ANÁLISIS DEL RETO

Sebastián Palma, 202222498, s.palma@uniandes.edu.co

Sara Leiva, 202220956, s.leivam@uniandes.edu.co

Andrés Rodríguez, 202222586, a.rodiguezs@uniandes.edu.co

Requerimiento 1

Descripción

```python
def req_1(data_structs):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento

    size = lt.size(data_structs["data"])

    lista = lt.newList(datastructure="ARRAY_LIST",cmpfunction = compare2)

    for i in range(1,size + 1):
        x = lt.getElement(data_structs["data"],i)

        estar = lt.isPresent(lista, x["Año"])

        if estar == 0:
            lt.addLast(lista, x)

        if estar >0:
            if int(x["Total saldo a pagar"]) > int(lt.getElement(lista,estar)["Total saldo a pagar"]):
                lt.changeInfo(lista,estar,x)



    return lista
```

Entrada	Estructura de datos
Salidas	Una lista con las actividades económicas con el mayor saldo total de impuestos a pagar para cada año
Implementado (Sí/No)	Si. Implementado por Sara Leiva.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Consultar el número de elementos del data_structs (lt.size)	O (1)

Crear una nueva lista vacía guardarla en una variable lista (lt.newList)	O (1)
Itera *i* sobre el número de elementos del data_structs	O (n)
Retorna un elemento en la posición *i* (Lt.getElement) y lo guarda en una variable x	O (1)
Se verifica si el valor del año del elemento en la posición *i* está dentro de la lista y almacena el valor de su posición en la lista en una variable estar (Lt.isPresent)	O (n)
Si estar es igual a cero, agrega al final de la lista el elemento de la posición x (lt.addLast)	O (1)
Si estar es mayor que cero, se compara el elemento en la posición *i* guardado en x y el elemento de la lista que corresponde a la posición guardada en estar, la comparación se hace con el valor de la llave "Total saldo a pagar" para ambos (Lt.getElement)	O (n)
En caso de que x tenga un valor mayor en "Total saldo a pagar", se actualiza el elemento en la posición guardada en estar de la lista con la información de x (lt.changeInfo)	O (1)
TOTAL	***O(n^2)***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

Entrada	**Tiempo (ms)**
small	0.38
5 pct	1.23
10 pct	3.03
20 pct	3.93
30 pct	5.57
50 pct	8.99
80 pct	14.26
large	18.02

Procesadores	Intel® Core™ i7-1165G7 @2.80GHz
Memoria RAM	15.6 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	Tiempo (ms)
small	0.21
5 pct	2.07
10 pct	3.57
20 pct	4.35
30 pct	8.1
50 pct	12.86
80 pct	15.87
large	20.69

Procesadores	AMD Ryzen 3 5300U with Radeon Graphics 2.60 GHz
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	Tiempo (ms)
small	0.41
5 pct	3.73
10 pct	5.59
20 pct	8.45
30 pct	10.39
50 pct	17.77
80 pct	28.51
large	36.7

Tablas de datos

Las tablas con la recopilación de datos promedio de las pruebas.

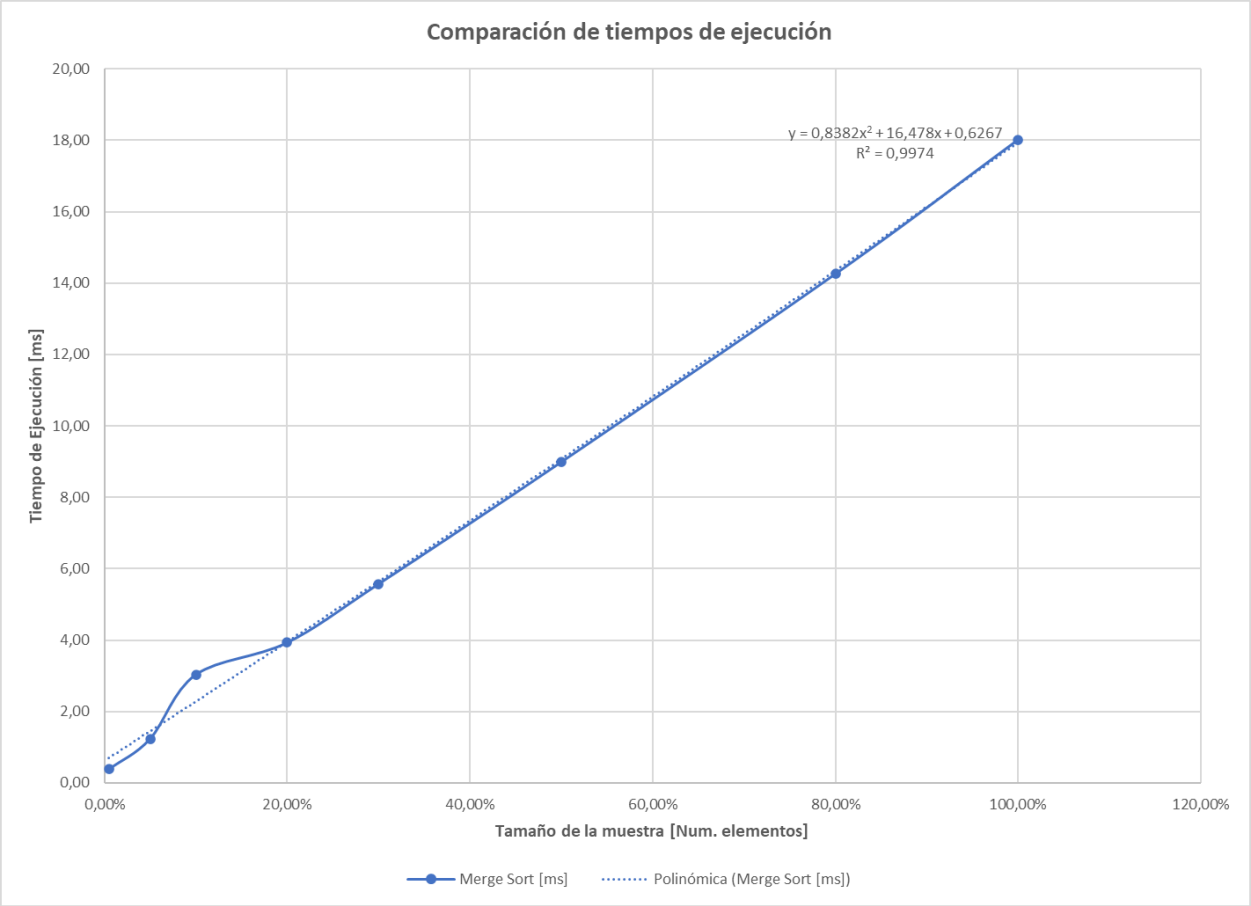
Muestra	Salida	Tiempo (ms)
small	Las actividades económicas con el mayor saldo total de impuestos a pagar para cada año	0,33

5 pct	Las actividades económicas con el mayor saldo total de impuestos a pagar para cada año	2.34
10 pct	Las actividades económicas con el mayor saldo total de impuestos a pagar para cada año	4.06
20 pct	Las actividades económicas con el mayor saldo total de impuestos a pagar para cada año	5.57
30 pct	Las actividades económicas con el mayor saldo total de impuestos a pagar para cada año	8.02
50 pct	Las actividades económicas con el mayor saldo total de impuestos a pagar para cada año	13.20
80 pct	Las actividades económicas con el mayor saldo total de impuestos a pagar para cada año	19.54
large	Las actividades económicas con el mayor saldo total de impuestos a pagar para cada año	25.13

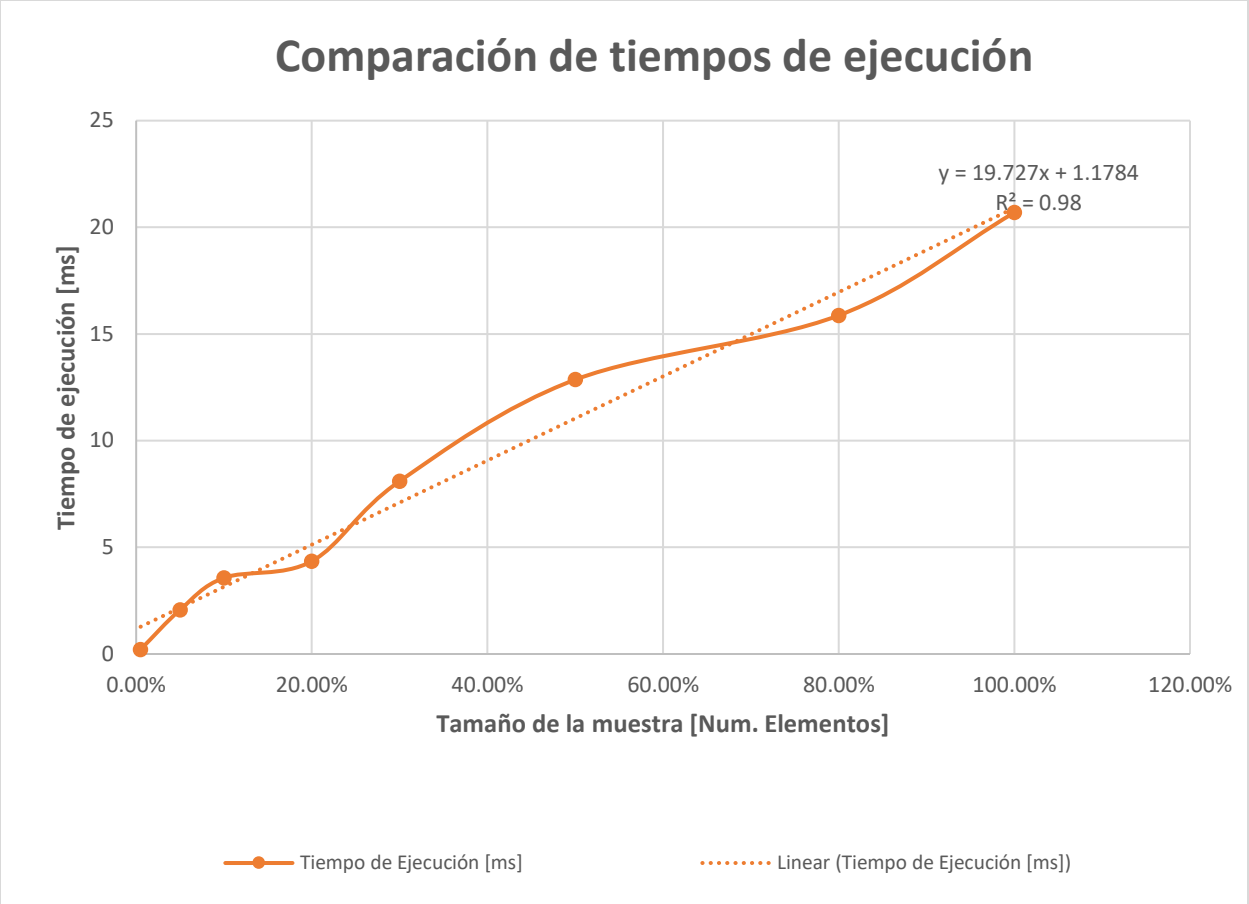
Graficas

Las gráficas con la representación de las pruebas realizadas.

- Máquina 1:


Comparación de tiempos de ejecución
20,00
18,00
16,00
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
Tiempo de Ejecución [ms]
y = 0,8382x² + 16,478x + 0,6267
R² = 0,9974
0,00%
20,00%
40,00%
60,00%
80,00%
100,00%
120,00%
Tamaño de la muestra [Num. elementos]
Merge Sort [ms]
Polinómica (Merge Sort [ms])

- Máquina 2:



- Máquina 3:

Análisis

Al implementar el requerimiento 1 se obtiene un Array_list de orden lineal a pesar de que la complejidad del algoritmo es $O(n^2)$. Esta complejidad se debe a la iteración que se hace sobre el data_structs en búsqueda de las actividades económicas con el mayor saldo total de impuestos a pagar en cada año, en el peor de los casos tiene que recorrer toda la lista para encontrar el mayor de alguna actividad económica de alguno de los años dentro del data_structs. También dentro de la iteración se usa la función lt.isPresent, donde se tiene que recorrer toda la lista, aumentando la complejidad total del algoritmo.

