Reto 1

Sebastian Guerrero – 202021249

Diego Alejandro Gonzalez – 202110240

Pautas Generales

De manera general, tras hacer las pruebas del Laboratorio 4, decidimos hace uso a través de toda la implementación de listas con la estructura de datos "ARRAY_LIST" y utilizar el método de ordenamiento MergeSort por su rapidez, estabilidad y que no conocemos bien la naturaleza de los datos (si están medio ordenados, desordenados, por grupos). Además, se escogió este algoritmo de ordenamiento ya que se sabe que las fechas y nacionalidades pueden tener "few uniques" por lo que MergeSort es la decisión más acertada.

	Máquina 1	Máquina 2
Procesadores	Intel(R) Core(TM) i7-10800H CPU @ 2.90 GHz	AMD Ryzen 7 5700U withRadeon Graphics 1.80 GHz
Memoria RAM (GB)	16,00 GB	8,00 GB
Sistema Operativo	Windows 10 –x64 bits	Windows 10

Requerimiento 1

Primero, se hizo un algoritmo que filtraba los datos haciendo comparaciones entre los N datos, para posteriormente ser ordenados con selection sort. Esta implementación se evidencia en la siguiente imagen. Este algoritmo tenia una complejidad de O(N) ya que hacía N comparaciones. Esta implementación se demoraba demasiado tiempo con archivos de mayor tamaño, razón por la cual se tuvo que modificar.

```
def GetArtistas(catalog, inicial, final, tipo):

    en_rango = lt.newList(tipo)
    artistas  = lt.iterator(catalog["Artist"])


    for artista in artistas:
        Date = int(artista["BeginDate"])
        if (Date >= inicial) and (Date <= final):
            lt.addLast(en_rango, artista)

    sa.sort(en_rango, compareBeginDate)

    return en_rango
```

Para la nueva implementación, se decidió ordenar la lista de datos antes y encontrar con búsqueda binaria los limites superiores y inferiores de la información que queremos. Esta implementación se evidencia en la siguiente imagen.

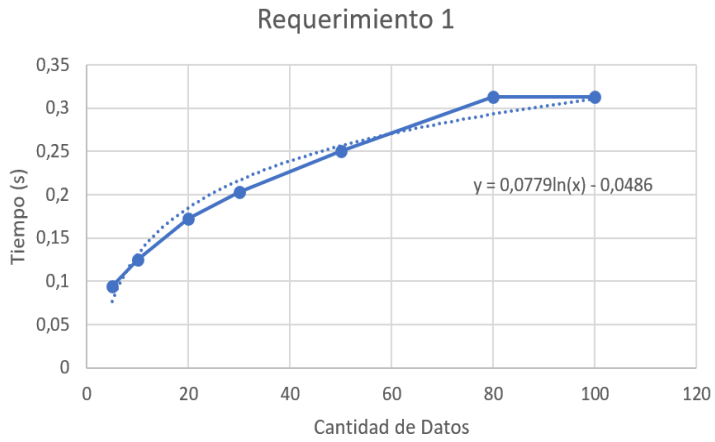
```
def funcionReqUno(catalog,minimo,maximo):
    ordenado = sa.sort(catalog['Artists'],cmpFunctionRuno)
    indexmin = binary_search_min(ordenado, int(minimo))
    indexmax = binary_search_max(ordenado, int(maximo))
    cant= indexmax-indexmin
    la_lista = lt.subList(ordenado, indexmin,cant+1)
    return la_lista
```

Como se puede ver en la imagen que primero se hace un MergeSort al catalogo, esto se hace en función del DateAcquired (O(nlog(n))). Posteriormente, se tiene dos búsquedas binarias y la creación de una sub lista. Finalmente, crea la sub lista a partir de los limites encontrados.

Por esta razón se puede decir que la complejidad de este algoritmo es de log n + log n + nlog n= 2nlog n. Igualmente se entiende la complejidad del sistema como O(nlog n). A continuación, se puede ver los tiempos registrados y la grafica de tendencia para este requerimiento teniendo en cuente la fecha inicial 1920 y fecha final 1985 (como en la respuesta ejemplo).

