ANÁLISIS DEL RETO

Juan Felipe Ochoa, 202320053, jf.ochoa@uniandes.edu.co

Juan David Guzmán, 202321801, jd.guzman23@uniandes.edu.co

David Alejandro Zambrano, 202321948, da.zambrano2@uniandes.edu.co

Requerimiento 1

Descripción

```python
def req1(data_structs,fecha_i,fecha_f):
    mapa_fechas= data_structs["jobs_fecha"]
    fecha_i= datetime.strptime(fecha_i,"%Y-%m-%d").date()
    fecha_f= datetime.strptime(fecha_f,"%Y-%m-%d").date()

    listas_ofertas= om.values(mapa_fechas,fecha_i,fecha_f)
    tamanio= 0
    ofertas= lt.newList()
    for lista in lt.iterator(listas_ofertas):
        tamanio+=lt.size(lista)
        for oferta in lt.iterator(lista):
            lt.addLast(ofertas,oferta)
    if lt.size(ofertas)>11:
        primeros= lt.subList(ofertas,1,5)
        ultimos=lt.subList(ofertas,lt.size(ofertas)-5,5)
        rta=lt.newList()
        for oferta in lt.iterator(primeros):
            lt.addLast(rta,oferta)
        for oferta in lt.iterator(ultimos):
            lt.addLast(rta,oferta)
        ofertas=rta
    return tamanio,ofertas
```

Este requerimiento se encarga de retornar dos datos, el tamaño se refiere a la cantidad de ofertas que están en el rango que el usuario da, el otro dato que retorna son las del rango dado por el usuario, específicamente las 5 más recientes y las 5 antiguas. Los parámetros de la función son precisamente toda la estructura de datos el rango de fechas que el usuario desea consultar. Inicia llamando el árbol creado con las llaves de las fechas y asignando al rango de fechas en el formato datetime.date. Luego, con la función om.values, logra optimizar la selección de los datos dentro del rango dado por el usuario del árbol de fechas. Luego, se recorren los valores del árbol de los nodos que se encuentran dentro del rango dado y se agregan a una lista ADT. Por último, se seleccionan los 5 más nuevos y 5 más antiguos

Entrada	Estructuras de datos del modelo, fecha inicial del rango, fecha final del rango (dados por el usuario
Salidas	Retorna las 5 primeras y 5 ultimas ofertas y el tamaño de todas las ofertas encontradas en el rango

Implementado (Sí/No)	Si. Implementado de forma Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Preparación de fechas	O(1)
Toma de listas de ofertas	O(logk + m)
Conteo y agregación de ofertas	O(n) (más pequeña que N ofertas)
sublistas	0(1)
TOTAL	***O(n) (más pequeña que todas las N ofertas)***

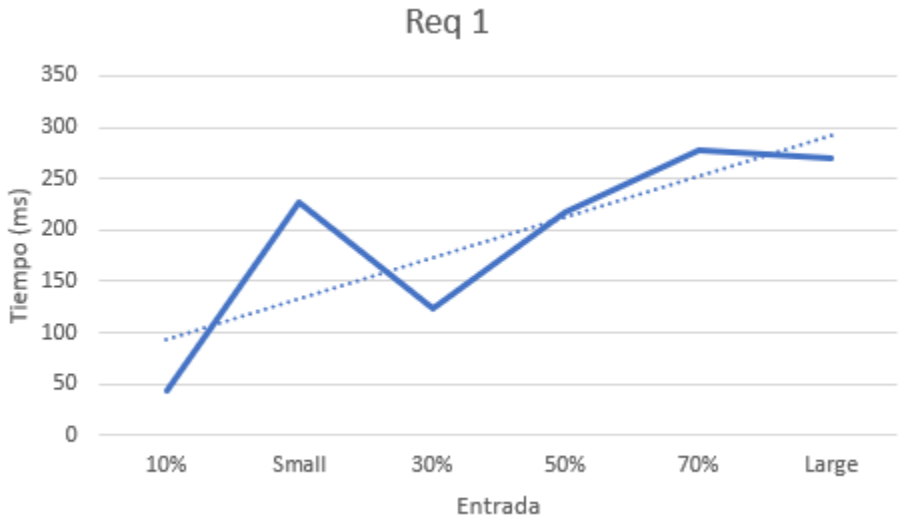
Pruebas Realizadas

Se hacen pruebas con los siguientes porcentajes: 10%,30%,50%,70%, Small y Large. Para hacer las pruebas, las entradas fueron: (fecha inicial del rango = 2022-04-20, fecha final del rango = 2022-12-20)

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	**Tiempo (ms)**
10%	42.56
small	226.19
30%	122.61
50%	216.28
70%	277.9
large	270.16

