ANÁLISIS DEL RETO

Felipe Gutiérrez Apráez Cod 202220848, f.gutierreza@uniandes.edu.co
Jacobo Morales Erazo Cod 202321072, j.morales1123@uniandes.edu.co
Pablo Sarmiento Tamayo Cod 202321369, p.sarmientot@uniandes.edu.co

Carga de datos

Plantilla para el documentar y analizar cada uno de los requerimientos.

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Lee cada fila de los CSVs	O(4N)
Crea una mega lista con todas las llaves ([‘jobs’])	O(4N)
Por cada uno de los 4 mapas agrega elementos	O(4N)
TOTAL	***O(12N)***

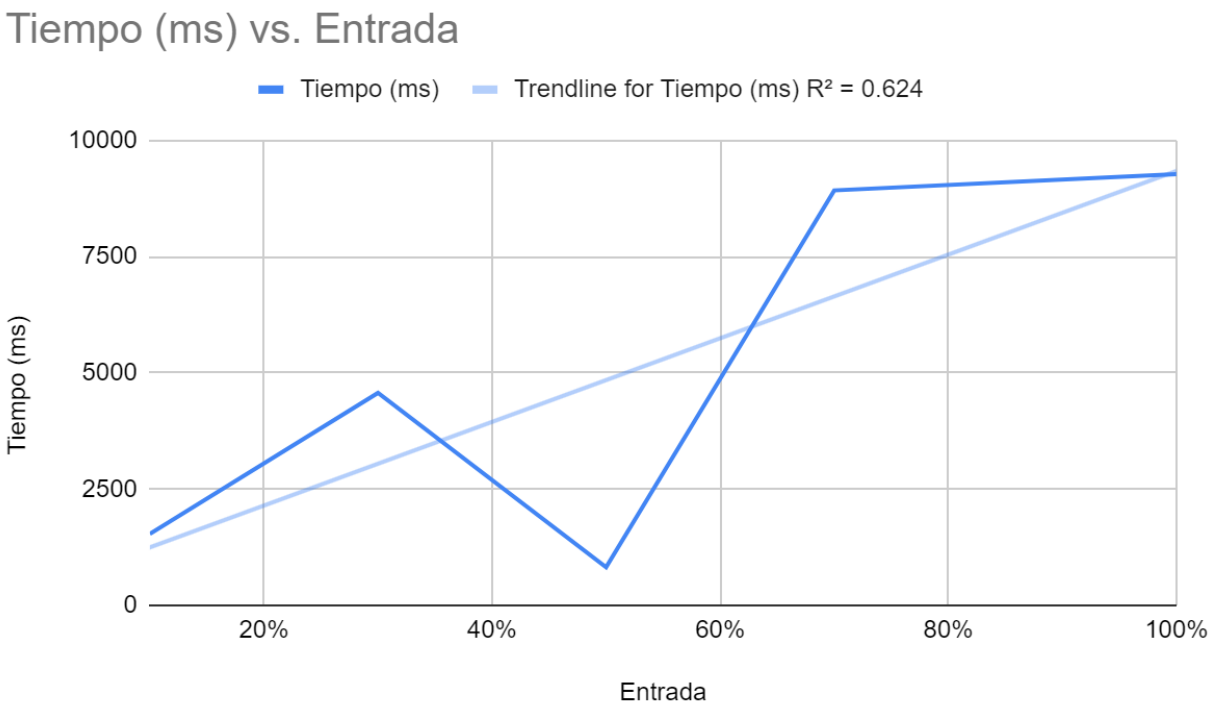
Se decide en una carga lenta que concatene además toda la información a una lista principal con el fin de aprovechar luego eso para rápido acceso.

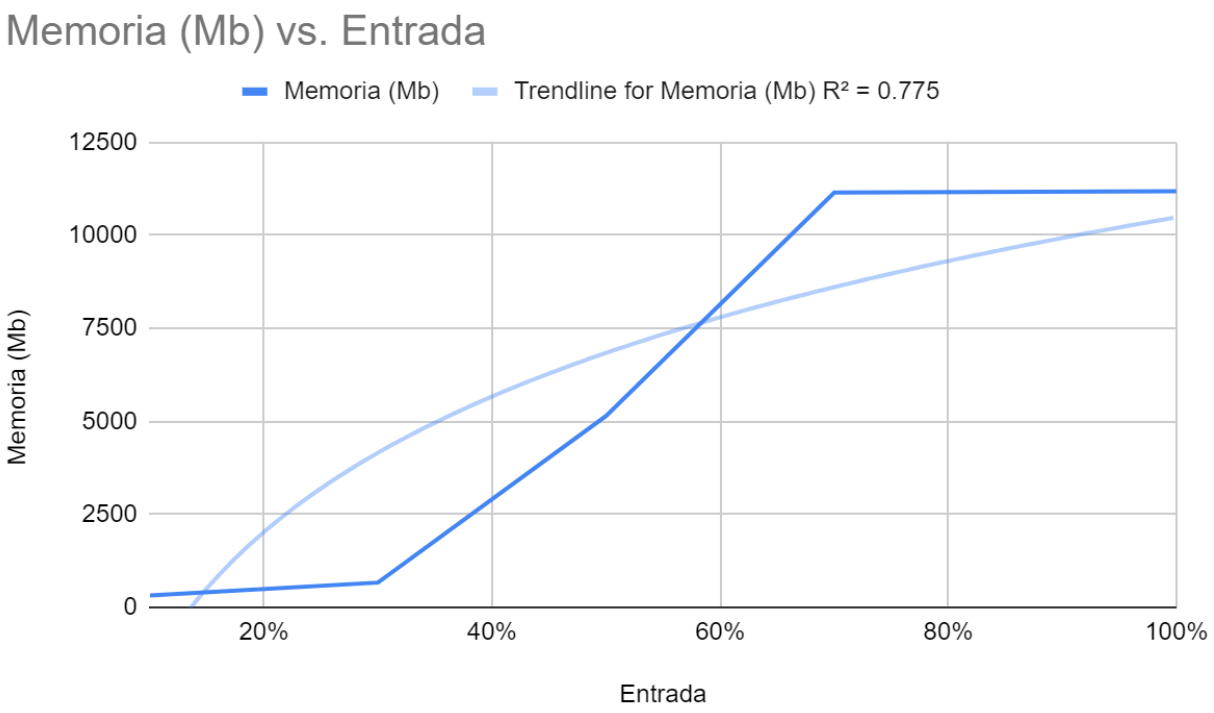
Pruebas Realizadas

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).

Entrada	**Tiempo (ms)**	**Memoria (Mb)**
10%	1537.43	322.38
30%	4568.84	665.57
50%	819.26	5144.16
70%	8923.62	1147.39
100%	9271.81	1188.41

Graficas


Tiempo (ms) vs. Entrada
Tiempo (ms)
Trendline for Tiempo (ms) R² = 0.624
10000
7500
5000
2500
0
Tiempo (ms)
20%
40%
60%
80%
100%
Entrada


Memoria (Mb) vs. Entrada
Memoria (Mb)
Trendline for Memoria (Mb) R² = 0.775
12500
10000
7500
5000
2500
0
Memoria (Mb)
20%
40%
60%
80%
100%
Entrada

Análisis

Cómo se puede ver en las gráficas. Dados los porcentajes de los archivos completos usados para la carga el tiempo suele tender a un incremento lineal entre los diferentes puntos. En cuanto a memoria, sin embargo, la tendencia parece ser logarítmico lo cual tendría sentido en mapas de linear probing con un factor de carga real muy cerca de 0.5 que hacen rehash cada vez que lo necesitan y así van abarcando más memoria para prevenirrse para el siguiente rehash.

