ANÁLISIS DEL RETO 3

Ana Fadul Ramos, 202320756, a.fadul@uniandes.edu.co

Manuela Galarza, 202320796, m.galarza@uniandes.edu.co

Nicolás Parra, 202322257, n.parraz@uniandes.edu.co

Carga de Datos

Descripción

```python
def load_data(control, memflag=True):
    """
    Carga los datos del reto
    """
    start_time = get_time()
    # inicializa el proceso para medir memoria
    if memflag is True:
        tracemalloc.start()
        start_memory = get_memory()

    catalog = control['model']
    load_jobs(catalog)
    load_skills(catalog)
    load_employment_types(catalog)
    load_multilocations(catalog)

    stop_time = get_time()
    time = delta_time(start_time, stop_time)

    if memflag is True:
        stop_memory = get_memory()
        tracemalloc.stop()
        memory = delta_memory(stop_memory, start_memory)
        return time, memory

    else:
        return time
```

Carga de datos de los archivos de jobs, skills, employment types, y multilocations.

Entrada	Archivos de jobs, skills, employment types, y multilocations.
Salidas	El model organizado en 12 diferentes indices.
Implementado (Sí/No)	Sí

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
La lectura de todos los trabajos en los archivos	O(N)
Agregar el trabajo a las listas, y hashes.	O(1)
Agregar los trabajos a los diferentes árboles.	O(logN)
Hacer conversiones entre salarios, eliminar el anterior, y volver a agregar en el caso que sea menor.	O(1)
Retomar información con get value de hashes para agregarla a otras listas/mapas/arboles.	O(1)
Total	**O(N)**

Pruebas Realizadas

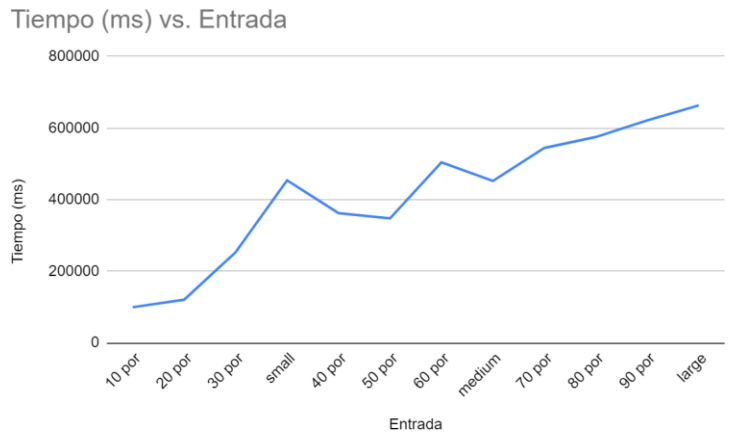
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el código de país PL, nivel de experiencia junior, y un número de 10 trabajos a visualizar.

Procesadores	**AMD Athlon Silver 3050U with Radeon Graphics 2.30 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Home Single Language

Tablas de datos

Entrada	Tiempo (ms)
10 por	99361
20 por	120470
30 por	253118
small	453953
40 por	362041
50 por	347490
60 por	504052
medium	452126
70 por	544390
80 por	574824
90 por	621343
large	663098

Graficas



Análisis

A pesar de tener una complejidad teórica de O(N), considerando N como el número de trabajos en los csv's de jobs, emplyment_types, skills y multilocations. Esto se toma un tiempo increiblemente extenuante considerando los 12 indices que se tienen para hacer los requerimientos lo más cortos posible. En estos indices se usan comandos como addLast y newList para arrays; y get_value, put, y createHash que tienen una complejidad de O(1). Además de los comandos de árboles getValue y add, que tienen una complejidad de O(logN). Aunque existen divergencias entre el gráfico de tiempo y la complejidad teórica, las pruebas de tiempo fueron efectivas, incrementando con respecto al incremento en el tamaño de datos ingresados. Sin embargo, diferencias entre los datos obtenidos pueden ser por la existencia de un elemento de suerte al ingresar y buscar información en hash tables (esto debido a nuestra implementación en linear probing).

Requerimiento 1

Descripción

```python
def req_1(data_structs, initialDate, finalDate):
    """
    Función que soluciona el requerimiento 1
    """
    lst = bst.keys(data_structs['dateIndex'], initialDate, finalDate, compareDates)
    ofertas= nf.newList()
    for i in range(nf.get_size(lst)):
        lista= nf.getElement(lst,i)['lstjobs']
        for j in range(nf.get_size(lista)) :
            oferta= nf.getElement(lista,j)
            nf.addLast(ofertas,oferta)

    totcrimes = nf.get_size(ofertas)
    Sorts.TimSort(ofertas,fecha_salary)
    if totcrimes>10:
        tabular= nf.first_last(ofertas)
    else:
        tabular= ofertas
    para_tabular= []
    for oferta in tabular['elements']:
        fila= []
        fila.append(oferta["published_at"])
        fila.append(oferta['title'] )
        fila.append(oferta['company_name'] )
        fila.append(oferta['experience_level'] )
        fila.append(oferta['country_code'] )
        fila.append(oferta['city'] )
        fila.append(oferta['company_size'] )
        fila.append(oferta['workplace_type'] )
        fila.append(ht.get_value(data_structs['id_employment_type'],oferta['id'])[1]['currency_salary'])
        fila.append(oferta.get('skill',habilidades(data_structs,oferta['id'])))
        para_tabular.append(fila)

    headers = ['Fecha de publicación','Título de la oferta', 'Nombre de la empresa','Nivel','Pais','Ciudad' ,

    return totcrimes, (para_tabular, headers)
```

Este requerimiento se encarga de dar las ofertas laborales publicadas dentro de un rango de fechas.