El comportamiento descrito es posible obsérvalo gracias a las gracias, ya que esta evidencia como a pesar de tener una complejidad del algoritmo $O(n^2)$, no se cuadruplican los tiempos de carga al duplicar los datos de entrada

Requerimiento 2

Descripción

```python
def req_2(data_structs):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    size = lt.size(data_structs["data"])

    lista = lt.newList(datastructure="ARRAY_LIST",cmpfunction = compare2)

    for i in range(1,size + 1):
        x = lt.getElement(data_structs["data"],i)

        estar = lt.isPresent(lista, x["Año"])

        if estar == 0:
            lt.addLast(lista, x)

        if estar >0:
            if int(x["Total saldo a favor"]) > int(lt.getElement(lista,estar)["Total saldo a favor"]):
                lt.changeInfo(lista,estar,x)


    return lista
```

Entrada	Estructura de datos
Salidas	Una lista con las actividades económicas con el mayor saldo total de impuestos a favor para cada año
Implementado (Sí/No)	Si. Implementado por Sara Leiva.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Consultar el número de elementos del data_structs (lt.size)	O (1)
Crear una nueva lista vacía guardarla en una variable lista (lt.newList)	O (1)
Itera *i* sobre el número de elementos del data_structs	O (n)
Retorna un elemento en la posición *i* (Lt.getElement) y lo guarda en una variable x	O (1)
Se verifica si el valor del año del elemento en la posición *i* está dentro de la lista y almacena el valor de su posición en la lista en una variable estar (Lt.isPresent)	O (n)
Si estar es igual a cero, agrega al final de la lista el elemento de la posición x (lt.addLast)	O (1)
Si estar es mayor que cero, se compara el elemento en la posición *i* guardado en x y el elemento de la lista que corresponde a la posición guardada en estar, la	O (n)

comparación se hace con el valor de la llave “Total saldo a favor” para ambos (Lt.getElement)	
En caso de que x tenga un valor mayor en “Total saldo a favor”, se actualiza el elemento en la posición guardada en estar de la lista con la información de x (lt.changeInfo)	O (1)
TOTAL	***O(n^2)***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

Entrada	**Tiempo (ms)**
small	0.39
5 pct	1.23
10 pct	1.92
20 pct	3.69
30 pct	5.20
50 pct	9.53
80 pct	15.24
large	18.04

Procesadores	**Intel® Core™ i7-1165G7 @2.80GHz**
Memoria RAM	15.6 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	**Tiempo (ms)**
small	0.71
5 pct	1.48
10 pct	4.88
20 pct	7.88
30 pct	13.94
50 pct	18.96
80 pct	20.24

large	23.05

Procesadores	**AMD Ryzen 3 5300U with Radeon Graphics 2.60 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	**Tiempo (ms)**
small	0.5
5 pct	2.72
10 pct	5.54
20 pct	9.38
30 pct	15.19
50 pct	20.05
80 pct	36.19
large	34.92

Tablas de datos

Las tablas con la recopilación de datos promedio de las pruebas.

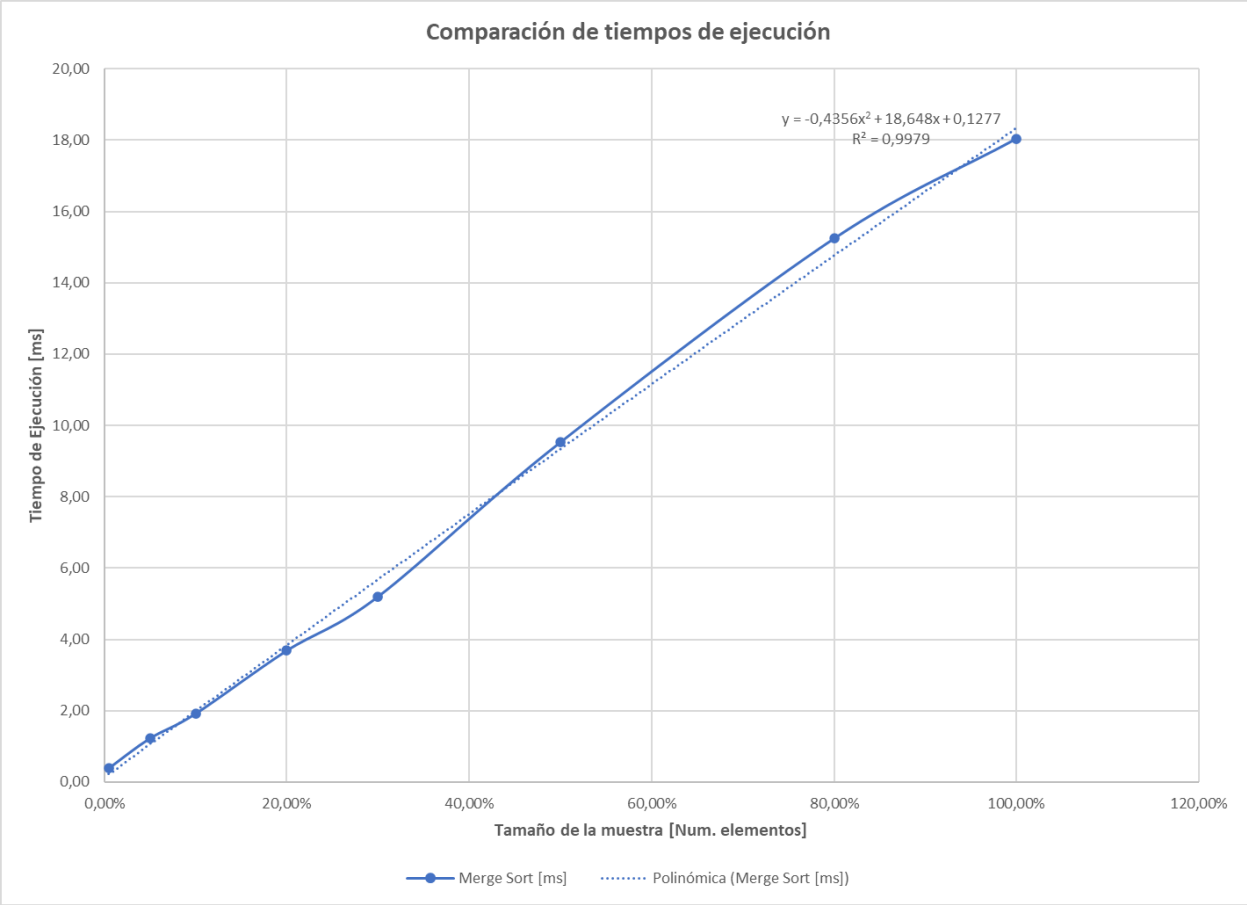
Muestra	Salida	Tiempo (ms)
small	Las actividades económicas con el mayor saldo total de impuestos a favor para cada año	0.34
5 pct	Las actividades económicas con el mayor saldo total de impuestos a favor para cada año	1.78
10 pct	Las actividades económicas con el mayor saldo total de impuestos a favor para cada año	5.72
20 pct	Las actividades económicas con el mayor saldo total de impuestos a favor para cada año	7.54
30 pct	Las actividades económicas con el mayor saldo total de impuestos a favor para cada año	14.53
50 pct	Las actividades económicas con el mayor saldo total de impuestos a favor para cada año	19.76

80 pct	Las actividades económicas con el mayor saldo total de impuestos a favor para cada año	23.34
large	Las actividades económicas con el mayor saldo total de impuestos a favor para cada año	27.98

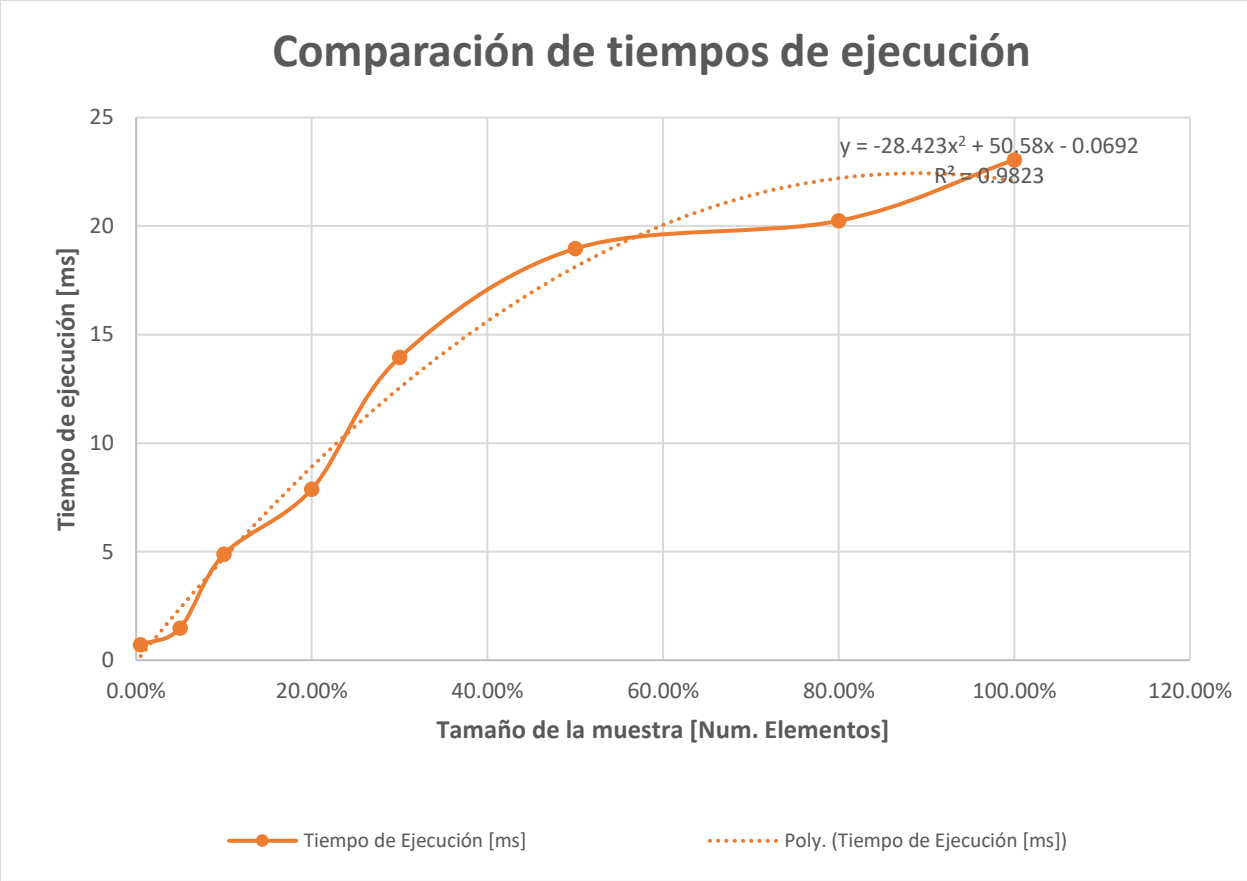
Graficas

Las gráficas con la representación de las pruebas realizadas.

- Máquina 1:



- Máquina 2:



- Máquina 3:

Análisis

Al implementar el requerimiento 2 se obtiene un Array_list de orden lineal a pesar de que la complejidad del algoritmo es $O(n^2)$. Esta complejidad se debe a la iteración que se hace sobre el data_structs en búsqueda de las actividades económicas con el mayor saldo total de impuestos a favor en cada año, en el peor de los casos tiene que recorrer toda la lista para encontrar el mayor de alguna actividad económica de alguno de los años dentro del data_structs. También dentro de la iteración se usa la función lt.isPresent, donde se tiene que recorrer toda la lista, aumentando la complejidad total del algoritmo.

El comportamiento descrito es posible obsérvalo gracias a las gracias, ya que esta evidencia como a pesar de tener una complejidad del algoritmo $O(n^2)$, no se cuadruplican los tiempos de carga al duplicar los datos de entrada

Requerimiento 3

Descripción

```
def req_3(data_structs):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    lista_sub, dic = organizar_retenciones(data_structs)
    lista_min = lt.newList(datastructure="ARRAY_LIST",cmpfunction=compare2)

    for elemento in lt.iterator(lista_sub):
        estar = lt.isPresent(lista_min, elemento["Año"])

        if estar == 0:
            lt.addLast(lista_min, elemento)

    tabla_grande = []
    table = []
```

```
for subsector in lt.iterator(lista_min):
    cod = subsector["Código subsector económico"]
    anio = subsector["Año"]

    lista = lt.newList(datastructure="ARRAY_LIST")
    for elemento in lt.iterator(data_structs["data"]):
        if elemento["Año"] == anio and elemento["Código subsector económico"] == cod:
            lt.addLast(lista,elemento)

    if lt.size(lista) > 1:
        merg.sort(lista,cmp_impuestos_by_retencion_act)

    if lt.size(lista) <= 6:
        for elemento in lista["elements"]:
            table = [elemento["Año"],elemento["Código subsector económico"],elemento["Código actividad económica"], elemento["Nombre activ
                     elemento["Total retenciones"],elemento["Total ingresos netos"],
                     elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
            tabla_grande.append(table)
```

```
    else:

        peores = lt.subList(lista,1,3)
        mejores = lt.subList(lista,lt.size(lista)-3,3)

        for elemento in peores["elements"]:
            table = [elemento["Año"],elemento["Código subsector económico"],elemento["Código actividad económica"], elemento["Nombre activ
                     elemento["Total retenciones"],elemento["Total ingresos netos"],
                     elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
            tabla_grande.append(table)
        for elemento in mejores["elements"]:
            table = [elemento["Año"],elemento["Código subsector económico"],elemento["Código actividad económica"], elemento["Nombre activ
                     elemento["Total retenciones"],elemento["Total ingresos netos"],
                     elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
            tabla_grande.append(table)

return lista_min, tabla_grande
```