Requerimiento 1

Plantilla para el documentar y analizar cada uno de los requerimientos.

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Sí, grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O(...)
Paso 2	O(...)
Paso ….	O(...)
TOTAL	***O(...)***

Pruebas Realizadas

Para medir los tiempos de ejecución y el consumo de ram frente al requerimiento se usaron las librerías de TraceMalloc y Time.

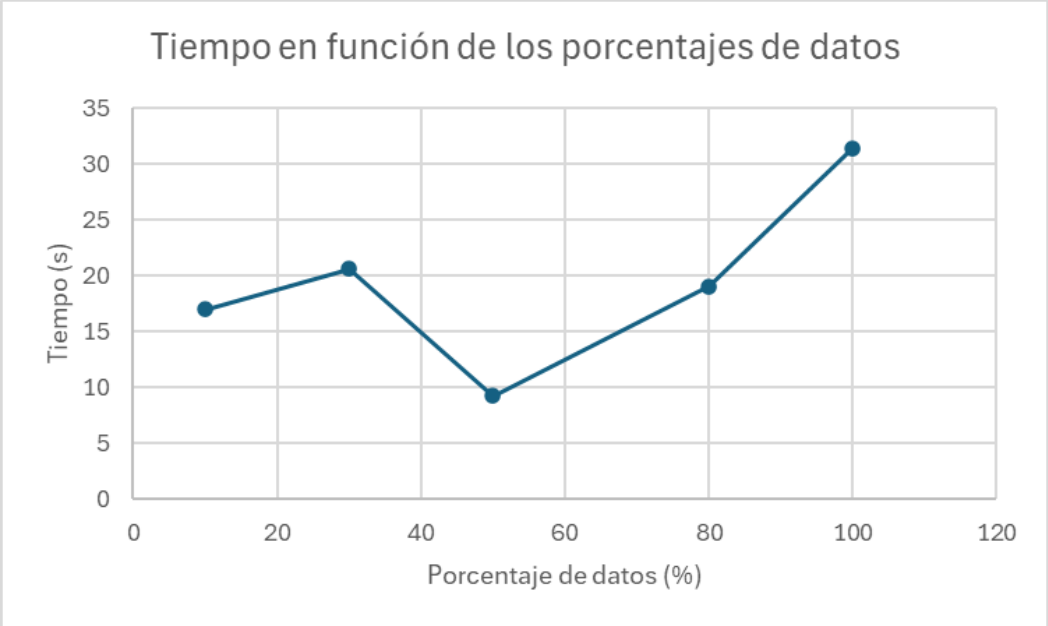
Parametros: PL, junior, 5

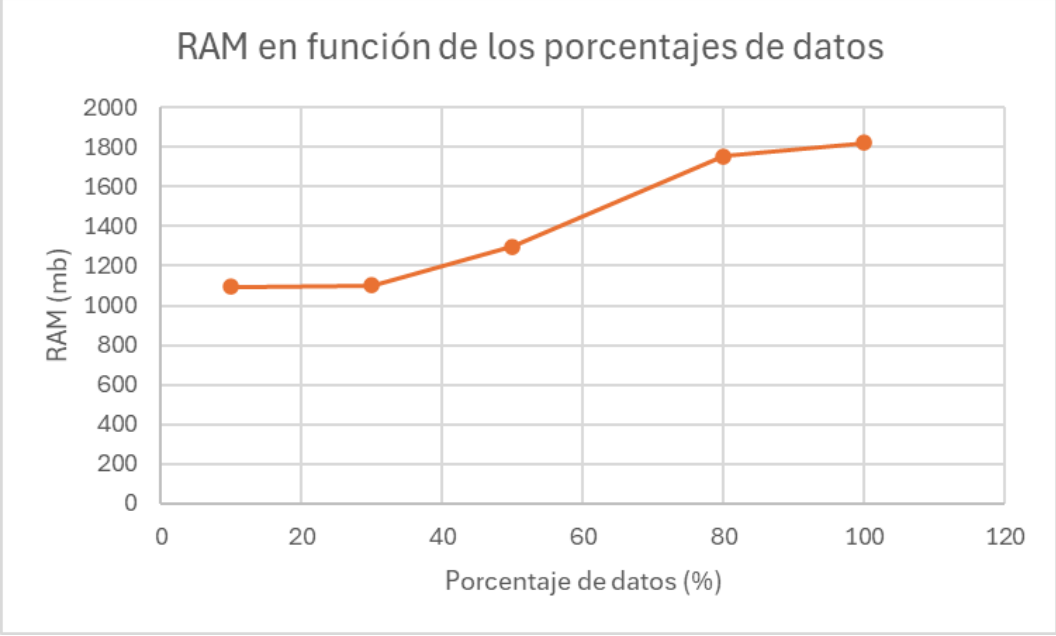
Tablas de datos

Entrada	**Tiempo (ms)**	**Ram (mb)**
Large	31419.796142578125	1823.32
Medium	9256.373291015625	1298.76
Small	20598.783203125	1101.65

10	17026.187255859375	1094.21
80	19063.528076171875	1754.34

Graficas





Análisis

La función **`req_1`** comienza accediendo al catálogo para obtener la lista de IDs asociados al país especificado y al nivel de experiencia requerido. Luego, realiza la intersección de estas listas para obtener las ofertas de trabajo que cumplen con ambos criterios. Posteriormente, extrae la información completa asociada a estos IDs, como el título del trabajo y la fecha de publicación. Además, ordena esta información por fecha de publicación. Finalmente, la función devuelve la información completa de las ofertas ordenadas, junto con el número total de ofertas ofrecidas según el país y el nivel de experiencia, así como el tamaño total de la lista de ofertas resultantes. Este proceso garantiza que se proporcionen de manera eficiente detalles relevantes sobre las ofertas de trabajo que cumplen con los criterios específicos de país y experiencia.

Requerimiento 2

Descripción

```python
def req_2(catalog, nombre_compañia, ciudad):

    #Organizar datos
    moam_c= catalog["model"]["city"]
    moam_e= catalog["model"]["company_name"]

    #Obtener ids de ciudad y compañia especifica
    lista_id_ciudad= mp.get(moam_c, ciudad)
    lista_id_ciudad= lista_id_ciudad["value"]
    lista_id_ciudad= lista_id_ciudad["elements"]
    lista_id_empresa= mp.get(moam_e, nombre_compañia)
    lista_id_empresa= lista_id_empresa["value"]
    lista_id_empresa= lista_id_empresa["elements"]

    #Total de ofertas ofrecidas por empresa y ciudad especifica
    total_ciudades= (len(lista_id_ciudad))
    total_empresas= (len(lista_id_empresa))

    #Encontrar ids comunes entre empresa y ciudad
    lista_ciudad_empresas= adi.getIdentical(lista_id_ciudad, lista_id_empresa)
    lista_ciudad_empresas= lista_ciudad_empresas["elements"]

    #Sacar info de los ids comunes
    ofertas_info= lt.newList("ARRAY_LIST")
    for offer in lista_ciudad_empresas:
        oferta= mp.get(catalog["model"]["jobs"], offer)
        oferta= oferta["value"][offer]
        oferta_info= {"Fecha": oferta["published_at"] ,
                      "Pais": oferta["country_code"] ,
                      "Ciudad":oferta["city"] ,
                      "Titulo_oferta": oferta["title"] ,
                      "Nivel_experiencia": oferta["experience_level"] ,
                      "Formato_aplicación": oferta["contract_type"],
                      "Tipo_trabajo": oferta["workplace_type"]}
        lt.addLast(ofertas_info, oferta_info)

    return total_ciudades, total_empresas, ofertas_info["elements"]
```

El requisito 2 se soluciona mediante listas de IDs en común. Se obtienen los IDs específicos basado en los parámetros pasados por el usuario, se encuentras los IDs entre las dos listas, se obtienen los datos requeridos del común entre las listas.

