ANÁLISIS DEL RETO

Andrés Felipe Rodriguez Acosta, af.rodrigueza12345@uniandes.edu.co, 202322292
Juan David Gutierrez Rodriguez, jd.gutierrezr123@uniandes.edu.co, 202316163
Samuel Rodriguez Robledo, s.rodriguezr234567@uniandes.edu.co, 202323878

Introducción

En este reto nuestro objetivo es aplicar los conocimientos aprendidos durante el módulo dos del curso Estructuras de datos y algoritmos. Nos centraremos en la implementación del TAD Map, la implementación de tablas de hash y la medición del uso de memoria. En este documento llevaremos a cabo un análisis de la complejidad de los algoritmos empleados y su tiempo de ejecución.

Funciones

Requerimiento 3: *Samuel Rodriguez Robledo*

Requerimiento 4: *Juan David Gutierrez Rodriguez*

Requerimiento 5: *Andres Felipe Rodriguez Acosta*

Índice

1) Carga de datos
2) Requerimiento 1
3) Requerimiento 2
4) Requerimiento 3
5) Requerimiento 4
6) Requerimiento 5
7) Requerimiento 6
8) Requerimiento 7
9) Requerimiento 8

Carga de datos

Descripción

Entrada	Modelo donde encontramos los índices implementados, tamaño de archivo, y si se desea medir la memoria. Para crear el modelo se necesita el tipo de tabla de hash con su factor de carga.
Salidas	Modelo con cada una de las ofertas agregadas
Implementado (Sí/No)	Si fue implementado por Juan David

Para la carga de los datos contamos con diferentes configuraciones para poder correctamente el modelo con cada uno de los índices implementados. Primero debemos elegir un tipo de tabla de hash, luego el tamaño del archivo y por último si desea medir memoria o no. Ya con esto podemos dirigirnos al modelo para llamar una a una las funciones que cargan los datos de cada archivo (ofertas, habilidades, tipos de contratación y locaciones).

```
def load_data(control, filename, memflag):
    """
    Carga los datos del reto
    """
    start_time = get_time()

    if memflag:
        tracemalloc.start()
        start_memory = get_memory()

    catalog = control['model']

    load_jobs(catalog, filename)
    load_skills(catalog, filename)
    load_employment_types(catalog, filename)
    load_multilocations(catalog, filename)

    sort(catalog['jobs'])

    stop_time = get_time()
    deltaTime = delta_time(start_time, stop_time)
    deltaMemory = 0

    if memflag:
        stop_memory = get_memory()
        tracemalloc.stop()
        deltaMemory = delta_memory(stop_memory, start_memory)

    return catalog['jobs'], deltaTime, deltaMemory
```

En el caso de la función load_jobs() se cambian las llaves que son Undefined por Desconocido. Seguido a esto agregamos la oferta a la lista y para cada índice de los otros archivos se agrega la llave del id de la oferta para que sea más fácil relacionar las ofertas con estos, y como valor creamos una lista vacía.

```python
def new_job(catalog, job):
    """
    Crea una nueva estructura para modelar las ofertas
    """
    for key in job:
        if job[key] == 'Undefined':
            job[key] = 'Desconocido'

    add_lst(catalog['jobs'], job)
    add_map(catalog['skills'], job['id'], lt.newList())
    add_map(catalog['employment_types'], job['id'], lt.newList())
    add_map(catalog['multilocations'], job['id'], lt.newList())
```

Para los otros archivos simplemente obtenemos las lista cuya llave es el id de la oferta y se agrega a esa lista el dato en el que estemos iterando. Sin embargo, para el caso de los tipos de contratación, primero, cambiamos los valores que estén vacíos por 0 y luego podremos obtener el salario promedio con la llave 'salary_from' y 'salary_to'. Ya con esto podremos agregarlo a la estructura de datos.

```python
def new_data(catalog, data):
    """
    Crea una nueva estructura para modelar los datos
    """
    lst = me.getValue(get_data(catalog, data['id']))
    add_lst(lst, data)
```

```python
def new_employment_type(catalog, data):
    """
    Crea una nueva estructura para modelar los tipos de contratacion
    """
    for key in data:
        if data[key] == '':
            data[key] = '0'

    data = {
        'type': data['type'],
        'id': data['id'],
        'currency_salary': data['currency_salary'],
        'salary': (int(data['salary_from']) + int(data['salary_to'])) / 2
    }

    new_data(catalog, data)
```

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Iterar sobre cada oferta del archivo	O(n)
Recorrer cada llave del diccionario de la oferta	O(16)
Agregar lista en la tabla de hash	O(1)
Iterar sobre los demás archivos	O(n)
Obtener la pareja en la tabla y agregar a la lista	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas con el tamaño de archivo -small en un maquina con las siguientes capacidades.

Procesadores	Intel Core i7 -13700H
Memoria RAM (GB)	16 GB
Sistema operativo	Windows 11 Home

Tablas de datos

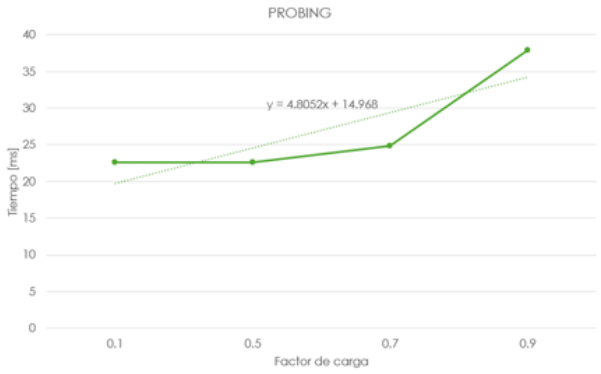
Las tablas con la recopilación de datos de las pruebas.

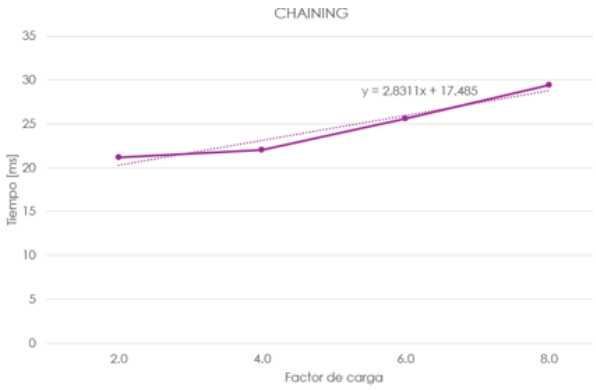
	Factor de carga	Salida	Memoria [Kb]	Tiempo [ms]
PROBING	0.1	34.470 ofertas	570984.43	22.595
	0.5	67.099 ofertas	570984.35	22.622
	0.7	103.709 ofertas	570984.30	24.828
	0.9	137.873 ofertas	570984.27	37.877
CHAINING	2.0	167.989 ofertas	694.62	21.173
	4.0	187.520 ofertas	694.62	22.058
	6.0	193.694 ofertas	694.61	25.586
	8.0	198.119 ofertas	694.61	29.434

Graficas

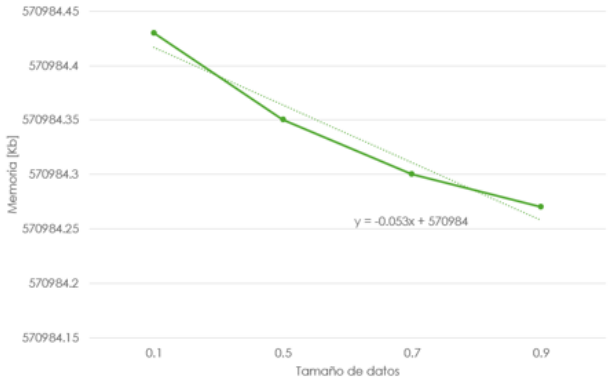
Las gráficas con la representación de las pruebas realizadas.