Entrada	La carga de datos, y la fecha mínima y máxima para evaluar.
Salidas	Total de trabajos que cumplen con estas condiciones, y una lista con la información de cada uno.
Implementado (Sí/No)	Sí

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Se utiliza un método bst.keys() para obtener las ofertas publicadas en el rango dado.	O(log n)
Se recorre la lista para obtener las ofertas dentro de cada fecha y agregarlas a una nueva lista.	O(n)
Se utiliza el algoritmo TimSort para ordenar la lista de ofertas según la fecha de publicación (si son iguales se ordena por salario).	O(n log n)
Si el tamaño de la lista de ofertas es superior a 10 se eligen las primeras y últimas 5 ofertas.	O(1)
Se recorre la lista de ofertas y se ordenan en un tabulate con la información solicitada.	O(m), donde m es el número de ofertas seleccionadas.
Total	**O(n log n)**

Pruebas Realizadas

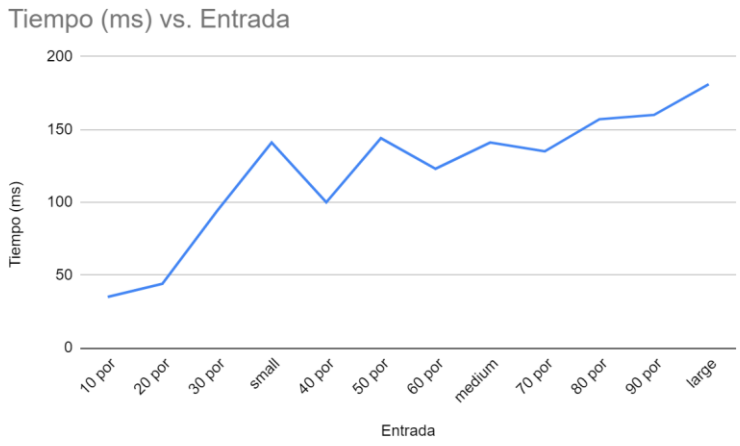
Las pruebas realizadas se realizaron en una maquina con estas especificaciones. Los datos de entrada fueron el rango 2022-04-01 y 2022-05-01.

Procesadores	**AMD Athlon Silver 3050U with Radeon Graphics 2.30 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Home Single Language

Tablas de datos

Entrada	**Tiempo (ms)**	**Memoria**
10 por	35	4.7
20 por	44	7.9
30 por	94	10.2
small	141	14.3
40 por	100	16.7
50 por	144	17.9
60 por	123	18.1
medium	141	19.3
70 por	135	19.8
80 por	157	22.5
90 por	160	23.4
large	181	24.2

Graficas



Análisis

A pesar de obtener en el retorno un ArrayList y una variable, que debería dar una complejidad constante, la implementación de este requerimiento tiene una complejidad de O(N*LogN). Esto debido a que al ir a través del for loop de los keys del árbol, después se tiene que organizar la lista con la información de todos los trabajos en esos nodos, esta lista teniendo ir a través de un TimSort, dejándo una complejidad de O(N*logN). Aunque existen divergencias entre el gráfico de tiempo y la complejidad teórica, las pruebas de tiempo fueron efectivas, incrementando proporcionalmente con respecto al incremento en el tamaño de datos ingresados. Es también relevante mencionar que el incremento en memoria, como se puede ver en el gráfico, es proporcional a como incrementa el tamaño del ArrayList al cual se le agregan la información de los trabajos que siguen los lineamientos delimitados por el usuario. Son anomalos los resultados del 40% en comparación con los otros, esto probablemente por un tema de probabilidad al crear los hashes en la carga.

Requerimiento 2

Descripción

```
def req_2(data_structs, initialDate, finalDate):
    """
    Función que soluciona el requerimiento 2
    """
    lst_keys = bst_rn.keys(data_structs['salario_min'], initialDate, finalDate, compare)
    totcrimes = nf.get_size(lst_keys)
    counter_real = 0
    total = nf.newList()
    for i in range(0, totcrimes):
        lugar = bst_rn.get(data_structs['salario_min'], lst_keys['elements'][i])['value']
        lugar = ht.valueSet(lugar)
        counter_real += data_size(lugar)
        item = Sorts.MergeSort(lugar, Sorts.mayor_menor_published)
        nf.extend(total, item)
    return counter_real, total
```

Este requerimiento se encarga de conocer las ofertas públicas en un rango de salario mínimo ofertado, tomando en cuenta a los dólares como moneda de referencia.

Entrada	La carga de datos, y salario mínimo y máximo a revisar.