Entrada	N, ciudad, empresa, control
Salidas	Total, de ofertas ofrecidas por una empresa y una ciudad. Si se usa N, se retornará la cantidad N de ofertas, si no retornara las primeros y últimos 5 elementos
Implementado (Sí/No)	Sí, Todos

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1(Obtener ids)	O(N)
Paso 2 (Total de ofertas)	O(1)
Paso 3(Encontrar ids comunes)	O(K)
Paso 4 (Sacar informacion de las ofertas)	O(M)
TOTAL	***O(N)***

Pruebas Realizadas

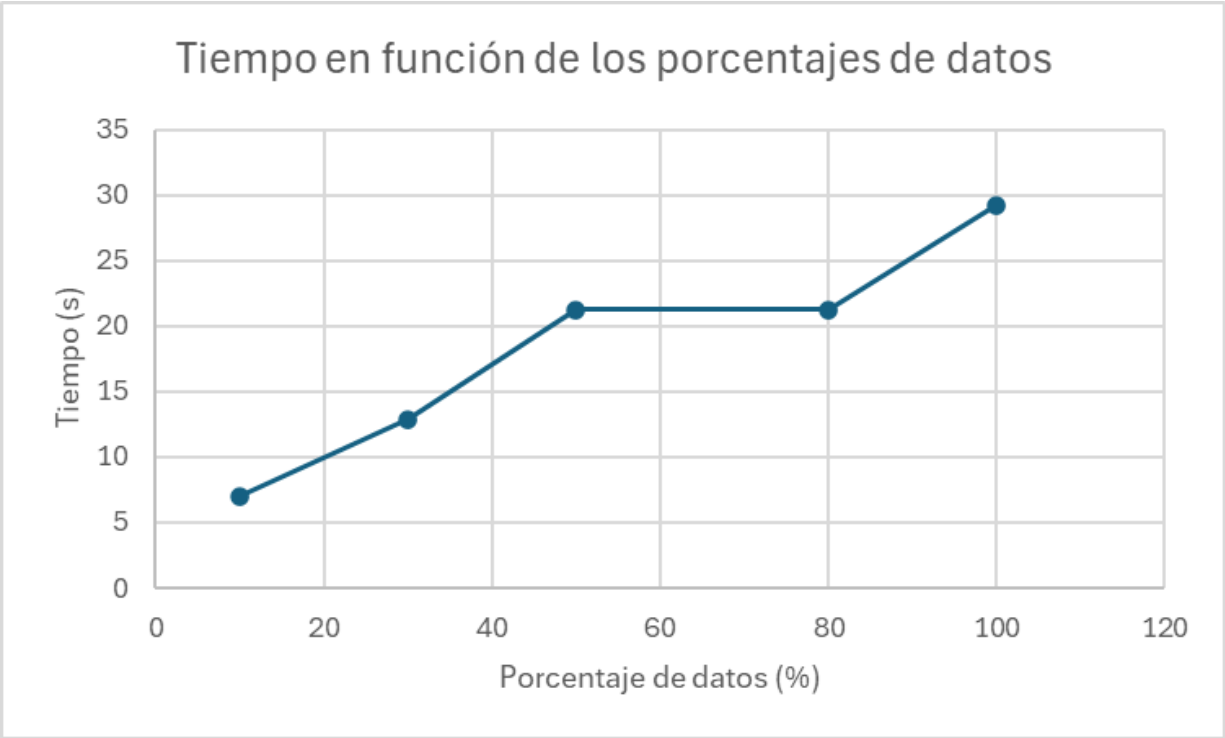
Para medir los tiempos de ejecución y el consumo de ram frente al requerimiento se usaron las librerías de TraceMalloc y Time.

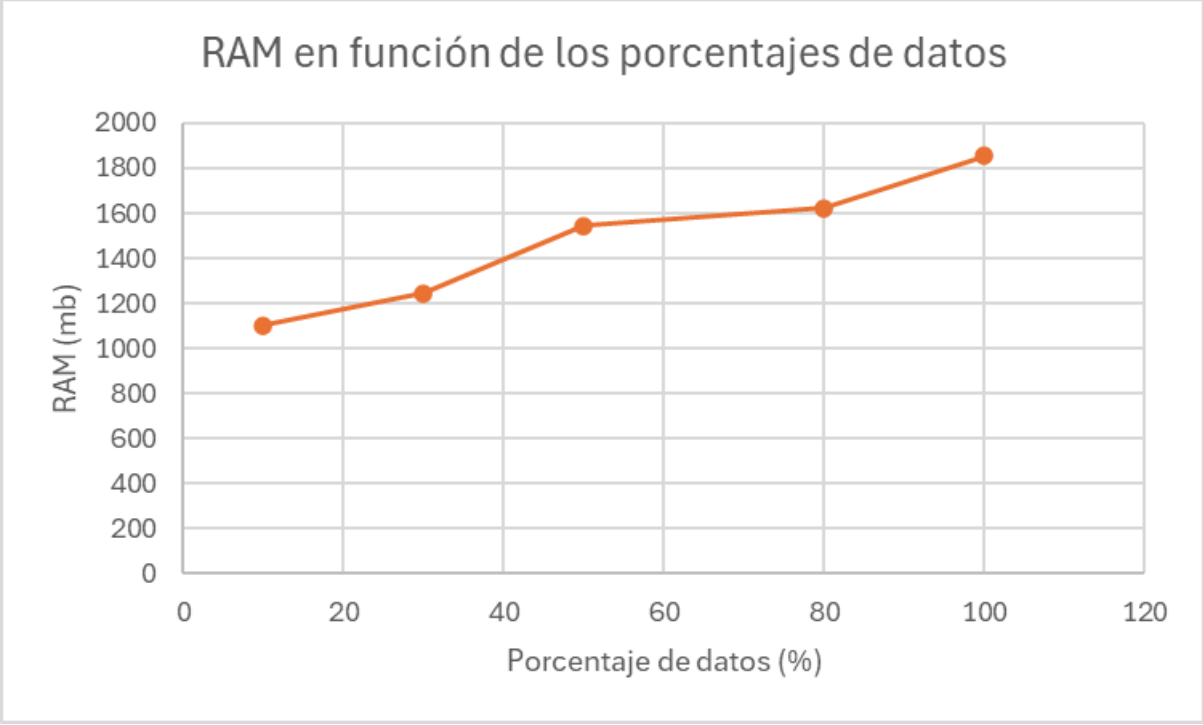
Parametros: 7N, Warszawa, 10

Tablas de datos

Entrada	**Tiempo (ms)**	**Ram (mb)**
Large	29263.19384765625	1854.23
Medium	21233.3798828125	1543.98
Small	12876.839599609375	1243.82
10	7023.10791015625	1103.67
80	21249.072021484375	1623.56