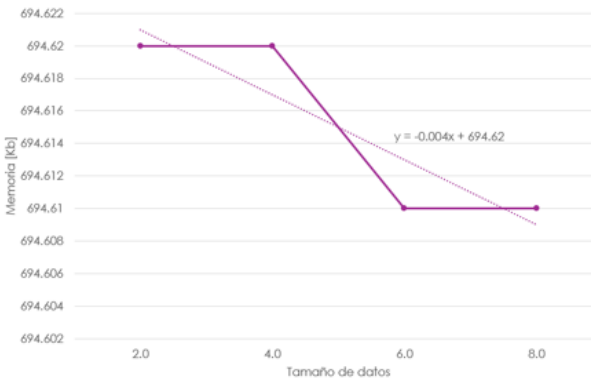
- **Factor de carga vs Tiempo**





- **Factor de carga vs Memoria**





Análisis

Nuestra carga de datos funciona correctamente con el archivo -large; sin embargo, al medir la memoria observamos que se acerca a la capacidad máxima de memoria permitida por nuestros computadores y es por eso que estas pruebas están desarrolladas con el archivo -small. En cuanto al resultado de estas, evidenciamos un notable cambio en cuanto el cambio de memoria respecto al factor de carga cuando cambiamos el mecanismo de colisiones. Para probing vemos que, cuando el factor de carga es mas grande, la memoria ocupada disminuye. Y en cuanto a chaining, existe un cambio más brusco en el consumo de memoria cuando usamos el factor de carga de 6.0. En cuanto a los tiempos de ejecución, podemos decir que a medida que aumenta el factor de carga en ambas implementaciones, el tiempo también aumenta gradualmente.

Volver al índice

Requerimiento 1

Descripción

Entrada	Código del país, nivel de experticia de las ofertas a consultar, número de ofertas a listar.
Salidas	Una lista con las N ofertas de trabajo ofrecidas en un país, filtradas por el nivel de experticia y la cantidad de ofertas publicadas.
Implementado (Sí/No)	Si fue implementado por Juan David.

Este requerimiento recibe como parámetro la cantidad de ofertas que se desea conocer, el nombre del país que se desea consultar y el nivel de experticia que requiere la oferta, para así retornar las empresas que cumplan con estas condiciones.

```
def req_1(catalog, num, country, exp):
    """
    Función que soluciona el requerimiento 1
    """
    jobs = catalog['jobs']
    filtered_jobs = lt.newList()
    country_offers = 0

    for job in lt.iterator(jobs):

        if job['country_code'] == country.upper():
            country_offers += 1

            if job['experience_level'] == exp.lower():
                lt.addLast(filtered_jobs, job)
            if data_size(filtered_jobs, lt) == num:
                return filtered_jobs, country_offers

    return filtered_jobs, country_offers
```

En el código se recorre la lista de ofertas de trabajo y se comparan las condiciones de cada elemento con las que entran por parámetro y se añade la oferta a una lista. Luego se verifica que el número de ofertas en la lista sea igual al que llega por parámetro para así retornar la cantidad indicada.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Declaración de variable Jobs y creación de lista.	O(1)
Recorrer la lista “Jobs”	O(n)
Comparaciones	O(1)
Agregar 1 al total de ofertas	O(1)
Agregar a la lista	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas con el tamaño de archivo -large en un maquina con las siguientes capacidades.

Procesadores	11th Gen Intel® Core™
Memoria RAM (GB)	8 GB
Sistema operativo	Windows 11 Home

Tablas de datos

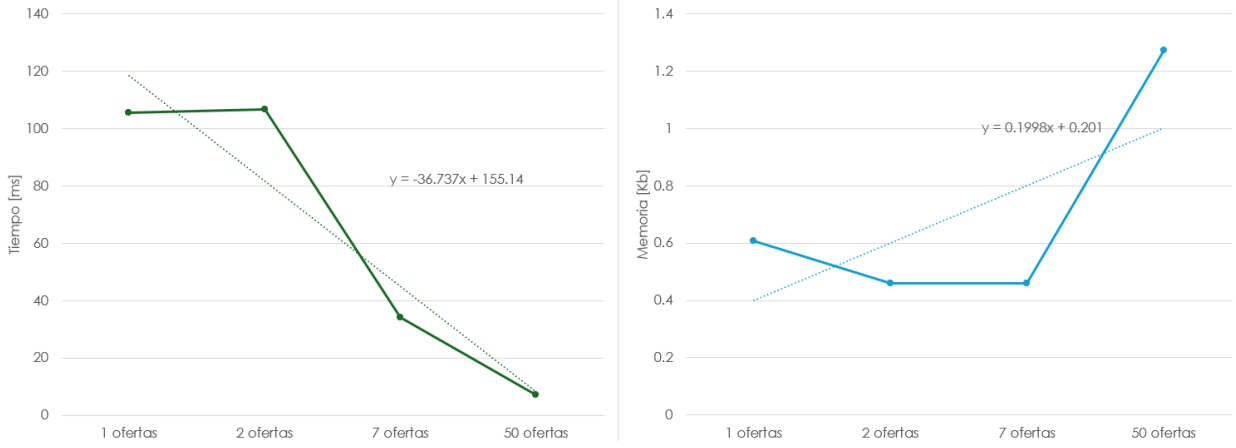
Las tablas con la recopilación de datos de las pruebas.

Entradas	Salida	Memoria [Kb]	Tiempo [ms]
FR, junior, 30	1 ofertas	0.61	105.442
AR, senior, 15	2 ofertas	0.46	106.539
CH, mid, 7	7 ofertas	0.46	34.085
PL, mid, 50	50 ofertas	1.27	7.138

Plot Area

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

En este requerimiento podemos observar que cuando el número de ofertas no es suficiente a las que quiere listar el usuario el tiempo de ejecución aumenta, y su complejidad es O(n). Esto se debe a que debe recorrer absolutamente todas las ofertas. Y en cuanto a la memoria, entre más ofertas obtenga, más memoria va a ocupar ya que esta almacenando esa cantidad de ofertas.

Volver al índice

Requerimiento 2

Descripción

Entrada	Lista de ofertas, número de elementos a listar, nombre de la empresa y ciudad.
Salidas	Lista con ofertas filtradas por nombre de empresa y ciudad.
Implementado (Sí/No)	Si fue implementado Juan David

```python
def req_2(catalog, num, company_name, city):
    """
    Función que soluciona el requerimiento 2
    """
    jobs = catalog['jobs']
    filtered_jobs = lt.newList('ARRAY_LIST')

    for job in lt.iterator(jobs):
        if job['company_name'].lower() == company_name.lower() and job['city'].lower() == city.lower():
            lt.addLast(filtered_jobs, job)
        if lt.size(filtered_jobs) == num:
            return filtered_jobs

    return filtered_jobs
```

Este requerimiento devuelve una cantidad de ofertas ingresada por el usuario comparándolas por el nombre de la empresa y la ciudad. Primero, agrega la oferta si el nombre y la ciudad son iguales a las ingresadas por el usuario. Luego, verifica si el tamaño de la lista es igual al número de ofertas que quiere ver el usuario. En caso de que esta última comparación sea verdadera, devolverá la lista.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Declaración de variable Jobs y creación de lista.	O(1)
Recorrer lista Jobs	O(n)
Comparaciones	O(1)
Agregar a la lista	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas con el tamaño de archivo -large en un maquina con las siguientes capacidades.