Salidas	El número de ofertas que están en este rango, y la información de las 5 primeras y últimas ofertas organizadas ascendientemente por salario mínimo, y si son iguales los salarios, por fecha más reciente.
Implementado (Sí/No)	Sí

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Keys_Range del arbol de salarios	O(logN)
Crear variables base de conteos, y listas	O(1)
For loop a través del keys_range	O(N)
Merge Sort por fechas de la lista dentro de cada arbol	O(K*N*logK) -> K es el número de elementos en cada lista
Extend de la lista completa	O(1)
Total	**O(K*N*logK)**

Pruebas Realizadas

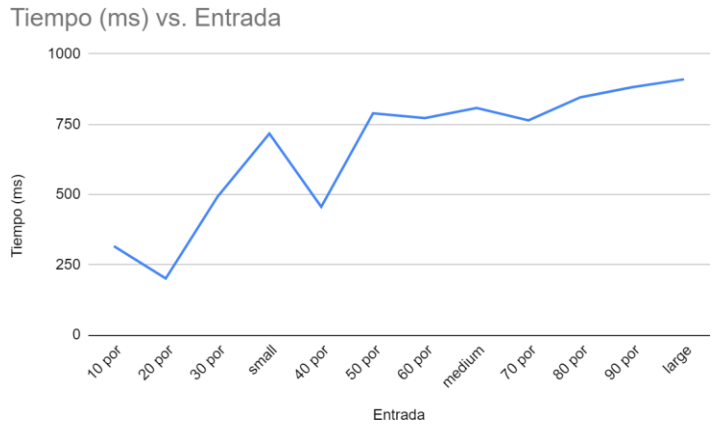
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el rango entre 3000 y 4000.

Procesadores	**AMD Athlon Silver 3050U with Radeon Graphics 2.30 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Home Single Language

Tablas de datos

Entrada	**Tiempo (ms)**	**Memoria**
10 por	316	53.2
20 por	201	99.4
30 por	492	163.2
small	717	200.1
40 por	456	205.3
50 por	789	254.3
60 por	772	280.9
medium	808	301.4
70 por	764	305
80 por	846	349.7
90 por	882	358.2
large	910	371.3

Graficas



Análisis

A pesar de obtener en el retorno un ArrayList y una variable, que debería dar una complejidad constante, la implementación de este requerimiento tiene una complejidad de O(K*N*LogK). Esto debido a que al ir a través del for loop de las keys en el árbol, después se tiene que organizar la lista con la información de todos los trabajos en esos nodos, esta lista teniendo un tamaño representado por la variable K en el peor caso, y debido a que se utiliza un MergeSort sobre esta, tenemos una complejidad de O(N*K*logN). Aunque existen divergencias entre el gráfico de tiempo y la complejidad teórica, las pruebas de tiempo fueron efectivas, incrementando proporcionalmente con respecto al incremento en el tamaño de datos ingresados. Es también relevante mencionar que el incremento en memoria, como se puede ver en el gráfico, es proporcional a como incrementa el tamaño del ArrayList al cual se le agregan la información de los trabajos que siguen los lineamientos delimitados por el usuario. Son anomalos los resultados del 40% en comparación con los otros, esto probablemente por un tema de probabilidad al crear los hashes en la carga.

Requerimiento 3

Descripción

```python
def req_3(data_structs, pais, experticia):
    """
    Función que soluciona el requerimiento 3
    """
    lista = data_structs['pais']
    lista = ht.get_value(lista, pais)[1]
    lista = ht.get_value(lista, experticia)[1]
    nueva = nf.addLast() #conversión de hash a lista - temas de tiempo en la carga (remove)
    for i in lista:
        nf.addLast(i[1])
    lista = Sorts.MergeSort(lista, Sorts.published_salario)
    return data_size(lista), lista
```

Este requerimiento se encarga de consultar las N ofertas más recientes para un país y que requieran un nivel de experiencia especifico.

Entrada	La carga de datos, el número de ofertas laborales a consultar, el nivel de experiencia, y el país para consultar.
Salidas	El total de ofertas que aplican bajo estas métricas y una lista con las N primeras ofertas organizadas por fecha de publicación y salario.
Implementado (Sí/No)	Sí, por Ana Fadul

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Get_Value de la información dentro de los hashes de país y experiencia.	O(1)
Value Set del Hash trabajos	O(N)
Merge Sort de los trabajos por publicidad y salario	O(N*logN)
Return del size de la lista y la lista	O(1)
Total	***O(N*logN)***

Pruebas Realizadas

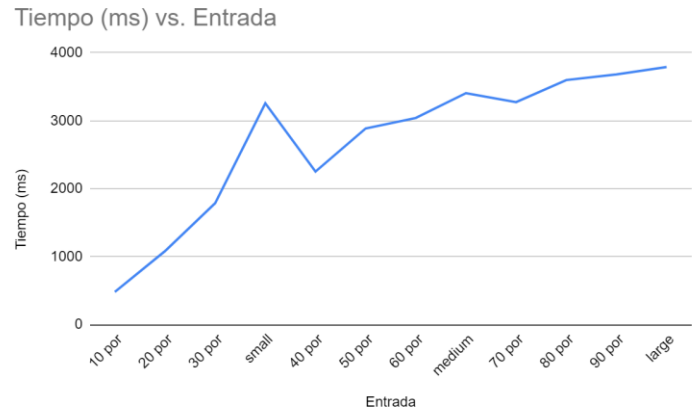
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el código de país PL, nivel de experiencia junior, y un número de 5 trabajos a visualizar.

