ANÁLISIS DEL RETO

Julian Pinto,202321207,jr.pinto@uniandes.edu.co

Estudiante 2, código 2, email 2

Estudiante 3, código 3, email 3

Carga de Datos:

```
def new_data_structs():
    data_struct={
    "job_offers": None,
    "employment_types" : None,
    "multilocations" : None,
    "skills" : None,
    "dateIndex": None
    }

    # g
    data_struct['job_offers']= lt.newList('ARRAY_LIST')
    data_struct['dateIndex'] = om.newMap(omaptype='RBT',
                                         cmpfunction=compareDates) # paramtros
    data_struct['dateIndex_exp'] = om.newMap(omaptype='RBT',
                                         cmpfunction=compareDates) # paramtros
    data_struct['req5'] = om.newMap(omaptype='RBT',
                                         cmpfunction=compareDates) # compare company size
    data_struct['job_offer_id'] = mp.newMap(101503,
                                 maptype='CHAINING',
                                 loadfactor=2)
    data_struct['employment_types_id'] =mp.newMap(101503,
                                 maptype='CHAINING',
                                 loadfactor=2)
                                # cmpfunction=compareemployment)
    data_struct['multilocation_id'] = mp.newMap(101503,
                                 maptype='CHAINING',
                                 loadfactor=2)
                                # cmpfunction=comparemultilocation)
    data_struct['skills_id'] = mp.newMap(101503,
                                 maptype='CHAINING',
                                 loadfactor=2)

    """
    Inicializa las estructuras de datos del modelo. Las crea de
    manera vacía para posteriormente almacenar la información.
```

En la carga de datos se usan los tres ADTs. La lista se usa para la representación de la carga de datos que vería el usuario. Los diccionarios que terminan en "_id" son mapas que guardan la información con la llave del id del trabajo. De esta manera podemos conectar los diferentes archivos. Para los requisitos con necesidad de árbol binario se crea en la carga de datos. Esto encaja con los requisitos 1, 5, 6 etc. En comparación con el reto pasado, podemos ver una similitud pues la idea de guardar trabajos e id se usó previamente. Sin embargo, el reto actual tiene varia bastante de los requisitos pasados y sus necesidades cambian. Por esta razón son importantes los árboles binarios. Esta carga de datos es mucho más lenta pues usamos arboles RBT para casi todos los casos lo cual tiene una complejidad de inserción alta.

Requerimiento <<1>>

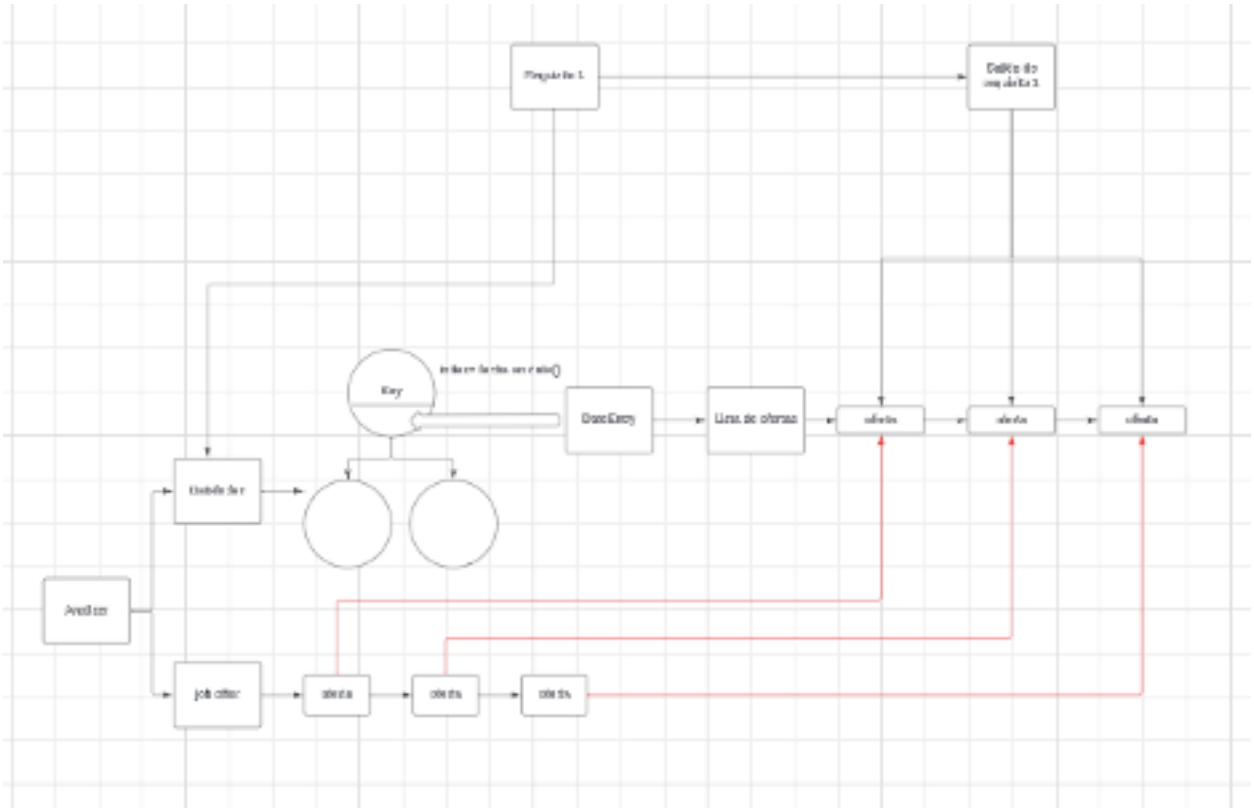
Plantilla para documentar y analizar los requerimientos.