Graficas



Análisis

Con la estructura de un árbol, que nos ayuda a optimizar la complejidad y según los resultados, podemos observar que la propiedad logarítmica se refleja en la gráfica, donde con más datos se demora un poco más pero cada vez se vuelve más lineal. Por otro lado, al reducir el recorrido a una lista menor, hace que la cantidad de tiempo en el proceso sea mucho menor. Esto quiere decir que se cumplen con los objetivos de optimización de tiempo al momento de la consulta

Requerimiento 2

Descripción

```python
def req2(data_structs, lim_inf_salary, lim_sup_salary):
    """
    Función que soluciona el requerimiento 2
    """
    mapa_fechas= data_structs["jobs_salary"]
    listas_salarios = om.values(mapa_fechas,lim_inf_salary,lim_sup_salary)
    tamanio= 0
    salarios = lt.newList()
    for lista in lt.iterator(listas_salarios):
        tamanio+=lt.size(lista)
        for salario in lt.iterator(lista):
            add_keys_jobs(data_structs, salario)
            lt.addLast(salarios, salario)

    if lt.size(salarios)>10:
        primeros= lt.subList(salarios,1,5)
        ultimos=lt.subList(salarios,lt.size(salarios)-5,5)
        rta=lt.newList()
        for salario in lt.iterator(primeros):
            lt.addLast(rta,salario)
        for salario in lt.iterator(ultimos):
            lt.addLast(rta,salario)
        salarios = rta
    return tamanio, salarios
```

Este requerimiento se encarga de retornar dos datos, el tamaño se refiere a la cantidad de ofertas que el usuario da, el otro dato que retorna son las ofertas del rango dado por el usuario, específicamente las 5 más recientes y las 5 antiguas. Los parámetros de la función son precisamente toda la estructura de datos, el rango de salarios que el usuario desea consultar. Inicia llamando el árbol creado con las llaves de salarios. Luego, con la función om.values, logra optimizar la selección de los datos dentro del rango dado por el usuario del árbol de salarios. Luego, se recorren los valores del árbol de los nodos que están en el rango dado y se agregan a una lista ADT, y se le agregan los datos de las ofertas a los datos según su id. Por último, se seleccionan los 5 más nuevos y 5 más antiguos

Entrada	Estructuras de datos del modelo, límite inferior del salario, limite máximo de salario
Salidas	Retorna las 5 primeras y 5 ultimas ofertas y el tamaño de todas las ofertas encontradas en el rango
Implementado (Sí/No)	Si. Implementado de forma Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Extraccion de salarios	O(logk + m)
Iteracion y procesamiento detallado	O(n(logn + m + j)
sublistas	0(1)
TOTAL	***O(nlog(n + m + j)***

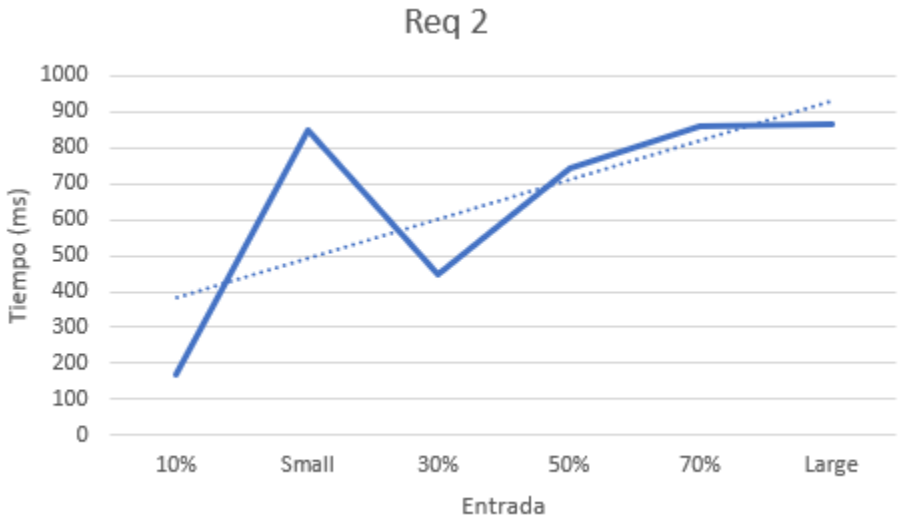
Pruebas Realizadas

Se hacen pruebas con los siguientes porcentajes: 10%,30%,50%,70%, Small y Large. Para hacer las pruebas, las entradas fueron: (límite inferior de salario = 1000, límite superior de salarios = 3000)

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	**Tiempo (ms)**
10%	171.15
small	846.46
30%	449.39
50%	744.45
70%	858.65
large	867.57

Graficas



Análisis

Como se refleja en la gráfica, la complejidad de este requerimiento es de carácter logarítmico, donde a medida que el número de datos es mayor, hasta cierto punto empieza a tomar una característica constante (del 70% al large), lo que quiere decir que este método utilizado en el requerimiento es eficiente al momento de revisar archivos con gran cantidad de datos. La propiedad logarítmica en la complejidad viene del uso de un árbol RBT, el cual esta balanceado y permite una búsqueda más eficiente en un rango dado. la complejidad O(nlog (n + m + j)) viene dada por n como numero de salarios, m como numero de habilidades y j como las ofertas dadas, todas dentro del logaritmo que es la demora que toma recoger los datos del árbol.