Procesadores	11th Gen Intel® Core™
Memoria RAM (GB)	8 GB
Sistema operativo	Windows 11 Home

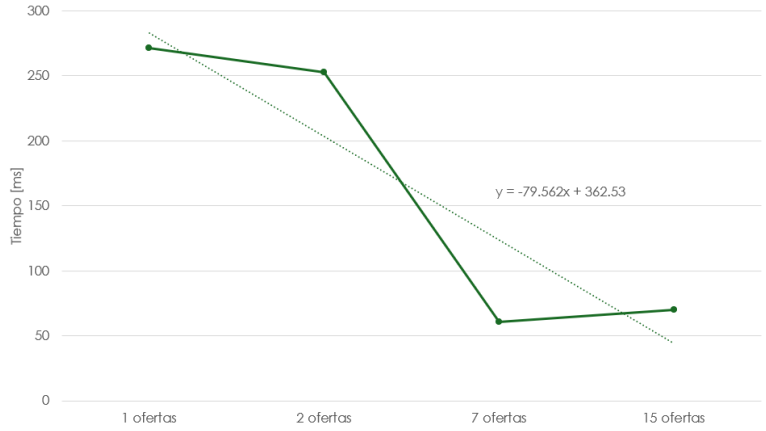
Tablas de datos

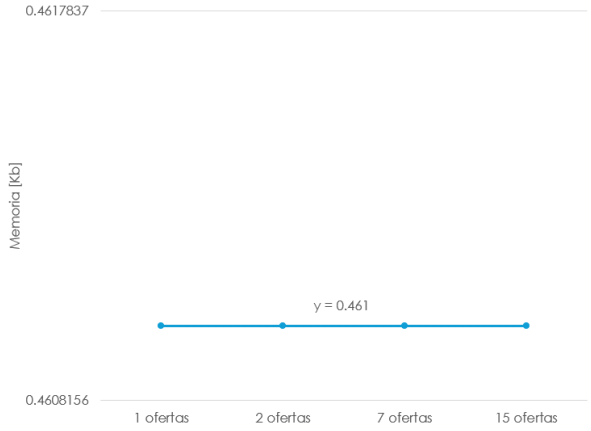
Las tablas con la recopilación de datos de las pruebas.

Entradas	Salida	Memoria [Kb]	Tiempo [ms]
Blockchain.com, Buenos Aires, 50	1 ofertas	0.46	271.181
Ericsson, Warszawa, 30	2 ofertas	0.46	252.783
Future Mind, Warszawa, 7	7 ofertas	0.46	60.466
Sopra Steria, Krakow, 15	15 ofertas	0.46	70.079

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

En este requerimiento podemos observar que cuando el número de ofertas no es suficiente a las que quiere listar el usuario el tiempo de ejecución aumenta, y su complejidad es O(n). Esto se debe a que debe recorrer absolutamente todas las ofertas. Y en cuanto a la memoria, podemos ver que tiene un comportamiento constante y siempre se mantiene bajo.

Requerimiento 3

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

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Se quiere analizar las ofertas de trabajado filtradas por el nombre de la empresa dentro de un rango de fechas, que entra como parámetro. Para empezar, abordamos haciendo la comparación para filtrar por nombre de empresa y rango de fechas, después creamos una copia de la lista filtrada para con una función auxiliar eliminar las llaves que no son necesarias. Después se suma a los contadores el número total de ofertas y cada nivel de experiencia la cantidad de ofertas. Finalmente se aplica el criterio de ordenamiento para ordenar de menor a mayor fecha, para devolver la lista filtrada, el número total de ofertas y el número de ofertas por cada nivel de experiencia.

Entrada	Nombre empresa, fecha límite inferior, fecha límite superior.
Salidas	Ofertas filtradas, total ofertas, total ofertas para cada experticia.

Implementado (Sí/No)	Si se implementó, Samuel

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Defino la variable offers como el catalog en la llave jobs	O(1)
Iterar offers	O(n)
Filtrar la empresa por nombre y por el rango de fecha que entro como parametro	O(1)
Crear un copia de la lista con los datos filtrados	O(1)
Eliminar las llaves que no necesito como parámetro con una función auxiliar	0(1)
Ordenar los datos	O(n log(n))
TOTAL	O(n log(n))

Pruebas Realizadas

Las pruebas fueron realizadas con el tamaño de archivo -large en un maquina con las siguientes capacidades.

	maquina
procesador	Intel Core i7 -13700H
RAM (GB)	16 GB
Sistema operativo	Windows 11 home

Tablas de datos

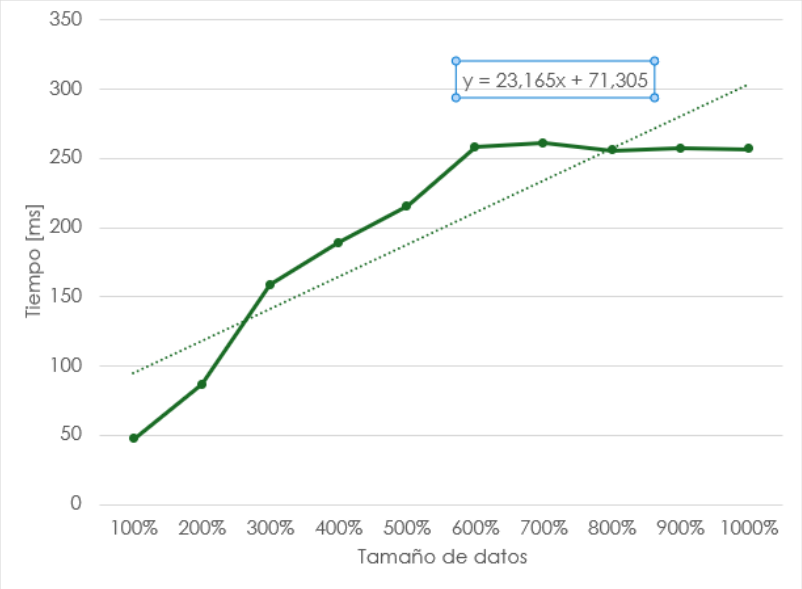
Las tablas con la recopilación de datos de las pruebas.

Entrada	**Salida**	**Memoria (Kb)**	**Tiempo (ms)**

Tamaño de datos	Salida	Tiempo [ms]
10%	2 ofertas	47,3
20%	3 ofertas	86,6
30%	4 ofertas	159,01
40%	12 ofertas	189,04
50%	12 ofertas	215,9
60%	12 ofertas	258,1
70%	12 ofertas	261,4
80%	12 ofertas	256
90%	12 ofertas	257,09
100%	12 ofertas	256,7

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.

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

Para este requerimiento hicimos pruebas con ARRAY_LIST ya que es más efectiva, notando que va a tener una complejidad O(n log(n)), pero esto puede cambiar de acuerdo al tipo de ordenamiento que se use, en este caso es esa, ya que la iteración que se hace recorre toda la lista comparando los valores que entran como parámetro con los del catalog y como algoritmo de ordenamiento se utilizo un merge . Las

demás cosas como las comparaciones, eliminar las llaves que no son necesarios, declaración de variables y agregar elementos en los “contadores” tienen complejidad 0(1)

Volver al índice

Requerimiento 4

Descripción

Entrada	País donde desea buscar, fecha límite inferior, fecha límite superior.
Salidas	Ofertas filtradas, ofertas publicadas en cada ciudad, número de compañías con al menos una oferta.
Implementado (Sí/No)	Si fue implementado por Juan David.