Entrada	Estructura de datos
Salidas	Una lista con los subsectores con las retenciones totales más bajas para cada año y otra lista con las actividades económica que más y menos aportaron a las retenciones totales para cada año
Implementado (Sí/No)	Si. Implementado por Sara Leiva.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Organizar el data_structs por retenciones y por la suma de retenciones con una función externa y guardarlo en una variable lista_sub	O (n)
Crear una nueva lista vacía y guardarla en una variable lista_min (lt.newList)	O (1)
Iterar *elemento* sobre lista_sub (lt.iterator)	O (n)
Se verifica si el año del *elemento* está presente en lista_min y se almacena su posición en la lista en una variable estar (lt.isPresent)	O (n)
Si estar es igual a 0, añadir a lo último el elemento dentro de lista_min (lt.addLast)	O (1)
Crear una lista vacía y guardarlo en la variable tabla_grande	O (1)
Crear una lista vacía guardarlo en la variable table	O (1)
Iterar *subsector* sobre lista_min (lt.iterator)	O (n)
Se saca el código del subsector económico del *subsector* se guarda en cod. También el año del *subsector* y se guarda en anio	O (1)
Crear una nueva lista y guardarla en una variable lista (lt.newList)	O (1)
Iterar *elemento* sobre el data_structs (lt.iterator)	O (n)
Si es año del *elemento* es igual a anio y el código del subsector económico del *elemento* es igual a cod, añadir el *elemento* al final de la lista (lt.addLast)	O (1)
Si el número de elementos de la lista es menor a 1, organizar por el valor de “Total de retenciones del subsector económico” con una función externa	O (1)
Si el número de elementos de la lista es menor o igual a 6, iterar *elemento* por todos los elementos de la lista e irlos agregando a tabla grande	O (1)
En el caso contrario que los elementos de la lista sean mayores que 6, sacar 2 sublistas en las cuales se guardan los primeros 3 elementos, en una variable llamada peores, y los últimos 3 elementos, en una variable llamada mejores (lt.subList)	O (n)
Iterar *elemento* sobre los elementos de la sublista peores	O (n)
Todos los elementos dentro de peores se van agregando a tabla_grande	O (1)
Iterar *elemento* sobre los elementos de la sublista mejores	O (n)
Todos los elementos dentro de mejores se van agregando a tabla_grande	O (1)

TOTAL	***O(n^2)***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

Entrada	**Tiempo (ms)**
small	28.98
5 pct	30.79
10 pct	32.77
20 pct	41.71
30 pct	46.47
50 pct	57.65
80 pct	71.90
large	75.44

Procesadores	**Intel® Core™ i7-1165G7 @2.80GHz**
Memoria RAM	15.6 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	**Tiempo (ms)**
small	3.25
5 pct	5.37
10 pct	6.87
20 pct	14.92
30 pct	22.09
50 pct	32.52
80 pct	48.08
large	55.97

Tablas de datos

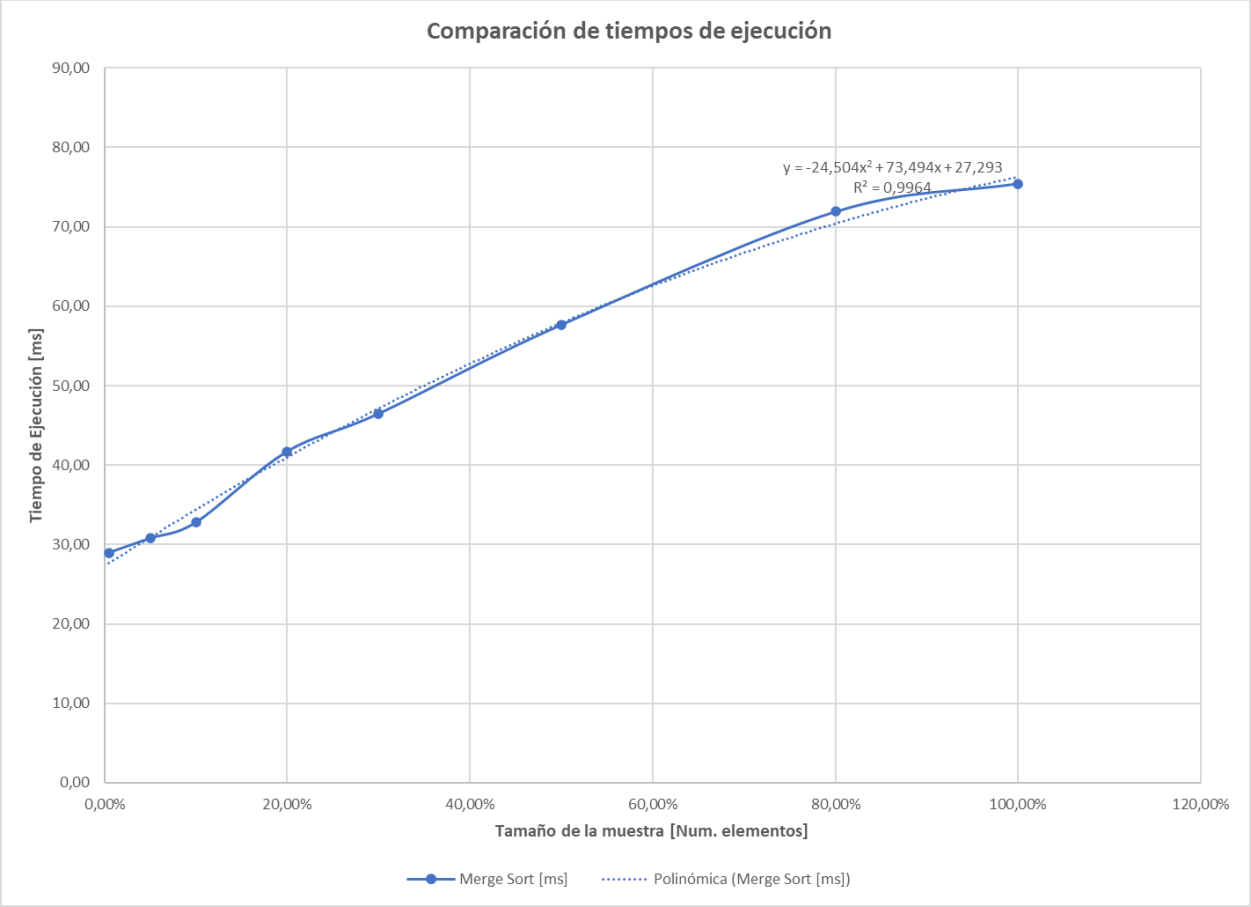
Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Los subsectores con las retenciones totales más bajas y las actividades económicas que más y menos aportaron a las retenciones totales para cada año	16.11
5 pct	Los subsectores con las retenciones totales más bajas y las actividades económicas que más y menos aportaron a las retenciones totales para cada año	18.08
10 pct	Los subsectores con las retenciones totales más bajas y las actividades económicas que más y menos aportaron a las retenciones totales para cada año	19.82
20 pct	Los subsectores con las retenciones totales más bajas y las actividades económicas que más y menos aportaron a las retenciones totales para cada año	28.31
30 pct	Los subsectores con las retenciones totales más bajas y las actividades económicas que más y menos aportaron a las retenciones totales para cada año	34.28
50 pct	Los subsectores con las retenciones totales más bajas y las actividades económicas que más y menos aportaron a las retenciones totales para cada año	44.95
80 pct	Los subsectores con las retenciones totales más bajas y las actividades económicas que más y menos aportaron a las retenciones totales para cada año	59.99
large	Los subsectores con las retenciones totales más bajas y las actividades económicas que más y menos aportaron a las retenciones totales para cada año	65.70

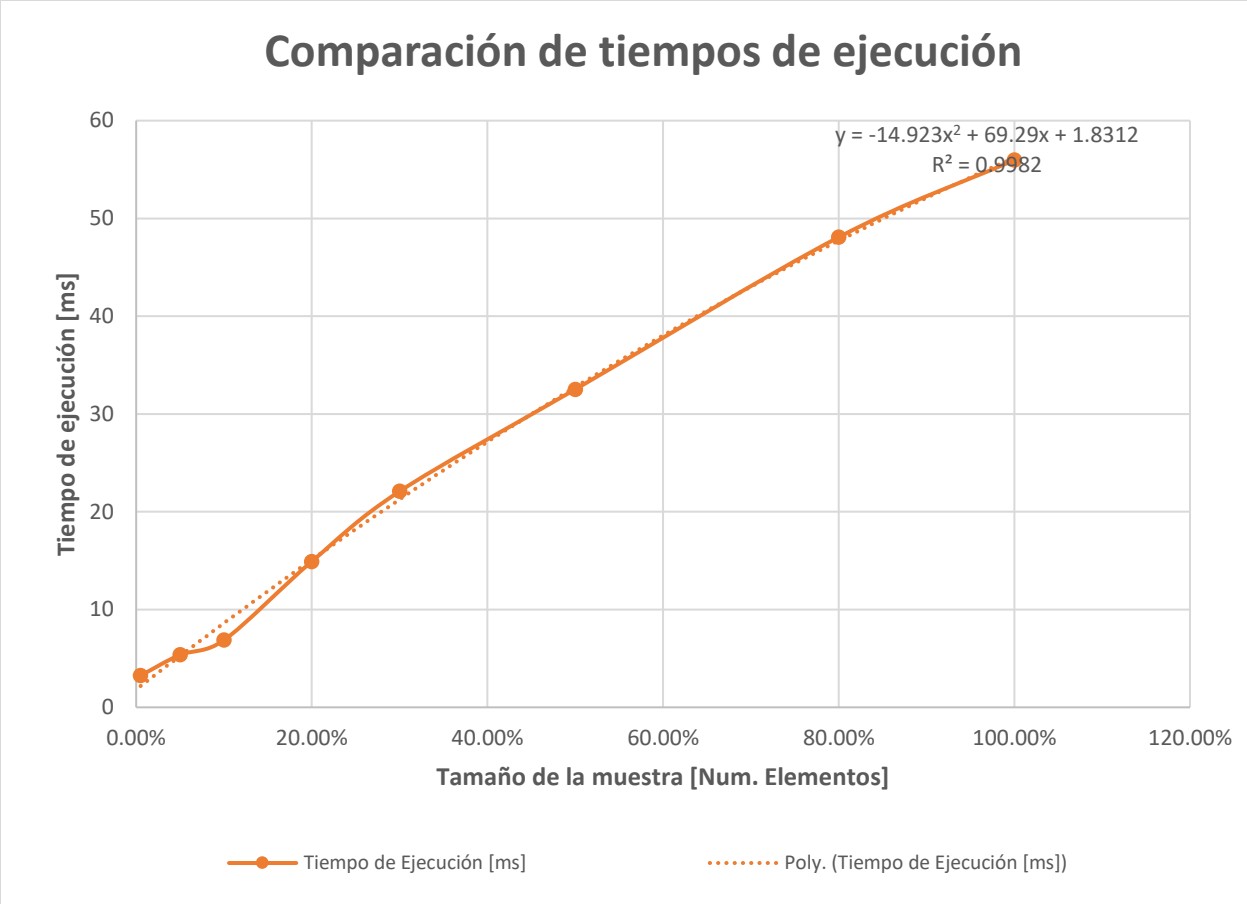
Graficas

Las gráficas con la representación de las pruebas realizadas.

- Máquina 1:



- Máquina 2:



- Máquina 3:

Análisis

Al implementar el requerimiento 3 obtenemos como resultado un Array_list de orden lineal a pesar de que la complejidad del algoritmo es $O(n^2)$**.** Este algoritmo presenta más complejidad que el requerimiento 1 y 2, ya que en este se presentan una serie de bucles anidados como lo son el for para recorrer la lista_sub y el for para recorrer la lista_min, que son los principales for de todo el código. Además, dentro de este primer for hay un lt.isPresent que tiene complejidad O(n) por lo que esto aumenta la complejidad total del algoritmo.

Lo descrito anteriormente es posible visualizarlo con las gráficas, puesto que esta muestra cómo se tiene un crecimiento de orden lineal, ya que al duplicar los datos de entrada los tiempos de ejecución no se duplican.