Procesadores	**AMD Athlon Silver 3050U with Radeon Graphics 2.30 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Home Single Language

Tablas de datos

Entrada	Tiempo (ms)	Memoria
10 por	480	170.1
20 por	1081	343.9
30 por	1786	471.3
small	3256	569.7
40 por	2251	702.1
50 por	2885	834.2
60 por	3040	923.7
medium	3405	1014.4
70 por	3273	1087.3
80 por	3598	1103.2
90 por	3680	1173.5
large	3790	1243.4

Graficas



Análisis

A pesar de obtener en el retorno un ArrayList y una variable, que debería dar una complejidad constante, la implementación de este requerimiento tiene una complejidad de O(N*LogN). Esto debido a que al ir a través del for loop de los values del árbol, después se tiene que organizar la lista con la información de todos los trabajos en esos nodos, esta lista teniendo ir a través de un MergeSort, dejando una complejidad de O(N*logN). Aunque existen divergencias entre el gráfico de tiempo y la complejidad teórica, las pruebas de tiempo fueron efectivas, incrementando proporcionalmente con respecto al incremento en el tamaño de datos ingresados. Es también relevante mencionar que el incremento en memoria, como se puede ver en el gráfico, es proporcional a como incrementa el tamaño del ArrayList al cual se le agregan la información de los trabajos que siguen los lineamientos delimitados por el usuario. Son anomalos los resultados del 40% en comparación con los otros, esto probablemente por un tema de probabilidad al crear los hashes en la carga.

Requerimiento 4

Descripción

```
def req_4(data_structs,ciudad,ubicacion):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    ofertas = data_structs['ciudad']
    ofertas = ht.get_value(ofertas, ciudad)[1]
    ofertas = ht.get_value(ofertas, ubicacion)[1]
    ofertas = ht.valueSet(ofertas)
    ofertas = Sorts.TimSort(ofertas, Sorts.published_salario)
    return data_size(ofertas), ofertas
```

Este requerimiento se encarga de

Entrada	El número de ofertas laborales para consulta, nombre de la ciudad y tipo de ubicación.
Salidas	Número total de ofertas publicadas con los requerimientos especificados, n ofertas publicadas más recientes con la información especificada.
Implementado (Sí/No)	Si, por Nicolás Parra

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Get_Value de la información en los hashes de ciudad y tipo de ubicación.	O(1)
Value Set del Hash de los trabajos.	O(N)
TimSort de los trabajos por publicidad y salario.	O(N*logN)
Return del size de la lista y la información en ella.	O(1)
Total	***O(N*logN)***

Pruebas Realizadas

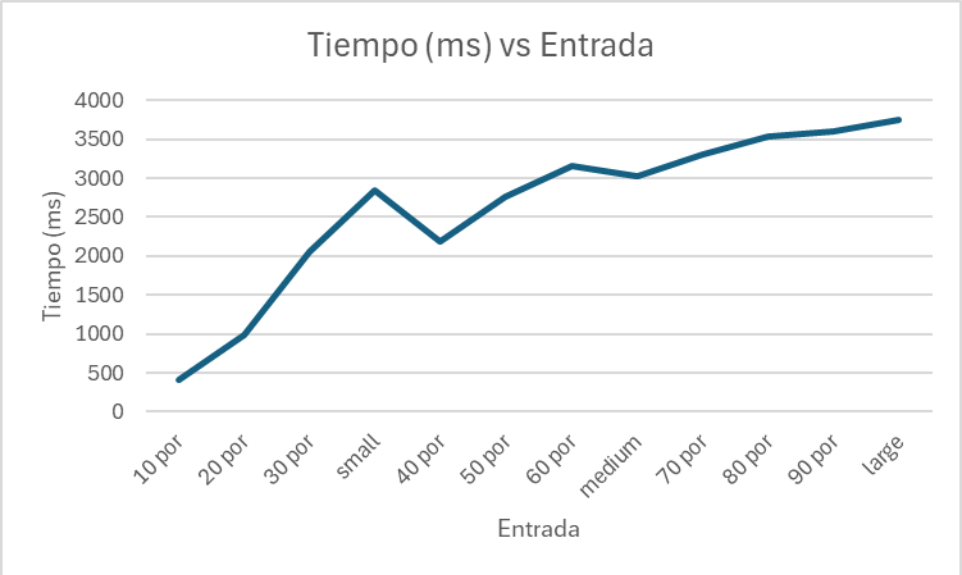
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron 10 ofertas laborales, ciudad Warszawa, tipo remote.