Máquina 1:

Porcentaje	**Tiempo(s)**
5	0,09375
10	0,125
20	0,171875
30	0,203125
50	0,25
80	0,3125
100	0,3125



Maquina 2:

PORCENTAJE	TIEMPO
5	0.125
10	0.1875
20	0.265625
30	0.265625
50	0.34375
80	0.390625
100	0.4375

En el grafico se evidencia como se observa una tendencia logarítmica en la cual se va estabilizando a medida que aumentan los datos. Estas pruebas también son un poco ambiguas debido a que cada vez puede haber más valores que toca rescatar y copiar en la sublista, por lo que las operaciones que se hacen incrementan a mayor medida al sacar las sublistas.

Requerimiento 2

Al enfrentarnos con el requerimiento 2 por primera vez, se pensó en colocar una función que recorriese todas las obras y filtrara en nuevo TAD lista aquellas que estuviesen en el rango de fechas estipulado. Posteriormente, la idea era buscar obra por obra los artistas que tuviera e ir agregando. Sin embargo, la complejidad temporal de este algoritmo planteado de esta forma podía ser exponencial de acuerdo con su implementación, por lo que se decidió cambiar a la implementación que se presenta a continuación:

Primero, se hace un filtro del String que llega por parámetro, para poder dejarlo como un entero que sea mucho más maniobrable. Para ello, se utilizaron 4 asignaciones con orden temporal O(1), como se muestra:

```
def funcionReqDos(catalog, minimo, maximo):
    mini= minimo[0:4]+minimo[5:7]+minimo[8:10]
    maxi= maximo[0:4]+maximo[5:7]+maximo[8:10]
    mini = int(mini)
    maxi = int(maxi)
```

Segundo, se ordenó el TAD lista de manera que las fechas de adquisición quedaran en orden ascendente. Esta operación, de acuerdo con el algoritmo merge utilizado, es de orden O(nlogn), siendo n el número de obras totales. Posteriormente, se realizaron 2 búsquedas binarias para determinar los límites inferior y superior dentro de los cuales se encontraban los datos de interés. Estas búsquedas son de orden O(logn). Finalmente, se crea el TAD que tendrá solo los datos entre estos límites(O(1)). La sección completa mencionada se encuentra a continuación:

```
ordenado = sa.sort(catalog['Artworks'],cmpFunctionRdos)
indexmin = binary_search_min2(ordenado, int(mini))+1
indexmax = binary_search_max2(ordenado, int(maxi))
cant= indexmax-indexmin
la_lista = lt.subList(ordenado, indexmin,cant+1)
```

Tercero, se inicia un ciclo que recorrerá la nueva lista para poder añadir a cada una de las obras sus respectivos autores. En este sentido, se hizo la depuración necesaria del String que se tiene como pauta de los autores y posteriormente se convirtieron los autores a una lista nativa de Python que fuese más sencilla de iterar. Todos estos procedimientos dentro del ciclo son de orden constante, y aunque el ciclo es de orden O(p), p representa las obras dadas en el rango de fechas, por lo que se considera que en los casos promedio esta magnitud va a ser tan pequeña que podría ser tomada como constante. No obstante, aquí cabe resaltar la importancia de la lista auxiliar, pues gracias a

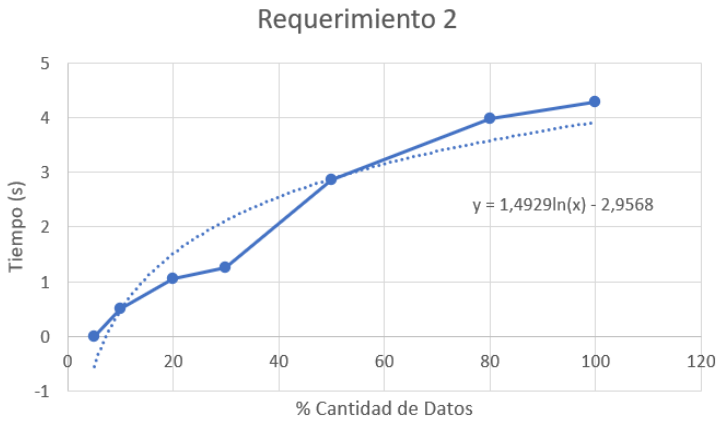
esta se puede a continuación realizar una búsqueda binaria de cada uno de los autores de acuerdo con sus referencias o ID(log(n)). Finalmente, se ingresan los datos ya organizados en el TAD que se va a devolver al visualizador, como se muestra a continuación:

```
la_rta= lt.newList("ARRAY_LIST")
for n in range(1,lt.size(la_lista)+1):
    elemento= lt.getElement(la_lista,n)
    buscar= elemento["ConstituentID"]
    byecorchetes = buscar.replace("[","")
    byecorchetedos = byecorchetes.replace("]","")
    authors = byecorchetedos.split(",")
    autores=""
    for x in authors:
        index = binary_search(catalog['Artists_Artworks'],int(x))
        ele = lt.getElement(catalog['Artists_Artworks'], index)
        autores = autores +"-"+ ele['DisplayName'] + "-"
    agregar = {
        'ObjectID':elemento['ObjectID'],
        'Title':elemento['Title'],
        'Artists':autores,
        'Medium':elemento['Medium'],
        'Dimensions':elemento['Dimensions'],
        'CreditLine':elemento['CreditLine'],
        'DateAcquired':elemento['DateAcquired'],
        'URL':elemento['URL']
    }
    lt.addLast(la_rta,agregar)
```

