
442 
443     return elapsed_time_mseg, sorted_list_by_date, purchase_artworks_num
```

La complejidad asociada a **FilteringArtworksByAduisitionDate()** en el Reto 1 es **O(n)**, dado que separan las obras de arte adquiridas en el rango de fechas indicado por medio de la comparación individual de cada elemento. Igualmente, la complejidad de **FilteringArtworksByAduisitionDate()** es **O(n)**, porque separa las obras de arte adquiridas por compra revisando individualmente cada obra de arte. De manera similar, la función **SortingMethodExecution()** realiza un ordenamiento en función de la fecha de adquisición de las obras adquiridas en el rango de fechas especificado. La complejidad de esta operación depende del tipo de algoritmo de ordenamiento utilizado. La complejidad es **O(n)**, **O(nlog(n)), O(nlog(n))** y **O(nlog(n))** para los algoritmos Insertion Sort, Shell Sort, Quick Sort y Merge Sort, respectivamente. Ello se debe a que las obras de arte fueron agregadas a la lista de clasificación a medida que fueron adquiridas por el museo. De ese modo, existe un orden dentro de la lista a ordenar. Finalmente, la complejidad es modelada en su mayor parte por la complejidad en **SortingMethodExecution()**.

La complejidad del requerimiento 2 es más favorable en el Reto 2 que en el Reto 1. La diferencia en complejidades de los retos se da en la búsqueda de las obras de arte adquiridas en un rango de fechas dado. Así la complejidad de búsqueda es **O(n)** y **O(1)** para el Reto 1 y el Reto 2, respectivamente. El Reto 2 se hace uso de una tabla de hash para encontrar las obras de arte.

Requerimiento 3

```
300  def getArtworksByMediumAndArtist(catalog, artist_name):
301      artist_names_map = catalog['artists_names']
302      num_more_artworks = 0
303      num_total_artworks = 0
304      num_total_mediums = 0
305      list_more_artworks = lt.newList()
306      name_more_artworks = ''
307      if mp.contains(artist_names_map, artist_name):
308          artist_Id = me.getValue(mp.get(catalog['artists_names'], artist_name))
309          mediums_keys_list = me.getValue(mp.get(catalog['artists_Ids'], artist_Id))['mediums_keys']
310          num_total_mediums = lt.size(mediums_keys_list)
311          mediums_map = me.getValue(mp.get(catalog['artists_Ids'], artist_Id))['mediums']
312          for medium_name in lt.iterator(mediums_keys_list):
313              medium_artworks = me.getValue(mp.get(mediums_map, medium_name))
314              num_medium = lt.size(medium_artworks)
315              num_total_artworks += num_medium
316              if num_medium > num_more_artworks:
317                  num_more_artworks = num_medium
318                  name_more_artworks = medium_name
319                  list_more_artworks = medium_artworks
320      return list_more_artworks, num_total_artworks, num_total_mediums, name_more_artworks
```

La complejidad asociada a **getArtworksByMediumAndArtist()** en el Reto 2 es **O(1)**. Ello se debe a que se hace uso de la tabla de hash **catalog[‘artists_Ids’]** y se simplifica la complejidad del proceso de búsqueda de las obras de arte asociadas al artista a **O(1)**. Las llaves de **catalog[‘artists_Ids’]** son los códigos de identificación únicos de los artistas, estos son obtenidos por medio de la tabla de hash **catalog[‘artists_names’]**, la cual a su vez tiene como llaves los nombres de los artistas y como valor el código de identificación asociado a los artistas. Los valores asociados a las llaves de **catalog[‘artists_Ids’]**, son diccionarios que contienen la información del artista y las obras de arte ordenadas por técnica. Tanto **catalog[‘artists_Ids’]**, como **catalog[‘artists_names’]** fueron creadas en el cargue de datos. Finalmente, solo es necesario comparar la cantidad de obras asociadas a una técnica de un artista utilizando la información del valor de la llave en **catalog[‘artists_Ids’]**.

```
447  def ClasifyArtistsTechnique(sub_list, lst, sorting_method, artist_name, data_structure):
448      start_time = time.process_time()
449      sub_list = sub_list.copy()
450
451      information = CreationArtistTechniquesInformation(sub_list, lst, artist_name, data_structure)
452      artist_artworks = information[0]
453      artist_techniques = information[1]
454      sorted_artist_techniques = SortingMethodExecution(sorting_method, artist_techniques, cmpTechniquesBySize)
455
456      stop_time = time.process_time()
457      elapsed_time_mseg = (stop_time - start_time)*1000
458
459      return elapsed_time_mseg, artist_artworks, sorted_artist_techniques
```

La función **CreationArtistsTechniquesInformation()** en el Reto 1 crea una TAD lista que contiene como elementos una lista con el nombre de la técnica y una TAD lista de todas la obras del autor que hacen uso de dicha técnica. La complejidad de esta función es **O(n)**, porque la función compara todas las obras respecto al autor y técnica utilizada en la mismas. De manera similar, la función **SortingMethodExecution()** realiza un ordenamiento en función de la cantidad de obras del artista en cuestión por técnica utilizada . La complejidad de esta operación depende del tipo de

algoritmo de ordenamiento utilizado. La complejidad es **O(n^2)**, **O(n^3/2)**, **O(n^2)** y **O(nlog(n))** para los algoritmos Insertion Sort, Shell Sort, Quick Sort y Merge Sort, respectivamente. Finalmente, la complejidad es modelada en su mayor parte por la complejidad en **SortingMethodExecution()**.

La complejidad del requerimiento 3 es mejor en el Reto 2 que en el Reto 1, dado que la complejidad asociada a la búsqueda de datos es **O(1)** y **O(n)** en los casos del Reto 2 y Reto 1, respectivamente.

Requerimiento 4