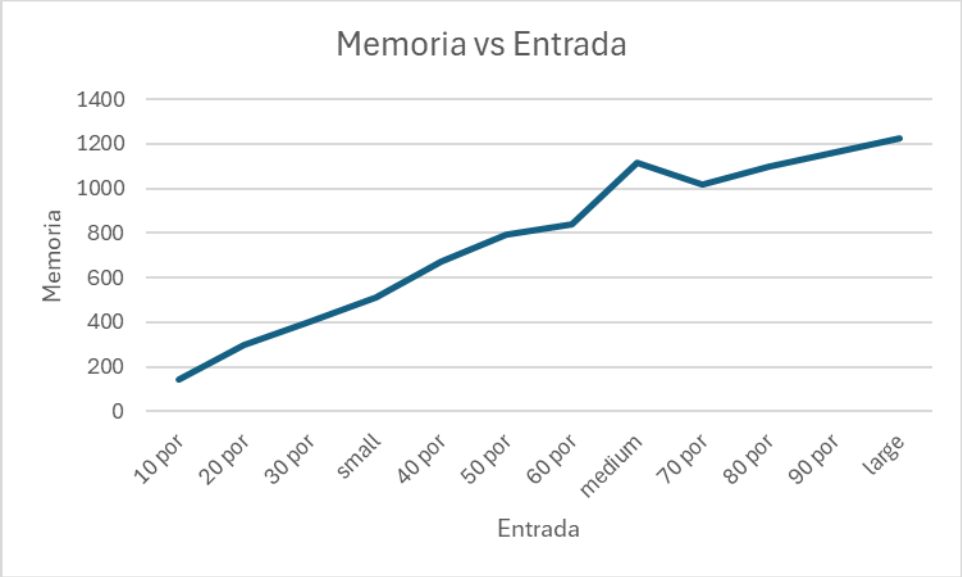
Procesadores	**AMD Athlon Silver 3050U with Radeon Graphics 2.30 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Home Single Language

Tablas de datos

Entrada	**Tiempo (ms)**	**Memoria**
10 por	410	141.3
20 por	984	298.6
30 por	2058	402.5
small	2841	512.8
40 por	2192	674.1
50 por	2763	793.2
60 por	3158	841.9
medium	3022	1118.2
70 por	3310	1017.8
80 por	3529	1096.3
90 por	3597	1162.7
large	3754	1224.9

Graficas





Análisis

A pesar de obtener en el retorno un ArrayList y una variable, que debería dar una complejidad constante, la implementación de este requerimiento tiene una complejidad de O(N*LogN). Este cambio en la complejidad se debe a que en una parte del algoritmo es necesario retornar todas las llaves de la lista y este proceso tiene una complejidad de O(N). Además, se requiere hacer un TimSort de la lista de

ofertas por fecha de publicación y filtrar la información, lo que genera una complejidad de (N*log(N)), dando así una complejidad final de O(N*log(N)). El filtrado de la información en las tablas de hash es constante y es por eso por lo que no se toma en cuenta para la complejidad final del requerimiento.

Requerimiento 5

Descripción

```
def req_5(data_structs,n,tamanio_low,tamanio_high,habilidad,nivel_low,nivel_high):
    """
    Función que soluciona el requerimiento 5
    """
    #Carga de datos: updateSizeIndex, addLevelTree, newSkillIndex, newLevelEntry, newSkillEntry
    arbol=data_structs['sizeIndex']
    en_rango= bst.keys(arbol,tamanio_low,tamanio_high,compare)
    arbol_nivel= nf.newList()
    lsthabilidad= nf.newList()
    lstfinal= nf.newList()
    for i in range(nf.get_size(en_rango)):
        fecha= nf.getElement(en_rango,i)
        mapa= fecha['skillIndex']
        la_habilidad= ht.get_value(mapa,habilidad)
        if la_habilidad!= None:
            nf.addLast(arbol_nivel,la_habilidad[1]['levelTree'])
            for i in range(nf.get_size(la_habilidad[1]['lstjobs'])):
                oferta= nf.getElement(la_habilidad[1]['lstjobs'],i)
                nf.addLast(lsthabilidad,oferta)
    total_ofertas= nf.get_size(lsthabilidad)
    for i in range(nf.get_size(arbol_nivel)):
        el_arbol= nf.getElement(arbol_nivel,i)
        nivel_rango= bst.keys(el_arbol,nivel_low,nivel_high,compare)
        for i in range(nf.get_size(nivel_rango)):
            lst= nf.getElement(nivel_rango,i)['lstjobs']
            for j in range(nf.get_size(lst)):
                oferta= nf.getElement(lst,j)
                nf.addLast(lstfinal,oferta)
    Sorts.TimSort(lstfinal,fecha_salary)
```

Este requerimiento se encarga de consultar las N ofertas más antiguas con un nivel mínimo y máximo para una habilidad solicitada para empresas en un rango de tamaño.