Requerimiento 4

Descripción

```
def req_4(data_structs):
    """
    Funcion que soluciona el requerimiento 4
    """

    lista_sub, dic = organizar_nomina(data_structs)
    lista_max = lt.newList(datastructure="ARRAY_LIST",cmpfunction=compare2)

    for elemento in lt.iterator(lista_sub):
        estar = lt.isPresent(lista_max, elemento["Año"])

        if estar == 0:
            lt.addLast(lista_max, elemento)

        if estar >0:
            if int(elemento["Total de costos y gastos nomina del subsector economico"]) > int(lt.getElement(lista_sub,estar)["Total de cost
                lt.changeInfo(lista_max,estar,elemento)

    tabla_grande = []
    table = []
```

```
for subsector in lt.iterator(lista_max):
    cod = subsector["Código subsector económico"]
    anio = subsector["Año"]

    lista = lt.newList(datastructure="ARRAY_LIST")
    for elemento in lt.iterator(data_structs["data"]):
        if elemento["Año"] == anio and elemento["Código subsector económico"] == cod:
            lt.addLast(lista,elemento)

    if lt.size(lista) > 1:
        merg.sort(lista,cmp_impuestos_by_nomina_act)

    if lt.size(lista) <= 6:
        for elemento in lista["elements"]:
            table = [elemento["Año"],elemento["Código subsector económico"],elemento["Código actividad económica"], elemento["Nombre activ
                     elemento["Costos y gastos nómina"],elemento["Total ingresos netos"],
                     elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
            tabla_grande.append(table)
```

```
    else:

        peores = lt.subList(lista,1,3)
        mejores = lt.subList(lista,lt.size(lista)-3,3)

        for elemento in peores["elements"]:
            table = [elemento["Año"],elemento["Código subsector económico"],elemento["Código actividad económica"], elemento["Nombre activ
                     elemento["Costos y gastos nómina"],elemento["Total ingresos netos"],
                     elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
            tabla_grande.append(table)
        for elemento in mejores["elements"]:
            table = [elemento["Año"],elemento["Código subsector económico"],elemento["Código actividad económica"], elemento["Nombre activ
                     elemento["Costos y gastos nómina"],elemento["Total ingresos netos"],
                     elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
            tabla_grande.append(table)


return lista_max, tabla_grande
```

Entrada	Estructura de datos
Salidas	Una lista con los subsectores con los mayores costos y gastos de nómina y otra lista con las actividades económica que más y menos aportaron a los costos y gastos de nómina para cada año
Implementado (Sí/No)	Si. Implementado por Sara Leiva.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Organizar el data_structs por nómina y por la suma de nómina con una función externa y guardarlo en una variable lista_sub	O (n)
Crear una nueva lista vacía y guardarla en una variable lista_max (lt.newList)	O (1)
Iterar *elemento* sobre lista_sub (lt.iterator)	O (n)
Se verifica si el año del *elemento* está presente en lista_max y se almacena su posición en la lista en una variable estar (lt.isPresent)	O (n)
Si estar es igual a 0, añadir a lo último el elemento dentro de lista_max (lt.addLast)	O (1)
Si estar es mayor que cero, se compara el *elemento* y el elemento de la lista que corresponde a la posición guardada en estar, la comparación se hace con el valor de la llave "Total de costos y gastos nómina del subsector económico" para ambos (Lt.getElement)	O (n)
En caso de que *elemento* tenga un valor mayor en "Total de costos y gastos nómina del subsector económico", se actualiza el elemento en la posición guardada en estar de la lista_max con la información de *elemento* (lt.changeInfo)	O (1)
Crear una lista vacía y guardarlo en la variable tabla_grande	O (1)
Crear una lista vacía guardarlo en la variable table	O (1)
Iterar *subsector* sobre lista_max (lt.iterator)	O (n)
Se saca el código del subsector económico del *subsector* se guarda en cod. También el año del *subsector* y se guarda en anio	O (1)
Crear una nueva lista y guardarla en una variable lista (lt.newList)	O (1)
Iterar *elemento* sobre el data_structs (lt.iterator)	O (n)
Si es año del *elemento* es igual a anio y el código del subsector económico del *elemento* es igual a cod, añadir el *elemento* al final de la lista (lt.addLast)	O (1)
Si el número de elementos de la lista es menor a 1, organizar con un merge sort por el valor de "Total de costos y gastos nómina del subsector económico" en una función externa	O (n Log(n))
Si el número de elementos de la lista es menor o igual a 6, iterar *elemento* por todos los elementos de la lista e irlos agregando a tabla grande	O (1)

En el caso contrario que los elementos de la lista sean mayores que 6, sacar 2 sublistas en las cuales se guardan los primeros 3 elementos, en una variable llamada peores, y los últimos 3 elementos, en una variable llamada mejores (lt.subList)	O (n)
Iterar *elemento* sobre los elementos de la sublista peores	O (n)
Todos los elementos dentro de peores se van agregando a tabla_grande	O (1)
Iterar *elemento* sobre los elementos de la sublista mejores	O (n)
Todos los elementos dentro de mejores se van agregando a tabla_grande	O (1)
TOTAL	***O(n²)***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

Entrada	**Tiempo (ms)**
small	29.96
5 pct	47.87
10 pct	66.65
20 pct	84.05
30 pct	99.42
50 pct	118.32
80 pct	130.93
large	158.55

Procesadores	**Intel® Core™ i7-1165G7 @2.80GHz**
Memoria RAM	15.6 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	**Tiempo (ms)**

small	2.92
5 pct	2.76
10 pct	4.78
20 pct	18.65
30 pct	21.64
50 pct	52.48
80 pct	85.98
large	100.72

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

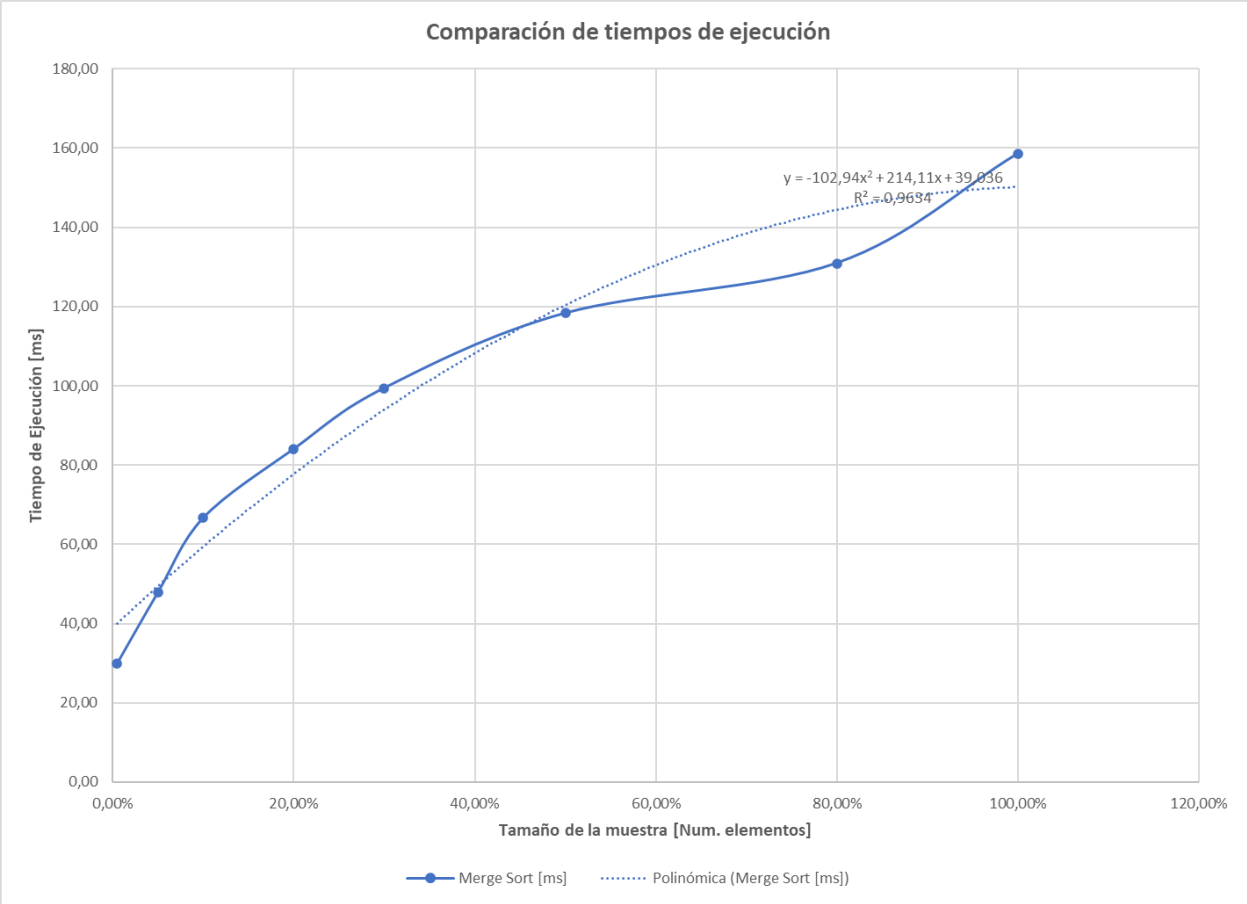
Muestra	Salida	Tiempo (ms)
small	Los subsectores con los mayores costos y gastos de nómina y las actividades económicas que más y menos aportaron a los costos y gastos de nómina para cada año	2.92
5 pct	Los subsectores con los mayores costos y gastos de nómina y las actividades económicas que más y menos aportaron a los costos y gastos de nómina para cada año	2.76
10 pct	Los subsectores con los mayores costos y gastos de nómina y las actividades económicas que más y menos aportaron a los costos y gastos de nómina para cada año	4.78
20 pct	Los subsectores con los mayores costos y gastos de nómina y las actividades económicas que más y menos aportaron a los costos y gastos de nómina para cada año	18.65
30 pct	Los subsectores con los mayores costos y gastos de nómina y las actividades económicas que más y menos aportaron a los costos y gastos de nómina para cada año	21.64
50 pct	Los subsectores con los mayores costos y gastos de nómina y las actividades económicas que más y	52.48

	menos aportaron a los costos y gastos de nómina para cada año	
80 pct	Los subsectores con los mayores costos y gastos de nómina y las actividades económicas que más y menos aportaron a los costos y gastos de nómina para cada año	85.98
large	Los subsectores con los mayores costos y gastos de nómina y las actividades económicas que más y menos aportaron a los costos y gastos de nómina para cada año	100.72

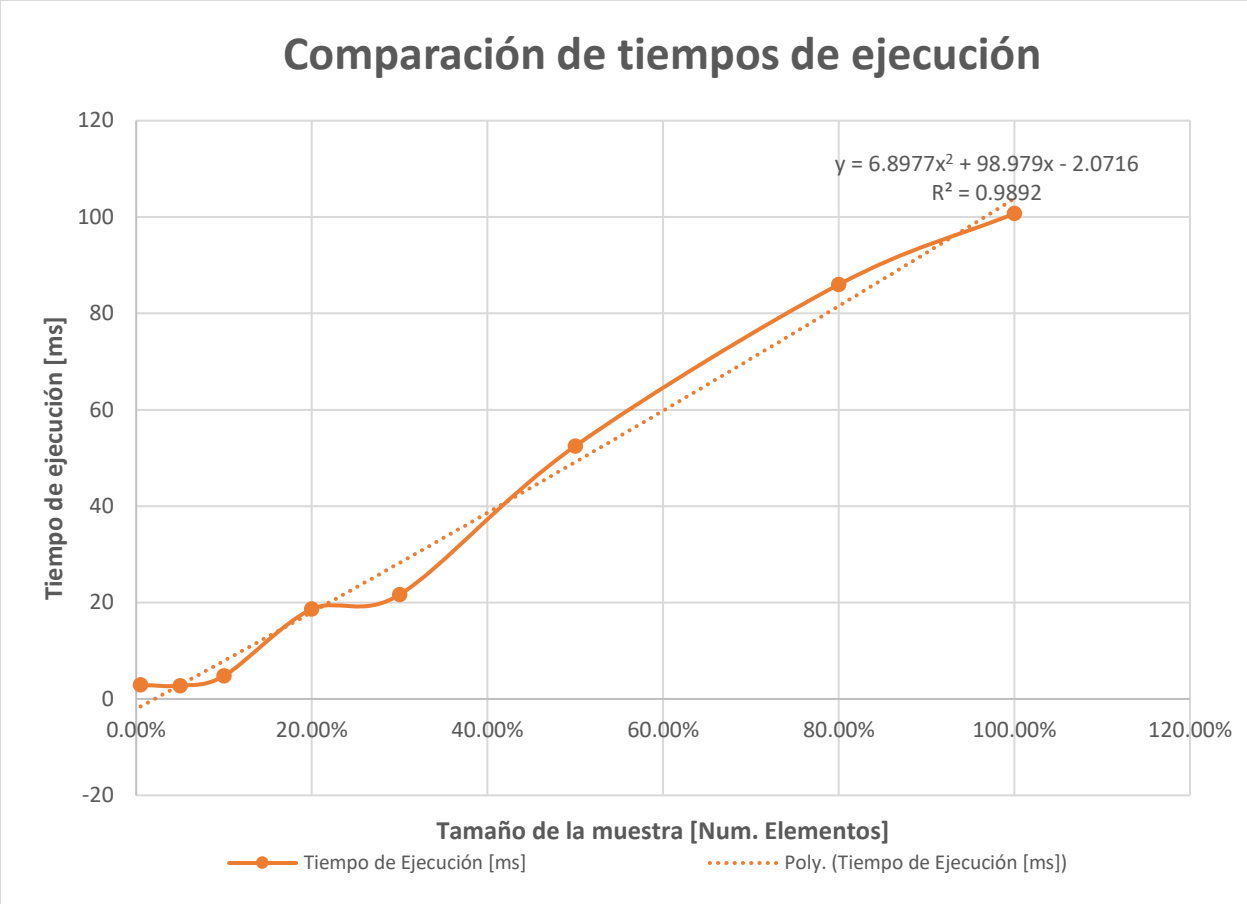
Graficas

Las gráficas con la representación de las pruebas realizadas.

- Máquina 1:



- Máquina 2:



- Máquina 3:

Análisis

Al implementar el requerimiento 4 obtenemos como resultado un Array_list de orden lineal a pesar de que la complejidad del algoritmo es $O(n^2)$**.** Este algoritmo presenta más complejidad que el requerimiento 1 y 2, ya que en este se presentan una serie de bucles anidados como lo son el for para recorrer la lista_sub y el for para recorrer la lista_max, que son los principales for de todo el código. Además, dentro de este primer for hay un lt.isPresent que tiene complejidad O(n) por lo que esto aumenta la complejidad total del algoritmo.