Este requerimiento busca las ofertas publicadas en un país entre un rango de fechas. Para esto le pedimos al usuario que ingrese el código de un país y dos limites (uno inferior y otro superior) de fechas de la forma año-mes-día. Primero se inicializa la tabla donde se guardarán las ofertas filtradas y dos tablas de hash que serán de ayuda para identificar el total de compañías y el total de ciudades.

```
req_4(catalog, country, min_date, max_date):
"""
Función que soluciona el requerimiento 4
"""
jobs = catalog['jobs']

filtered_jobs = lt.newList()

total_companies = mp.newMap(lt.size(jobs),
                            maptype='CHAINING',
                            loadfactor=4)

counted_cities = mp.newMap(lt.size(jobs),
                           maptype='CHAINING',
                           loadfactor=4)

min_date, max_date = datetime.strptime(min_date, '%Y-%m-%dT%H:%M:%S.%fZ'), datetime.strptime(max_date, '%Y-%m-%dT%H:%M:%S.%fZ')
```

Luego, recorremos las ofertas para comparar el país y verificar que se encuentre dentro de las fechas limite. Si eso se cumple, agregaremos la oferta a la lista y podremos hacer el conteo también de las empresas y de las ciudades.

```
for job in lt.iterator(jobs):

    if job['country_code'] == country.upper():

        # Obtener la fecha de la oferta para la comparacion
        current_date = datetime.strptime(job['published_at'], '%Y-%m-%dT%H:%M:%S.%fZ')

        if min_date < current_date < max_date:
            # Agregar la oferta a la lista
            lt.addLast(filtered_jobs, job)
            # Agregar la empresa a la tabla que contara el total de empresas con ofertas
            mp.put(total_companies, job['company_name'], None)

            # Manipular las ciudades y el numero de ofertas en cada ciudad
            if mp.contains(counted_cities, job['city']):
                # Obtener el diccionario de la ciudad y agregar 1
                city = me.getValue(get_data(counted_cities, job['city']))
                city['offers'] += 1

            else:
                # Crear estructura para modelar los datos
                city = {'name': job['city'],
                        'offers': 1}
                # Añadir ciudad a la tabla de las ciudades
                mp.put(counted_cities, job['city'], city)
```

Por último, verificaremos que la lista no este vacía. Con esto ya podemos ordenar las ciudades de mayor a menor número de ofertas y devolver los datos necesarios para implementar este requerimiento.

```
if lt.isEmpty(filtered_jobs):
    # Si no se encuentra ninguna oferta devolver None
    return None, None, None
else:
    # Ordenar las ciudades de mayor a menor, por el numero de ofertas publicadas
    counted_cities = sort(mp.valueSet(counted_cities), sort_req_4)

    return filtered_jobs, counted_cities, mp.size(total_companies)
```

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Recorrer ofertas	O(n)
Condicionales	O(1)
Agregar elementos a estructuras de datos	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas con el tamaño de archivo -large en un maquina con las siguientes capacidades.

Procesadores	11th Gen Intel® Core™
Memoria RAM (GB)	8 GB
Sistema operativo	Windows 11 Home

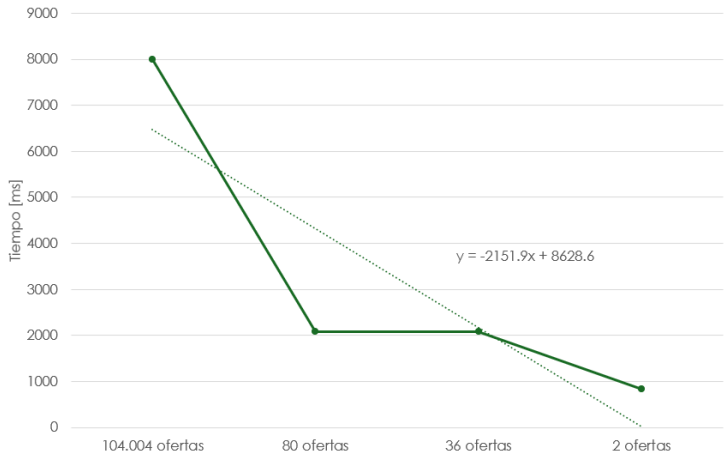
Tablas de datos

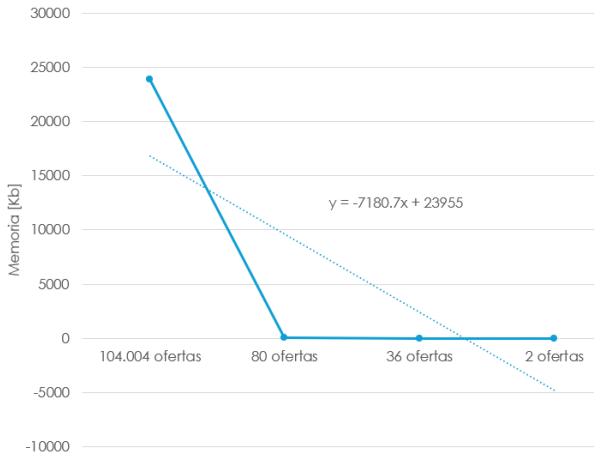
Las tablas con la recopilación de datos de las pruebas.

Entradas	Salida	Memoria [Kb]	Tiempo [ms]
PL, 2022-01-01, 2023-01-01	104.004 ofertas	23921.98	7999.048
CZ, 2023-05-20, 2023-12-20	80 ofertas	71.94	2084.110
FR, 2022-01-06, 2022-06-06	36 ofertas	12.62	2087.771
AR, 2022-01-01, 2022-12-31	2 ofertas	6.06	824.970

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

En este requerimiento podemos observar que cuando el número de ofertas es demasiado alto, tanto su consumo de memoria como su tiempo de ejecución crecen bastante. Sin embargo, a medida que el programa encuentra menos ofertas, estas dos mediciones también tienden a bajar. Esto quiere decir que entre mas ofertas se encuentren, mayor tiempo y memoria consumirá este requerimiento.

Volver al índice

Requerimiento 5

Descripción

Este requerimiento pide retornar las ofertas que se publicaron en una ciudad en rango de fechas, para empezar a abordar este requerimiento, se empieza pidiendo al usuario que ingrese los datos para usarlos como parámetro. Posteriormente se inician las variables “offer_ciudad” el cual va a ser un contador para el total de ofertas, “rta” la cual será una lista para guardar los datos filtrados, “empresas” para hacer un conteo de las empresas que tienen por lo menos una oferta publicada y “mayor” que es un mapa para usarlo posteriormente para hallar las empresas con mayor y menor número de ofertas y su conteo:

```
def req_5(catalog, ciudad, fecha_inicio, fecha_fin):
    """
    Función que soluciona el requerimiento 5
    """
    offers = catalog["jobs"]
    offer_ciudad = 0
    rta = lt.newList("ARRAY_LIST")
    ofertas = lt.newList("ARRAY_LIST")
    mayor = mp.newMap(100,
                                maptype='CHAINING',
                                loadfactor=4)
```

Despues hacemos la respectiva iteración y filtramos los datos en las fechas determinadas y que esten publicadas en la ciudad que entro como parametro para asi ingresar los respectivos datos.

```
for job in lt.iterator(offers):

    if job["city"].lower() == ciudad.lower() and fecha_inicio < job["published_at"] < fecha_fin:

        if mp.contains(mayor, job['company_name']):
            # Obtener el diccionario de la ciudad y agregar 1
            company = me.getValue(get_data(mayor, job['company_name']))
            company['total'] += 1
        else:
            # Crear estructura para modelar los datos
            company = {'name': job['company_name'],
                       'total': 1}
            mp.put(mayor, job['company_name'], company)

        if not lt.isPresent(ofertas, job["company_name"]):
            lt.addLast(ofertas, job["company_name"])

        offer_ciudad += 1
```

```
            if not lt.isPresent(ofertas, job["company_name"]):
                lt.addLast(ofertas, job["company_name"])

            offer_ciudad += 1
            res = {
                "fecha publicacion": job["published_at"],
                "titulo oferta" : job["title"],
                "Nombre empresa" : job["company_name"],
                "lugar de trabajo": job["workplace_type"],
                "tamaño empresa": job["company_size"]
            }

            lt.addLast(rta, res)

    counted_cities = sort(mp.valueSet(mayor), sort_req_5)
    maximo = lt.firstElement(counted_cities)
    minimo = lt.lastElement(counted_cities)
    empresas = lt.size(ofertas)
    sort(rta, sort_criteria_andres)

    return  offer_ciudad, empresas, maximo, minimo, rta
```

Por último, ordenamos el mapa de" mayor de mayor a menor sacamos el máximo y el mínimo de los valores de las empresas filtradas.