Entrada	El número de ofertas laborales para consulta. El límite inferior y superior del tamaño de la compañía. El nombre de la habilidad solicitada. El límite inferior y superior del nivel de la habilidad
Salidas	El número total de ofertas laborales publicadas para las compañías que tengan un tamaño en un rango y que requieran una habilidad específica. Las N ofertas laborales publicadas más antiguas que cumplan con las condiciones especificadas. Cada una con cierta información.
Implementado (Sí/No)	Si, por Manuela Galarza

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Se utiliza el método bst.keys() para obtener las llaves dentro del rango dado en el árbol de tamaños.	O(log n)

Se recorren los nodos dentro del rango y se obtienen las ofertas que contienen la habilidad requerida (mapa). Se recorren las ofertas asociadas a la habilidad requerida y se agregan a una lista.	O(m * n), donde m es el número de fechas dentro del rango y n es el número máximo de ofertas asociadas a una habilidad
Se recorren los árboles de nivel obtenidos de los mapas el paso anterior y se obtienen las llaves (niveles) dentro del rango especificado. Y la lista de ofertas asociadas a cada nivel.	O(p * q), donde p es el número de árboles de nivel obtenidos, y q es el número máximo de ofertas asociadas a un nivel. No se tiene en cuenta número de claves dentro del rango de niveles porque máximo es 5.
Se utiliza el algoritmo TimSort para ordenar la lista de ofertas por fecha.	O(r log r), donde r es el número de ofertas en la lista final.
Se recorre la lista seleccionada de ofertas y se forma una estructura de datos para tabular.	O(t), donde t es el numero ingresado de ofertas para consulta, o r si hay menos ofertas de las solicitadas.
Total	***O(m * n + p * q + r log r)***

Pruebas Realizadas

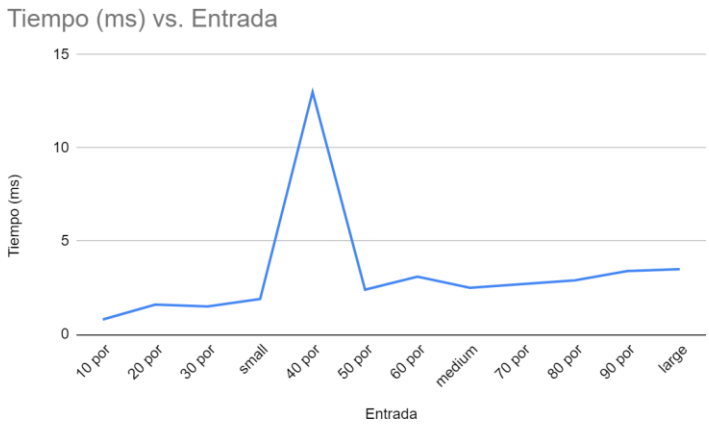
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron 5 trabajos, con una compañía entre 200 y 500, y habilidad GO entre 1 y 5.

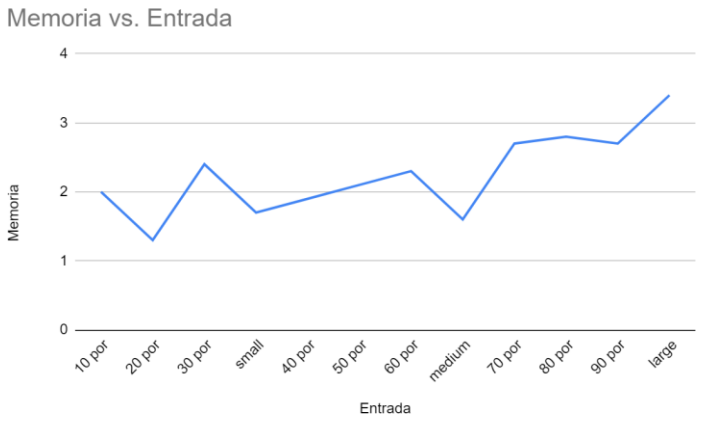
Procesadores	**AMD Athlon Silver 3050U with Radeon Graphics 2.30 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Home Single Language

Tablas de datos

Entrada	**Tiempo (ms)**	**Memoria**
10 por	0.8	2.
20 por	1.6	1.3
30 por	1.5	2.4
small	1.9	1.7
40 por	13	1.9
50 por	2.4	2.1
60 por	3.1	2.3
medium	2.5	1.6
70 por	2.7	2.7
80 por	2.9	2.8
90 por	3.4	2.7
large	3.5	3.4

Graficas


Tiempo (ms) vs. Entrada
15
10
5
0
Tiempo (ms)
10 por
20 por
30 por
small
40 por
50 por
60 por
medium
70 por
80 por
90 por
large
Entrada


Memoria vs. Entrada
4
3
2
1
0
Memoria
10 por
20 por
30 por
small
40 por
50 por
60 por
medium
70 por
80 por
90 por
large
Entrada

Análisis

Requerimiento 6

Descripción

```
def req_6(data_structs,n,initialDate,finalDate,salMin,salMax):
    """
    Función que soluciona el requerimiento 6
    """
    mapa_ciudades= ht.hash_table(200,1factor)
    en_rango= bst.keys(data_structs['dateIndex'], initialDate, finalDate, compareDates)
    lstfinal= nf.newList()
    for i in range(nf.get_size(en_rango)):
        fecha= nf.getElement(en_rango,i)
        arbol2= fecha['salaryTree']
        en_salario=bst.keys(arbol2,salMin, salMax, compare)
        for j in range(nf.get_size(en_salario)):
            salario= nf.getElement(en_salario,j)
            oferta= salario['lstjobs']
            for k in range(nf.get_size(oferta)):
                el_salario= nf.getElement(oferta,k)
                nf.addLast(lstfinal,el_salario)
                ciudad=el_salario['city']
                if ht.get_value(mapa_ciudades,ciudad):
                    e = (ht.get_value(mapa_ciudades,ciudad))[1]
                else:
                    e = nf.newList()
                    ht.put(mapa_ciudades, ciudad, e)
                nf.addLast(e,el_salario)
```