```
324 def getNationalitiesByNumArtworks(catalog, data_structure, sorting_method):
325     nationalities_names_list = catalog['nationalities_keys']
326     num_nationalities_list = lt.newList(data_structure)
327     major_nationality_artworks_list = lt.newList()
328     if lt.size(nationalities_names_list) >= 1:
329         for nationality in lt.iterator(nationalities_names_list):
330             num_artworks = lt.size(me.getValue(mp.get(catalog['nationalities'], nationality)))
331             lt.addLast(num_nationalities_list, (nationality, num_artworks))
332         SortingMethodExecution(sorting_method, num_nationalities_list, cmpNationalityByNumArtworks)
333         major_nationality_name = lt.getElement(num_nationalities_list, 1)[0]
334         major_nationality_artworks_list = me.getValue(mp.get(catalog['nationalities'], major_nationality_name))
335     return major_nationality_artworks_list, num_nationalities_list
```

La complejidad asociada a **getNationalitiesByNumArtworks()** en el Reto 2 es aproximadamente **O(1)**. Ello se debe a que existe una cantidad fija de nacionalidades en la tabla de hash **catalog[‘nationalities’]**, la cual tiene como llaves el nombre de las nacionalidades y como valores asociados las obras de arte de la nacionalidad asociada. Así, la clasificación de nacionalidades por número de obras de arte se hace recorriendo la lista de nacionalidades **catalog[‘nationalities_keys’]**, la cual tiene tamaño constante a medida que se agregan datos, y comparando el tamaño de las listas de obras de arte asociadas. Tanto **catalog[‘nationalities’]**, como **catalog[‘nationalities’]** fueron creadas en el cargue de datos.

```
463 def ClasifyArtworksByNationality(sub_list, sorting_method, artists_ID_dict, data_structure):
464     start_time = time.process_time()
465     sub_list = sub_list.copy()
466 
467     num_artworks_nationalities = CreateDictNumPerNationality(sub_list, artists_ID_dict)
468     artworks_nationalities_list = CreateNationalityNumList(num_artworks_nationalities, data_structure)
469     sorted_list = SortingMethodExecution(sorting_method, artworks_nationalities_list, cmpNationalitiesBySize)
470 
471     stop_time = time.process_time()
472     elapsed_time_mseg = (stop_time - start_time)*1000
473 
474     return elapsed_time_mseg, sorted_list
```

La función **CreateDictNumPerNationality()** en el Reto 1 cuenta la cantidad de obras de arte de cada nacionalidad y guarda dicha información en un diccionario donde la nacionalidad es la llave y el número de obras de arte es el valor de dicha llave. La función **CreateNationalityNumList()** convierte el diccionario generado en **CreateDictNumPerNationality()** en un TAD lista donde los elementos es son lista que contienen las nacionalidades y sus respectivos números de obras. La complejidad del proceso anteriormente mencionado es **O(n)**, dado que para completarlo es necesario recorrer la lista exactamente una vez comparando todas las obras de arte. De manera similar, la función **SortingMethodExecution()** realiza un ordenamiento en función de la cantidad

de obras del artista en cuestión por nacionalidad . La complejidad de esta operación depende del tipo de algoritmo de ordenamiento utilizado. La complejidad es **O(n^2)**, **O(n^3/2)**, **O(n^2)** y **O(nlog(n))** para los algoritmos Insertion Sort, Shell Sort, Quick Sort y Merge Sort, respectivamente. Finalmente, la complejidad es modelada en su mayor parte por la complejidad en **SortingMethodExecution()**.

La complejidad del requerimiento 4 es mejor en el Reto 2 que en el Reto 1, dado que la complejidad en el Reto 2 es constante y aquella del Reto 1 depende del algoritmo de ordenamiento.

Requerimiento 5

La complejidad de la función **getTransportationByDepartment()** en el Reto 2 está determinada por el algoritmo de ordenamiento utilizado. Por un parte, obtener las obras de arte del departamento tiene una complejidad de **O(1)**, dado que este proceso se hace mediante la lista de hash **catalog['departments']**. Las llaves de **catalog['departments']** son los nombres de los departamentos del museo y los valores asociados a dichas llaves son las listas de las respectivas obras de arte del departamento. Por otra parte, es necesario ordenar la lista de las obras de arte del departamento. De ese modo, la complejidad es **O(n^2)**, **O(n^3/2)**, **O(n^2)** y **O(nlog(n))** para los algoritmos Insertion Sort, Shell Sort, Quick Sort y Merge Sort, respectivamente.

