Universidad de los Andes
Facultad de Ingeniería

Departamento de Ingeniería de Sistemas y Computación
Estructuras de Datos y Algoritmos
ISIS-1225

ABET Engineering Accreditation Commission

ANÁLISIS DEL RETO

Alejandro Narváez Arias, 202123110, a.narvaeza@uniandes.edu.co

franklin hernandez, 202214547, f.hernandezg@uniandes.edu.co

Requerimiento 1

Descripción

```
def req_1(data_structs, anio, codigo_sector):
    """
    Función que soluciona el requerimiento 1
    """
    tamanio = data_size(data_structs)
    anios = crear_diccionario(data_structs, "data","Año", tamanio)
    datos_anio = anios[anio]
    lista_dicts = datos_anio["elements"]
    mayor = lt.newList(datastructure="ARRAY_LIST")
    alto = 0
    for dict in lista_dicts:
        if dict["Código sector económico"] == codigo_sector:
            if int(dict["Total saldo a pagar"]) > alto:
                if lt.isEmpty(mayor) == True:
                    lt.addFirst(mayor, dict)
                else:
                    lt.deleteElement(mayor, 0)
                    lt.addFirst(mayor, dict)

    return (mayor)
```

Este requerimiento se encarga de retornar un dato de una lista dado su ID. Lo primero que hace es verificar si el elemento existe. Dado el caso que exista, retorna su posición, lo busca en la lista y lo retorna. De lo contrario, retorna None.

Entrada	Estructuras de datos del modelo, ID.
Salidas	actividad económica con mayor saldo a pagar para un sector económico
Implementado (Sí/No)	Si. Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Obtener el tamaño de la lista de datos mediante la función data size	O(1)

Crear un diccionario de listas que contenga los datos agrupados por año mediante la función crear_diccionario Obtener la lista de diccionarios correspondiente al año especificado en el parámetro anio del requerimiento mediante la variable datos anio Recorrer la lista de diccionarios y encontrar el mayor saldo a pagar que corresponda al código de sector económico especificado en el parámetro código sector	O(n) O(1) O(n)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el ID 1.

Procesadores	**apple chip m2**
Memoria RAM	8 GB
Sistema Operativo	macOS Monterey

Entrada	**Tiempo (ms)**
small	0.64
5 pct	1.03
10 pct	2.13
20 pct	4.99
30 pct	6.22
50 pct	8.63
large	15.16