Finalmente, se dice que la complejidad total sería O(nlogn)+O(p)+ plog(m), donde p hace referencia a los elementos de la sublista, n a las obras, y m a la lista auxiliar, teniendo en cuenta que el factor más grande es O(nlogn), se dice que la complejidad total es de este tipo.

Maquina 1

Porcentaje	Tiempo(s)
5	0.234375
10	0,5
20	1,0625
30	1,25
50	2,8587
80	3,9873
100	4,290625



Máquina 2:

PORCENTAJE	TIEMPO
5	0.34375
10	0.703125
20	1.515625
30	2.125

50	3.8125
80	6.296875
100	7.765625

Requerimiento 3(Diego Alejandro González Vargas)

Para la implementación del tercer requerimiento, su complejidad algoritmica va dictada por el siguiente análisis de secciones:

Primero, se realiza una búsqueda del nombre del artista solicitado por el usuario, dicha búsqueda se hace sobre el TAD lista auxiliar creado en la función cargar datos. Este TAD no tiene los nombres de los artistas organizados. En consecuencia, se optó por una búsqueda común, donde se compara elemento por elemento hasta encontrar el String que corresponda. Esta búsqueda tiene una complejidad de O(n), siendo n el número de autores totales que maneja el MoMA, como se puede ver a continuación:

```
def funcionReqTres(catalog, nombre):
    index = normal_search_nombre(catalog['Artists_Artworks'],nombre)
```

```
def normal_search_nombre(arr, x):
    largo  = lt.size(arr)
    for artist in range(1,largo+1):
        artista = lt.getElement(arr,artist)
        name = artista["DisplayName"]
        if (name == x):
            return artist
    return -1
```

Segundo, una vez obtenida la supuesta ubicación del artista buscado, se realiza una doble comprobación: Por un lado, se comprueba que la posición exista dentro del arreglo, pues como se puede ver de la función de búsqueda anterior, si el artista no existe en la base de datos la “ubicación” encontrada será –1. Por otra parte, se comprueba que el artista ya existente tenga además obras para analizar, por lo que se comprueba que tenga ID’s de objetos en su elemento. Estas 2 operaciones son de orden O(1), como se puede ver a continuación:

```
if index != (-1):
    autor = lt.getElement(catalog['Artists_Artworks'],index)
    objetos = autor["ObjectID"]
    if (objetos==None) or (objetos==''):
        vacio = "NOHAYOBRAS"
        tupla = vacio,vacio
        return tupla
    else:
```

Tercero, una vez se tiene el String con los objetos del autor, se procede a cambiar dicho String por una lista de objetos que se puedan buscar más fácilmente. Esta operación tiene orden de crecimiento

O(1). Así mismo, se organiza el TAD lista con las obras, pensado también en las búsquedas posteriores. Este segundo preparativo tiene una complejidad de O(mlogm), donde m es el número de obras totales del MoMA. Finalmente se crean los TAD lista nuevos en los que se va a almacenar los datos respuesta para el usuario, proceso de complejidad O(1). Esta sección se muestra de la siguiente forma:

```
lt_objetos = objetos.split(",")
ordenado = sa.sort(catalog["Artworks"], cmpobjectid)
tad_objetos = lt.newList("ARRAY_LIST")
tad_medios = lt.newList("ARRAY_LIST")
```