Entrada	Nombre ciudad, fecha inicial, fecha final

Salidas	Total, de ofertas publicadas, total empresas que publicaron por lo menos 1 oferta, empresa con más ofertas y su conteo, empresa con menor número de ofertas y su conteo, listado de ofertas publicadas
Implementado (Sí/No)	Si se implementó, Andres Felipe Rodriguez.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Iterar en “jobs”	O(n)
Filtrar los datos: condicionales	O(1)
Agregar al contador o a las listas el dato correspondiente dependiendo el caso	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas con el tamaño de archivo -large en un maquina con las siguientes capacidades.

	maquina
procesador	Ryzen 7 5700u
RAM (GB)	16 GB
Sistema operativo	Windows 10 home 64 bits

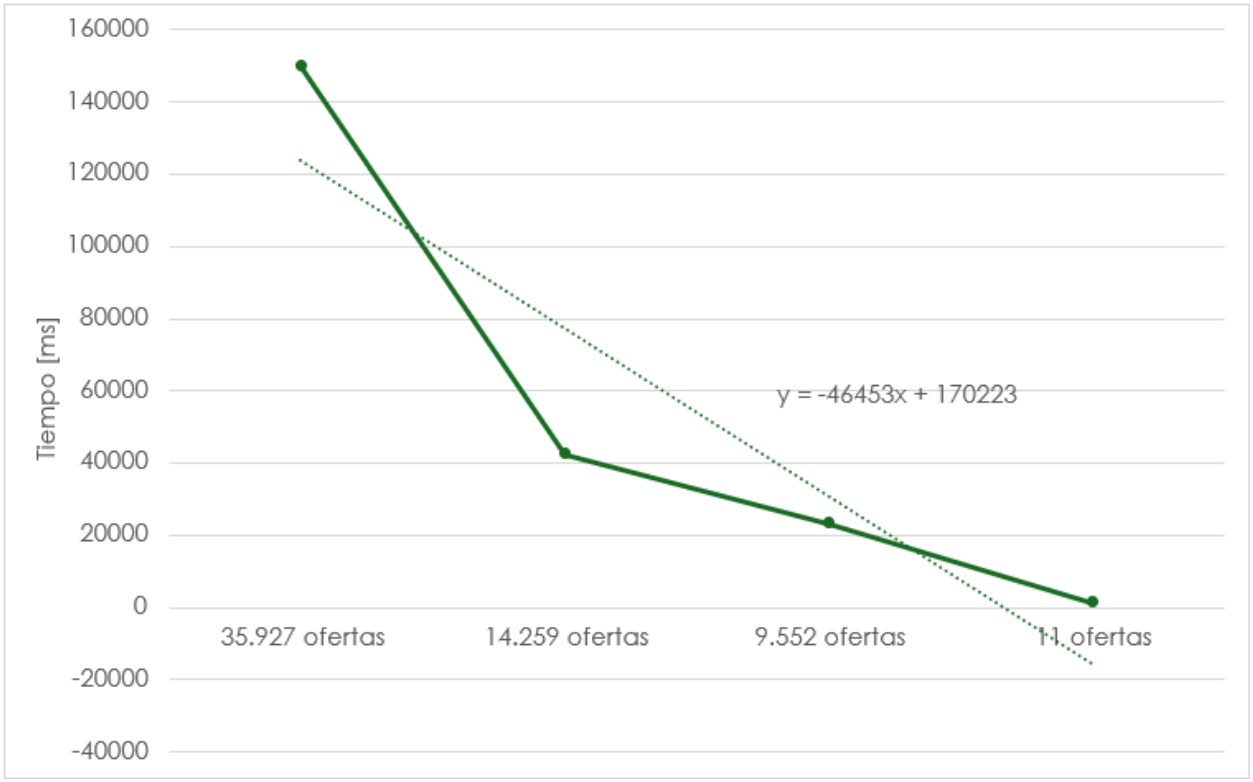
Tablas de datos

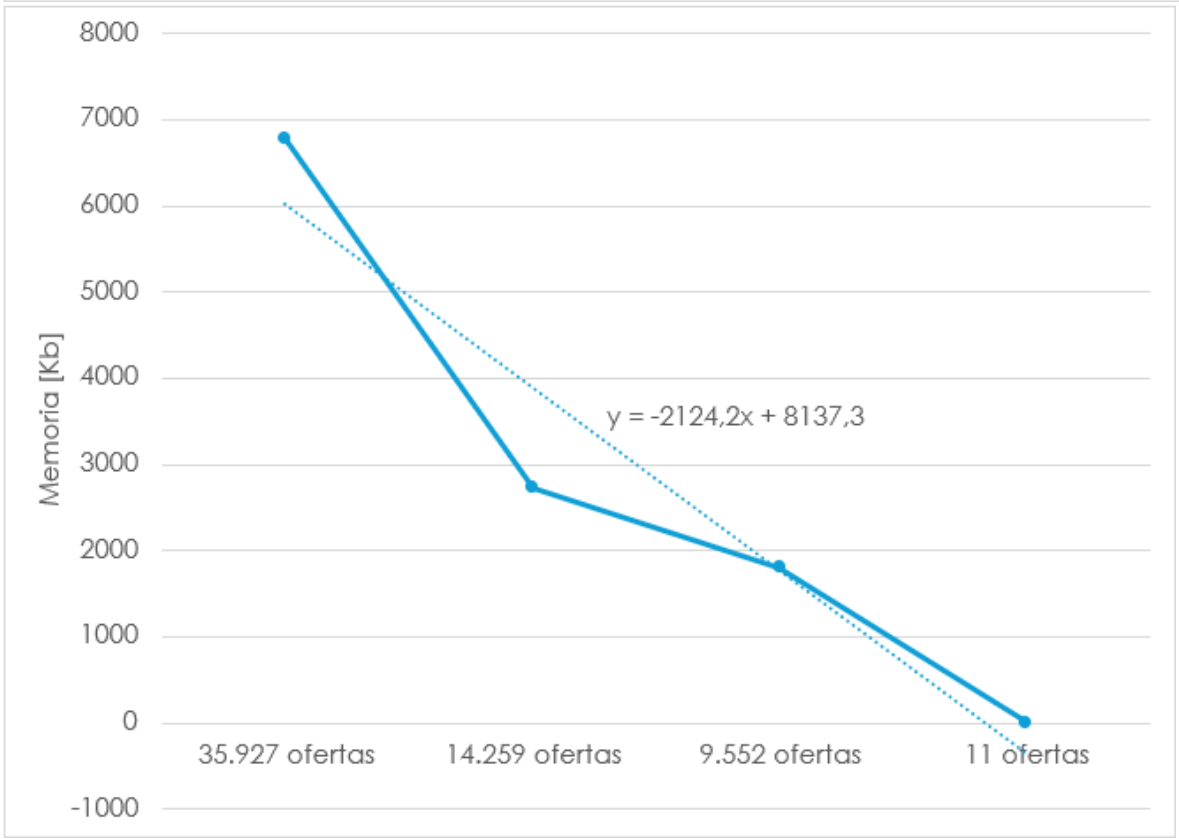
Las tablas con la recopilación de datos de las pruebas.