Armando la estructura de datos

```
def update_date_index(map, data):
    occurreddate = data['published_at']
    jobdate = datetime.datetime.strptime(occurreddate, "%Y-%m-%dT%H:%M:%S.%fZ")

    if not om.contains(map, jobdate.date()):
        date_entry = lt.newList()
        lt.addLast(date_entry, data)
        om.put(map,jobdate.date(), date_entry )
    else:
        date_entry = me.getValue(om.get(map, jobdate.date()))
        lt.addLast(date_entry,data)
    return map
```

Buscando en la estructura de datos

```
def req_1(data_structs, keylo, keyhi):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    keylo = datetime.datetime.strptime(keylo, "%Y-%m-%d").date()
    keyhi = datetime.datetime.strptime(keyhi, "%Y-%m-%d").date()
    mapa = data_structs['dateIndex']
    # comentario date time
    lista_return = lt.newList('ARRAY_LIST')
    lista_valores = om.values(mapa, keylo, keyhi)
    for valor in lt.iterator(lista_valores):
        for job in lt.iterator(valor):
            lt.addLast(lista_return, job)
            # tal vez sort
    return lista_return
```



Descripción

Para este requerimiento, usamos una estructura de árbol tipo RBT en la cual tenemos como llaves los fechas en formato year, months y days. De esta manera, el input del usuario encaja con el árbol y es eficiente. Por otro lado, como valor se guardan listas en las cuales guardamos los trabajos.

El requerimiento como tal usa la función om.values() para hacer una lista de listas de cada nodo que encaja con el rango. Después a eso se usan dos for para iterar y hacer una lista que devuelve los trabajos.

Entrada	Fecha inicial y fecha final
Salidas	Lista de trabajos que encajan en las fechas
Implementado (Sí/No)	Si, grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1 - filtrar	O(logn)
Paso 2 - iterar por lista para mostrar resultado	O (1)
TOTAL	***O(log(n))***

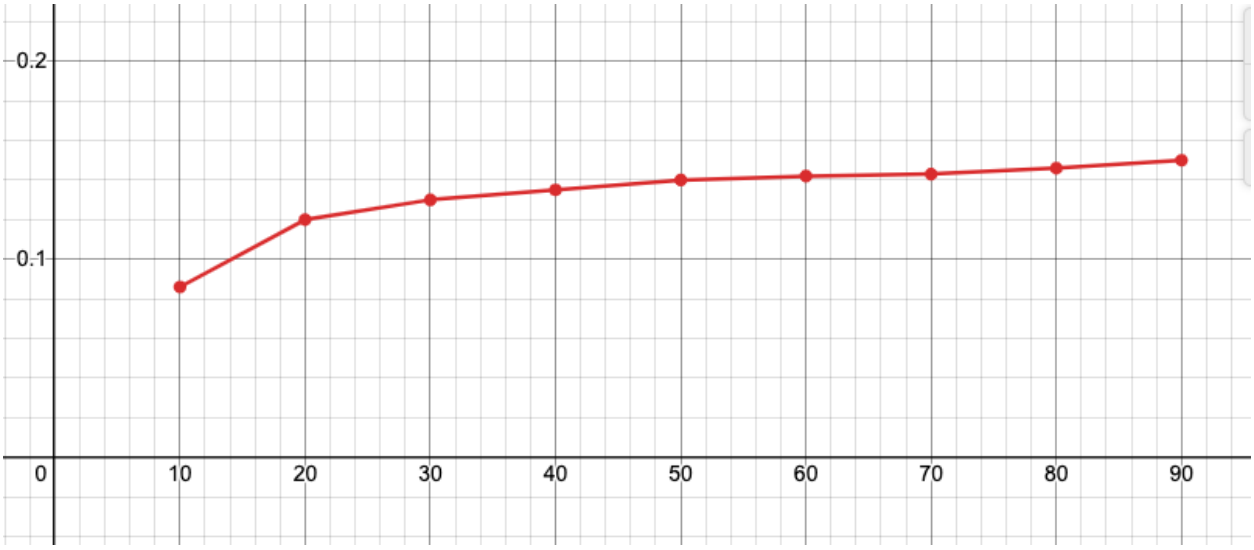
Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**AMD Ryzen 7 4800HS with Radeon Graphics**
Memoria RAM	8 GB
Sistema Operativo	mac