Este requerimiento se encarga de

Entrada	Numero de ciudades a consultar, fecha inicial del periodo, fecha final del periodo, salario mínimo ofertado, salario máximo ofertado.
Salidas	Número total de ofertas publicadas con los requerimientos especificados, numero total de ciudades que cumplen con las especificaciones, n ciudades que cumplen con las especificaciones ordenadas alfabéticamente, para cada una de las ofertas de la ciudad con mayor cantidad de ofertas que cumplen con las condiciones imprimir la información especificada.
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
KeySet del arbol de Fechas	O(N)

Búsqueda por rangos con base a una fecha inicial y una fecha final	O (log N)
Búsqueda por rangos con base en un salario mínimo inferior y un salario mínimo superior.	O (log N)
For loop a través del rango de fechas, salarios, y dentro de ese todos los trabajos guardados dentro del valor.	O(N^3)
Conteos, get de hashes para inforamación, crear variables, y agregar a listas.	O(1)
Hacer size para hallar la cantidad de ofertas publicadas por ciudad	O (1)
KeySet del hash de ciudades	O (N)
Merge Sort a través de la lista de ciudades	O(N*logN)
For loops a través de lista de ciudades y retomar información en hashes de ciudades	O(N)
Total	***O (N^3)***

Pruebas Realizadas

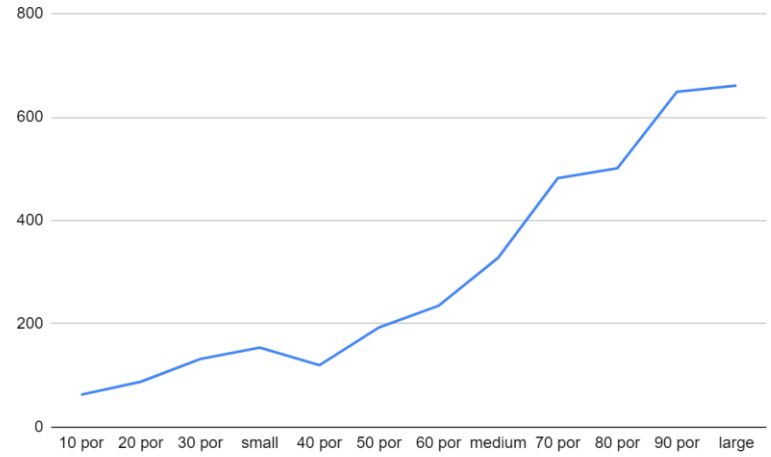
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el código de país PL, nivel de experiencia junior, y un número de 10 trabajos a visualizar.

Procesadores	**AMD Athlon Silver 3050U with Radeon Graphics 2.30 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Home Single Language

Tablas de datos

Entrada	**Tiempo (ms)**	**Memoria**
10 por	63	7
20 por	88	8
30 por	132	10
small	154	8
40 por	120	11
50 por	193	12
60 por	235	15
medium	328	14
70 por	482	14
80 por	501	15
90 por	649	16
large	661	15

Graficas



Análisis

A pesar de obtener en el retorno un ArrayList, una variable, que debería dar una complejidad constante, la implementación de este requerimiento tiene una complejidad de O(N^3). Esto debido a las búsquedas que se deben hacer en los keys_set de los rangos de fecha y salarios, y después a través de los trabajos dentro de este rango. Aunque existen divergencias entre el gráfico de tiempo y la complejidad teórica, las pruebas de tiempo fueron efectivas, incrementando proporcionalmente con respecto al incremento en el tamaño de datos ingresados. Es también relevante mencionar que el incremento en memoria, como se puede ver en el gráfico, es proporcional a como incrementa el tamaño del ArrayList y hash los cuales se les está agregando la información de los trabajos que siguen los lineamientos delimitados por el usuario. Son anomalos los resultados del 40% en comparación con los otros, esto probablemente por un tema de probabilidad al crear los hashes en la carga.

Requerimiento 7

Descripción

```
def req_7(data_structs,anio,pais,conteo):
    """
    Función que soluciona el requerimiento 7
    """
    el_anio= ht.get_value(data_structs['anio'],anio)[1]
    lst_anio= el_anio['lstjobs']
    mapa_paises= el_anio['countryMap']
    total_anio= nf.get_size(lst_anio)
    el_pais=ht.get_value(mapa_paises,pais)[1]
    lst_pais= el_pais['lstjobs']
    total_histograma= nf.get_size(lst_pais)
    if conteo=='experticia':
        propiedad= 'levelMap'
    elif conteo== 'habilidad':
        propiedad= 'skillMap'
    else:
        propiedad= 'locationMap'
    el_mapa= el_pais[propiedad]
    llaves= ht.keySet(el_mapa)
    frecuencias= nf.newList()
```