Requerimiento 3

Descripción

```
def req_3(control,N, country_code, experience_level):
    """
    Función que soluciona el requerimiento 3
    """
    ans = lt.newList('ARRAY_LIST')
    country = control['jobs_pais']
    mapa_country = mp.get(country, country_code)
    if mapa_country:
        lista = me.getValue(mapa_country)
        for job in lt.iterator(lista):
            add_keys(control,job)
            if job['experience_level'] == experience_level:

                dic_respuesta={"published_at":job["published_at"],
```

```
                        "título":job["title"],
                        'company_name': job['company_name'],
                        "nivel_experticia":job["experience_level"],
                        "ciudad":job["city"],
                        "pais":job["country_code"],
                        "tamanio":job["company_size"],
                        "tipo_lugar":job["workplace_type"],
                        "salario_minimo":job["salary_in_usd"],
                        "habilidad": job['name_skill']
                        }
            lt.addLast(ans, dic_respuesta)

    quk.sort(ans, compare_Dates)
    if lt.size(ans) <= N:
        ofertas_final = ans
    else:
        ofertas_final = lt.subList(ans, 1, N)
    return ofertas_final
```

Este requerimiento se encarga de retornar dos datos, el número total de ofertas laborales publicadas para un país y que requieran un nivel específico, el otro dato que retorna son las N ofertas laborales publicadas más recientes que cumplan con las condiciones. Los parámetros de la función son precisamente toda la estructura de datos, el número de ofertas a evaluar, el código del país y el nivel de experticia que el usuario desea consultar. Inicia llamando el árbol creado con las llaves de países. Luego, con la función de mp.get obtenemos las ofertas con el país que indicó el usuario y con el me.getvalue se obtiene la lista. Después se itera sobre esta lista y se le añade a cada oferta las llaves necesarias con la función auxiliar add_keys. Posteriormente, se verifica cuales ofertas tienen el nivel de experticia que pregunta el usuario y se retorna en una lista “ans” las ofertas con dichos requerimientos y solo con las llaves especificadas. Finalmente se hace un quick sort para organizar la lista final por fecha. Si la cantidad de ofertas con estos requerimientos es menor a la que pide el usuario, solo se retornan esas ofertas.

Entrada	Estructuras de datos del modelo, número de ofertas a evaluar, código de país, y nivel de experiencia.
Salidas	Retorna las N ofertas laborales más recientes con los requerimientos y el número total de ofertas publicadas con estos requerimientos
Implementado (Sí/No)	Si. Implementado de forma Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Acceso a mapa	O(1)
Iteracion y procesamiento detallado	O(m)
Ordenamiento de la lista	O(n log n)
TOTAL	***O(nlog(n +m)***

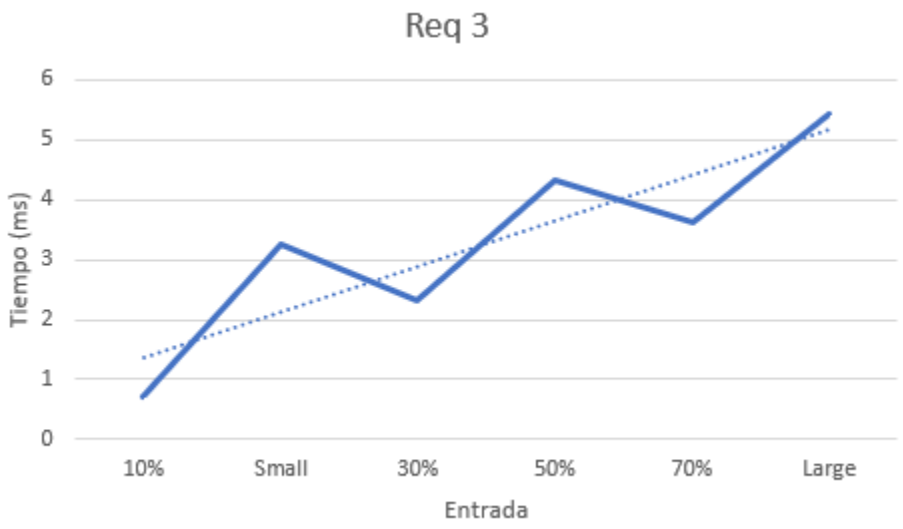
Pruebas Realizadas

Se hacen pruebas con los siguientes porcentajes: 10%,30%,50%,70%, Small y Large. Para hacer las pruebas, las entradas fueron: (Número de ofertas: 10, Código: FR, nivel: Junior)