Entrada	**Tiempo (s)**
small	
5 pct	
10 pct	0.009
20 pct	0.016
30 pct	0.034
50 pct	0.072
80 pct	0.147
large	

Graficas



Análisis

Podemos ver una gráfica que se asimila a una búsqueda logarítmica. Podemos ver en la función que es una complejidad simple pues no requiere de muchas iteraciones. Está condicionada a unas constantes, pero no son N. En la recolección de datos hubo una gran variación de datos, por lo que no hay mucha

certeza de lo que la gráfica pueda representar. Hubo alores extremadamente bajos con un porcentaje alto del archivo y otros muy altos en porcentajes bajos.

Comparaciones entre retos pasados y reto actual:

Este requisito cambia bastante del requisito uno del reto 2. Sin embargo, se pueden hacer unas comparaciones. En el reto pasado, este reto tenía como intención hacer dos filtros de ciudad y empresa. Este requisito filtra por un rango de fechas. Podemos ver como buscar en un rango de fecha, lo cual había sido una complicación mayor previamente, ahora es facilísimo. Esto se debe a la estructura de árboles. La complejidad del reto pasado era de O (1) pero es por lo que solo se requieren llaves para resolver el problema. Este, tiene una complejidad e O(logn) por su mecanismo de búsqueda. No es justo hacer una comparación uno a uno entre requisitos pues piden diferentes valores con diferentes parámetros

Requerimiento <<5>>

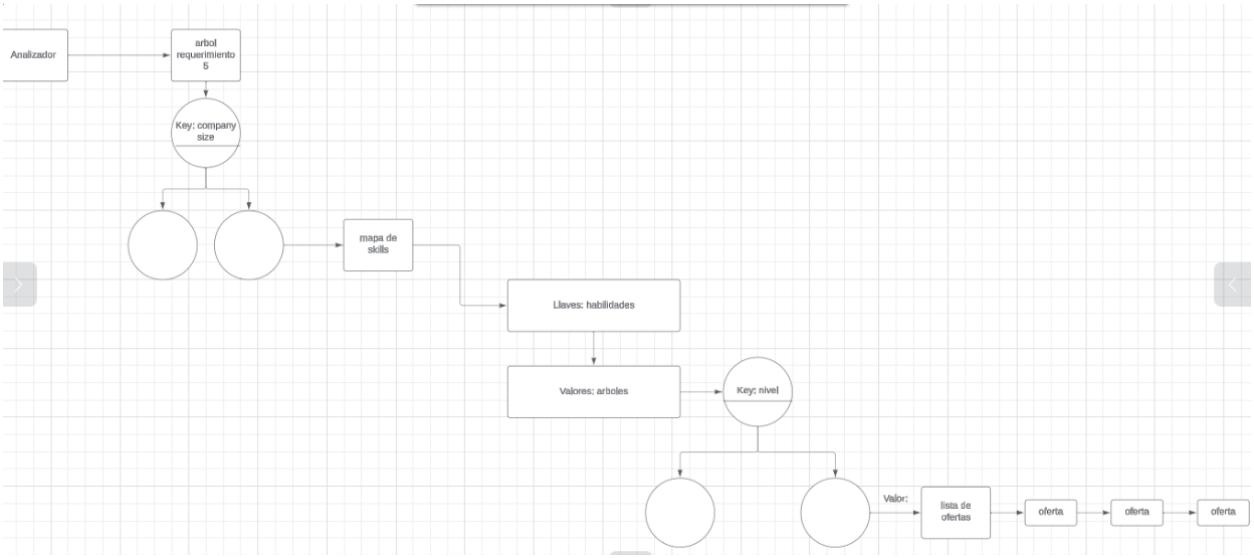
Plantilla para el documentar y analizar cada uno de los requerimientos.