Entrada	**Salida**	**Memoria(Kb)**	**Tiempo(ms)**
Warszawa, 2021-02-12, 2023-10-10	35927 ofertas	6779,20	149625,290
Gdansk, 2021-02-12, 2023-10-10	14259 empresas	2717.443	42323.854
Katowice, 2021-02-12, 2023-10-10	9552	1809.102	23281,508
Paris, 2021-02-12, 2023-10-10	11 ofertas	1.234	1127.947

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

En este requerimiento se puede notar de que tanto la cantidad de memoria y el tiempo de respuesta depende de la cantidad de datos resultantes después de haberlos filtrado con los parámetros de entrada. Entre más ciudades cumplan con los parámetros más memoria y más tiempo se usará. Además, en la mayoría de los casos los valores de retorno serán mayores

Volver al índice

Requerimiento 6

Descripción

Entrada	Modelo donde se guardaron la información de las ofertas, número de ciudades que desea listar el usuario, nivel de experiencia y año.
Salidas	Lista de ciudades con mayor número de ofertas, tupla que contiene el total de ofertas y el total de compañías con al menos una oferta.
Implementado (Sí/No)	Si fue implementado por Juan David

Este requerimiento devuelve una cantidad de ciudades que tienen ofertas en un año determinado y un nivel de experiencia. Con estas ciudades se devuelven estadísticas como el número total de ofertas, empresas con al menos oferta, empresa con mayor número de ofertas y las ciudades con mayor y menor número de ofertas.

Para abordar este requerimiento primero se creó una tabla de hash para almacenar las ciudades y otra para almacenar las compañías. Luego se itero sobre cada una de las ofertas y se usó un condicional para saber si el año y la experiencia coincidían con la ingresada por el usuario. Con esto ya podíamos empezar a agregar las ciudades a nuestra tabla de hash; sin embargo, para esto es necesario saber si la llave de esa ciudad ya existe o no con el fin de no sobrescribir la información.

```
def req_6(catalog, num_cities, exp, year):
    """
    Función que soluciona el requerimiento 6
    """
    jobs = catalog['jobs']
    employment_types = catalog['employment_types']

    cities = mp.newMap(1000,
                       maptype='PROBING',
                       loadfactor=0.1)

    total_companies = mp.newMap(5000,
                                maptype='CHAINING',
                                loadfactor=10)

    total_offers = 0
```

Si la ciudad no ha sido agregada, se creará un diccionario el cual nos permitirá almacenar la información que necesitemos de cada ciudad. En este caso, el país, la cantidad de ofertas, el número de compañías,

entre otras. Y por otro lado, si la ciudad ya fue agregada, primero obtenemos el valor de la llave para luego poder manipular esa información agregando los datos necesarios.

```
for job in lt.iterator(jobs):

    if year in job['published_at']:

        if exp.lower() == job['experience_level'] or exp.lower() == 'indiferente':

            # Obtener el salario promedio de la oferta
            salaries = me.getValue(get_data(employment_types, job['id']))
            job['salary'] = get_job_salary(salaries)
            # Obtener la pareja de la ciudad en la tabla de ciudades, si no existe devuelve None
            city = get_data(cities, job['city'])

            if city:
                # Obtener el diccionario de la ciudad
                city = me.getValue(city)

                # Agregar 1 a la cantidad de ofertas
                city['offers'] += 1

                # Sumar el salario de la oferta a el salario promedio de la ciudad
                city['average_salary'] += job['salary']

                # Obtener la oferta con mejor y peor salario
                if 0 < job['salary'] > city['highest_salary']['salary']:
                    city['highest_salary'] = job
                if 0 < job['salary'] < city['lowest_salary']['salary']:
                    city['lowest_salary'] = job
```

```
                # Manipular el contador de empresas que tienen ofertas en la ciudad
                num = 0
                company = get_data(city['companies'], job['company_name'])
                if company:
                # Si la empresa ya fue agregada, entonces obtener el numero de ofertas de esa empresa.
                    num = me.getValue(company)

                # Agregar 1 a las ofertas totales de esa empresa
                mp.put(city['companies'], job['company_name'], num+1)

                # Obtener la mejor compañia de la ciudad y comparar el numero de ofertas
                if me.getValue(get_data(city['companies'], city['best_company'])) < num+1:
                    # Si es mayor, entonces actualizar 'best_company'
                    city['best_company'] = job['company_name']

            else:
                # Crear tabla de hash para almacenar las empresas que tienen ofertas en la ciudad
                companies = mp.newMap(100,
                                      maptype='CHAINING',
                                      loadfactor=10)
                mp.put(companies, job['company_name'], 1)

                # Crear diccionario para modelar los datos
                city = {
                    'name': job['city'],
                    'country': job['country_code'],
                    'offers': 1,
                    'average_salary': job['salary'],
                    'highest_salary': job,
                    'lowest_salary': job,
                    'companies': companies,
                    'best_company': job['company_name']
                }

                # Agregar la ciudad a la tabla de hash de las ciudades
                mp.put(cities, city['name'], city)

            # Agregar la empresa de esa oferta a la tabla que contara el total de empresas
            mp.put(total_companies, job['company_name'], None)
```

Finalmente, debemos ordenar las ciudades de mayor a menor número de ofertas y corroborar que el número de ciudades no supere la cantidad que quiere listar el usuario. Para esto, convertimos la tabla de hash en una lista con la función “ValueSet” e implementamos la función que nos devuelve una sublista. Para terminar de cuadrar la información de las ciudades con lo que nos pide el requerimiento iteramos sobre esas ciudades para establecer el número total de ofertas, el salario promedio, la mejor compañía y el número de compañías.

```
# Ordenar las ciudades de mayor a menor, por el numero de ofertas publicadas
cities = sort(mp.valueSet(cities), sort_req_6)

# Obtener solo las n primeras ciudades en caso de que hayan mas de la cantidad ingresada por el usuario
if data_size(cities, lt) > num_cities:
    cities = get_sublist(cities, 1, num_cities)

for city in lt.iterator(cities):
    # Sumar el numero total de ofertas publicadas
    total_offers += city['offers']

    # Obtener el salario promedio de la ciudad
    city['average_salary'] /= city['offers']

    # Obtener la mejor compañia con el conteo de ofertas
    city['best_company'] = get_data(city['companies'], city['best_company'])
    # Obtener cuantas empresas publicaron al menos una oferta
    city['companies'] = data_size(city['companies'], mp)

return cities, (total_offers, data_size(total_companies, mp))
```

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar tablas y valores	O(1)
Recorrer ofertas	O(n)
Condicionales	O(1)
Manipular diccionarios de ciudades	O(1)
Recorrer lista de ciudades	O(n)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas con el tamaño de archivo -large en un maquina con las siguientes capacidades.

Procesadores	11th Gen Intel® Core™
Memoria RAM (GB)	8 GB
Sistema operativo	Windows 11 Home

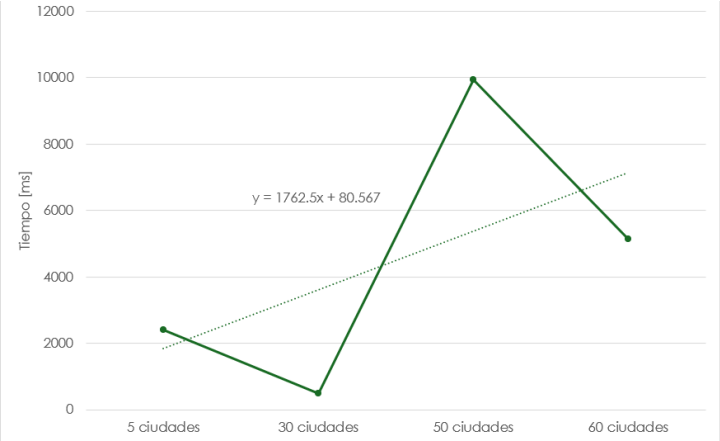
Tablas de datos

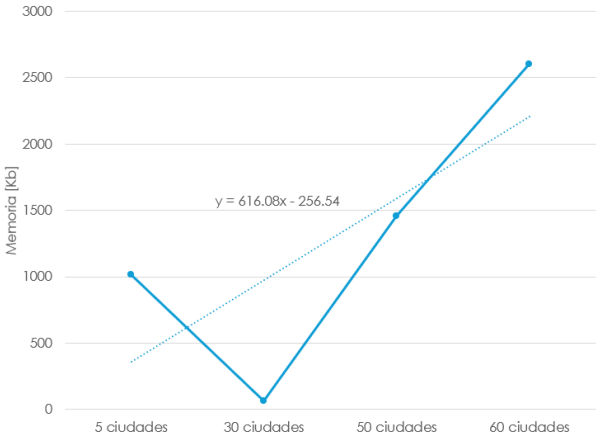
Las tablas con la recopilación de datos de las pruebas.