Cuarto, se realiza un ciclo que revise cada uno de los ID de los objetos del artista, dicho ciclo tendra una complejidad O(1), pues se considera que la cantidad de obras de un autor es enormemente inferior a la de lista completa de obras, por lo que su recorrido no tendría un gasto temporal considerable. Luego, dentro del ciclo se realiza el siguiente procedimiento: Para empezar, se busca de manera binaria el objeto dentro del TAD lista que tiene toda la información adicional del mismo. Estas búsquedas tienen una complejidad de O(log n). Luego se organiza el nuevo elemento para el TAD objetos del artista, y se agrega dicho objeto. Este último proceso tiene complejidad O(1). La sección mencionada se muestra a continuación:

```
for i in lt_objetos:
    index = binary_search_id(ordenado, i)
    elemento = lt.getElement(ordenado, index)
    agregar = {
        'ObjectID':elemento['ObjectID'],
        'Title':elemento['Title'],
        'Medium':elemento['Medium'],
        'Dimensions':elemento['Dimensions'],
        'DateAcquired':elemento['DateAcquired'],
        'Department':elemento['Department'],
        'Classification':elemento['Classification'],
        'URL':elemento['URL'],
        'ConstituentID':elemento['ConstituentID']
    }
    lt.addLast(tad_objetos, agregar)
```

Quinto, aún dentro del ciclo, se procede a modificar el TAD contador de los medios usados por el artista. Para este procedimiento se creó una función auxiliar que busca la ubicación del medio del objeto en el TAD de medios. Si encuentra el medio, le añade 1 a su contador. En caso contrario, se añade el medio al TAD. Tomando en cuenta que la cantidad de medios es considerablemente más baja que la cantidad de artistas u obras se tomara a esta función como O(1) para su complejidad temporal. Esta función se muestra a continuación:

```
def cambiarTADmedios(arr, x):
    pos=0
    final=lt.size(arr)
    for i in range(1,final+1):
        cosa=lt.getElement(arr,i)
        medio= cosa["Medium"]
        if medio==x["Medium"]:
            pos=i
    if pos>0:
        o = lt.getElement(arr,pos)
        coso= int(o["Count"])
        cambio= coso + 1
        cambiodict = {
            'Medium': o["Medium"],
            'Count': str(cambio)
        }
        lt.changeInfo(arr,pos,cambiodict)
    else:
        nuevodict={
            'Medium': x["Medium"],
            'Count': "1"
        }
        lt.addLast(arr,nuevodict)
```

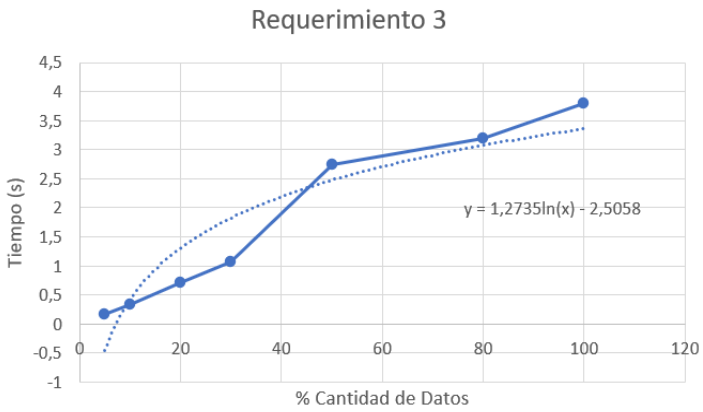
Finalmente, se ordena con el algoritmo Merge el TAD de los medios, para obtener el más repetido. Esta operación se va considerar convenientemente de orden O(1). Adicionalmente, se hace un nuevo TAD para almacenar los objetos de este medio y se hace un recorrido por el TAD de los objetos para filtrar solo los del medio seleccionado y agregarlos al TAD que finalmente será mostrado al usuario. Dicho recorrido, como se mencionó anteriormente, no merece ser considerado para la complejidad temporal. Para terminar, se organiza una tupla con la información que se va a enviar al controlador:

```
sa.sort(tad_medios,cmpcount)
mediorep = lt.getElement(tad_medios, 1)
rtafinal= lt.newList("ARRAY_LIST")
objetos  = lt.size(tad_objetos)
for ob in range(1,objetos+1):
    objeto=lt.getElement(tad_objetos,ob)
    name = objeto["Medium"]
    if (name == mediorep["Medium"]):
        lt.addLast(rtafinal, objeto)
tuplarta= tad_medios,rtafinal
return tuplarta
```