Armando la estructura de datos

```
def add_req5 (datastructs, data_skill):

    data = me.getValue(mp.get(datastructs['job_offer_id'],data_skill['id']))
    if 'skills' not in data:
        skills = lt.newList()
        lt.addLast(skills, data_skill)
        data['skills']= skills
    else:
        skills = data['skills']
        lt.addLast(skills, data_skill)
    req5 = datastructs['req5']
    if data['company_size'] == 'Undefined':
        data['company_size'] = 0
    if not om.contains(req5, int(data['company_size'])):
        lista = lt.newList()
        arbol_nivel = om.newMap(omaptype='RBT',
                                cmpfunction=compareDates)
        skill = mp.newMap()
        om.put(arbol_nivel,data_skill['level'], lista)
        mp.put(skill, data_skill['name'],arbol_nivel)
        om.put(req5, int(data['company_size']), skill)
    else:
        skill = me.getValue(om.get(req5, int(data['company_size'])))
        if not mp.contains(skill, data_skill['name']):

            lista = lt.newList()
            arbol_nivel = om.newMap(omaptype='RBT',
                                    cmpfunction=compareDates)
            om.put(arbol_nivel,data_skill['level'], lista )
            mp.put(skill, data_skill['name'],arbol_nivel)
        else:
            arbol_nivel = me.getValue(mp.get(skill, data_skill['name']))
            if not om.contains(arbol_nivel, data_skill['level']):
                lista = lt.newList()
                om.put(arbol_nivel, data_skill['level'], lista)
            else:
                lista = me.getValue(om.get(arbol_nivel, data_skill['level']))
    lt.addLast(lista, data)
```

Buscando en la estructura de datos

```python
def req_5(data_structs, mincomp,maxcomp, skill, minskill, maxskill):
    peso  = 0
    arbol = data_structs['req5']
    mincomp = int(mincomp)
    maxcomp = int(maxcomp)
    lista_return = lt.newList()
    lista_val = om.values(arbol, mincomp,maxcomp)
    for tablas in lt.iterator(lista_val):
        arbol_it = mp.get(tablas, skill)
        if not arbol_it is None:
            arbol_it = me.getValue(arbol_it)
            listas_datos = om.values(arbol_it,om.minKey(arbol_it), om.maxKey(arbol_it))
            for niveles in lt.iterator(listas_datos):
                peso += lt.size(niveles)
            lista_niveles = om.values(arbol_it,minskill,maxskill)
            for lista in lt.iterator(lista_niveles):
                for oferta in lt.iterator(lista):
                    lt.addLast(lista_return, oferta)



    return peso , first_last(lista_return,['published_at','city','title', 'country_code', 'skills','company_name','company_size',
```



Descripción

La primera carga de datos se enfoca en la creación de un árbol de tamaños de compañías. A esto le sigue un mapa de habilidades (skills) el cual contiene un árbol de niveles de experticia (anos de experticia). La estructura termina con una lista donde se guardan los trabajos. De esta manera, ya en la función del requerimiento, se usa la estructura para extraer la información. Para esto, la funcion necesita hartas funciones for() pues se extraen listas de arboles, listas de mapas y listas en las cuales se debe iterar para conseguir la informacion completa y relevante

Entrada	Tamaño mínimo y máximo de las compañías, nivel mínimo y máximo de las compañías.
Salidas	Lista de trabajos que encajan en las fechas, peso de la lista
Implementado (Sí/No)	Si, Lucas Ruiz

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O (logn)
Paso 2	O(r)
Paso 3	O(p)
Paso 4	O(s)
TOTAL	***O(r*p*s*log(n))***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Parametros:

F1: 2023-01-01

F2: 2023-05-01

Habilidad: JAVA

Nivel 1: 2

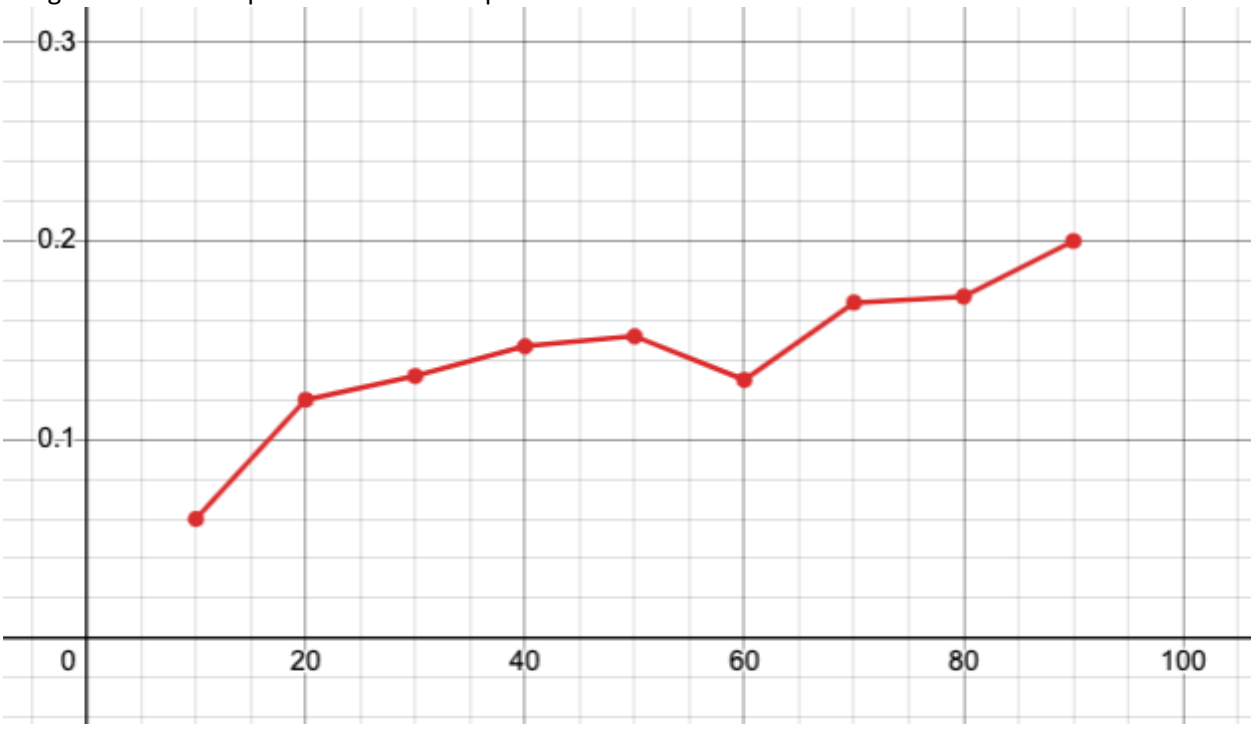
Nivel 2: 4

Procesadores	**AMD Ryzen 7 4800HS with Radeon Graphics**
Memoria RAM	8 GB
Sistema Operativo	mac

Entrada	**Tiempo (ms)**
10 pct	0.06
20 pct	0.120
30 pct	0.133
40 pct	0.147
50 pct	0.154
70 pct	0.161
80 pct	0.169
large	0.2

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

De manera muy similar al requerimiento 1, este se comporta siguiendo una tendencia logarítmica. Podemos ver que este comportamiento actúa acorde con el código pues este se en cagarga de hacer búsquedas logarítmicas. Los for simplemente recorren a complejidad de O (p) y otras constantes. En la recolección de datos hubo una gran variación de datos, por lo que no hay mucha certeza de lo que la gráfica pueda representar. Hubo alores extremadamente bajos con un porcentaje alto del archivo y otros muy altos en porcentajes bajos.

Comparaciones entre retos pasados y reto actual:

Este requisito cambia completamente a comparación del reto 1 y 2 pues las herramientas del árbol hacen que la dificultad de los pasados retos sea casi nula. Este cambia drásticamente por lo que la comparación no es tan útil. Podemos ver que, si se usan tablas de hash y listas, como en el reto pasado pero la implementación de árboles hace que este sea superior en términos de complejidad.