Graficas





Análisis

La función req_2 opera en 4 pasos principales seguidos por un retorno de valore. En primer lugar, se obtienen las listas de identificadores de ciudad y empresa específicos del catálogo, con una posible complejidad de búsqueda de O(n) cada una. Estas listas son importantes para identificar las ofertas relevantes relacionadas con la ciudad y la empresa especificadas. Luego, en el tercer paso, se calcula el

total de ofertas por ciudad y empresa, lo que toma tiempo constante O(1) proporcionando la información sobre la distribución de las ofertas en función de la ciudad y la empresa especifica. Después, en el cuarto paso, se encuentran los identificadores comunes entre empresa y ciudad, cuya complejidad depende de getIdentical, O(K). Este paso es crítico para identificar las ofertas que cumplen con los criterios especificados. Finalmente, se obtiene información detallada de las ofertas comunes, lo que implica un ciclo sobre estas ofertas con una complejidad de O(m), donde m es el tamaño de la lista de ofertas comunes entre la ciudad y la empresa seleccionada. En este paso se obtendrá la información necesaria por oferta. En resumen, la función req_2 tiene una complejidad temporal de O(N) en el peor caso, principalmente debido a la búsqueda en el mapa, mientras que la complejidad espacial dependerá de la cantidad de ofertas y empresas involucradas. Esta función demuestra una forma eficiente de analizar y obtener información detallada sobre las ofertas de trabajo relacionadas con una empresa y una ciudad específicas demostrándolo en sus tiempos de ejecución.

Requerimiento 3

Plantilla para el documentar y analizar cada uno de los requerimientos.

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	catalog, company, startDate, endDate
Salidas	totaldeOffer, noByJunior, noByMid, noBySenior, dsfullInfos
Implementado (Sí/No)	Sí, Felipe Gutiérrezz Apráez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Acceder los diferentes mapas del catalogo	O(0)
Acceder la lista de IDs por byCompany	O(1)
Sacar rango de fechas	O(k*1) Dónde K << N
Sacar IDs correspondientes a esas fechas	O(n) Dónde n << N
Sacar los Ids que cumplen con ambos criterios Compañía ^ Rang	O(n) Dónde n << N
Total,de Ofertas	O(L) Dónde L << n
Acceder a mapa de ofertas por experiencia	O(1)
Número total de ofertas con experticia junior	O(1)
adi.getIdentical(meetsCriteria, byJunior)	O(L) Dónde L << n
Número total de ofertas con experticia mid	O(1)
adi.getIdentical(meetsCriteria, bymid	O(L) Dónde L << n
Número total de ofertas con experticia senior.	O(1)
adi.getIdentical(meetsCriteria, bysenior)	O(L) Dónde L << n
TOTAL	***O(L+n)*** Dónde L + n <<N

Pruebas Realizadas

Se realizan las pruebas para los peores casos. En este caso, sacar la lista de IDs para cualquier empresa es O(1). Sin embargo, la cantidad de retornos de IDs que tenga esta empresa sí impactará el costo de tiempo de procesamiento. Esto es debido a que el método de *adi.getIdentical* identificará la lista más grande de elementos (en este caso 7N tendrá `2073` apariciones lo largo de jobs), luego tendrá que hacer un gasto de "K" elementos para irlos metierndo dentro de un mapa y luego tendrá que recorrer la lista más pequeña para conocer si el elemento está rpesente. Es decir, el costo mayor de procesamiento lo dará la lista más larga de todas las presentes en el algoritmo. Se elige también un año en intervalo de tiempo de

tal manera que la empresa tenga múltiples menciones "n" a lo largo de el año (suponiendo peor caso de 365) y que no sean tan distantes entre sí.

Aquí se puede evidenciar la construcción de el método getIdentical que retorna una lista de IDs similar dadas dos listas de IDs. Esta función hace uso de mapas y de identificar cúal es la lista más costosa de rercorrer con el fin de evitar que llegue a (N^2) en cualquier momento dado garantizando recorridos máximos de O(K) dónde K son los k-elementos de la lista más pequeña de IDs que está gartantizado a ser **mucho menor que N.**

```
def getIdentical(ARRLiA, ARRLiB):
    # Sacamos los tamaños otra vez
    ARRLiA = ile.EnsureADTLi(ARRLiA)
    ARRLiB = ile.EnsureADTLi(ARRLiB)
    sizeA = lt.size(ARRLiA) ## Sacar el tamaño de las ARRAY_LISTS
    sizeB = lt.size(ARRLiB)
    if sizeA > sizeB:
        larger = ARRLiA
        smaller = ARRLiB
    elif sizeB > sizeA:
        larger = ARRLiB
        smaller = ARRLiA
    # Ahora metamos todas las llaves del más grande en un mapa
    largerSize = int(lt.size(larger))
    IdMap = mp.newMap(numelements= (largerSize*(1 + 0.1)),
                      maptype='PROBING',
                      loadfactor=0.5)
    for element in lt.iterator(larger):
        # Dado que son Ids no nos importa casos de repetición (pues no los hay)
        mp.put(IdMap, element, element)
        # Suena estúpido pero no lo es
    common = lt.newList('ARRAY_LIST')
    for element in lt.iterator(smaller):
        if mp.contains(IdMap, element):
            lt.addLast(common, element)
    return common
```

Se exponen a continuación los vectores de prueba realizados.

Parámetros Usados:

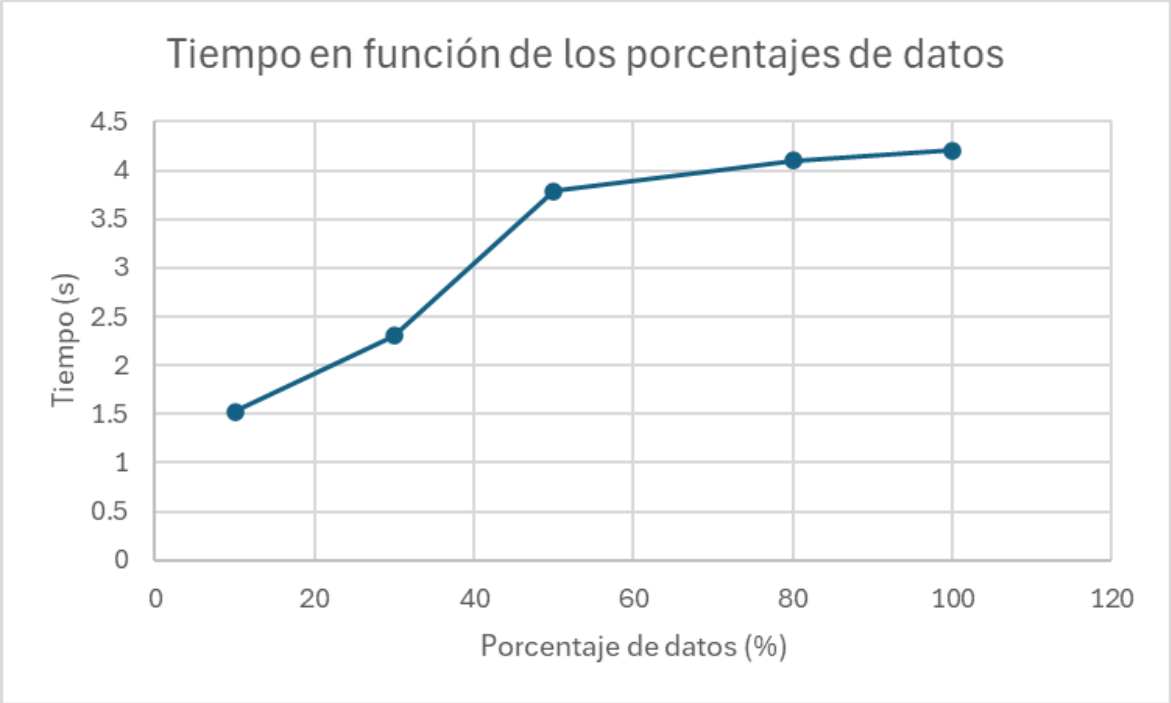
Company: 7N StartDate: 2022-07-07 EndDate: 2023-07-07