En conclusión, se dice que se si se toman en cuenta las variables que más datos tienen, la complejidad temporal para el peor de los casos sería: O(n + mlogm), donde m hace referencia a las obras y n a los artistas

Maquina 1

Porcentaje	Tiempo(s)
5	0,17187
10	0,32812
20	0,71875
30	1,0625
50	2,75
80	3,1875
100	3,796875



Maquina 2:

PORCENTAJE	TIEMPO
5	0.21875
10	0.453125
20	1.015625
30	1.4375
50	2.578125
80	4.34375
100	5.375

Requerimiento 4 (Sebastian Guerrero Ríos)

Al enfrentarse con este reto la primera vez, se pensó en hacer comparaciones y listas de apoyo para guardar cada nacionalidad. Pero al hacer pruebas y ver que la complejidad llega a ser n o n^2 en algunos casos ya que se comparan todos los datos y se extraen para poder sacar las listas ayuda, se decidió tomar otro acercamiento al problema. Esta segunda instancia se puede ver en la siguiente imagen.

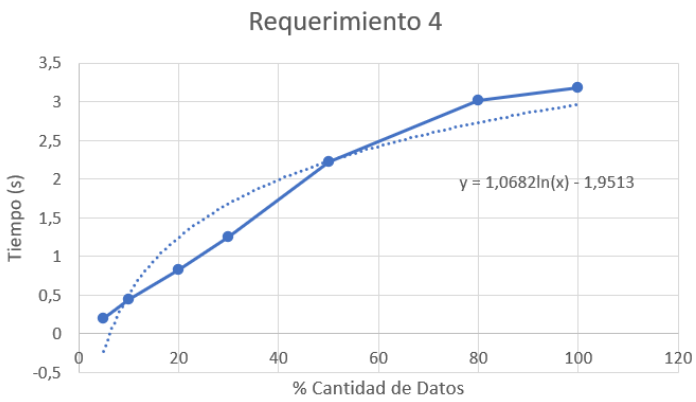
```python
def funcionReqCuatro(catalog):
    data = sa.sort(catalog['Nationality_Artworks'], cmpnationality)
    nat = lt.newList("ARRAY_LIST")
    low = 1
    size = lt.size(data)
    while low <= size:

        if low == size:

            temp = lt.subList(data, low, 1)

            pais = {'Nationality': lt.getElement(data, low)['Nationality'],
                    'obras': temp}

            lt.addLast(nat,pais)

            low += 1

        else:

            high = binmax(data, lt.getElement(data, low)["Nationality"])
            temp = lt.subList(data, low, high - low + 1)

            pais = {'Nationality': lt.getElement(data, low)['Nationality'],
                    'obras': temp}

            lt.addLast(nat,pais)

            low = high + 1

    a = sa.sort(nat, cmpsize)

    return a
```

En la imagen superior se evidencia el código generado para el requerimiento 4, para este, primero se ordena la lista en términos de la nacionalidad del autor, esto conlleva una complejidad de n log n ya que se usa el método de MergeSort. Posterior mente se tiene un while el cual se va a repetir M veces donde M representa la cantidad de nacionalidades diferentes. Posteriormente, se hace una búsqueda binaria para encontrar la posición del ultimo elemento que tenga la misma nacionalidad del que estamos buscando. Este proceso de búsqueda tiene una complejidad de log n, en el mejor caso, pero este puede aumentar debido a que se buscan todos los términos que sean de la misma nacionalidad, al encontrar uno con la búsqueda binaria, se avanza uno a uno hasta llegar a un elemento que no tenga la misma nacionalidad. En el peor caso, la mitad de las obras son de la misma nacionalidad y la otra mitad tiene una nacionalidad única, en este caso, la búsqueda binaria encontraría el termino en la mitad y tendría que recorrer y hacer n/2 comparaciones para llegar hasta el final y comprobar que no haya más iguales. Esta implementación se evidencia en la siguiente imagen.

```
def binmax(arr, x):
    low = 0
    high = lt.size(arr)-1
    mid = 0

    while low <= high:

        mid = low + (high - low) // 2
        ele=lt.getElement(arr, mid+1)
        begin=ele["Nationality"]

        # If x is greater, ignore left half

        if x < begin:
            high = mid - 1

        elif x > begin:
            low = mid + 1

        # means x is present at mid
        else:
            if mid < lt.size(arr)-1:
                while begin==x and mid < lt.size(arr):
                    mid=mid+1
                    ele=lt.getElement(arr, mid)
                    begin=ele["Nationality"]
                return mid
            else:
                return mid+1
    # If we reach here, then the element was not present
    return -1
```