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

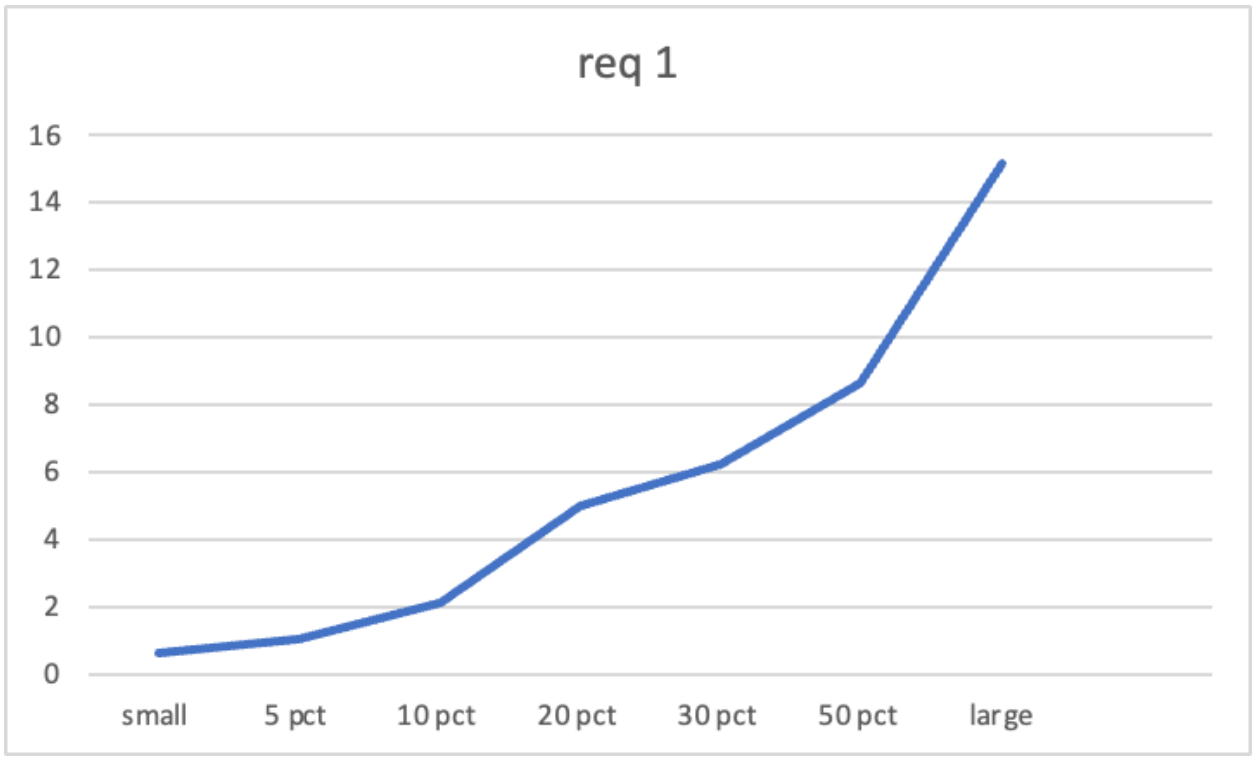
input:2015, sub 3

Muestra	**Salida**	**Tiempo (ms)**
small	1	0.64
5 pct	1	1.03
10 pct	1	2.13
20 pct	1	4.99

30 pct	1	6.22
50 pct	1	8.63
large	1	15.16

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

La obtención del tamaño de la lista y la creación del diccionario tienen una complejidad lineal O(n), la implementación del requerimiento 1 también tiene una complejidad lineal O(n). Esto se debe a que se recorre la lista de datos una vez por cada año, lo que puede resultar en un recorrido de toda la lista en el peor de los casos. Además, cada año se realiza una búsqueda lineal en la lista de elementos correspondiente, lo que aumenta aún más la complejidad.

Este comportamiento se puede observar en la gráfica experimental, que muestra una curva que coincide con el comportamiento lineal esperado. Por lo tanto, se podría decir que la implementación del requerimiento 1 tiene una complejidad lineal O(n), lo que puede afectar su desempeño para conjuntos de datos muy grandes.

Requerimiento 2

Descripción

```python
def req_2(data_structs, anio, codigo_sector):
    """
    Función que soluciona el requerimiento 2
    """
    tamanio = data_size(data_structs)
    anios = crear_diccionario(data_structs, "data","Año", tamanio)
    datos_anio = anios[anio]
    lista_dicts = datos_anio["elements"]
    mayor = lt.newList(datastructure="ARRAY_LIST")
    alto = 0
    for dict in lista_dicts:
        if dict["Código sector económico"] == codigo_sector:
            if int(dict["Total saldo a favor"]) > alto:
                if lt.isEmpty(mayor) == True:
                    lt.addFirst(mayor, dict)
                else:
                    lt.deleteElement(mayor, 0)
                    lt.addFirst(mayor, dict)

    return (mayor)
```

Este requerimiento se encarga de retornar un dato de una lista dado su ID. Lo primero que hace es verificar si el elemento existe. Dado el caso que exista, retorna su posición, lo busca en la lista y lo retorna. De lo contrario, retorna None.

Entrada	Estructuras de datos del modelo, ID.
Salidas	actividad económica con mayor saldo a favor
Implementado (Sí/No)	Si. Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Obtener el tamaño de la lista de datos mediante la función data size	O(1)
Crear un diccionario de listas que contenga los datos agrupados por año mediante la función crear_diccionario. Obtener la lista de diccionarios correspondiente al año especificado en el parámetro anio del requerimiento mediante la variable datos anio.	O(n) O(1)

Recorrer la lista de diccionarios y encontrar el valor más alto de la categoría "Total saldo a favor" que corresponda al código de sector económico especificado en el parámetro código sector .	O(n)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el ID 1.