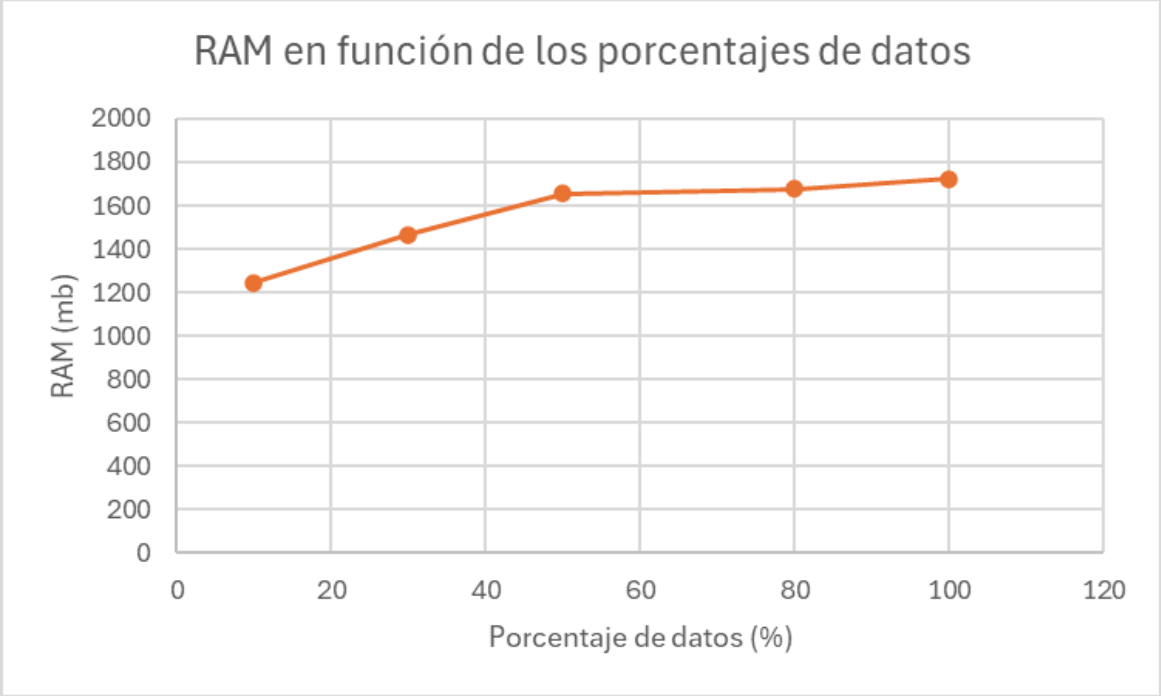
Tablas de datos

Entrada	**Tiempo (ms)**	**Ram (mb)**
Large	4207.04833984375	1723.45
Medium	3791.31396484375	1654.32
Small	2300.675048828125	1465.55
10	1526.119873046875	1245.78
80	4098.92041015625	1676.54

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Requerimiento 4

Descripción

```python
def req_4(control, CODpais, Ffirst, Flast):
    """
    Función que soluciona el requerimiento 4
    """
    # jacome: Realizar el requerimiento 4
    mapcountries=control["country_code"]
    keyscountries=mp.keySet(mapcountries)
    moal2=control["jobs"]
    listaIDsSBCountry=lt.newList("ARRAY_LIST")
    for i in lt.iterator(keyscountries):
        if i == CODpais:
            dato=mp.get(mapcountries,i)
            lt.addLast(listaIDsSBCountry,dato["value"])

    valuesSBCountry=lt.newList(datastructure="ARRAY_LIST")
    for i in lt.iterator(listaIDsSBCountry):
        for j in lt.iterator(i):
            dato = mp.get(moal2,j)
            lt.addLast(valuesSBCountry,dato["value"][j])
    DateSort(valuesSBCountry)
    valuesSBCountryANDDate=lt.newList("ARRAY_LIST")
    for i in lt.iterator(valuesSBCountry):
        fecha=i["published_at"][0:10]
        if Ffirst <= fecha and fecha <= Flast:
            lt.addLast(valuesSBCountryANDDate,i)

    NUMofertas = lt.size(valuesSBCountryANDDate)
    CompanieswithOfertas = ContadorMapasNOList(valuesSBCountry,"company_name")
    NUMcompanies = mp.size(CompanieswithOfertas)
    CitieswithOfertas = ContadorMapasNOList(valuesSBCountry,"city")
    NUMcities = mp.size(CitieswithOfertas)
    keyscities = mp.keySet(CitieswithOfertas)
    MAXofertasCity = mp.get(CitieswithOfertas,lt.getElement(keyscities,1))
    MINofertasCity = mp.get(CitieswithOfertas,lt.getElement(keyscities,1))
    for i in lt.iterator(keyscities):
        dato = mp.get(CitieswithOfertas,i)
        if MAXofertasCity["value"] < dato["value"]:
            MAXofertasCity = dato
        if MINofertasCity["value"] > dato["value"]:
            MINofertasCity = dato
    sa.sort(valuesSBCountryANDDate,CompareCompanies)
    DateSort(valuesSBCountryANDDate)

    return NUMofertas, NUMcompanies, NUMcities, MAXofertasCity, MINofertasCity, valuesSBCountryANDDate
```

Este requerimiento se encarga de retornar Total de ofertas en el país, Total de empresas con más de 1 publicación, Número total de ciudades del país, Ciudad del país con mayor número de ofertas, Ciudad del país con menor número de ofertas, Listado de ofertas publicadas.

Lo primero que hace es sacar los IDs del mapa “country_code” con el parámetro código de país ingresado por el usuario. Luego, saca los valores del mapa “jobs” en el cual esta toda la información como (“key”: ID (único),”value”: todos los datos de todos los csv). Posteriormente, con los parámetros

de entrada de Fecha inicial y Fecha final se revisan las fechas de las ofertas en el país especificado y se añaden a una nueva lista. Así logrando tener como resultado una lista con las ofertas en el país especificado que estén en el rango de fechas especificado.

Ya teniendo la lista se usan funciones de DISClib para sacar el tamaño de la lista y una función implementada llamada: "ContadorMapasNOList" la cual cuenta todas las ocurrencias de la llave determinada en la lista especificada y lo añade a un mapa, como resultado queda una tupla ("key": valor de llave,"value": cantidad de ocurrencias mayor a 0). Con esta función se sacan los tamaños de la cantidad de ofertas y se usa un algoritmo de mayor y menor para sacar la ciudad con más ofertas y la ciudad con menos ofertas. Finaliza organizando la lista de datos para la impresión.