Lo descrito anteriormente es posible visualizarlo con las gráficas, puesto que esta muestra cómo se tiene un crecimiento de orden lineal, ya que al duplicar los datos de entrada los tiempos de ejecución no se duplican.

Requerimiento 5

```
def req_5(data_structs):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    lista_sub = organizar_nomina(data_structs)
    lista_max = lt.newList(datastructure="ARRAY_LIST",cmpfunction=compare2)

    for elemento in lt.iterator(lista_sub):
        estar = lt.isPresent(lista_max, elemento["Año"])

        if estar == 0:
            lt.addLast(lista_max, elemento)

        if estar >0:
            if int(elemento["Total de impuestos a cargo para el subsector"]) > int(lt.getElement(lista_sub,estar)["Total de impuestos a car
                lt.changeInfo(lista_max,estar,elemento)

    tabla_grande = []
    table = []
```

```
for subsector in lt.iterator(lista_max):
    cod = subsector["Código subsector económico"]
    anio = subsector["Año"]

    lista = lt.newList(datastructure="ARRAY_LIST")
    for elemento in lt.iterator(data_structs["data"]):
        if elemento["Año"] == anio and elemento["Código subsector económico"] == cod:
            lt.addLast(lista,elemento)

    if lt.size(lista) > 1:
        merg.sort(lista,cmp_by_descuento_act)

    if lt.size(lista) <= 6:
        for elemento in lista["elements"]:
            table = [elemento["Año"],elemento["Código subsector económico"],elemento["Código actividad económica"], elemento["Nombre activi
                    elemento["Descuentos tributarios"],elemento["Total ingresos netos"],
                    elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
            tabla_grande.append(table)
```

```
    else:

        peores = lt.subList(lista,1,3)
        mejores = lt.subList(lista,lt.size(lista)-3,3)

        for elemento in peores["elements"]:
            table = [elemento["Año"],elemento["Código subsector económico"],elemento["Código actividad económica"], elemento["Nombre activi
                    elemento["Descuentos tributarios"],elemento["Total ingresos netos"],
                    elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
            tabla_grande.append(table)
        for elemento in mejores["elements"]:
            table = [elemento["Año"],elemento["Código subsector económico"],elemento["Código actividad económica"], elemento["Nombre activi
                    elemento["Descuentos tributarios"],elemento["Total ingresos netos"],
                    elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
            tabla_grande.append(table)


return lista_max, tabla_grande
```

Descripción

Entrada	Estructura de datos
Salidas	Una lista con subsectores con los mayores descuentos tributarios y otra lista con las actividades económicas que más y menos aportaron a los descuentos tributarios para cada año
Implementado (Sí/No)	Si. Implementado por Sara Leiva.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Organizar el data_structs por nómina y por la suma de nómina con una función externa y guardarlo en una variable lista_sub	O (n)

Crear una nueva lista vacía y guardarla en una variable lista_max (lt.newList)	O (1)
Iterar *elemento* sobre lista_sub (lt.iterator)	O (n)
Se verifica si el año del *elemento* está presente en lista_max y se almacena su posición en la lista en una variable estar (lt.isPresent)	O (n)
Si estar es igual a 0, añadir a lo último el elemento dentro de lista_max (lt.addLast)	O (1)
Si estar es mayor que cero, se compara el *elemento* y el elemento de la lista que corresponde a la posición guardada en estar, la comparación se hace con el valor de la llave "Total de impuestos a cargo del subsector económico" para ambos (Lt.getElement)	O (n)
En caso de que *elemento* tenga un valor mayor en "Total de impuestos a cargo del subsector económico", se actualiza el elemento en la posición guardada en estar de la lista_max con la información de *elemento* (lt.changeInfo)	O (1)
Crear una lista vacía y guardarlo en la variable tabla_grande	O (1)
Crear una lista vacía guardarlo en la variable table	O (1)
Iterar *subsector* sobre lista_max (lt.iterator)	O (n)
Se saca el código del subsector económico del *subsector* se guarda en cod. También el año del *subsector* y se guarda en anio	O (1)
Crear una nueva lista y guardarla en una variable lista (lt.newList)	O (1)
Iterar *elemento* sobre el data_structs (lt.iterator)	O (n)
Si es año del *elemento* es igual a anio y el código del subsector económico del *elemento* es igual a cod, añadir el *elemento* al final de la lista (lt.addLast)	O (1)
Si el número de elementos de la lista es menor a 1, organizar con un merge sort por el valor de "Total de costos y gastos nómina del subsector económico" en una función externa	O (n Log(n))
Si el número de elementos de la lista es menor o igual a 6, iterar *elemento* por todos los elementos de la lista e irlos agregando a tabla grande	O (1)
En el caso contrario que los elementos de la lista sean mayores que 6, sacar 2 sublistas en las cuales se guardan los primeros 3 elementos, en una variable	O (n)

llamada peores, y los últimos 3 elementos, en una variable llamada mejores (lt.subList)	
Iterar *elemento* sobre los elementos de la sublista peores	O (n)
Todos los elementos dentro de peores se van agregando a tabla_grande	O (1)
Iterar *elemento* sobre los elementos de la sublista mejores	O (n)
Todos los elementos dentro de mejores se van agregando a tabla_grande	O (1)
TOTAL	***O(n²)***

Pruebas Realizadas

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

Entrada	**Tiempo (ms)**
small	1.17
5 pct	4.07
10 pct	7.84
20 pct	11.04
30 pct	18.94
50 pct	22.95
80 pct	48.17
large	49.3

Procesadores	**Intel® Core™ i7-1165G7 @2.80GHz**
Memoria RAM	15.6 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	**Tiempo (ms)**
small	0.22
5 pct	3.51
10 pct	5.76
20 pct	8.96
30 pct	14.32
50 pct	19.45
80 pct	24.56

large	26.53

Tablas de datos

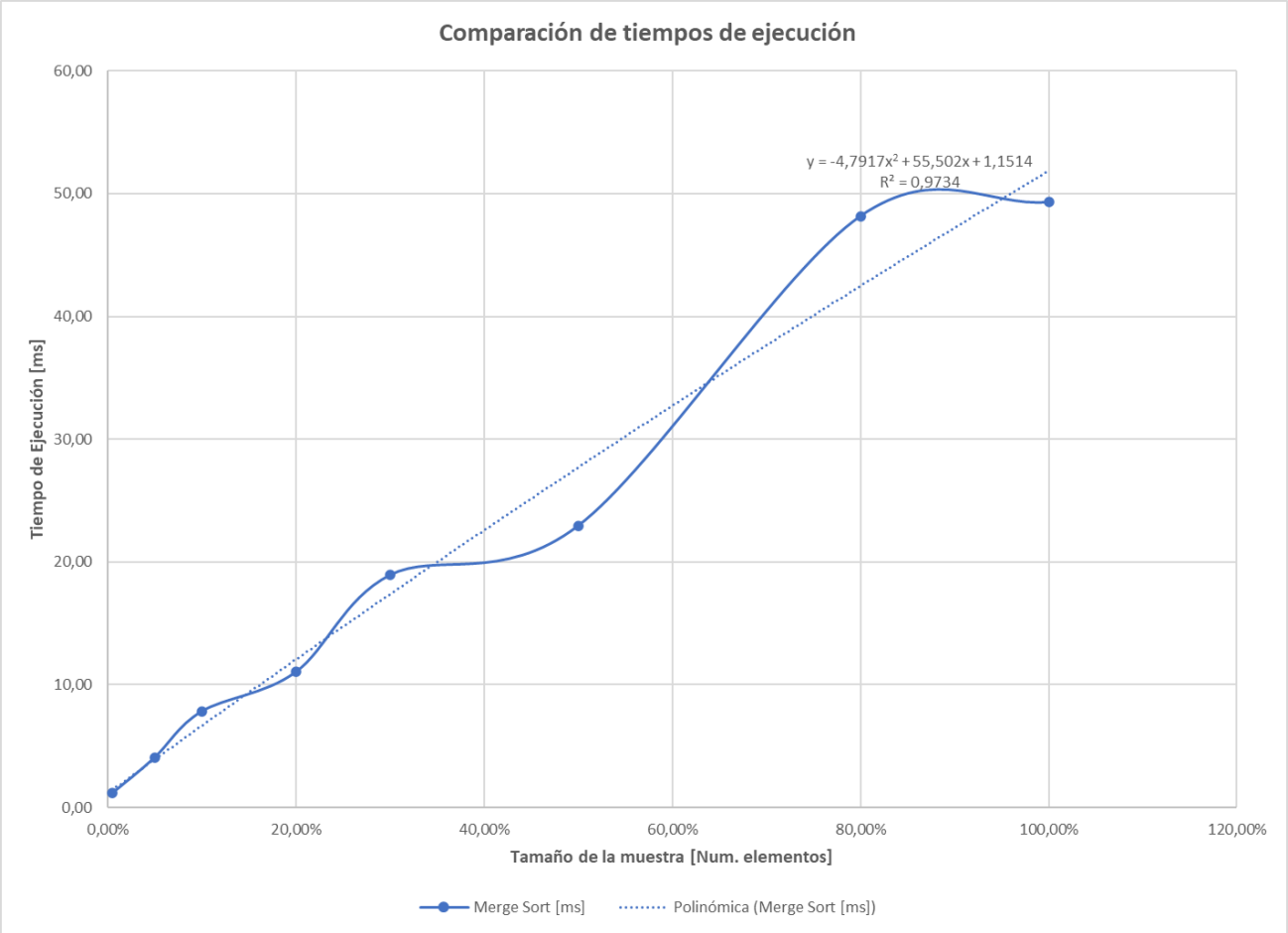
Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Los subsectores con los mayores descuentos tributarios y las actividades económicas que más y menos aportaron al descuento tributario para cada año	0.22
5 pct	Los subsectores con los mayores descuentos tributarios y las actividades económicas que más y menos aportaron al descuento tributario para cada año	3.51
10 pct	Los subsectores con los mayores descuentos tributarios y las actividades económicas que más y menos aportaron al descuento tributario para cada año	5.76
20 pct	Los subsectores con los mayores descuentos tributarios y las actividades económicas que más y menos aportaron al descuento tributario para cada año	8.96
30 pct	Los subsectores con los mayores descuentos tributarios y las actividades económicas que más y menos aportaron al descuento tributario para cada año	14.32
50 pct	Los subsectores con los mayores descuentos tributarios y las actividades económicas que más y menos aportaron al descuento tributario para cada año	19.45
80 pct	Los subsectores con los mayores descuentos tributarios y las actividades económicas que más y menos aportaron al descuento tributario para cada año	24.56
large	Los subsectores con los mayores descuentos tributarios y las actividades económicas que más y menos aportaron al descuento tributario para cada año	26.53

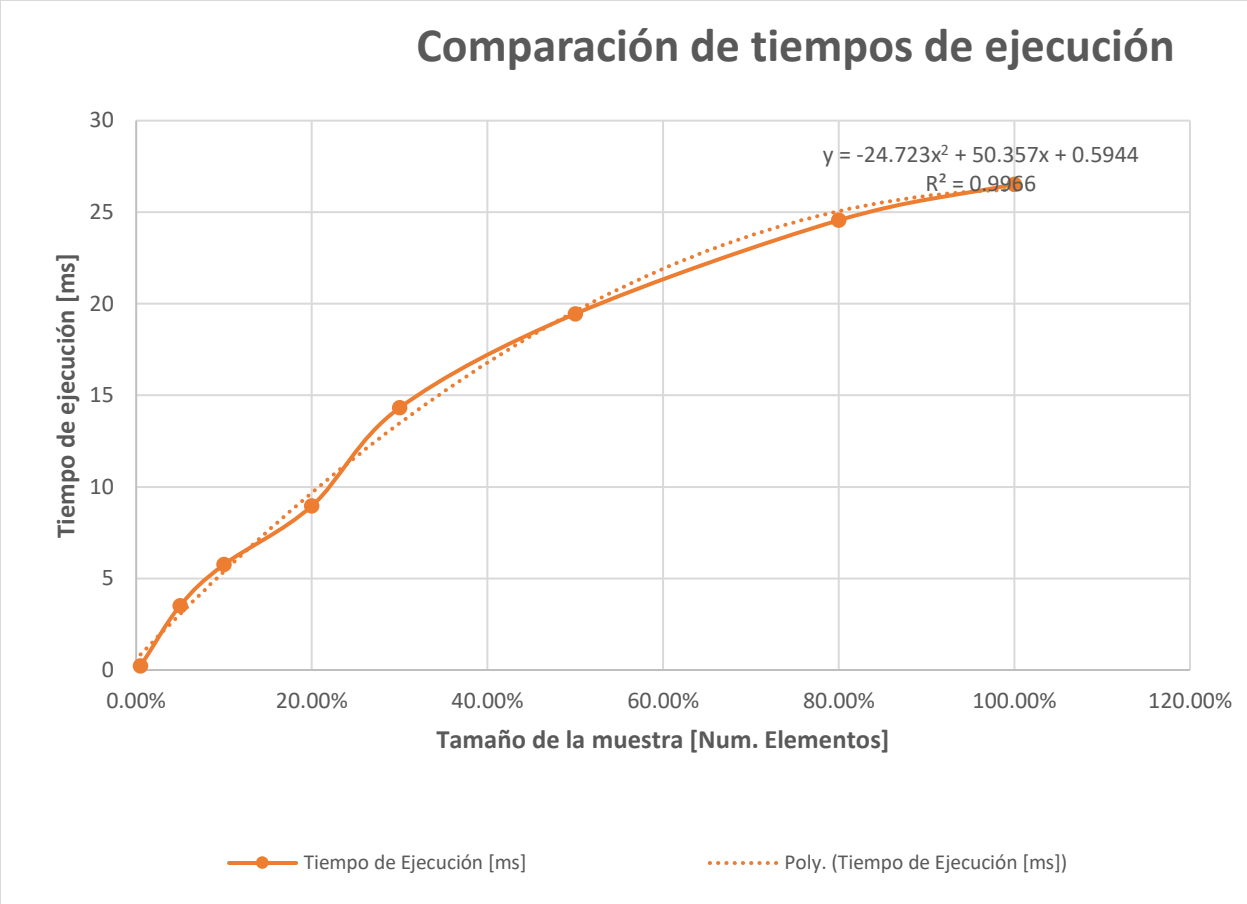
Graficas

Las gráficas con la representación de las pruebas realizadas.