Entradas	Salida	Memoria [Kb]	Tiempo [ms]
5, senior, 2023	5 ciudades	1012.92	2395.264
30, junior, 2023	30 ciudades	61.75	498.395
50, mid, 2022	50 ciudades	1459.29	9925.933
60, indiferente, 2022	60 ciudades	2600.68	5127.788

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

En este requerimiento tanto su consumo de memoria como su tiempo de ejecución pueden variar ya que estas ciudades se componen de diferentes elementos que pueden alterar estas mediciones como la cantidad total de ofertas, el total de compañías, las ofertas por compañía, etc. Sin embargo, podemos observar que estas mediciones tienden a crecer a medida que encuentra mas ciudades que cumplan con los parámetros ingresados por el usuario.

Volver al índice

Requerimiento 7

Descripción

Asignación de variables:

```
def req_7(catalog, num_countries, year, month):
    """
    Función que soluciona el requerimiento 7
    """
#nombramiento de variables

    # Archivos

    skills = catalog["skills"]
    jobs = catalog["jobs"]
    jobs2 = lt.newList()
    jobs3 = lt.newList()

    #--------------------------

    # Filtro 1
    countries_map = mp.newMap(1000,
                              maptype='PROBING',
                              loadfactor=0.1)
    countries_dic = {}
    #1.0
    total_offers = 0
    2.0
    cities_map = mp.newMap(1000,
                              maptype='PROBING',
                              loadfactor=0.1)
    cities_dic = {}

    skills_senior_map = mp.newMap

    senior_list = lt.newList()
```

Recorrido en jobs y organización de países de mayor a menor en nuevo mapa.

```
#recorrido en jobs
for job in lt.iterator(jobs):

    if (str(year) in job["published_at"]) and (str(month) in job["published_at"]) :

        #avanze en dic para saber el numero de ofertas de cada pais. FILTRO3

        if mp.contains(countries_map, job["country_code"]):
            countries_dic["offers"] += 1
            mp.put(countries_map,job["country_code"],countries_dic)
        else:
            countries_dic["name"] = job["country_code"]
            countries_dic["offers"] = 1
            mp.put(countries_map,job["country_code"],countries_dic)

        #5.0 recorrido en mapa de skills
        sk = me.getValue(mp.get(skills,job["id"]))

        for i in (sk):
            if str(i) == "name":
                job["skill"] = sk[i]
            if str(i) == "level":
                job["skill_level"] = sk[i]
```

Entrada	Número de países a consultar, fecha limite inferior, fecha limite superior
Salidas	Ofertas filtradas, total de ofertas, numero de ciudades, países con mas ofertas y su conteo, ciudad con más ofertas y su conteo
Implementado (Sí/No)	Si se implementó, Samuel

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar tablas y valores	O(1)
Recorrer ofertas	O(n)
Condicionales	O(1)
Manipular diccionarios de ciudades	O(1)
Recorrer lista de ciudades	O(n)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas con el tamaño de archivo -large en un maquina con las siguientes capacidades.

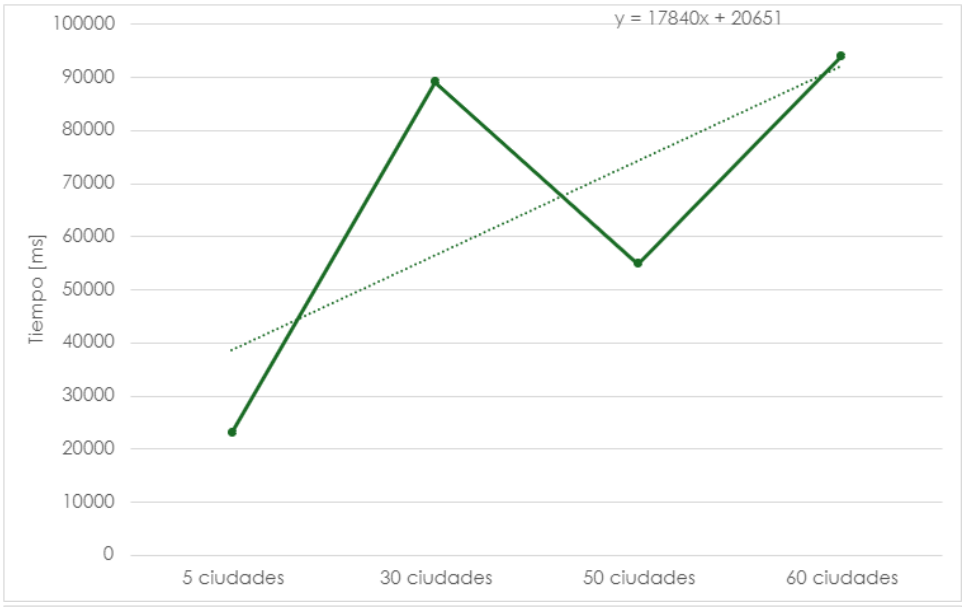
	maquina
procesador	Intel Core i7 -13700H
RAM (GB)	16 GB
Sistema operativo	Windows 11 home

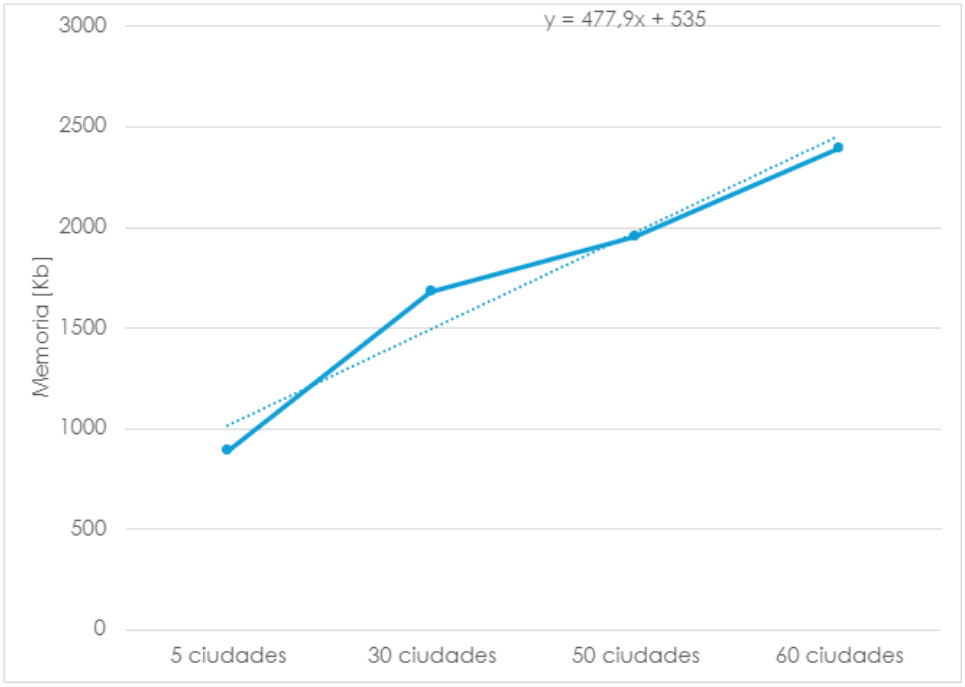
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Entrada	**Salida**	**Memoria (Kb)**	**Tiempo (ms)**
6 – 2020 - 07	38	892	23095
10 – 2023 - 07	61	1680	89095
18 – 2022 - 09	89	1953	54828
24 – 2022 - 02	101	2394	93984

Graficas





Análisis

Este requerimiento tiene una complejidad mayor a los individuales, pues requiere de un loop dentro de otro al tratarse de dos archivos en los cuales buscar informacion y comparar, sin emabrgo no consume una memoria ni un tiempo excesivo.