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	Tiempo (ms)
10%	0.71
small	3.25
30%	2.31
50%	4.32
70%	3.61
large	5.42

Graficas



Análisis

La complejidad aumento al momento de realizar el recorrido en la lista filtrada y el ordenamiento de la lista final. Sin embargo, este recorrido for no es tan grande gracias a la filtración que se hace por el código de país, y solo se realiza comparaciones con el nivel de experticia. Lo mismo ocurre con el ordenamiento por fecha, este aumenta la complejidad en O(nlogn), no obstante, es una lista bastante reducida. Se observa cómo aumenta la complejidad con el archivo, pero no demasiado, el algoritmo funciona correctamente.

Requerimiento 4

Descripción

```
def req_4(control, N, city, tipo_ubicacion):
    """
    Función que soluciona el requerimiento 4
    """
    ans = lt.newList('ARRAY_LIST')
    ciudad = control['jobs_ciudad']
    mapa_city = mp.get(ciudad, city)
    if mapa_city:
        lista = me.getValue(mapa_city)
        for job in lt.iterator(lista):
            add_keys(control,job)
            if job['workplace_type'] == tipo_ubicacion:

                dic_respuesta={"published_at":job["published_at"],
                           "título":job["title"],
                           'company_name': job['company_name'],
```

```
                        "nivel_experticia":job["experience_level"],
                        "ciudad":job["city"],
                        "pais":job["country_code"],
                        "tamanio":job["company_size"],
                        "tipo_lugar":job["workplace_type"],
                        "salario_minimo":job["salary_in_usd"],
                        "habilidad": job['name_skill']
                        }
            lt.addLast(ans, dic_respuesta)
    quk.sort(ans, compare_Dates)
    if lt.size(ans) <= N:
        ofertas_final = ans
    else:
        ofertas_final = lt.subList(ans, 1, N)
    return ofertas_final
```

.

Entrada	Estructuras de datos del modelo, número de ofertas a evaluar, ciudad, y ubicación
Salidas	Retorna las N ofertas laborales más recientes con los requerimientos y el número total de ofertas publicadas con estos requerimientos
Implementado (Sí/No)	Si. Implementado de forma Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Acceso a mapa	O(1)
Iteración y procesamiento detallado	O(m)
Ordenamiento de la lista	O(n log n)
TOTAL	***O(nlog(n +m)***

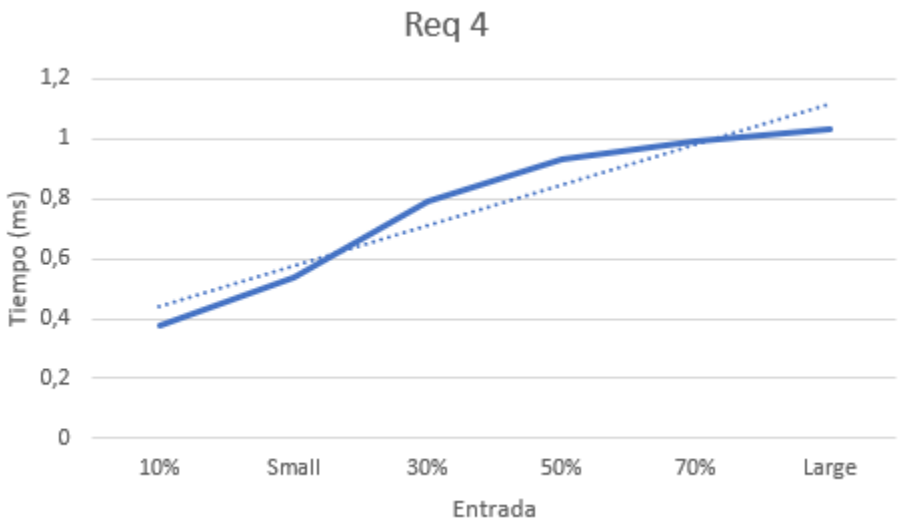
Pruebas Realizadas

Se hacen pruebas con los siguientes porcentajes: 10%,30%,50%,70%, Small y Large. Para hacer las pruebas, las entradas fueron: (número de ofertas: 10, ciudad: Paris, ubicación: remote)

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	Tiempo (ms)
10%	0.38
small	0.54
30%	0.79
50%	0.93
70%	0.99
large	1.03

Graficas



Análisis

La complejidad aumento al momento de realizar el recorrido en la lista filtrada y el ordenamiento de la lista final. Sin embargo, este recorrido for no es tan grande gracias a la filtración que se hace por el código de país, y solo se realiza comparaciones con el nivel de experticia. Lo mismo ocurre con el ordenamiento por fecha, este aumenta la complejidad en O(nlogn), no obstante, es una lista bastante reducida. Se observa cómo aumenta la complejidad con el archivo, pero no demasiado, el algoritmo funciona correctamente.