Requerimiento <<6>>

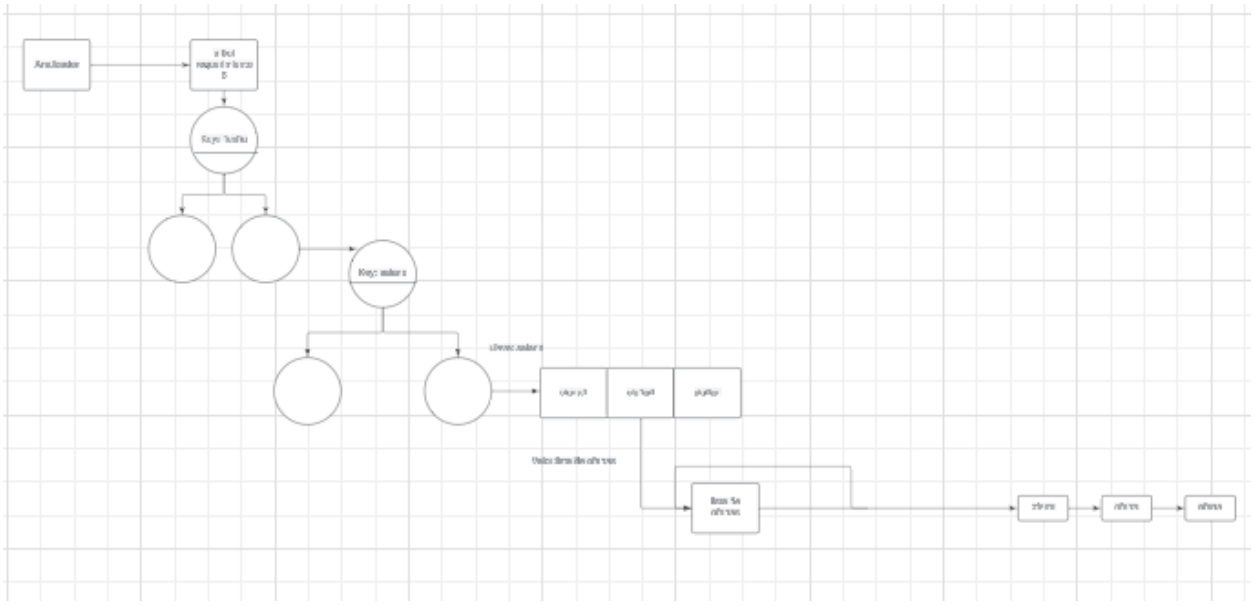
Plantilla para documentar y analizar los requerimientos.

Armando la estructura de datos

```
def add_req_6_nuevo(data_structs, data_emloy):
    ids = data_emloy['id']
    data = me.getValue(mp.get(data_structs['job_offer_id'], ids))
    if data_emloy['currency_salary'] == "Desconocido" or data_emloy['salary_from'] == 'Desconocido':
        salario_min = 0
    else:
        salario_min =salario(data_emloy['currency_salary'],data_emloy['salary_from'])
    data['salary'] = salario_min
    arbol_fechas = data_structs['req6']
    occurreddate = data['published_at']
    ciudad = data['city']
    fecha = datetime.datetime.strptime(occurreddate, "%Y-%m-%dT%H:%M:%S.%fZ")
    if not om.contains(arbol_fechas, fecha.date()):
        arbol_salario = om.newMap(omaptype='RBT',
                                  cmpfunction=compareDates)
        mapa_ciudades = mp.newMap()
        lista_ofertas = lt.newList()
        om.put(arbol_fechas, fecha.date(), arbol_salario)
        om.put(arbol_salario, salario_min, mapa_ciudades)
        mp.put(mapa_ciudades, ciudad, lista_ofertas)
    else:
        arbol_salario = me.getValue(om.get(arbol_fechas, fecha.date()))
        if not om.contains(arbol_salario, salario_min):
            lista_ofertas = lt.newList()
            mapa_ciudades = mp.newMap()
            mp.put(mapa_ciudades, ciudad, lista_ofertas )
            om.put(arbol_salario, salario_min, mapa_ciudades)
        else:
            mapa_ciudades = me.getValue(om.get(arbol_salario, salario_min))
            if not mp.contains(mapa_ciudades,ciudad):
                lista_ofertas = lt.newList()
                mp.put(mapa_ciudades, ciudad, lista_ofertas)
            else:
                lista_ofertas = me.getValue(mp.get(mapa_ciudades, ciudad))

    lt.addLast(lista_ofertas, data)
```

Buscando en la estructura de datos

```
    """
    # TODO: Realizar el requerimiento 6
    salarioMax = int(salarioMax)
    salarioMin = int(salarioMin)
    arbol_fechas=data_structs["req6"]
    diccionario_ciudades=om.newMap(omaptype="RBT", cmpfunction=compareDates)
    fechaInicial=datetime.datetime.strptime(fechaInicial, "%Y-%m-%d").date()
    fechaFinal=datetime.datetime.strptime(fechaFinal, "%Y-%m-%d").date()
    arboles_salario=om.values(arbol_fechas, fechaInicial, fechaFinal)
    max_ofertas=0
    max_ciudad=None
    cantidad_ofertas=0
    for arbol in lt.iterator(arboles_salario):
        mapas_ciudades=om.values(arbol,salarioMin,salarioMax)
        for mapa_ciudad in lt.iterator(mapas_ciudades):
            for ciudad in lt.iterator(mp.keySet(mapa_ciudad)):
                entry=om.get(diccionario_ciudades,ciudad)
                if entry==None:
                    lst=lt.newList()
                    om.put(diccionario_ciudades, ciudad, lst)
                else:
                    lst=me.getValue(entry)
                ofertas=me.getValue(mp.get(mapa_ciudad, ciudad))
                for oferta in lt.iterator(ofertas):
                    lt.addLast(lst,oferta)
                cantidad_ofertas+=lt.size(ofertas)
                if lt.size(lst)>max_ofertas:
                    max_ofertas=lt.size(lst)
                    max_ciudad=ciudad
    ciudades_ordenadas=om.keySet(diccionario_ciudades)
    lista_de_ciudades=lt.newList(datastructure="ARRAY_LIST")
    for ciudad in lt.iterator(ciudades_ordenadas):
        lt.addLast(lista_de_ciudades, ciudad)
    cantidad_ciudades=om.size(diccionario_ciudades)
    lista_ofertas_max_ciudad=me.getValue(om.get(diccionario_ciudades, max_ciudad))
    return cantidad_ofertas, cantidad_ciudades,lista_de_ciudades,first_last(lista_ofertas_max_ciudad, ['published_at','city','compan
```