Entrada	Estructura de datos con los mapas que contienen la información completa, Código del país, Fecha inicial, Fecha final
Salidas	Total de ofertas en el país Total de empresas con más de 1 publicación Número total de ciudades del país Ciudad del país con mayor número de ofertas Ciudad del país con menor número de ofertas Listado de ofertas publicadas
Implementado (Sí/No)	Sí, implementado por Jacobo Morales Erazo

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad (aclaración N<K<L)**
Paso 1 (sacar IDs del país especificado)	O(K)
Paso 2 (sacar valores de jobs)	O(K)
Paso 3 (filtrar los valores por fecha)	O(K)
Paso 4 (Utilizar función contadora por mapas)	O(L)
Paso 5 (Obtener sizes)	O(1)
Paso 6 (Obtener mayor y menor valor)	O(L)
Paso 7 (Organizar para la impresión)	O(L)
TOTAL	***O(3K+3L+1) = O(K)***

Pruebas Realizadas

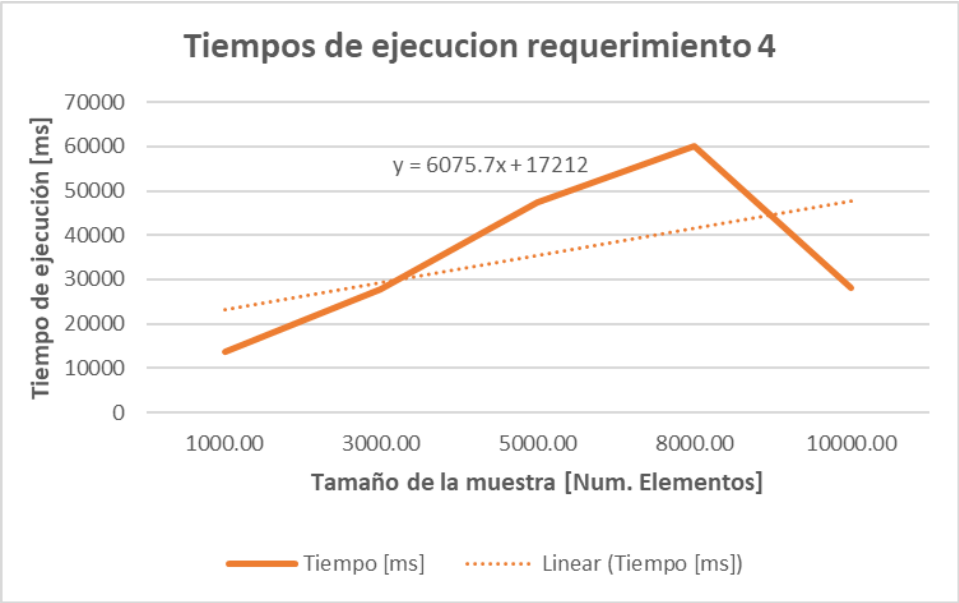
Para medir los tiempos de ejecución frente al requerimiento se usó la librería Time.

Parámetros: usado PL, 2022-04-01, 2023-01-01

Tablas de datos

Entrada	Tiempo (ms)
Large	28115.39208984375
Medium	47504.8134765625
Small	27791.27392578125
10	13822.173583984375
80	59961.4873046875

Graficas



Análisis

Si se omite el dato anormal con el 100% de los datos se puede ver la tendencia clara que se espera con el análisis de complejidad. La complejidad debería ser de O(K) el cual es menor que O(N). Esto lo soporta la gráfica, donde se ve la función de la tendencia de crecimiento lineal del tiempo a medida que aumenta la cantidad de datos y que no es del tamaño de los datos (un poco menor) esto se puede correlacionar con el cálculo teórico de la complejidad que debería ser K.

Requerimiento 5

Descripción

```python
def req_5(catalog, ciudad, fecha_inicial, fecha_final):
    moam_c= catalog["model"]["city"]
    moam_p= catalog["model"]["published_at"]
    moam_e= catalog["model"]["company_name"]
    moam_j= catalog["model"]["jobs"]

    lista_empresas= mp.keySet(moam_e)
    lista_id_empresas= mp.valueSet(moam_e)

    #Transformar lista_id_empresas a array
    lista_id_empresas_array= lt.newList("ARRAY_LIST")
    for id in lt.iterator(lista_id_empresas):
        lt.addLast(lista_id_empresas_array, id["elements"][0])
    lista_id_empresas_array= lista_id_empresas_array["elements"]

    #Obtener ids de una ciudad especifica
    lista_id_ciudad= mp.get(moam_c, ciudad)["value"]["elements"]

    lista_id_fechas= mp.valueSet(moam_p)

    #Volver lista_id_fechas a array
    fecha_array= lt.newList("ARRAY_LIST")
    for id in lt.iterator(lista_id_fechas):
        lt.addLast(fecha_array, id["elements"][0])
    fecha_array= fecha_array["elements"]

    #Sorteo por fechas
    fullinfo= adi.getFullInfos(moam_j, fecha_array)
    fullinfo= adi.deboneIds(fullinfo)["elements"]

    #Filtro por rango de fechas
    lista_id_tiempo_filtradas= lt.newList("ARRAY_LIST")
    for offer in fullinfo:
        published_at = dt.fromisoformat(offer['published_at'][:-5])
        id= offer["id"]
        if fecha_inicial <= published_at <= fecha_final:
            lt.addLast(lista_id_tiempo_filtradas, id)
    lista_id_tiempo_filtradas= lista_id_tiempo_filtradas["elements"]
```

```python
#Filtro especifico por ciudad y por empresas
id_repetidas_ciudad_tiempo= adi.getIdentical(lista_id_tiempo_filtradas, lista_id_ciudad)["elements"]
id_repetidas_empresa= adi.getIdentical(id_repetidas_ciudad_tiempo, lista_id_empresas_array)["elements"]

total_ciudad= len(id_repetidas_ciudad_tiempo)
total_empresas= len(id_repetidas_empresa)

#Empresas max y min y sus contadores
empresas_dict= {}
for empresa in lt.iterator(lista_empresas):
    ids= mp.get(moam_e, empresa)["value"]
    longitud= lt.size(ids)

    empresas_dict[empresa]= longitud

if (max(empresas_dict, key=empresas_dict.get)) == 1:
    empresa_max = max(empresas_dict, key=empresas_dict.get)
    empresa_min= empresa_max
else:
    empresa_max = max(empresas_dict, key=empresas_dict.get)
    empresa_min = min(empresas_dict, key=empresas_dict.get)

#Organizacion y filtro para sortear primero por fecha y luego nombre
fullinfo2= adi.getFullInfos(moam_j, id_repetidas_ciudad_tiempo)
fullinfo2= adi.deboneIds(fullinfo2)["elements"]

fullinfo2_1= lt.newList("ARRAY_LIST")
for dict in fullinfo2:
    lt.addLast(fullinfo2_1, dict)

lista_organizada_todo= (merg.sort(fullinfo2_1, Compare_fecha_nombre))["elements"]
```

```python
#sacar informacion de lista sorteada
ofertas_sorted= lt.newList("ARRAY_LIST")
for oferta in lista_organizada_todo:
    oferta_info= {"Fecha": oferta["published_at"],
                  "Titulo": oferta["title"],
                  "Nombre_empresa_oferta": oferta["company_name"],
                  "Tipo_lugar": oferta["workplace_type"],
                  "Tamaño_empresa": oferta["company_size"]
    }
    lt.addLast(ofertas_sorted, oferta_info)

tupla_max= empresa_max, empresas_dict[empresa_max]
tupla_min= empresa_min, empresas_dict[empresa_min]

return total_ciudad, total_empresas, tupla_max, tupla_min, ofertas_sorted["elements"]
```

El requisito 5 se aborda organizando los datos del catálogo y extrayendo información relevante basada en la ciudad y el rango de fechas especificados. Se filtran las ofertas de trabajo según el criterio de fecha, luego se comparan con las empresas y la ciudad especificadas para identificar las coincidencias. Se cuentan las ofertas por ciudad y empresa, y se determinan las empresas con más y menos ofertas. Finalmente, se ordenan las ofertas restantes primero por fecha y luego por nombre de empresa, proporcionando así una lista de ofertas clasificada para su presentación al usuario.