Volver al índice

Requerimiento 8

Descripción

Entrada	Experticia, divisa, fecha limite inferior, fecha límite superior
Salidas	Países filtrados, empresas que publicaron por lo menos 1 empresa, total ofertas, total países, total ciudades, ofertas con rango salarial, ofertas con rango fijo, ofertas sin salario
Implementado (Sí/No)	Si fue implementado por Andrés y Juan David

Este requerimiento devuelve los países que contengan ofertas dada una experiencia y un rango de fechas. Con esto podemos saber también los salarios ofertados en cada país los cuales nos permitirán ordenarlos de mayor a menor y así conocer cuál es el país que tiene una oferta salarial más alta, y cuál es el que tiene la más baja.

```
def req_8(catalog, exp, min_date, max_date):
    """
    Función que soluciona el requerimiento 8
    """
    jobs = catalog['jobs']
    employment_types = catalog['employment_types']
    skills = catalog['skills']

    total_companies = mp.newMap(100,
                                maptype='CHAINING',
                                loadfactor=10)
    total_cities = mp.newMap(100,
                                maptype='CHAINING',
                                loadfactor=10)
    countries = mp.newMap(100,
                          maptype='PROBING',
                          loadfactor=0.5)

    highest_country = lt.newList()
    lowest_country = lt.newList()

    offers_with_salary = 0
    offers_without_salary = 0
    total_offers = 0

    min_date, max_date = datetime.strptime(min_date, '%Y-%m-%dT%H:%M:%S.%fZ'), datetime.strptime(max_date, '%Y-%m-%dT%H:%M:%S.%fZ')
```

Primero, inicializamos todas las variables necesarias para almacenar los valores que nos pedía este requerimiento. Para almacenar los países utilizamos una tabla de hash de tipo probing. Luego, empezamos a iterar sobre cada una de las ofertas y hacer las comparaciones en cuanto al nivel de experiencia y el rango de fechas.

```
for job in lt.iterator(jobs):

    if exp.lower() == job['experience_level'] or exp.lower() == 'indiferente':

        # Obtener la fecha de la oferta para la comparacion
        current_date = datetime.strptime(job['published_at'], '%Y-%m-%dT%H:%M:%S.%fZ')

        if min_date < current_date < max_date:

            # Obtener el salario promedio de la oferta
            salaries = me.getValue(get_data(employment_types, job['id']))
            job['salary'] = get_job_salary(salaries)

            # Obtener la cantidad de habilidades requeridas de la oferta
            job_skills = me.getValue(get_data(skills, job['id']))
            job['skills'] = data_size(job_skills, lt)

            # Buscar la pareja del pais de la oferta
            country = mp.get(countries, job['country_code'])
```

Después, debemos empezar a manipular la información de cada país que nos será fundamental para responder a este requerimiento. Para esto, crearemos para cada país un diccionario lo que nos permitirá ir almacenando datos, operando sobre estos y obteniendo la información precisa de cada ciudad para poder imprimirla correctamente en la consola.

```
            # Crear estructura para modelar los datos
            country = {
                'code': job['country_code'],
                'average_salary': 0,
                'companies': companies,
                'cities': cities,
                'offers': 1,
                'offers_with_salary': 0,
                'highest_salary': 0,
                'lowest_salary': 1000000,
                'average_skills': job['skills']
            }

            if job['salary'] != 0:
                # Agregar al contador de ofertas con salario, tanto el general como el del pais
                offers_with_salary += 1
                country['offers_with_salary'] += 1
                # Sumar al promedio de los salarios del pais
                country['average_salary'] += job['salary']
                #Agregar al mayor y menor salario
                country['highest_salary'] = job['salary']
                country['lowest_salary'] = job['salary']

            else:
                # Sumar al contador de ofertas sin salario
                offers_without_salary += 1

            # Agregar el pais a la tabla de hash de los paises
            mp.put(countries, country['code'], country)
```

Por último, convertimos la tabla de ciudades en una lista para poder iterar sobre esta. Con esto logramos obtener los últimos datos y al final, usamos un criterio de ordenamiento que se encargue de ordenar los países de mayor a menor, por oferta salarial promedio. Ya con esto podemos devolver los datos que se requieren para implementar este requerimiento.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Inicializar variables	O(1)
Recorrer ofertas	O(n)
Condicionales	O(1)

Manipular o inicializar diccionario del país	O(1)
Agregar elementos a las estructuras de datos	O(1)
Recorrer lista de países	O(n)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas fueron realizadas con el tamaño de archivo -large en un maquina con las siguientes capacidades.

Procesadores	11th Gen Intel® Core™
Memoria RAM (GB)	8 GB
Sistema operativo	Windows 11 Home

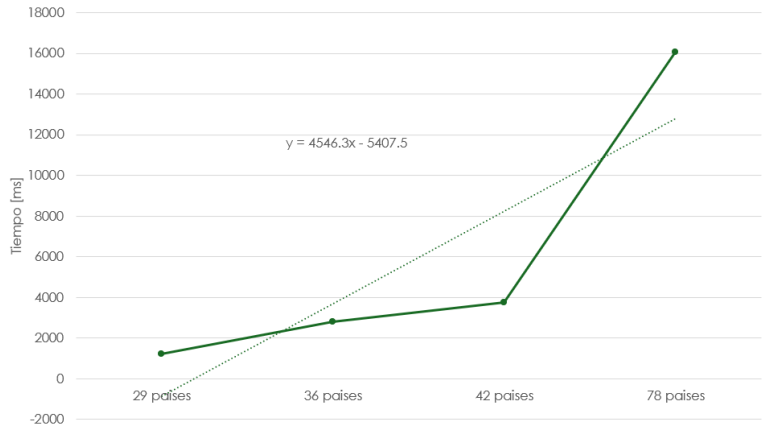
Tablas de datos

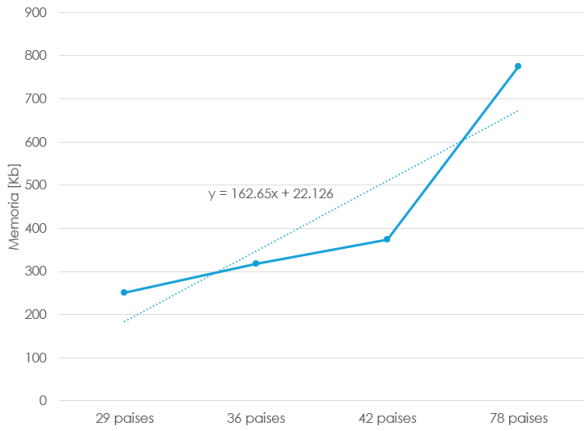
Las tablas con la recopilación de datos de las pruebas.

Entradas	Salida	Memoria [Kb]	Tiempo [ms]
junior, 2022-01-01, 2023-01-01	29 paises	250.43	1212.404
mid, 2023-05-20, 2023-12-20	36 paises	317.25	2816.845
senior, 2022-01-06, 2022-06-06	42 paises	373.48	3746.507
indiferente, 2022-01-01, 2022-12-31	78 paises	773.85	16056.742

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

En este requerimiento tanto su consumo de memoria como su tiempo de ejecución tienden a crecer a medida que encuentra más países que cumplan con los parámetros ingresados por el usuario. Esto se debe a que ocupa más tiempo recorriendo las ofertas y manipulando lo información. Y ocupa más espacio guardando toda la información requerida para implementar este requerimiento.

Volver al índice