- Máquina 1:



- Máquina 2:



Análisis

Al implementar el requerimiento 5 obtenemos como resultado un Array_list de orden lineal a pesar de que la complejidad del algoritmo es $O(n^2)$**.** Este algoritmo presenta más complejidad que el requerimiento 1 y 2, ya que en este se presentan una serie de bucles anidados como lo son el for para recorrer la lista_sub y el for para recorrer la lista_max, que son los principales for de todo el código. Además, dentro de este primer for hay un lt.isPresent que tiene complejidad O(n) por lo que esto aumenta la complejidad total del algoritmo.

Lo descrito anteriormente es posible visualizarlo con las gráficas, puesto que esta muestra cómo se tiene un crecimiento de orden lineal, ya que al duplicar los datos de entrada los tiempos de ejecución no se duplican.

Requerimiento 6

Descripción

```python
def req_6(data_structs, year = 2019):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6

    anio = lt.newList("ARRAY_LIST")

    for act_eco in lt.iterator(data_structs["data"]):
        if int(act_eco["Año"]) == year:
            lt.addLast(anio, act_eco)

    sectores = lt.newList("ARRAY_LIST")
    cod_sectores = lt.newList("ARRAY_LIST")

    for actividad in lt.iterator(anio):
        if actividad["Código sector económico"] not in lt.iterator(cod_sectores):
            lt.addLast(cod_sectores, actividad["Código sector económico"])
            sector = lt.newList("ARRAY_LIST")
            lt.addLast(sector, actividad["Código sector económico"])
            actividades = lt.newList("ARRAY_LIST")
            lt.addLast(actividades, actividad)
            lt.addLast(sector, actividades)
            lt.addLast(sectores, sector)
        else:
            for sec in lt.iterator(sectores):
                if int(lt.firstElement(sec)) == int(actividad["Código sector económico"]):
                    lt.addLast(lt.getElement(sec, 2), actividad)
```

```python
for sect in lt.iterator(sectores):
    sum_in = 0
    sum_cg = 0
    sum_sp = 0
    sum_sf = 0
    for act in lt.iterator(lt.getElement(sect, 2)):
        sum_in = sum_in + int(act["Total ingresos netos"])
        sum_cg = sum_cg + int(act["Total costos y gastos"])
        sum_sp = sum_sp + int(act["Total saldo a pagar"])
        sum_sf = sum_sf + int(act["Total saldo a favor"])
    lt.addLast(sect, sum_in)
    lt.addLast(sect, sum_cg)
    lt.addLast(sect, sum_sp)
    lt.addLast(sect, sum_sf)

sector_max = lt.firstElement(sectores)

for s in lt.iterator(sectores):
    if int(lt.getElement(s, 3)) > int(lt.getElement(sector_max, 3)):
        sector_max = s

sub_sectores = lt.newList("ARRAY_LIST")
cod_subsectores = lt.newList("ARRAY_LIST")
```

```
for actividad in lt.iterator(lt.getElement(sector_max, 2)):
    if not actividad["Código subsector económico"] in lt.iterator(cod_subsectores):
        lt.addLast(cod_subsectores, actividad["Código subsector económico"])
        subsector = lt.newList("ARRAY_LIST")
        lt.addLast(subsector, actividad["Código subsector económico"])
        actividades = lt.newList("ARRAY_LIST")
        lt.addLast(actividades, actividad)
        lt.addLast(subsector, actividades)
        lt.addLast(sub_sectores, subsector)
    else:
        for sec in lt.iterator(sub_sectores):
            if int(lt.firstElement(sec)) == int(actividad["Código subsector económico"]):
                lt.addLast(lt.getElement(sec, 2), actividad)

for subsect in lt.iterator(sub_sectores):
    sum_in = 0
    sum_cg = 0
    sum_sp = 0
    sum_sf = 0
    for act in lt.iterator(lt.getElement(sect, 2)):
        sum_in = sum_in + int(act["Total ingresos netos"])
        sum_cg = sum_cg + int(act["Total costos y gastos"])
        sum_sp = sum_sp + int(act["Total saldo a pagar"])
        sum_sf = sum_sf + int(act["Total saldo a favor"])
    lt.addLast(subsect, sum_in)
    lt.addLast(subsect, sum_cg)
    lt.addLast(subsect, sum_sp)
    lt.addLast(subsect, sum_sf)
```

```
merg.sort(sub_sectores, cmp_t_ing)

sub_max = lt.firstElement(sub_sectores)
sub_min = lt.lastElement(sub_sectores)

merg.sort(lt.getElement(sub_max, 2), compere_actividad)
merg.sort(lt.getElement(sub_min, 2), compere_actividad)

max_act = (lt.firstElement(lt.getElement(sub_max, 2)), lt.lastElement(lt.getElement(sub_max, 2)))
min_act = (lt.firstElement(lt.getElement(sub_min, 2)), lt.lastElement(lt.getElement(sub_min, 2)))

return sector_max, sub_max, sub_min, max_act, min_act
```

Entrada	Lista de datos, int año
Salidas	7 listas ordenadas con, la primera siendo del subsector, las dos siguientes del subsector que aporto más y menor respectivamente y las ultimas 4 listas están agrupadas por subsector con la actividad que más y menos aporto a dicho subsector. Todas las listas cuentan con: código, nombre, total de ingresos, total costo y gastos, total saldo a pagar y total saldo a favor
Implementado (Sí/No)	Si. Implementado por Andres Rodriguez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Crear una lista y guardarla en una variable llamada anio	O(1)
Iterar act_eco sobre el data_structs	O(n)
Si el año de act_eco es igual al año ingresado, agregar act_eco al final de la lista anio	O(1)

Crear una lista y guardarla en una variable llamada sectores	O(1)
Crear una lista y guardarla en una variable llamada cod_sectores	O(1)
Iterar actividad sobre la lista anio	O(n)
Si el código sector económico de actividad no está en la lista cod_sectores, agregar al final el codigo sector economico de actividad en la lista cod_sectores	O(n)
Crea una lista y la guardarla en una variable llamada sector	O(n)
Añadir al final de sector el codigo sector economico de actividad	O(1)
Crear una lista y guardarla en una variable llamada actividades	O(1)
Agregar al final de la lista actividades actividad	O(1)
Agregar al final de la lista sector actividades	O(1)
Agregar al final de la lista sectores sector	O(1)
Si lo anterior no se cumple, iterar sec sobre sectores	O(n)
Si el primer elemento sec es igual al código sector económico de actividad, agregar al final del segundo elemento de sec actividad	O(1)
Iterar sect sobre sectores	O(n)
Asigna la varibles sum_in	O(1)
Asigna la variable sum_cg	O(1)
Asigna la variable sum_sp	O(1)
Asigna la variable sum_sf	O(1)
Recorrer sect	O(n)
Añadir al final de los array lists las variables asignadas	O(1)
TOTAL	O(n^2)

Pruebas Realizadas

Procesadores	**Intel® Core™ i7-1165G7 @2.80GHz**
Memoria RAM	15.6 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	**Tiempo (ms)**
small	0.22
5 pct	0.75
10 pct	3.56
20 pct	6.76
30 pct	8.54
50 pct	15.65

80 pct	20.23
large	32.45

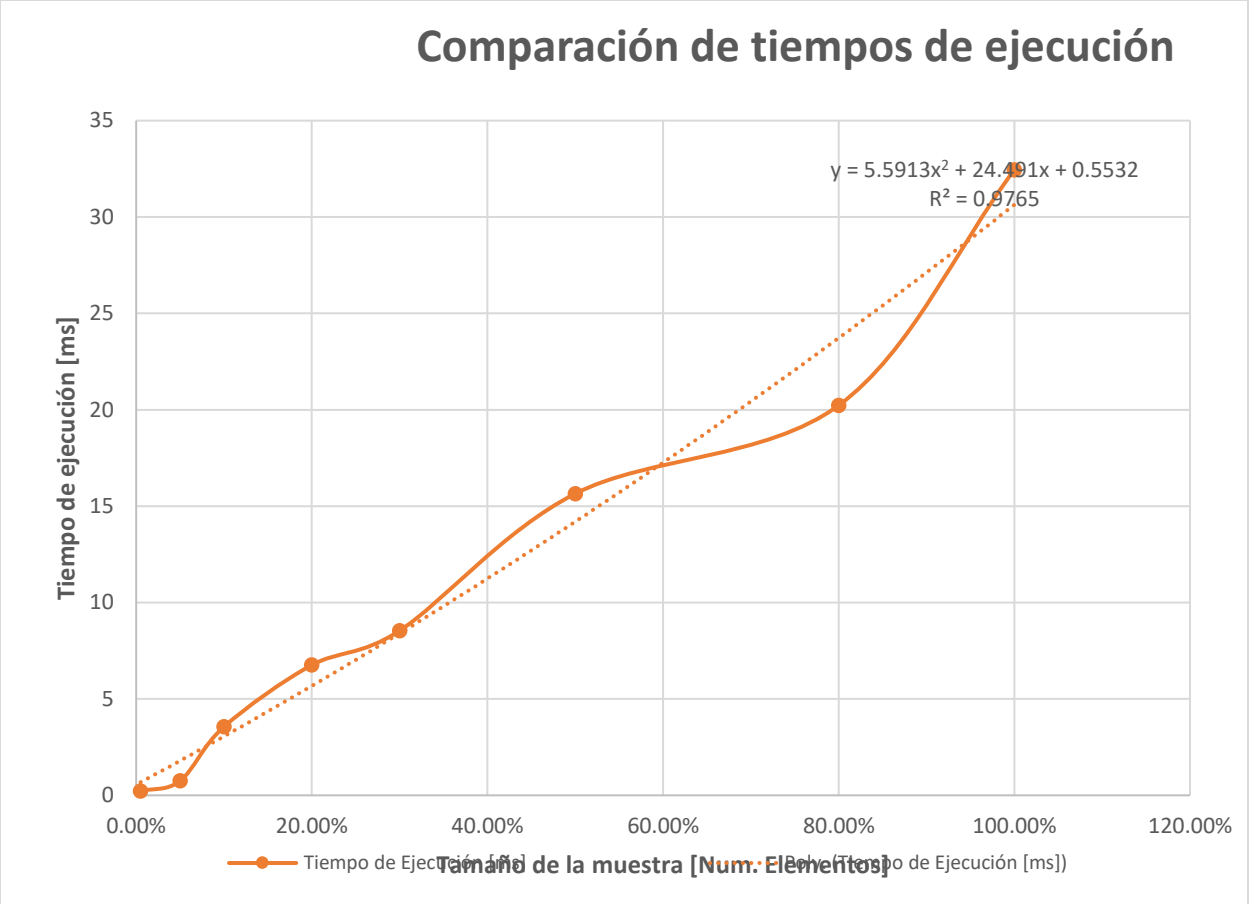
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	7 listas ordenadas	0.22
5 pct	7 listas ordenadas	0.75
10 pct	7 listas ordenadas	3.56
20 pct	7 listas ordenadas	6.76
30 pct	7 listas ordenadas	8.54
50 pct	7 listas ordenadas	15.65
80 pct	7 listas ordenadas	20.23
large	7 listas ordenadas	32.45

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Las gráficas y el tiempo de ejecución demuestran que el código tiene una complejidad que lo permite ser eficiente al tratar con una carga de datos de gran tamaño. Así mismo, evidencia que el código tiene una complejidad de O(n^2).