Entrada	Ciudad y rango de fechas.
Salidas	Total de ofertas ciudad y empresas Empresa con max y min de ofertas Lista sorteada con información.
Implementado (Sí/No)	Pablo Sarmiento

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1	O(n)
Paso 2	O(1)
Paso 3	O(m)
Paso 4	O(k)
Paso 5	O(f)
Paso 6	O(f)
Paso 7	O(m log m)
Paso 8	O(m)
TOTAL	***O(n)***

Pruebas Realizadas

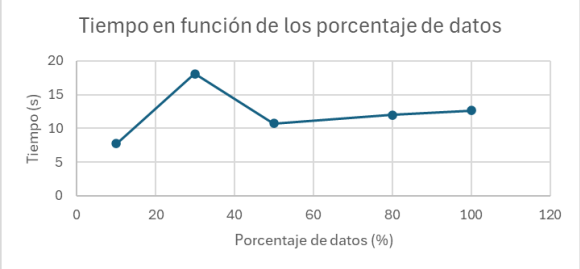
Para medir los tiempos de ejecución y el consumo de ram frente al requerimiento se usaron las librerías de TraceMalloc y Time.

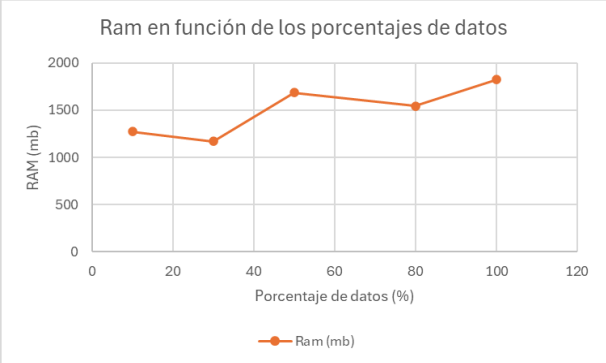
Parametros: Warszawa, 2022-01-01, 2024-01-01

Tablas de datos

Entrada	Tiempo (s)	Ram (mb)
Large	12657.25390625	1822.34
Medium	10743.641357421875	1684.76
Small	18073.159912109375	1168.34
10	7738.35009765625	1275.54
80	11973.078369140625	1543.23

Graficas





Análisis

La función req_5 opera en varios pasos para procesar el catálogo de ofertas de trabajo y proporcionar información específica sobre las ofertas en una ciudad dada dentro de un rango de fechas especificado. En primer lugar, se organizan los datos del catálogo, lo que implica acceder a diferentes campos como ciudad, fecha de publicación, nombre de la empresa y detalles de la oferta, con una complejidad de acceso de O(1) para cada uno. Luego, se obtienen listas de identificadores de empresas y se transforman en matrices, lo que implica recorrer y agregar elementos a una lista, con una complejidad de O(n) para la transformación. Se extraen los identificadores de la ciudad especificada del catálogo, lo que implica acceso a través del mapa con complejidad O(1). Posteriormente, se procesan las fechas de publicación para filtrar las ofertas dentro del rango especificado, lo que implica recorrer y comparar las fechas con una complejidad de O(m), donde m es el número de ofertas. Se filtran las ofertas específicas de la ciudad y las empresas, utilizando la función getIdentical, que tiene una complejidad de O(k), donde k es el tamaño de la lista de identificadores de tiempo filtradas. Luego, se cuentan las ofertas por ciudad y empresa, lo que implica recorrer las listas resultantes con una complejidad de O(n) para las empresas y O(m) para la ciudad. Se determinan las empresas con más y menos ofertas, lo que implica iterar sobre las empresas con una complejidad de O(n). A continuación, se organizan las ofertas primero por fecha y luego por nombre de empresa, lo que implica ordenar las ofertas con una complejidad de O(m log m) por el uso de Merge sort. Finalmente, se extrae información detallada de las ofertas ordenadas, lo que implica recorrer la lista ordenada con una complejidad de O(m). En resumen, la función req_5 proporciona una solución eficaz para analizar y presentar información detallada sobre las ofertas de trabajo en una ciudad específica durante un rango de fechas dado, con una complejidad temporal total que depende del tamaño de los datos de entrada, siendo dominada por la la obtención de la lista de IDs con notación O(N).

Requerimiento 6

Descripción

```python
def req_6(catalog, exp, year, N):
    start = time.time()*1000
    #* EdgeCases
    if exp != 'indiferente':
        #* 1) Debemos sacar todos los IDs correspondientes al nivel de experiencia ~(1/3 N)
        expMap = catalog['experience_level']
        byExp = mp.get(expMap, exp) # Acceder al mapa
        byExp = byExp['value'] # (Esto es una ARRAY_LIST con todos los IDs)
    elif exp == 'indiferente':
        byExp = mp.valueSet(expMap)
    ##* Ahora debemos acceder a los elementos del año indicado.
    datemap = catalog['published_at']
    keysByYear = adi.getKeysByYear(datemap, year)
    IdsByYear = lt.newList("ARRAY_LIST")
    for key in lt.iterator(keysByYear):
        # Acá tenemos que sacar los IDs asociados a la llave y ver si también pertenecen a el nivel
        InternalIds = mp.get(datemap, key)
        InternalIds = InternalIds['value']
        for idi in lt.iterator(InternalIds):
            lt.addLast(IdsByYear, idi)
    meetsCriterias = adi.getIdentical(byExp, IdsByYear) # Usamos la función para encontrar similares
```

Disminuye la cantidad de datos de las listas

Entrada	Catalog, exp, year, N
Salidas	Total de conteos y ofertas
Implementado (Sí/No)	Sí, grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O(...)
Paso 2	O(...)
Paso ….	O(...)
TOTAL	***O(...)***

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).