Procesadores	**apple chip m2**
Memoria RAM	8 GB
Sistema Operativo	macOS Monterey

Entrada	Tiempo (ms)
small	0.62
5 pct	1.19
10 pct	2.91
20 pct	4.06
30 pct	5.27
50 pct	8.01
large	14.40

Tablas de datos

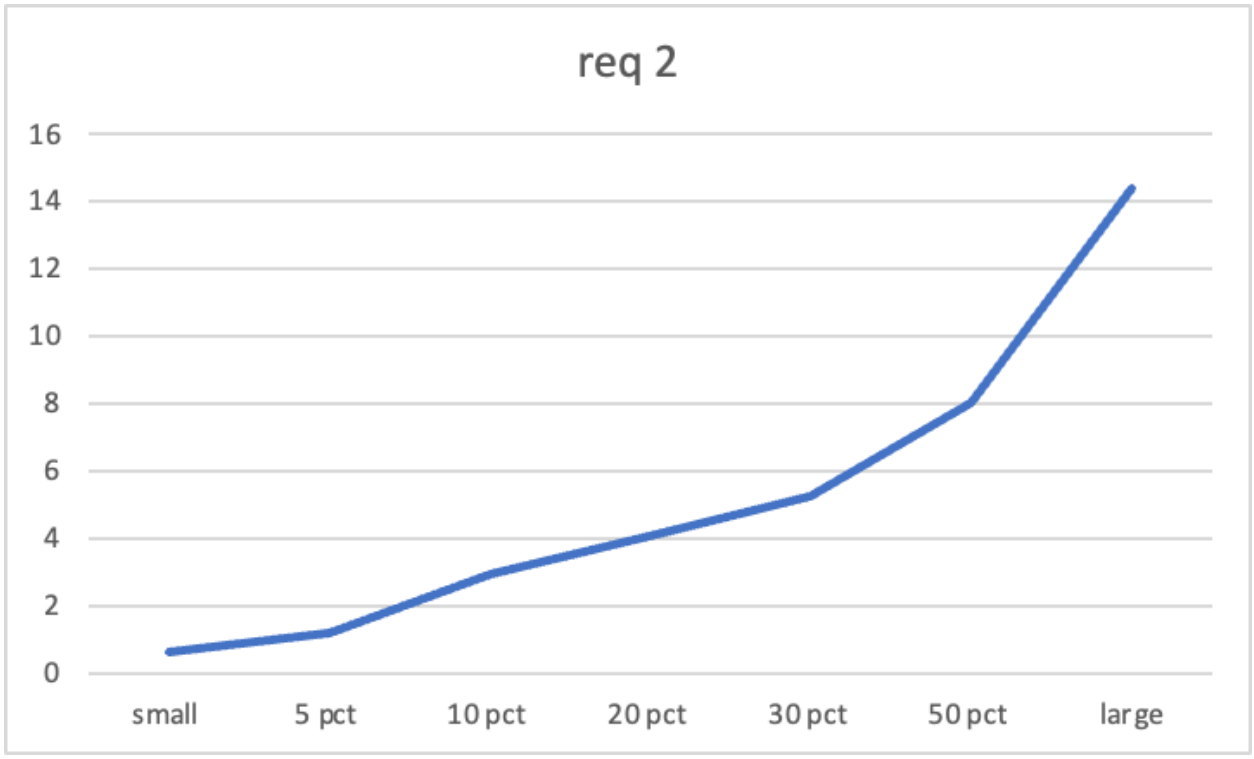
Las tablas con la recopilación de datos de las pruebas.

input:2012, sub 3

Muestra	Salida	Tiempo (ms)
small	Dato1	0.62
5 pct	Dato2	1.19
10 pct	Dato3	2.91
20 pct	Dato4	4.06
30 pct	Dato5	5.27
50 pct	Dato6	8.01
large	Dato8	14.40

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

la implementación del requerimiento 2 tiene una complejidad temporal de orden lineal O(n), debido a que se recorre toda la lista de elementos para buscar aquellos que correspondan al sector económico y se compara el valor de "Total saldo a favor" para determinar si es mayor al valor más alto encontrado hasta el momento. De esta manera, en el peor de los casos, se tendría que recorrer toda la lista de elementos, lo que implica una complejidad lineal.

Esta conclusión se puede verificar gráficamente, donde se observa una tendencia lineal en el comportamiento de la función de tiempo de ejecución en función del tamaño de los datos de entrada.