Descripción

La estructura de datos se encarga de hacer un árbol de fechas al que le sigue un árbol de salarios donde se guardan tablas hash que tienen de llaves las ciudades y de valor una lista que guarda todas las ofertas que encajan. De esta manera, la función del requisito tiene que recorrer estos y luego iterar las veces

necesarias para sacar las listas y las ofertas. Con las tablas hash y las listas se puede resolver todos los requisitos del requerimiento y así no se gasta mucho tiempo en complejidades.

Entrada	Fecha mínima y máxima, salario mínimo y máximo
Salidas	Lista de ofertas de la ciudad máxima, lista de ciudades, ofertas en total, numero de ciudades
Implementado (Sí/No)	Si, grupo

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O (logn)
Paso 2	O(logn)
Paso 3	O(p)
Paso 4	O(s)
TOTAL	***O(p*s*log(n))***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Parametros:

F1: 2023-01-01

F2: 2023-10-01

Salario 1: 6000$

salario2: 8000$

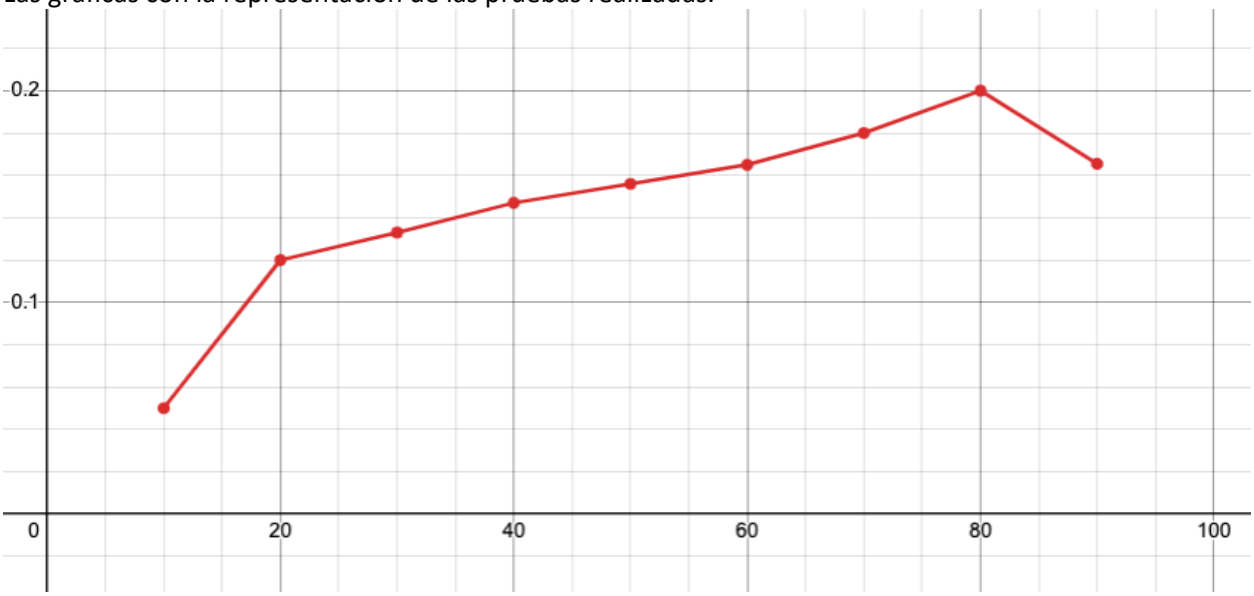
Procesadores	**AMD Ryzen 7 4800HS with Radeon Graphics**
Memoria RAM	8 GB
Sistema Operativo	mac

Entrada	**Tiempo (ms)**
10 pct	0.05
20 pct	0.110
30 pct	0.133

40 pct	0.146
50 pct	0.157
70 pct	0.165
80 pct	0.18
large	0.164

Graficas

Las gráficas con la representación de las pruebas realizadas.