Entrada	**Tiempo (s)**

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Requerimiento 7

Descripción

```
def req_7(control, numeroPaises, año, mes):
    """
    Función que soluciona el requerimiento 7
    """

    moal2=control["jobs"]
    #filtrar por fecha
    fecha=año+"-"+mes
    mapa_date=control["published_at"]
    lista_fechas = mp.keySet(mapa_date)
    # IDsSWD = datos Shorten With Date
    IDsSWD=lt.newList(datastructure="ARRAY_LIST")
    for i in lt.iterator(lista_fechas):
        if i[0:7]==fecha:
            IDs=mp.get(mapa_date,i)
            lt.addLast(IDsSWD,IDs["value"])
    # sacar elemento de moal2 con IDsSWD
    # values = valores Shorten With Date
    valuesSWD=lt.newList(datastructure="ARRAY_LIST")
    for i in lt.iterator(IDsSWD):
        for j in lt.iterator(i):
            dato = mp.get(moal2,j)
            lt.addLast(valuesSWD,dato["value"][j])

    #Organizar la lista IDsSWD
    valuesSWD=DateSort(valuesSWD)

    # Solo dejar los que tengan los paises pedidos
    #MAcountriesCount = Max Amount of countries Count
    MAcountriesCount = mp.newMap(numelements=3,maptype="PROBING",loadfactor=0.5)
    for i in lt.iterator(valuesSWD):
        i=i["country_code"]
        existcountry = mp.contains(MAcountriesCount,i)
        if existcountry:
            entry = mp.get(MAcountriesCount,i)
            pais = me.getValue(entry)
        else:
            pais = 0
        pais = pais + 1
        mp.put(MAcountriesCount,i,pais)
    listcountries=mp.keySet(MAcountriesCount)
    # MAcountries = Max Amount of countries
    MAcountries = lt.newList(datastructure="ARRAY_LIST")
    listMAcountriesCount = lt.newList(datastructure="ARRAY_LIST")
    for i in lt.iterator(listcountries):
        dato=mp.get(MAcountriesCount,i)
        lt.addLast(listMAcountriesCount,dato)

    merg.sort(listMAcountriesCount,CompareValuesMapTuple)

    for i in lt.iterator(listMAcountriesCount):
        key=mp.get(MAcountriesCount,i["key"])["key"]
        present = lt.isPresent(MAcountries,key)
        if lt.size(MAcountries) == numeroPaises:
            break
        elif present == 0:
            lt.addLast(MAcountries,key)

    #valuesFbCs = values Filtered by Countries
    valuesFbCs = lt.newList(datastructure="ARRAY_LIST")
    for i in lt.iterator(valuesSWD):
        for j in lt.iterator(MAcountries):
            icountry=i["country_code"]
            if icountry == j:
                lt.addLast(valuesFbCs,i)

    #* total de ofertas
    NUMofertas=lt.size(valuesFbCs)

    #* Número ciudades donde se ofertó en los países resultantes
    #CitiesCount = número de ofertas por ciudad
    CitiesCount = mp.newMap(numelements=11,maptype="PROBING",loadfactor=0.5)
    for i in lt.iterator(valuesFbCs):
        city = i["city"]
        existcity = mp.contains(CitiesCount,city)
        if existcity:
            entry = mp.get(CitiesCount, city)
            ciudad = me.getValue(entry)
        else:
            ciudad = 0
        ciudad = ciudad + 1
        mp.put(CitiesCount,city,ciudad)
    NUMofertasCities=mp.size(CitiesCount)

    listCities=mp.keySet(CitiesCount)
    listCitiesCount = lt.newList(datastructure="ARRAY_LIST")
    for i in lt.iterator(listCities):
        dato=mp.get(CitiesCount,i)
        lt.addLast(listCitiesCount,dato)

    merg.sort(listCitiesCount,CompareValuesMapTuple)

    MAcities = lt.getElement(listCitiesCount,1)

    #* País con mayor cantidad de ofertas
    MAXNUMofertasCountry = lt.getElement(listMAcountriesCount,1)
    for i in lt.iterator(listMAcountriesCount):
        value=i["value"]
        if MAXNUMofertasCountry["value"]<value:
            MAXNUMofertasCountry=i


    juniorinfo= manejoinfoExperience(valuesFbCs,"junior")
    middleinfo= manejoinfoExperience(valuesFbCs,"mid")
    seniorinfo= manejoinfoExperience(valuesFbCs,"senior")

    """valuesFbCs = todos los elements"""
    return  NUMofertas, MAXNUMofertasCountry, MAcities, NUMofertasCities, juniorinfo, middleinfo, seniorinfo, valuesFbCs
```

En este requerimiento entran los parámetros: control (estructura de datos con los mapas con la información de los csvs), Número de países, año y mes. Con estos datos se accede a los mapas con los IDs y la información. Primero, se reduce el tamaño de los datos viendo que datos tienen la misma fecha que la especificada. Segundo, se obtienen los valores de los IDs resultantes. Tercero, se organizan por fecha para un uso posterior. Cuarto, se cuenta la cantidad de ofertas por país con la función

“ContadorMapasNOList”. Quinto, se mantienen (añadiéndolos a una nueva lista) los países que tengan mayor cantidad de ofertas siempre siendo menor o igual a la cantidad especificada. Con eso queda la lista con los datos para obtener las salidas.

Posteriormente, se cuentan los datos necesarios para las salidas, como la cantidad de ofertas, la cantidad ofertas del país con más ofertas, la cantidad de ofertas de las ciudades, la cantidad de ofertas de la ciudad con más ofertas.

Por último, se crea una función por aparte llamada “manejoinfoExperience” la cual cuenta las habilidades, compañías y sedes de acuerdo a un nivel de experiencia.

Entrada	Data structure, Número (N) de países, Fecha inicial consulta, Fecha final consulta
Salidas	Total de ofertas, Número ciudades, Nombre país con mayor cantidad y conteo, Nombre de la ciudad con mayor cantidad y conteo, conjunto de datos para junior, mid y senior calculando diversos datos.
Implementado (Sí/No)	Sí, grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad (Aclaración N>K>L)**
Paso 1	O(K)
Paso 2	O(K)
Paso 3	O(K)
Paso 4	O(K)
Paso 5	O(K)
Paso 6	O(L)
Paso 7 se repite 3 veces	O(L)
TOTAL	***O(5K+2L)=O(K)***

Pruebas Realizadas

Para medir los tiempos de ejecución frente al requerimiento se usó la librería Time.

Número paises: 8

Año: 2022

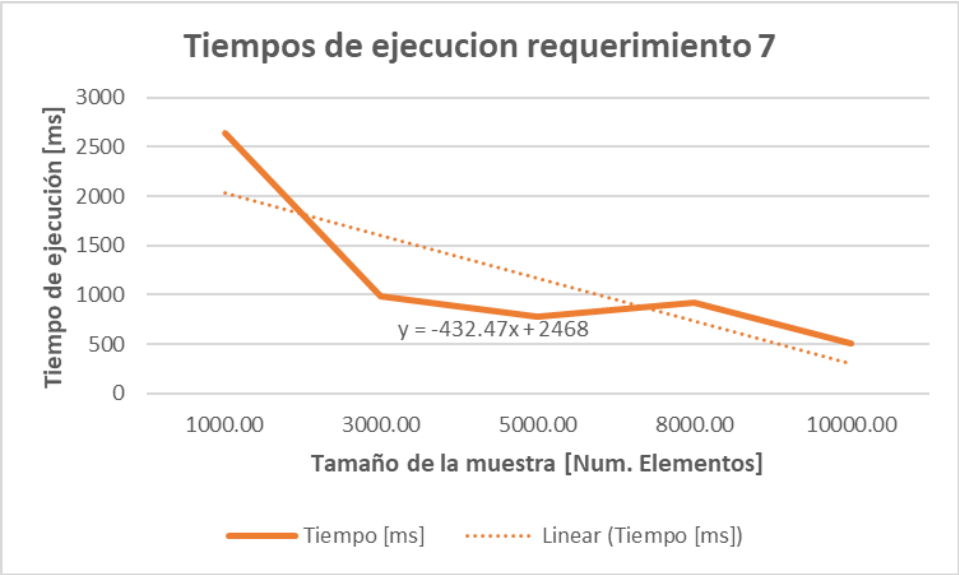
Mes: 04

Tablas de datos

Entrada	**Tiempo (ms)**

Large	990.1633
Medium	785.3489
Small	511.1475
10	2641.0205
80	925.2019

Graficas



Análisis

La gráfica muestra un comportamiento anormal a comparación del esperado en el análisis de complejidad puesto que la complejidad debería de disminuir a medida que disminuya la cantidad de datos. Además, se ve un comportamiento anormal en el dato de 10% de los datos. A pesar de esto la gráfica muestra un comportamiento opuesto al esperado. Esto se podría explicar con la idea de que al haber más datos estos se guardan de una forma más eficiente en los mapas por lo tanto el K se va haciendo menor al N proporcionalmente a medida que el N aumenta. Es decir, tiene una gran eficiencia en grandes cantidades de datos porque la repetición de datos similares al ser guarda en mapas tiene una complejidad muy baja para acceder a estos.