Considerando lo anterior se puede decir que la complejidad del requerimiento depende fuertemente de la cantidad de nacionalidades diferentes que se encuentran, O(M n log n) en el peor caso. Donde M es la cantidad de nacionalidades diferentes y n es la cantidad de obras.

Porcentaje	**Tiempo(s)**
5	0,203125
10	0,4375
20	0,828125
30	1,25
50	2,215625
80	3,015625
100	3,1812



Maquina 2:

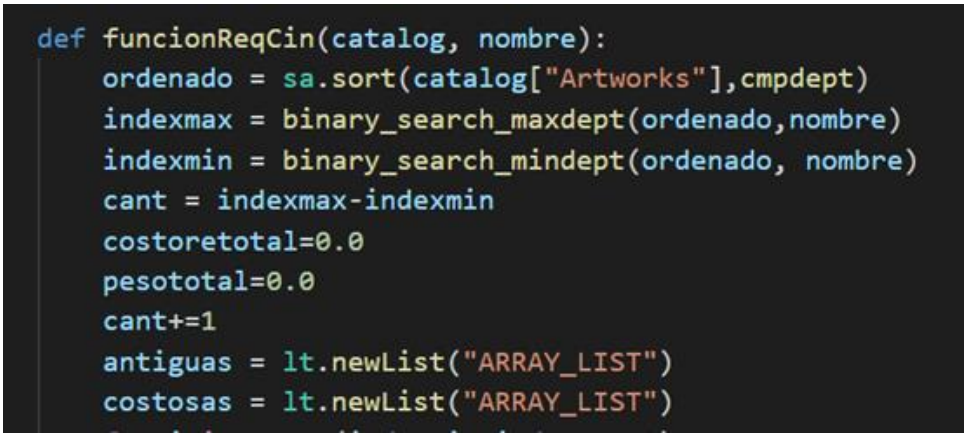
PORCENTAJE	TIEMPO
5	0.453125
10	0.921875
20	1.828125
30	2.5
50	4.1875
80	6.671875
100	8.109375

Al hacer pruebas con el código, se evidencia que la tendencia no es completamente logarítmica. Esto se debe a que a medida que aumenta la cantidad de datos también aumenta la cantidad de nacionalidades diferentes, por lo que la complejidad aumenta. En los últimos casos, entre 80 y 100 porciento, nos percatamos de que es menos probable que haya una nueva nacionalidad por lo que comienza a estabilizarse y a llegar a un punto de equilibrio.

Requerimiento 5

Para el caso del requerimiento, de acuerdo con la disposición inicial elegida para almacenar los datos, se consideró que la manera más eficiente de realizar la implementación es la siguiente:

Primero, se organizó el TAD lista que almacena las obras de acuerdo con el departamento al que pertenecen cada una de ellas, permitiendo agruparlas de manera pertinente para el ejercicio. Este ordenamiento se dice que es de complejidad temporal O(nlogn) siendo n la cantidad de obra totales.Además, se realizaron busquedas binarias sobre la lista ordenada para poder marcar los límites del grupo de obras pertenecientes al departamento que se está buscando. Dichas búsquedas son de orden O(logn) Posteriormente se crearon las variables necesarias durante toda la función, así como los TAD que puedan ayudar para la respuesta. Todas estas asignaciones son de orden O(1):



```
def funcionReqCin(catalog, nombre):
    ordenado = sa.sort(catalog["Artworks"],cmpdept)
    indexmax = binary_search_maxdept(ordenado,nombre)
    indexmin = binary_search_mindept(ordenado, nombre)
    cant = indexmax-indexmin
    costoretotal=0.0
    pesototal=0.0
    cant+=1
    antiguas = lt.newList("ARRAY_LIST")
    costosas = lt.newList("ARRAY_LIST")
```