Este requerimiento se encarga de contabilizar las ofertas laborales publicadas para un país y un año específico según alguna propiedad de interés como lo son el nivel de experticia requerido, el tipo de ubicación del trabajo, o habilidad específica

Entrada	El año y país para la consulta. La propiedad de conteo (experticia, ubicación, o habilidad). El número de segmentos o casillas (bins) en los que se divide el histograma
Salidas	El número de ofertas laborales publicadas dentro del periodo anual relevante. El número de ofertas laborales publicadas utilizados para crear el histograma de la propiedad. El histograma con la distribución de las ofertas laborales publicadas según la propiedad. Listado de las ofertas laborales publicadas que cumplen las condiciones de conteo para el histograma. Cada uno de los eventos en la consulta con cierta información.
Implementado (Sí/No)	Si

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Se obtienen las listas de ofertas correspondientes al año y al país especificados.	O(1)
Dependiendo del valor de conteo, se selecciona la propiedad correspondiente (levelMap, skillMap o locationMap).	O(1)

Se recorre el mapa de la propiedad seleccionada para obtener las listas de ofertas asociadas a cada clave.	O(m*n), donde el tamaño del mapa es m y el número máximo de ofertas asociadas a una clave es n
Se recorre la lista seleccionada de ofertas y se forma una estructura de datos para tabular.	O(k), donde k es el tamaño de la lista de ofertas seleccionadas.
Total	***O(m * n)***

Pruebas Realizadas

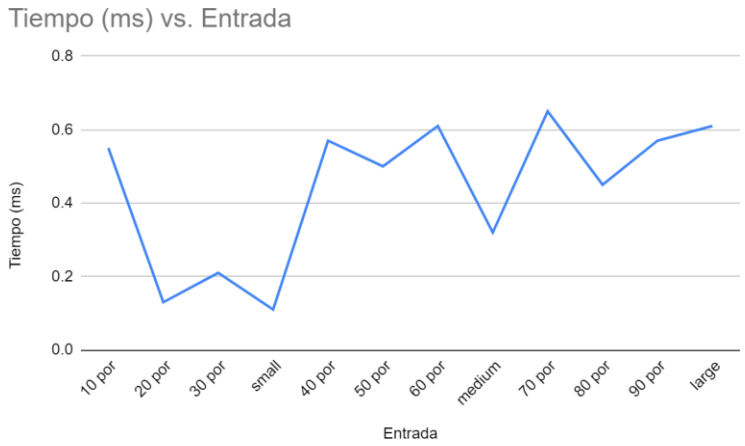
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron 3 bins, en el año 2022, con el código de país US, y sobre experticia.

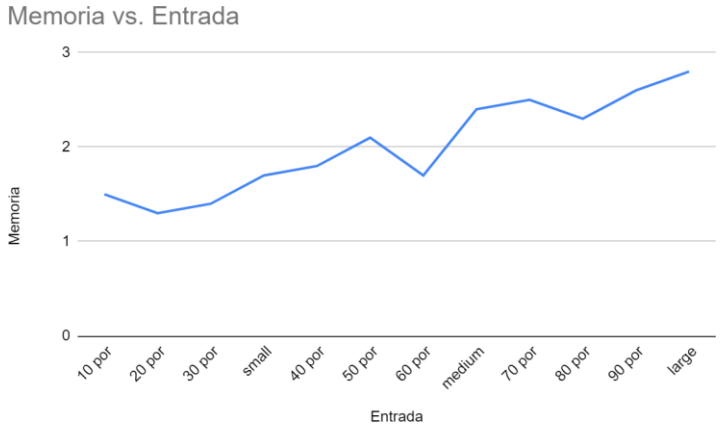
Procesadores	**AMD Athlon Silver 3050U with Radeon Graphics 2.30 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11 Home Single Language

Tablas de datos

Entrada	**Tiempo (ms)**	**Memoria**
10 por	0.55	1.5
20 por	0.13	1.3
30 por	0.21	1.4
small	0.11	1.7
40 por	0.57	1.8
50 por	0.5	2.1
60 por	0.61	1.7
medium	0.32	2.4
70 por	0.65	2.5
80 por	0.45	2.3
90 por	0.57	2.6
large	0.61	2.8

Graficas





Análisis

A pesar de obtener en el retorno un ArrayList, una variable y una gráfica de la libreria matplotpy, que debería dar una complejidad constante, la implementación de este requerimiento tiene una complejidad de O(M*N). Esto debido a que se debe recorrer con un for loop la lista de mapas, y después recorrer todas las ofertas M que están asociadas con este. Aunque existen divergencias entre el gráfico de tiempo y la complejidad teórica, las pruebas de tiempo fueron efectivas, incrementando proporcionalmente con respecto al incremento en el tamaño de datos ingresados. Es también relevante mencionar que el incremento en memoria, como se puede ver en el gráfico, es proporcional a como incrementa el tamaño del ArrayList al cual se le agregan la información de los trabajos que siguen los lineamientos delimitados por el usuario. Son anomalos los resultados del 40% en comparación con los otros, esto probablemente por un tema de probabilidad al crear los hashes en la carga.