Requerimiento 5

```
def req_5(control, N, lim_inf_tam, lim_sup_tam, skill, lim_inf_nh, lim_sup_nh):
    """
    Función que soluciona el requerimiento 5
    """
    mapa_tamano= control["jobs_tamano"]

    listas_ofertas= om.values(mapa_tamano,lim_inf_tam,lim_sup_tam)
    tamanio= 0
    ofertas= lt.newList()
    for lista in lt.iterator(listas_ofertas):
        tamanio+=lt.size(lista)
        for oferta in lt.iterator(lista):
            add_keys(control, oferta)
            if skill in oferta["name_skill"]:
                if(oferta['level_skill'][skill] >=lim_inf_nh) and (oferta['level_skill'][skill] <= lim_sup_nh):
                    lt.addLast(ofertas,oferta)
    quk.sort(ofertas, compare_dates)
    if lt.size(ofertas) <= N:
        ofertas_final = ofertas
    else:
        ofertas_final = lt.subList(ofertas, 0, N)
    """if lt.size(ofertas)>10:
        primeros= lt.subList(ofertas,1,5)
        ultimos=lt.subList(ofertas,lt.size(ofertas)-4,5)
        rta=lt.newList()
        for oferta in lt.iterator(primeros):
            lt.addLast(rta,oferta)
        for oferta in lt.iterator(ultimos):
            lt.addLast(rta,oferta)
        ofertas=rta"""
    return tamanio,ofertas_final
```

Descripción

Primero se llama al árbol jobs_tamano, donde están todas las ofertas de trabajo organizadas por el tamaño de la compañía. Después se extraen con om.values() todas las llaves del árbol que están entre el rango del tamaño de la compañía. Se hace un recorrido donde primero se usa un contador para contar el total de ofertas encontradas, después itera en la lista de diccionarios. Acá se adicionan llaves de los otros archivos, como las habilidades. Hace un recorrido encontrando la habilidad buscada y si esta se encuentra dentro del rango de habilidad. Por último, después de encontrar todas las ofertas que cumplen con lo buscado, se le hace un quk.sort() para organizarlas por fecha, y de ahí se obtienen las “N” ofertas buscadas. Se retorna tanto el tamaño, que son todas las ofertas encontradas, y también la lista final.

Entrada	Control (todos los mapas y árboles), el número de ofertas buscadas, El límite inferior del tamaño de la compañía, El límite superior del tamaño de la compañía, Nombre de la habilidad solicitada, El límite inferior del nivel de la habilidad, El límite superior del nivel de la habilidad.
Salidas	El número de ofertas encontradas, la lista de ofertas encontradas ya previamente filtradas

Implementado (Sí/No)	Implementado por Juan Felipe Ochoa

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

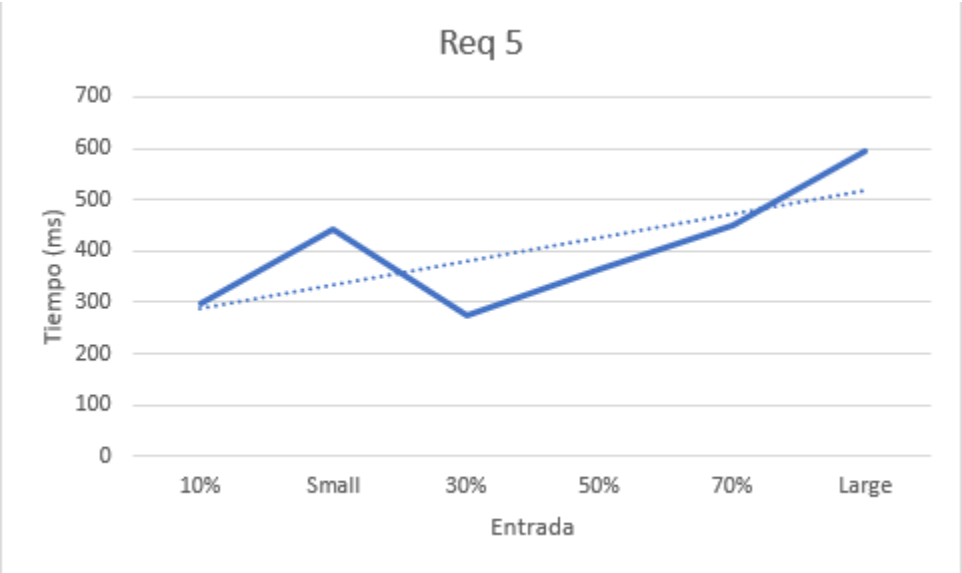
Pasos	**Complejidad**
Extracción de trabajos por salario	O(logk + m)
Recorrido	O(k)
quk.sort	O(nlog(n))
sublistas	O(1)
TOTAL	***O(n log n)***