Requerimiento 7

```python
def req_7(data_structs,n,anio_inicial,anio_final):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7

    anio1 = int(anio_inicial)
    anio2 = int(anio_final)

    lista = lt.newList(datastructure="ARRAY_LIST")


    for elemento in lt.iterator(data_structs["data"]):
        if int(elemento["Año"]) in range(anio1,anio2+1):
            lt.addLast(lista,elemento)

    merg.sort(lista,cmp_impuestos_by_costos_gastos_act)

    sublista = lt.subList(lista,1,n)
    merg.sort(sublista,cmp_impuestos_by_costos_gastos_act_anio)

    return sublista
```

Descripción

Entrada	Estructura de datos, año inicial y año final
Salidas	Array list conformado por diccionarios con la información de las actividades económicas que cumplen con lo requerido.
Implementado (Sí/No)	Si. Implementado por Sara Leiva.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Convertir el año inicial que entra por parámetro a entero	O(1)
Convertir el año final que entra por parámetro a entero	O(1)
Crear un array list	O(1)
Recorrer los elementos de la estructura de datos	O(n)

Evaluar si el elemento cumple con las condiciones por medio de un “if”	O(1)
Añadir un elemento al final de la lista	O(1)
Organizar la lista por Merge Sort	O(nlogn)
Crear una sublista	O(n)
Organizar la sublista por Merge Sort	O(nlogn)
TOTAL	**O(n^2)**

Pruebas Realizadas

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

Entrada	**Tiempo (ms)**
small	0.83
5 pct	3.79
10 pct	9.60
20 pct	19.43
30 pct	30.49
50 pct	46.77
80 pct	76.52
large	99.63

Procesadores	**Intel® Core™ i7-1165G7 @2.80GHz**
Memoria RAM	15.6 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	**Tiempo (ms)**
small	0.61
5 pct	2.61
10 pct	7.29
20 pct	8.33
30 pct	9.42
50 pct	11.54
80 pct	24.54
large	33.77

Tablas de datos

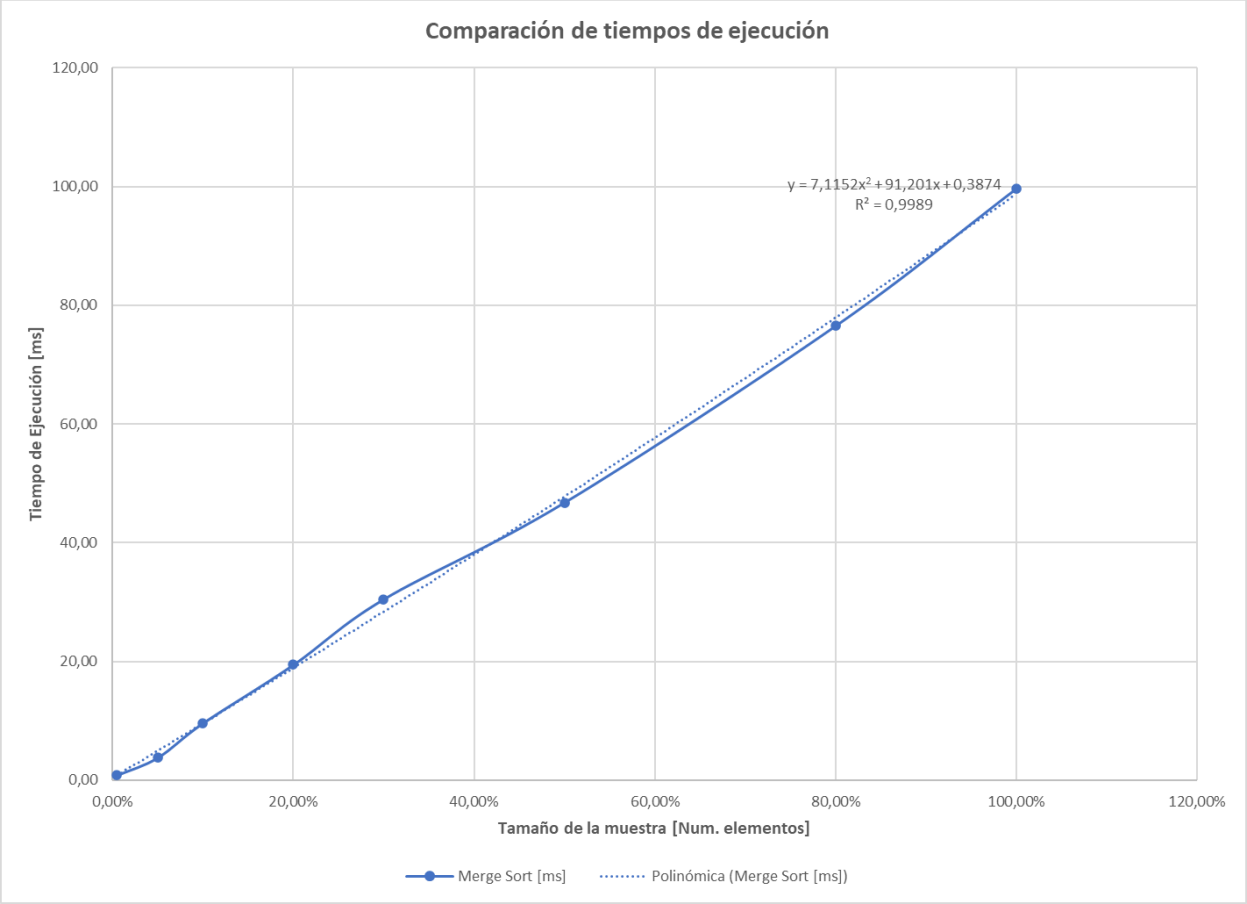
Las tablas con la recopilación de datos promedio de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Las actividades económicas con el menor total de costos y gastos para en un periodo de tiempo dado	0.61
5 pct	Las actividades económicas con el menor total de costos y gastos para en un periodo de tiempo dado	2.61
10 pct	Las actividades económicas con el menor total de costos y gastos para en un periodo de tiempo dado	7.29
20 pct	Las actividades económicas con el menor total de costos y gastos para en un periodo de tiempo dado	8.33
30 pct	Las actividades económicas con el menor total de costos y gastos para en un periodo de tiempo dado	9.42
50 pct	Las actividades económicas con el menor total de costos y gastos para en un periodo de tiempo dado	11.54
80 pct	Las actividades económicas con el menor total de costos y gastos para en un periodo de tiempo dado	24.54
large	Las actividades económicas con el menor total de costos y gastos para en un periodo de tiempo dado	33.77

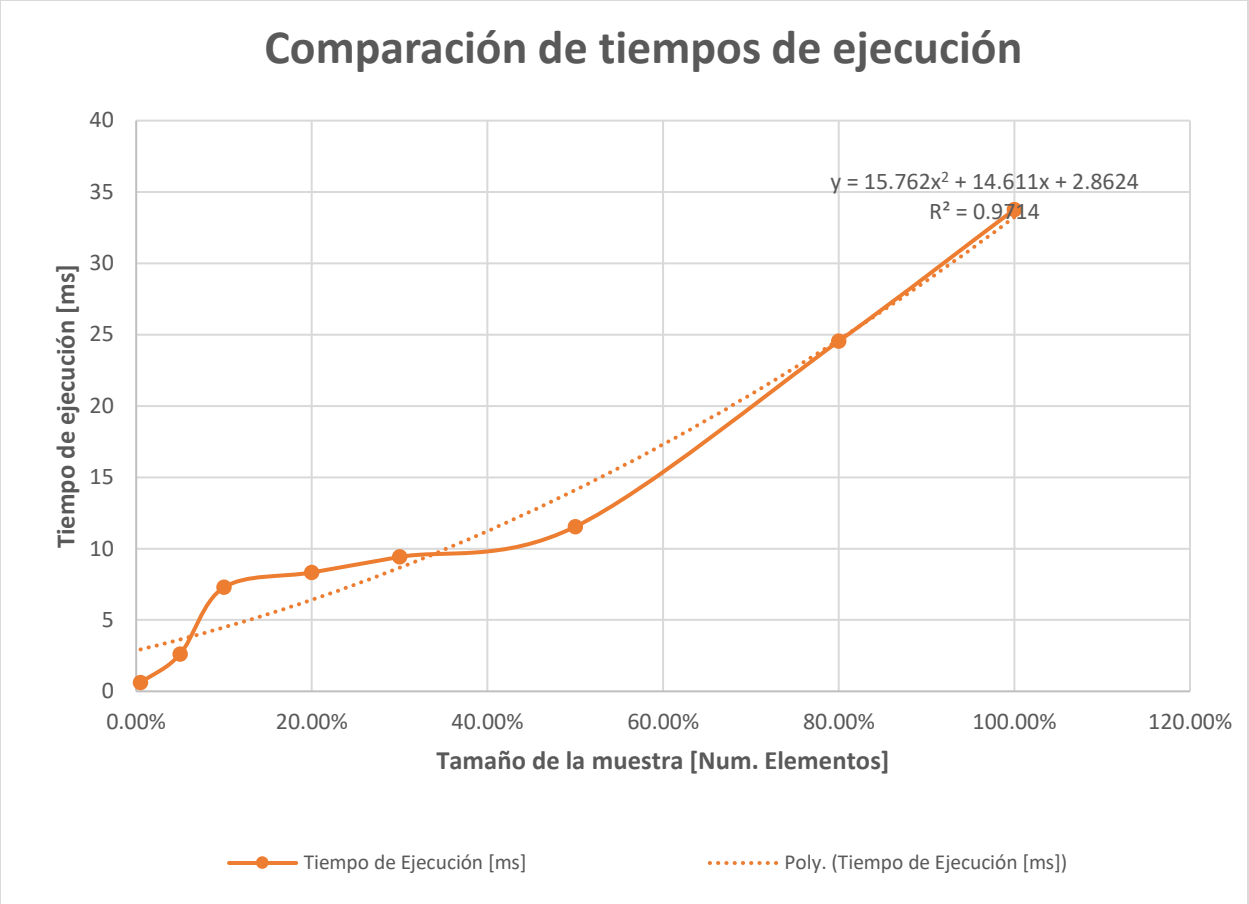
Graficas

Las gráficas con la representación de las pruebas realizadas.

- Máquina 1:


Comparación de tiempos de ejecución
120,00
100,00
80,00
60,00
40,00
20,00
0,00
Tiempo de Ejecución [ms]
y = 7,1152x² + 91,201x + 0,3874
R² = 0,9989
0,00%
20,00%
40,00%
60,00%
80,00%
100,00%
120,00%
Tamaño de la muestra [Num. elementos]
Merge Sort [ms]
Polinómica (Merge Sort [ms])

- Máquina 2:



- Máquina 3:

Análisis

Este código tiene una complejidad de O(n^2), que significa que se demora aproximadamente el número de elementos de la carga de datos, n, * n. Así mismo, esto puede ser evidenciado a través de las gráficas, que nos muestran que los tiempos obtenidos por las máquinas son acotados por una función cuadrática.

Requerimiento 8

```
def req_8(data_structs,n,anio_inicial,anio_final):
    """
    Función que soluciona el requerimiento 8
    """
    # TODO: Realizar el requerimiento 8

    lista_cod_disjuntos = organizar_impuesto_cargo(data_structs,anio_inicial,anio_final)
    lista_cod_unidos = lt.newList(datastructure="ARRAY_LIST")
    merg.sort(lista_cod_disjuntos,cmp_by_cod)

    num = int(n)

    anio1 = int(anio_inicial)
    anio2 = int(anio_final)

    for elemento in lt.iterator(lista_cod_disjuntos):

        if lt.size(lista_cod_unidos) <1:
            lt.addLast(lista_cod_unidos,elemento)

        else:
            i = lt.size(lista_cod_unidos)
            if elemento["Código sector económico"] == lt.getElement(lista_cod_unidos,i)["Código sector económico"]:
                costos_gastos = elemento["Total costos y gastos del subsector"] + int(lt.getElement(lista_cod_unidos,i)["Total costos y gas
                ingresos = elemento["Total ingresos netos del subsector economico"] + int(lt.getElement(lista_cod_unidos,i)["Total ingresos
                saldo_pagar = elemento["Total saldo a pagar del subsector"] + int(lt.getElement(lista_cod_unidos,i)["Total saldo a pagar de
                saldo_favor = elemento["Total saldo a favor del subsector"] + int(lt.getElement(lista_cod_unidos,i)["Total saldo a favor de
                impuestos = elemento["Total de impuestos a cargo para el subsector"] + int(lt.getElement(lista_cod_unidos,i)["Total de impu
                dic = {'Código sector económico': elemento["Código sector económico"], 'Nombre sector económico': elemento["Nombre sector e
                       'Código subsector económico': elemento["Código subsector económico"], 'Nombre subsector económico': elemento["Nomb
                       'Total ingresos netos del subsector economico': ingresos,
                       'Total costos y gastos del subsector': costos_gastos, 'Total saldo a pagar del subsector': saldo_pagar,
                       'Total saldo a favor del subsector': saldo_favor, 'Total de impuestos a cargo para el subsector': impuestos}
                lt.changeInfo(lista_cod_unidos,i,dic)
            else:
                lt.addLast(lista_cod_unidos,elemento)

    sa.sort(lista_cod_unidos,cmp_by_letra)
```

```
    for subsector in lt.iterator(lista_cod_unidos):
        cod = subsector["Código subsector económico"]
        tabla_grande = []
        table = []
        lista = lt.newList(datastructure="ARRAY_LIST")
        for elemento in lt.iterator(data_structs["data"]):
            if elemento["Código subsector económico"] == cod and int(elemento["Año"]) in range(anio1,anio2+1):
                lt.addLast(lista,elemento)

        if lt.size(lista) > 1:
            merg.sort(lista,cmp_impuestos_by_nomina_act)

        if lt.size(lista) <= num:
            for elemento in lista["elements"]:
                table = [elemento["Código actividad económica"], elemento["Nombre actividad económica"],
                         elemento["Total Impuesto a cargo"],elemento["Total ingresos netos"],
                         elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
                tabla_grande.append(table)


        elif lt.size(lista)>= num:

            top = lt.subList(lista,1,num)

            for elemento in top["elements"]:
                table = [elemento["Código actividad económica"], elemento["Nombre actividad económica"],
                         elemento["Total Impuesto a cargo"],elemento["Total ingresos netos"],
                         elemento["Total saldo a pagar"],elemento["Total saldo a favor"]]
                tabla_grande.append(table)

        print("La(s)",len(tabla_grande),"actividades economicas que más y menos aportaron en el subsector " + cod + " son: ")

        print(tabulate(tabla_grande,headers,tablefmt="grid",maxcolwidths=14, maxheadercolwidths=10,numalign="right"), "\n")



    return lista_cod_unidos
```