Requerimiento 3

Descripción

```
def req_3(data_structs, anio):
    """
    #Función que soluciona el requerimiento 3
    """
    tamanio = data_size(data_structs)
    anios = crear_diccionario(data_structs, "data","Año", tamanio)
    datos_anio = anios[anio]
    lista_dics = datos_anio["elements"]
    menor_subsector = lt.newList(datastructure="ARRAY_LIST")
    lt.addFirst(menor_subsector, lista_dics[0])
    bajo_subsector = int(lista_dics[0]["Total retenciones"])
    for dict in lista_dics:
        if int(dict["Total saldo a pagar"]) > bajo_subsector:
                lt.deleteElement(menor_subsector, 0)
                lt.addFirst(menor_subsector, dict)
    valores_max = encontrar_mayores_retenciones(lista_dics)
    valores_min = encontrar_menores_retenciones(lista_dics)
    return (menor_subsector, valores_max, valores_min)
```

```
def encontrar_mayores_retenciones(lista_de_diccionarios):
    valores_mayores = lt.newList(datastructure="ARRAY_LIST")
    lt.addFirst(valores_mayores, lista_de_diccionarios[0])

    for diccionario in lista_de_diccionarios:
        tot_retenciones = float(diccionario["Total retenciones"])
        if lt.size(valores_mayores) < 2:
            mayor = lt.firstElement(valores_mayores)
            if mayor["Total retenciones"] < diccionario["Total retenciones"]:
                lt.addFirst(valores_mayores, diccionario)
            else:
                lt.addLast(valores_mayores, diccionario)

        elif lt.size(valores_mayores) < 3:
            dic_0 = lt.getElement(valores_mayores, 0)
            dic_1 = lt.getElement(valores_mayores, 1)
            if tot_retenciones > float(dic_0["Total retenciones"]):
                lt.insertElement(valores_mayores, diccionario, 0)
            elif tot_retenciones > float(dic_1["Total retenciones"]):
                lt.insertElement(valores_mayores, diccionario, 1)

        else:
            dic_0 = lt.getElement(valores_mayores, 0)
            dic_1 = lt.getElement(valores_mayores, 1)
            dic_2 = lt.getElement(valores_mayores, 2)
            if tot_retenciones > float(dic_0["Total retenciones"]):
                lt.insertElement(valores_mayores, diccionario, 0)
                lt.deleteElement(valores_mayores, 2)
            elif tot_retenciones > float(dic_1["Total retenciones"]):
                lt.insertElement(valores_mayores, diccionario, 1)
                lt.deleteElement(valores_mayores, 2)
            elif tot_retenciones > float(dic_2["Total retenciones"]):
                lt.insertElement(valores_mayores, diccionario, 2)
                lt.deleteElement(valores_mayores, 2)
    return valores_mayores
```

```python
def encontrar_menores_retenciones(lista_de_diccionarios):
    valores_menores = lt.newList(datastructure="ARRAY_LIST")
    lt.addFirst(valores_menores, lista_de_diccionarios[0])

    for diccionario in lista_de_diccionarios:
        tot_retenciones = float(diccionario["Total retenciones"])
        if lt.size(valores_menores) < 2:
            menor = lt.firstElement(valores_menores)
            if menor["Total retenciones"] < diccionario["Total retenciones"]:
                lt.addFirst(valores_menores, diccionario)
            else:
                lt.addLast(valores_menores, diccionario)

        elif lt.size(valores_menores) < 3:
            dic_0 = lt.getElement(valores_menores, 0)
            dic_1 = lt.getElement(valores_menores, 1)
            if tot_retenciones > float(dic_0["Total retenciones"]):
                lt.insertElement(valores_menores, diccionario, 0)
            elif tot_retenciones > float(dic_1["Total retenciones"]):
                lt.insertElement(valores_menores, diccionario, 1)

        else:
            dic_0 = lt.getElement(valores_menores, 0)
            dic_1 = lt.getElement(valores_menores, 1)
            dic_2 = lt.getElement(valores_menores, 2)
            if tot_retenciones > float(dic_0["Total retenciones"]):
                lt.insertElement(valores_menores, diccionario, 0)
                lt.deleteElement(valores_menores, 2)
            elif tot_retenciones > float(dic_1["Total retenciones"]):
                lt.insertElement(valores_menores, diccionario, 1)
                lt.deleteElement(valores_menores, 2)
            elif tot_retenciones > float(dic_2["Total retenciones"]):
                lt.insertElement(valores_menores, diccionario, 2)
                lt.deleteElement(valores_menores, 2)
    return valores_menores
```

Este requerimiento se encarga de retornar el subsector economico con menor total de retenciones y las actividades con más y menos retenciones.