Pruebas Realizadas

Se hacen pruebas con los siguientes porcentajes: 10%,30%,50%,70%, Small y Large. Para hacer las pruebas, las entradas fueron: (número de ofertas = 10, Límite inferior del tamaño = 300, Límite superior del tamaño = 800, Habilidad = GO, Límite inferior de habilidad = 3 y Límite superior de habilidad = 5)

Entrada	**Tiempo (ms)**
10%	297.28
Small	441.64
30%	272.86
50%	363.90
70%	451.86
Large	593.65

Graficas



Análisis

Aunque el comportamiento de los datos parezca O(N), al ser filtrado varias veces, se obtienen recorridos mucho más cortos y un ordenamiento más sencillo. Por eso, al final termina siendo un O(nlog(n)), con una “n” mucho menor a “N”, siendo “N”, la lista total de ofertas y “n” una lista con las ofertas previamente filtradas. Esto se evidencia un poco en la gráfica, porque a pesar de ir creciendo en tiempo, el crecimiento no es tan significativo, haciendo que sea eficiente.

Requerimiento 6

```python
def req_6(data_structs,N_ciudades,fecha_i,fecha_f, lim_inf_sal, lim_sup_sal):
    """
    mapa_fechas= data_structs["jobs_fecha"]

    # TODO: Realizar el requerimiento 6
    fecha_i= datetime.strptime(fecha_i,"%Y-%m-%d").date()
    fecha_f= datetime.strptime(fecha_f,"%Y-%m-%d").date()
    listas_ofertas= om.values(mapa_fechas,fecha_i,fecha_f)
    tamanio= 0
    ofertas= lt.newList()
    for lista in lt.iterator(listas_ofertas):
        tamanio+=lt.size(lista)
        for oferta in lt.iterator(lista):
            add_keys(data_structs, oferta)
            if oferta["salary_in_usd"]>= lim_inf_sal and oferta["salary_in_usd"]<=lim_sup_sal:
                lt.addLast(ofertas,oferta)
    dic_ciudades, mayor_ciudad, numero_ciudades = encontrar_mayor_ciudad(ofertas)
    ciudades = list(dic_ciudades.items())
    ciudades_tad = lt.newList()
    for ciudad in ciudades:
        lt.addLast(ciudades_tad,ciudad)
    quk.sort(ciudades_tad, compare_ciudades)
    ciudades_mayores = lt.newList("ARRAY_LIST")
    for ciudad in lt.iterator(ciudades_tad):
        N_ciudades-=1
        lt.addLast(ciudades_mayores,ciudad)
        if N_ciudades<=0:
            break
    ans = lt.newList("ARRAY_LIST")
    for oferta in lt.iterator(ofertas):
        if oferta['city'] == mayor_ciudad:
            lt.addLast(ans,oferta)
    if lt.size(ans)>10:
        primeros= lt.subList(ans,1,5)
        ultimos=lt.subList(ans,lt.size(ans)-4,5)
        rta=lt.newList()
        for oferta in lt.iterator(primeros):
            lt.addLast(rta,oferta)
        for oferta in lt.iterator(ultimos):
            lt.addLast(rta,oferta)
        ans=rta
    return tamanio,numero_ciudades,ciudades_mayores, ans
```

(parameter)

Descripción

La función analiza el árbol de ofertas de empleo, filtrando y clasificando información según rangos de fecha y salario. Primero, convierte las fechas y recupera las ofertas de empleo que caen dentro de este intervalo. Luego, itera sobre estas ofertas para seleccionar aquellas que cumplen con criterios específicos de salario. Posteriormente, identifica la ciudad con más ofertas y organiza las ciudades por la cantidad de ofertas. Se extraen las ciudades con más ofertas hasta alcanzar el número deseado. Finalmente, selecciona y limita las ofertas de la ciudad predominante a las primeras y últimas cinco, en caso de que haya más de diez, y devuelve un resumen de los resultados obtenidos. Este proceso implica manipulación de fechas, filtrado basado en salarios, y ordenamiento y reducción basados en la ubicación geográfica de las ofertas de trabajo.