Descripción

Entrada	Estructura de datos, año inicial y año final

Salidas	Array list con los subsectores con el mayor total de impuestos a cargo entre los años indicados por parámetro.
Implementado (Sí/No)	Si. Implementado por Sara Leiva.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Llamar a una función.	O(1)
Crear un array list.	O(1)
Organizar una lista con Merge Sort.	O(nlogn)
Establecer los parámetros de entrada como enteros.	O(1)
Recorrer los elementos de la lista.	
Sí el tamaño de la lista es menor a uno, añadir todos los elementos de la lista a la nueva lista creada.	O(1)
Establecer una variable i como el tamaño de la lista.	O(1)
Si el tamaño de la lista es mayor a uno, comparar los elementos que están dentro de la lista. Si hay dos elementos con el mismo código de sector económico, sumar sus valores para obtener sólo un elemento que represente en totalidad sus valores.	O(n)
Cambiar el elemento que se encontraba en la lista por el recién creado.	O(1)
Si el código del elemento no está en la lista, añadirlo al final de la lista.	O(1)
Organizar la lista por Merge Sort.	O(nlogn)
Recorrer los elementos de la lista recién organizada “lista_cod_unidos”.	O(n)
Asignar variables, crear listas y crear un array list “lista”.	O(1)
Recorrer los elementos de la lista recibida por parámetro (la lista con toda la información obtenida por la carga de datos).	O(n)
Sí el elemento de la lista con toda la información coincide con algún código que se encuentre en los elementos de “lista_cod_unidos”, añadir el elemento a “lista”.	O(1)
Si el tamaño de “lista” es mayor a uno, organizar la lista con Merge Sort.	O(nlogn)
Si el tamaño de la “lista” es menor al número ingresado por parámetro, añadir todos los elementos de la “lista” a una lista.	O(1)
Añadir esa lista a otra lista.	O(1)

Si el tamaño de la “lista” es mayor al número ingresado por parámetro, crear una sublista de la “lista” con num elementos.	O(num)
Recorrer los elementos de la sublista y guardarlos en una lista.	O(num)
Agregar esta lista a otra lista.	O(1)
TOTAL	**O(n^4)**

Pruebas Realizadas

Procesadores	**Intel® Core™ i5-9300H CPU @2.40GHz**
Memoria RAM	8.0 GB
Sistema Operativo	Windows 10 Home Single – 64 bits

Entrada	**Tiempo (ms)**
small	75.74
5 pct	104.47
10 pct	133.8
20 pct	142.71
30 pct	148.61
50 pct	167.35
80 pct	199.94
large	216.56

Procesadores	**Intel® Core™ i7-1165G7 @2.80GHz**
Memoria RAM	15.6 GB
Sistema Operativo	Windows 11 Pro – 64 bits

Entrada	**Tiempo (ms)**
small	28.83
5 pct	44.17
10 pct	45.29
20 pct	52.81
30 pct	55.71
50 pct	57.8
80 pct	83.66
large	90.16

Tablas de datos

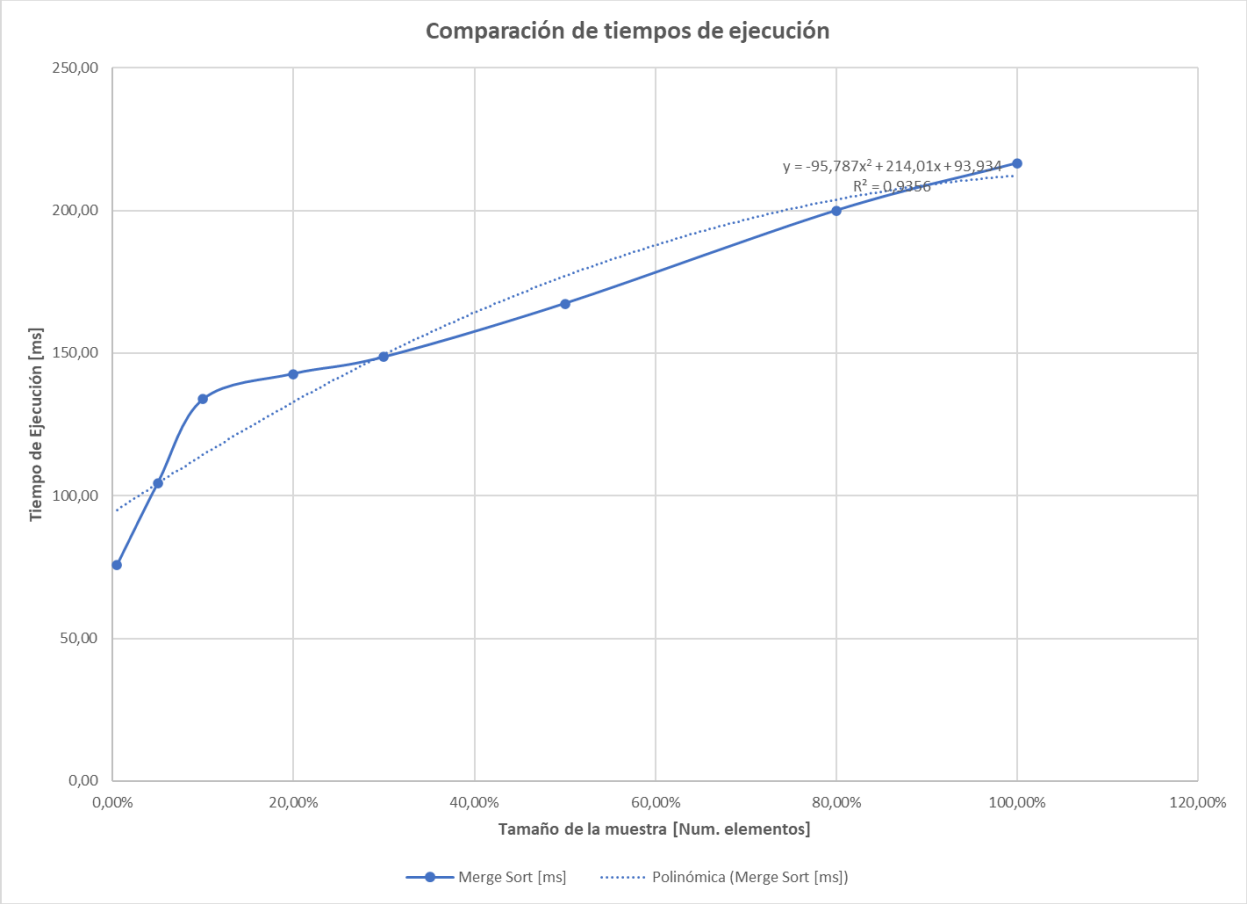
Las tablas con la recopilación de datos promedio de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Las actividades económicas de cada subsector con los mayores totales de impuestos a cargo en un periodo de tiempo dado	28.83
5 pct	Las actividades económicas de cada subsector con los mayores totales de impuestos a cargo en un periodo de tiempo dado	44.17
10 pct	Las actividades económicas de cada subsector con los mayores totales de impuestos a cargo en un periodo de tiempo dado	45.29
20 pct	Las actividades económicas de cada subsector con los mayores totales de impuestos a cargo en un periodo de tiempo dad	52.81
30 pct	Las actividades económicas de cada subsector con los mayores totales de impuestos a cargo en un periodo de tiempo dado	55.71
50 pct	Las actividades económicas de cada subsector con los mayores totales de impuestos a cargo en un periodo de tiempo dad	57.8
80 pct	Las actividades económicas de cada subsector con los mayores totales de impuestos a cargo en un periodo de tiempo dado	83.66
large	Las actividades económicas de cada subsector con los mayores totales de impuestos a cargo en un periodo de tiempo dado	90.16

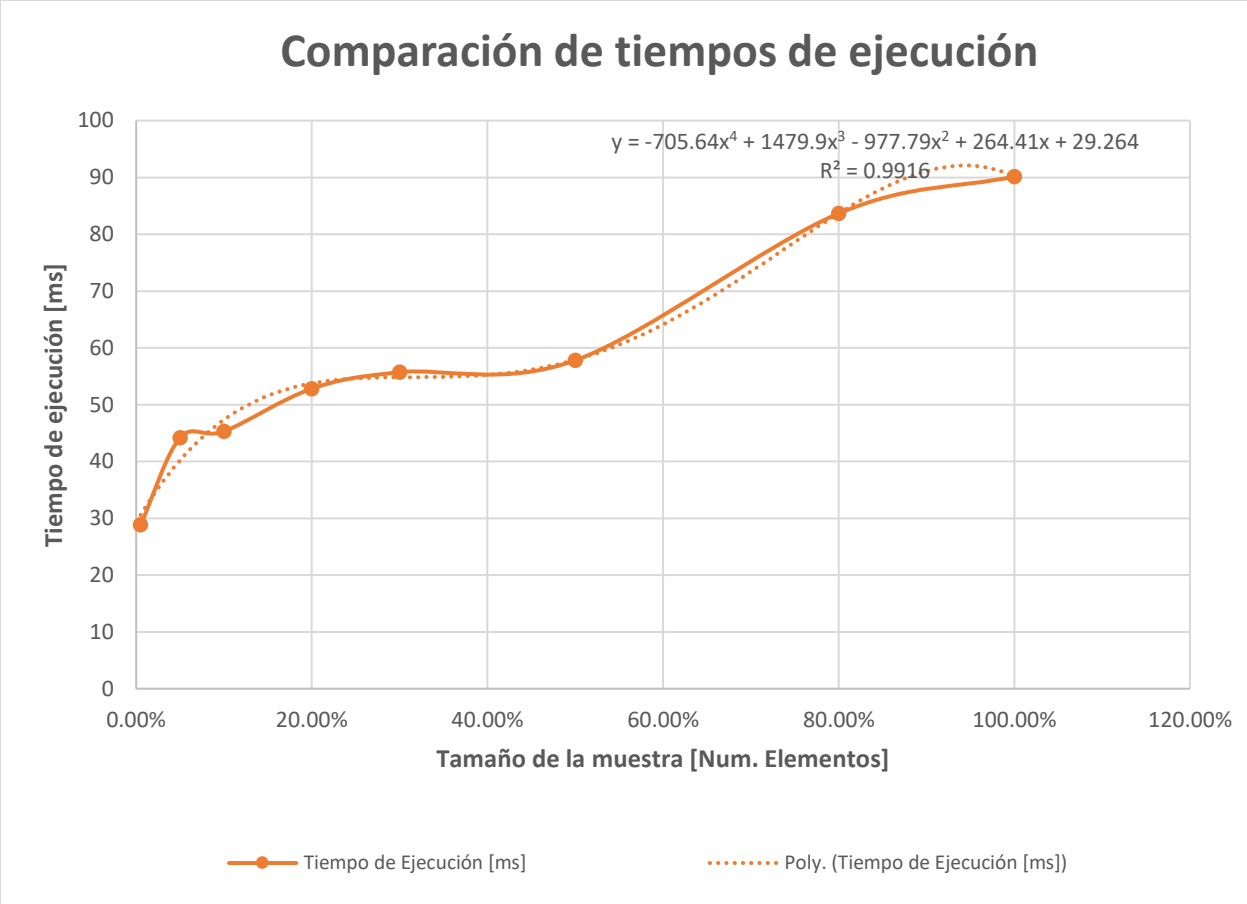
Graficas

Las gráficas con la representación de las pruebas realizadas.

- Máquina 1:


Comparación de tiempos de ejecución
250,00
200,00
150,00
100,00
50,00
0,00
Tiempo de Ejecución [ms]
y = -95,787x² + 214,01x + 93,934
R² = 0,9356
0,00%
20,00%
40,00%
60,00%
80,00%
100,00%
120,00%
Tamaño de la muestra [Num. elementos]
Merge Sort [ms]
Polinómica (Merge Sort [ms])

- Máquina 2:



Análisis

El anterior código tiene una complejidad alta, gracias a que está acotado por O(N^4). Por lo tanto, se vuelve un código lento e ineficaz. No obstante, tiene la capacidad de mostrar la información pedida y no presenta errores en su ejecución.