Entrada	Estructuras de datos del modelo, Año
Salidas	Subsector económico con el menor total de retenciones para un año específico
Implementado (Sí/No)	Si/ Estudiante 1

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Asignaciones	9

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(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	Tiempo (s)
1%	1.1505999999353662
5%	1.8248000000603497
10%	2.0304999999934807
20%	2.6656000000657514
30%	3.1420999999390915
50%	8.18840000021737
100%	15.060200000065379

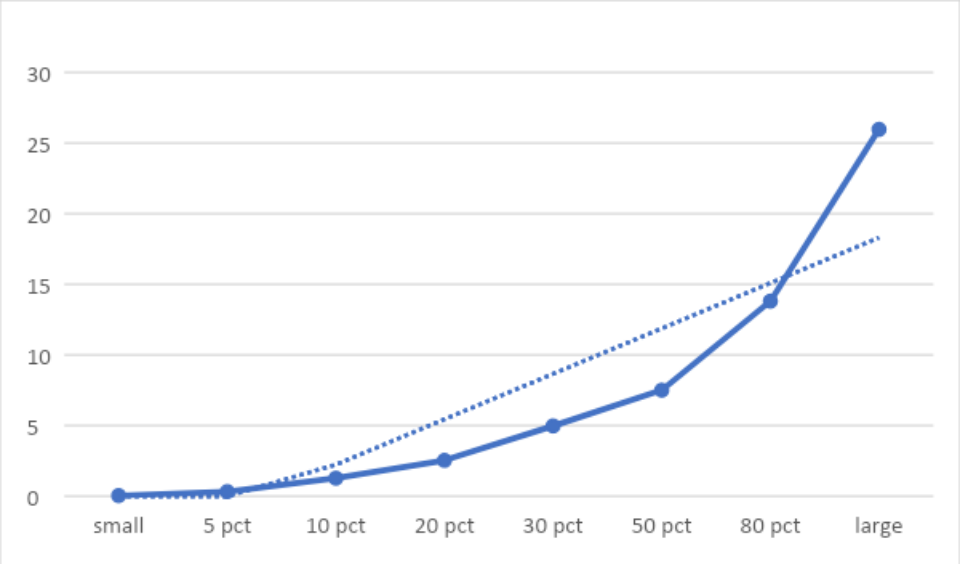
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	1	1.1505999999353662
5 pct	1	1.8248000000603497
10 pct	1	2.0304999999934807
20 pct	1	2.6656000000657514
30 pct	1	3.1420999999390915
50 pct	1	8.18840000021737
large	1	15.060200000065379

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

A pesar de que obtener un elemento en un *ArrayList,* dada su posición, tiene complejidad constante, la implementación de este requerimiento tiene un orden lineal O(n). Esto debido a que, lo primero que se hace es verificar si el elemento hace parte de la lista. Específicamente, a la hora de buscar un elemento en una lista, en el peor de los casos es necesario recorrer toda la lista, es decir, complejidad lineal.

Este comportamiento se puede evidenciar experimentalmente en la gráfica. Ya que, gracias a que los datos no se encuentran tan dispersos con respecto a la línea de tendencia, la curva coincide con el comportamiento lineal esperado.

Requerimiento 4

Descripción

encontrar el subsector con los mayores costos y gastos de un año y llama una función auxiliar que ordena la información por subsector mientras hace la suma de costos y gastos nominas del subsector

Entrada	Estructuras de datos del modelo, ID.
Salidas	actividad económica con mayor saldo a favor
Implementado (Sí/No)	Si. franklin

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad

Paso 1: Crea la lista nombres donde para guardar la información del subsector mayor y la lista filas_respuesta donde para guardar la información de las 3 mejores y peores actividades económica de cada subsector	O (1)

Paso 3: Se llama la función auxiliar ordenar_por_columna que organiza los datos por subsector y guarda la suma costos y gastos nómina del subsector.	O(N)
Paso 4: Se llama la función auxiliar buscar_mayor_suma que encuentra y retorna la mayor suma de costos y gastos nomina entre los subsectores disponibles.	O(N)
Paso 5: Crea el diccionario donde guarda la información correspondiente (información para mostrar después al usuario)	O(1)
Paso 6: Recorre las filas que pertenecen al subsector mayor para hacer las sumas de Total ingresos netos, Total costos y gastos, Total sado a pagar y Total saldo a favor para el subsector	O(m) con m<N
Paso 7: Si las filas del subsector son menores a 6 llama la función auxiliar encontrar_menores_dic que retorna las primeras n actividades económicas ordenadas de menor a mayor. Si hay al menos 6 actividades llama la función encontrar_menores_dic y	O(k^2) con k<7

encontrar_mayores_dic (que ordenan como selection sort) que retorna 3 actividades económicas menores/mayores.	