Segundo, de acuerdo con los índices encontrados anteriormente, se inicia un ciclo el cual recorre todos los elementos pertenecientes al departamento buscado, y de acuerdo con una serie de condicionales, determina el precio de transporte de cada uno de los objetos pertenecientes al departamento de interés. Este proceso es de orden O(p), donde p es el número de obras del departamento:

```
for i in range(indexmin,indexmax+1):
    costototal=0.0
    ele = lt.getElement(ordenado,i)
    costopeso=0
    if (ele["Weight (kg)"]==None) or (ele["Weight (kg)"]==''):
        costopeso=float(-1)
    else:
        costopeso = float(ele["Weight (kg)"])*72
        pesototal+= float(ele["Weight (kg)"])
    costomedidas=-1
    if (ele["Height (cm)"]!=None) and (ele["Height (cm)"]!=''):
        if (ele["Width (cm)"]!=None) and (ele["Width (cm)"]!=''):
            if (ele['Depth (cm)']!=None) and (ele['Depth (cm)']!=''):
                if float(ele['Depth (cm)'])>0:
                    depth= float(ele['Depth (cm)'])/100
                    width = float(ele["Width (cm)"])/100
                    height = float(ele["Height (cm)"])/100
                    m3= depth*width*height
                    costomedidas=m3*72
                else:
                    width = float(ele["Width (cm)"])/100
                    height = float(ele["Height (cm)"])/100
                    m2= width*height
                    costomedidas=m2*72
            else:
                width = float(ele["Width (cm)"])/100
                height = float(ele["Height (cm)"])/100
                m2= width*height
                costomedidas=m2*72
        elif (ele["Diameter (cm)"]!=None) and (ele["Diameter (cm)"]!=''):
            radio = float(ele["Diameter (cm)"])/200
            height = float(ele["Height (cm)"])/100
            m3= radio*height*radio*math.pi
```

Cuarto, una vez establecido el precio de transporte de cada implemento, se procede a llenar los 2 TAD lista creados para satisfacer los requisitos. Para ello, se usan tanto los datos que ya tiene cada elemento, como las búsquedas binarias en la lista auxiliar para encontrar los artistas asociados. Este proceso tiene un orden de O(p) para todas las asignaciones comunes y de O(plogm) para las busquedas binarias, donde m es la cantidad de artistas totales y p las repeticiones del ciclo.

```
costototal=max(costopeso,costomedidas)
if costototal<=0:
    costototal=48.0
buscar= ele["ConstituentID"]
byecorchetes = buscar.replace("[","")
byecorchetedos = byecorchetes.replace("]","")
authors = byecorchetedos.split(",")
autores=""
for x in authors:
    index = binary_search(catalog['Artists_Artworks'],int(x))
    elemento = lt.getElement(catalog['Artists_Artworks'], index)
    autores = autores +"-"+ elemento['DisplayName'] + "-"
agregar = {
    'ObjectID':ele['ObjectID'],
    'Title':ele['Title'],
    'Artists':autores,
    'Medium':ele['Medium'],
    'Dimensions':ele['Dimensions'],
    'DateAcquired':ele['DateAcquired'],
    'Classification':ele['Classification'],
    'TransCost (USD)':str(costototal),
    'URL':ele['URL'],
    'Date':ele['Date']
}
lt.addLast(antiguas,agregar)
lt.addLast(costosas,agregar)
costoretotal+=costototal
```

Finalmente, se realizan los ordenamientos correspondientes a las necesidades expresadas en el requerimiento (antigüedad y costo), que tendrán una complejidad de O(plogp), y se retorna una tupla con toda la información

```
ordenantiguas = sa.sort(antiguas,cmpdate)
ordencostosas = sa.sort(costosas,cmpcost)
tuplatriple = costoretotal,ordenantiguas,ordencostosas,pesototal
return tuplatriple
```

En conclusión, para casos promedio en donde las obras p (pertenecientes a la obra), sean prominencialmente inferiores a n(cantidad de obras totales), la operación y aproximación de la complejidad temporal del algortimo sería la siguiente:

O(nlogn) +2 O(logn)+ O(p)

-O(nlogn)

Maquina 1:

Porcentaje	Tiempo(s)
5	0,34375
10	0,71875
20	1,53125
30	2,354
50	2,66875
80	2,790625
100	2,9375

Maquina 2:

PORCENTAJE	TIEMPO
5	0,484375
10	1.03125
20	2.125
30	3.03125
50	5.265625
80	8.734375
100	10.921875