```
478 def TransportArtworksDepartment(sub_list, sorting_method, department, data_structure):
479     start_time = time.process_time()
480     information = CreateArtworkTransportationCostList(sub_list, department, data_structure)
481     artworks_by_date = information[0]
482     artworks_by_cost = information[1]
483     total_cost = information[2]
484     total_weight = information[3]
485     oldest_artworks = CreationOrderedListByDate(artworks_by_date, sorting_method)
486     most_expensive_artworks = CreationOrderedListByCost(artworks_by_cost, sorting_method)
487 
488     stop_time = time.process_time()
489     elapsed_time_mseg = (stop_time - start_time)*1000
490 
491     return elapsed_time_mseg, artworks_by_date, total_cost, total_weight, most_expensive_artworks, oldest_artworks
```

La función **CreateArtworkTransportationCostList()** en el Reto 1 calcula el valor de transporte de cada obra de arte del departamento ingresado por el usuario. La complejidad de esta función es **O(n)**, dado que la misma recorre todas las obras de arte y verifica si pertenecen al departamento indicado y calcula el costo de transporte simultáneamente. Posteriormente, las funciones **CreationOrderedListByDate()** y **CreationOrderedListByCost** ordenan la lista generada en **CreateArtworkTransportationCostList()** en base a la fecha de creación y costo de transporte, respectivamente. La complejidad de esta operación depende del tipo de algoritmo de ordenamiento utilizado. La complejidad es **O(n^2)**, **O(n^3/2)**, **O(n^2)** y **O(nlog(n))** para los algoritmos Insertion Sort, Shell Sort, Quick Sort y Merge Sort, respectivamente. Finalmente, la complejidad es modelada en su mayor parte por la complejidad en **SortingMethodExecution()**.

La complejidad del requerimiento 5 es mejor en el Reto 2 que en el Reto 1, dado que el proceso inicial de búsqueda de las obras pertenecientes al departamento ingresado por el usuario tiene un complejidad de **O(1)** y **O(n)** para el Reto 2 y el Reto 1, respectivamente.

Requerimiento 6

```
360 def getMostProlificArtists(catalog, data_structure, sorting_method,
361                                                 initial_birth_year, end_birth_year, num_artists):
362     most_prolific_artists_list = lt.newList(data_structure)
363     birth_years_map = catalog['birth_years']
364     for year in range(initial_birth_year, end_birth_year + 1):
365         year = str(year)
366         if mp.contains(birth_years_map, year):
367             artists_birth_year = me.getValue(mp.get(birth_years_map, year))
368             for artist in lt.iterator(artists_birth_year):
369                 artists_Id = artist['ConstituentID']
370                 artist_dict = me.getValue(mp.get(catalog['artists_Ids'], artists_Id))
371                 artist_name = artist_dict['info']['DisplayName']
372                 artist_medium_info = getArtworksByMediumAndArtist(catalog, artist_name)
373                 artworks_most_used_medium = artist_medium_info[0]
374                 if lt.size(artworks_most_used_medium) >= 5:
375                     artworks_most_used_medium = lt.subList(artist_medium_info[0], 1, 5)
376                 num_more_artworks = lt.size(artworks_most_used_medium)
377                 num_total_artworks = artist_medium_info[1]
378                 num_total_mediums = artist_medium_info[2]
379                 name_most_used_medium = artist_medium_info[3]
380                 lt.addLast(most_prolific_artists_list, (artist_name, artworks_most_used_medium,
381                 num_total_artworks, num_total_mediums, num_more_artworks, name_most_used_medium))
382     SortingMethodExecution(sorting_method, most_prolific_artists_list, cmpMostProlificArtist)
383     if lt.size(most_prolific_artists_list) > num_artists:
384         requirement_list = lt.subList(most_prolific_artists_list, 1, num_artists)
385     else:
386         requirement_list = most_prolific_artists_list
387     return requirement_list
```

La complejidad de la función **getMostProlificArtists()** está determinada por el algoritmo de ordenamiento utilizado. Por una parte, obtener los artistas nacidos en un rango de fechas por medio del uso de la tabla de hash **catalog[‘birth_years]** tiene una complejidad de **O(1)**. Así mismo, la obtención de la cantidad de obras y medios utilizados por cada artista nacido en el rango de fechas por medio de la función **getArtworkByMediumAndArtist()** tiene una complejidad de **O(1)**. Por otra parte, el proceso de ordenamiento de los artistas en función de la cantidad de obras y medios registrados tiene una complejidad acorde al algoritmo de ordenamiento utilizado. De ese modo, la complejidad es **O(n^2)**, **O(n^3/2)**, **O(n^2)** y **O(nlog(n))** para los algoritmos Insertion Sort, Shell Sort, Quick Sort y Merge Sort, respectivamente.