Entrada	El número de ciudades a consultar, La fecha inicial, La fecha final, El límite inferior del salario mínimo ofertado y El límite superior del salario mínimo ofertado.
Salidas	El número de ofertas encontradas, el número de ciudades en las cuales se encontraron las ofertas, una lista con las ciudades con mayores ofertas, y una lista con todas las ofertas encontradas.
Implementado (Sí/No)	Se implementó grupalmente

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

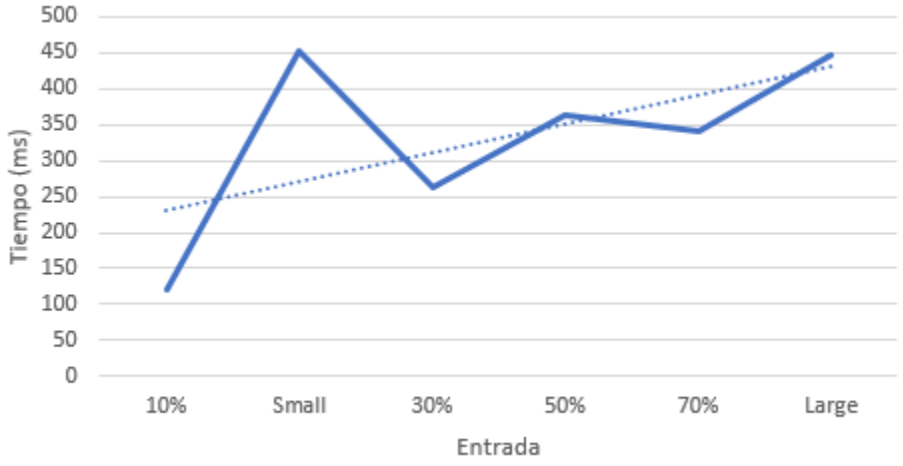
Pasos	**Complejidad**
Om.values()	O(log(n)+k)
Primer for (iteración sobre lista de ofertas)	O(k)
Encontrar_mayor_ciudad()	O(k)
Recorrido para mayor ciudad	O(c)
Recorrido para agregar a la lista	O(log(c))
TOTAL	***O(log(n)+ k+ c(log(c))***

Pruebas Realizadas

Se hacen pruebas con los siguientes porcentajes: 10%,30%,50%,70%, Small y Large. Para hacer las pruebas, las entradas fueron: (número de ofertas = 10, fecha inicial = 2023-03-01, fecha final = 2023-04-20, Límite inferior del salario= 1000 y Límite superior del salario = 3000)

Entrada	**Tiempo (s)**
10%	120.37
Small	453.22
30%	262.93
50%	364.38
70%	393.90
Large	447.61

Graficas



Análisis

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

La función de om.values implica un aumento en la complejidad algorítmica del requerimiento ya que se filtra el mapa_fechas por un rango de fechas específico y tiene que hacer esta búsqueda. Posteriormente el recorrido for que se hace sobre la lista de ofertas tiene una complejidad de O(k) ya que es una lista filtrada. Lo mismo ocurre con la función auxiliar de encontrar la ciudad con mayor número de ofertas. Al encontrar la mayor ciudad se hace otro recorrido con complejidad de O(c) y, finalmente, el recorrido para agregar a la lista resultado tiene una complejidad de O(log(c)). Al sumar todo se obtiene una complejidad algorítmica total de O (log(n)+k+c+og(c)). Asimismo, en la gráfica observamos un aumento en el archivo small, el cual es parecido al large. Por ende, se evidencia que a pesar de tener un aumento considerable, la función mantiene un tiempo estable y entrega los resultados rápidamente.

Requerimiento 7

```python
def req_7(data_structs, anio, codigo_pais, propiedad):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    tipo_propiedad = None
    if propiedad.lower() == "experticia":
        tipo_propiedad = "experticia"
    elif propiedad.lower() == "ubicacion":
        tipo_propiedad = "ubicacion"
    else:
        tipo_propiedad = "habilidad"

    entrada_mapa_pais = mp.get(data_structs["mapa_pais"], codigo_pais)
    arbol_pais = me.getValue(entrada_mapa_pais)
    empieza_el_rango = datetime(year=int(anio), month=1, day=1)
    termina_el_rango = datetime(year=int(anio), month=12, day=31)
    lista_ofertas_anio = om.values(arbol_pais, empieza_el_rango, termina_el_rango)
    dicc_ans = {}
    cantidad_ofertas = 0
    if tipo_propiedad == "experticia":
        dicc_ans["junior"] = lt.newList("ARRAY_LIST")
        dicc_ans["mid"] = lt.newList("ARRAY_LIST")
        dicc_ans["senior"] = lt.newList("ARRAY_LIST")
        for lista_por_fechas in lt.iterator(lista_ofertas_anio):
            for oferta in lt.iterator(lista_por_fechas):
                cantidad_ofertas += 1
                lt.addLast(dicc_ans[oferta["experience_level"]], oferta)
```

```python
elif tipo_propiedad == "ubicacion":
    dicc_ans["remote"] = lt.newList("ARRAY_LIST")
    dicc_ans["partly_remote"] = lt.newList("ARRAY_LIST")
    dicc_ans["office"] = lt.newList("ARRAY_LIST")
    for lista_por_fechas in lt.iterator(lista_ofertas_anio):
        for oferta in lt.iterator(lista_por_fechas):
            cantidad_ofertas += 1
            lt.addLast(dicc_ans[oferta["workplace_type"]], oferta)
else:
    for lista_por_fechas in lt.iterator(lista_ofertas_anio):
        for oferta in lt.iterator(lista_por_fechas):
            cantidad_ofertas += 1
            add_keys(data_structs,oferta)
            for habilidad in oferta["name_skill"]:
                if habilidad in dicc_ans:
                    lt.addLast(dicc_ans[habilidad], oferta)
                else:
                    dicc_ans[habilidad] = lt.newList("ARRAY_LIST")
                    lt.addLast(dicc_ans[habilidad], oferta)
return dicc_ans, cantidad_ofertas
```