Paso 8: Finalmente se agregan las filas con la O(1) información a la segunda lista creada.
TOTAL O(N)

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el ID 1.

Procesadores	**apple chip m2**
Memoria RAM	8 GB
Sistema Operativo	macOS Monterey

Entrada	Tiempo (ms)
small	0.87
5 pct	1.670
10 pct	2.89
20 pct	4.91
30 pct	6.57
50 pct	9.91
large	18.88

Tablas de datos

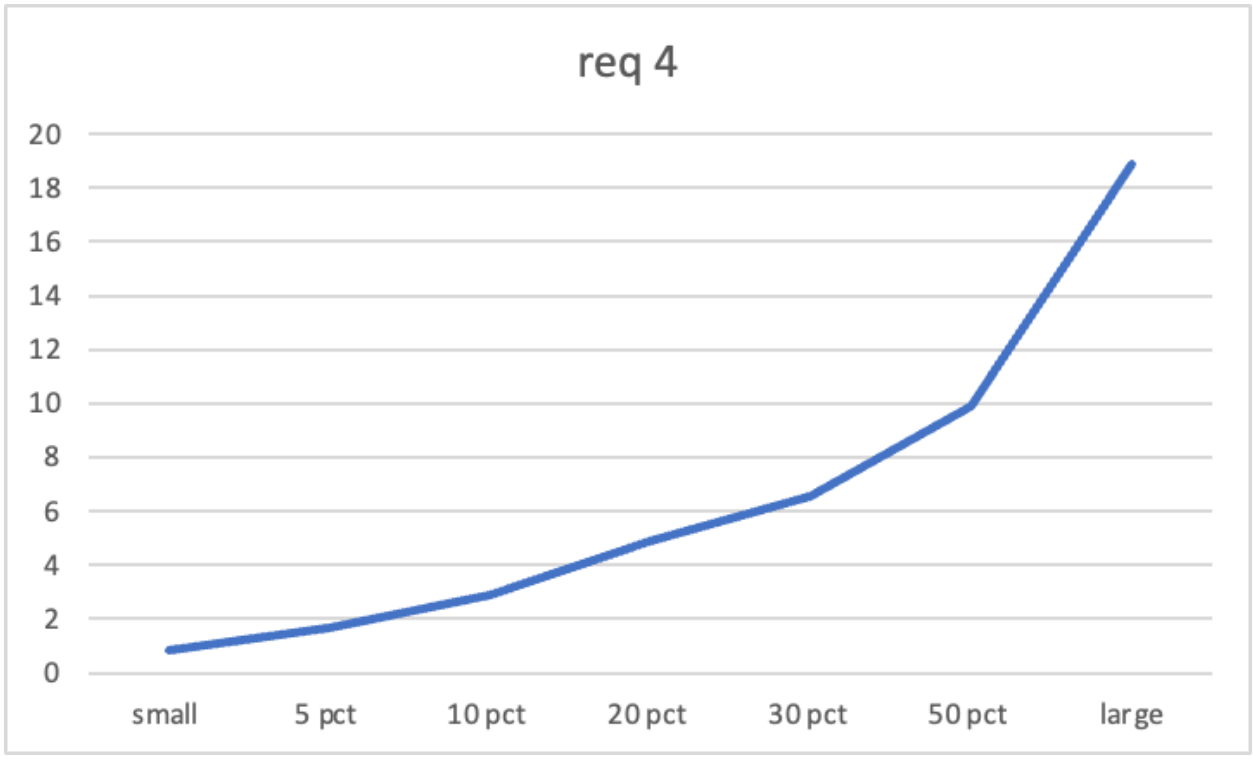
Las tablas con la recopilación de datos de las pruebas.

input:2012

Muestra	Salida	Tiempo (ms)
small	Dato1	0.87
5 pct	Dato2	1.670
10 pct	Dato3	2.89
20 pct	Dato4	4.91
30 pct	Dato5	6.57
50 pct	Dato6	9.91
large	Dato8	18.88

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

re recorre una vez el archivo CSV y se realizan algunas operaciones auxiliares como la ordenación de datos y la búsqueda de la mayor suma de costos y gastos de nómina.