Descripción

La función realiza la clasificación de ofertas de empleo basándose en un año específico, un código de país y una propiedad definida (experticia, ubicación o habilidad). Primero, el código determina el tipo de propiedad a clasificar, luego recupera las ofertas de empleo de un árbol asociado al país indicado, dentro de un rango de fechas que abarca el año completo. Dependiendo de la propiedad elegida, clasifica y organiza las ofertas en categorías apropiadas (como nivel de experiencia, tipo de lugar de trabajo o habilidades específicas), y almacena estas en diccionarios o listas. Finalmente, devuelve un resumen de

las categorías junto con la cantidad total de ofertas procesadas, permitiendo una visión estructurada según los criterios especificados.

Entrada	El año, el código del país para la consulta, La propiedad de conteo (experticia, ubicación, o habilidad).
Salidas	Se devuelve el diccionario con las llaves dependiendo del caso, y la lista de ofertas como valores. También se devuelven la cantidad de ofertas totales.
Implementado (Sí/No)	Se implementó grupalmente

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

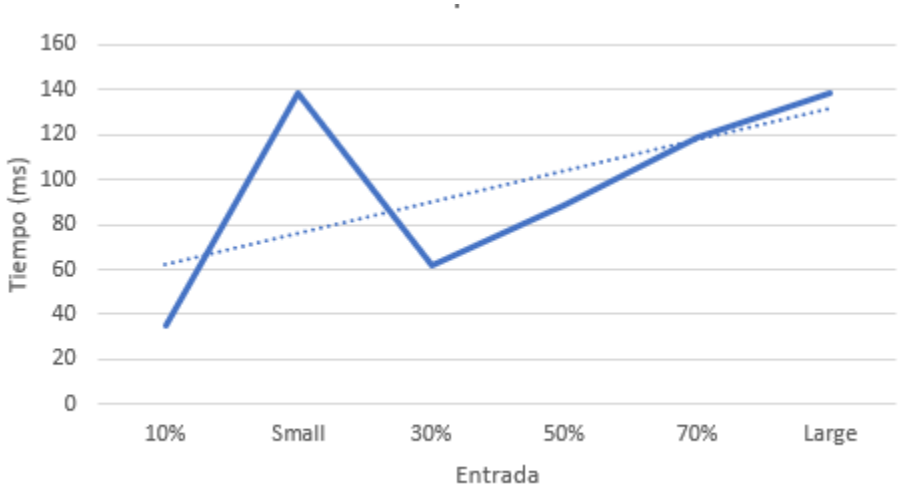
Pasos	**Complejidad**
Acceder al mapa	O(1)
Acceder al árbol	O(log(n)+k)
Recorrido sobre la lista	O(k)
TOTAL	***O(log(n)+k)***

Pruebas Realizadas

Se hacen pruebas con los siguientes porcentajes: 10%,30%,50%,70%, Small y Large. Para hacer las pruebas, las entradas fueron: (año = 2023, Código del País = PL, tipo de propiedad = ubicación)

Entrada	**Tiempo (s)**
10%	35.18
Small	138.78
30%	62.08
50%	88.18
70%	118.94
Large	138.69

Graficas



Análisis

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

Este requerimiento tiene una complejidad algorítmica total de O(log(n)+k) ya que al acceder al árbol con los valores específicos la complejidad aumenta O(log(n)+k) y posteriormente el recorrido sobre la lista que se obtiene es de O(k). En ambos casos, se evalúa sobre estructuras que están filtradas, por ende, la cantidad de datos es mucho menor y aumenta la velocidad de la función. Esto se evidencia en la gráfica en donde el small y large tienen la mayor duración, con un aproximado de 140 milisegundos. Es decir que el requerimiento es aplicado de forma correcta y no se demora más de lo necesario.