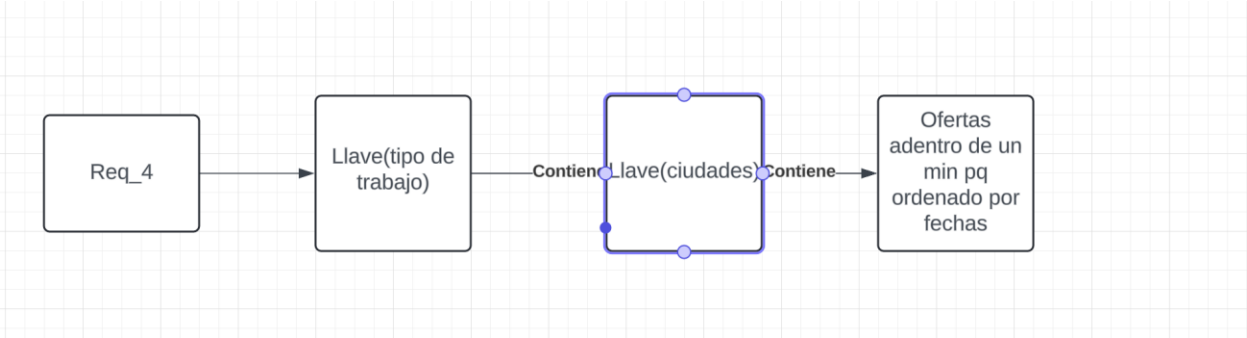
Análisis

En este requerimiento, la carga de datos hace una gran parte del reto, pero de todas maneras es necesario iterar varias veces por unas constantes. Al tener muchos filtros, la cantidad de iteraciones no es tanta, pero igual afectan la complejidad del algoritmo. En la recolección de datos hubo una gran variación de datos, por lo que no hay mucha certeza de lo que la gráfica pueda representar. Hubo alores extremadamente bajos con un porcentaje alto del archivo y otros muy altos en porcentajes bajos.

Comparaciones entre retos pasados y reto actual:

Este requerimiento es diferente al de los retos pasados. Igual hay un componente de comparación de tiempos en la cual esta es más fácil por la dinámica que brinda los árboles binarios. Incluso, la complejidad de buscar por fecha se disminuye ya que se puede implementar desde la carga de datos. Este nos pide más relación entre los archivos de skills y employment types. De esta manera se diverge del reto pasado y le suma a la complejidad y al sustento del requerimiento. Sin embargo, con las nuevas herramientas podemos solucionar estas complicaciones.

Requerimiento 4 Julián Pinto



Descripción

```
def req_4(data,n,ciudad,ubi):
    info=data['req_4']
    info=me.getValue(mp.get(info,ubi))
    info=me.getValue(mp.get(info,ciudad))
    tamanio=mpq.size(info)
    retorno=lt.newList(datastructure='ARRAY_LIST')
    for i in range (0,int(n)):
        oferta=mpq.delMin(info)
        lt.addLast(retorno,oferta)
        id=oferta['id']
        salariomin=me.getValue(mp.get(data['employment_types_id'],id))['salary_from']
        oferta['salario']=salariomin
        skills=(me.getValue(mp.get(data['skills_id'],id)))
        skills=skills_to_str(skills)
        oferta['skills']=skills
    retorno=first_last(retorno,['published_at','title','company_name','experience_level','country_code','city','company_size','workplace_type','salario','skills'])
    return retorno,tamanio
```

Este requerimiento se encarga de tabular N ofertas laborales dado un tipo de trabajo y una ciudad. Lo primero que hace es entrar a una llave de la carga de datos que las llaves son el tipo de trabajo. Una vez entra acá entra a otro diccionario que contiene ciudades, acá encuentra la ciudad buscada y asociada a esta hay un heap el cual está orientado de menor fecha (más reciente) a mayor. Una vez encuentra este heap le saca las N ofertas más recientes, una vez saca estas ofertas, limpia la info para que quede solo que necesitamos y lo mete a una lista que luego se le pasara al tabulador

Entrada	Data_structs,N,ciudad y ubi
Salidas	La lista que se tabula y el conteo de ofertas que cumplen este requisito
Implementado (Sí/No)	Si. Implementado por Julian Pinto

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Parametros(10,Warszawa y 2)

Pasos	**Complejidad**
Buscar el heap dados los criterios	O(1)
Extraer N elementos menores del heap	O (Nlog(n) n es el tamaño del heap
TOTAL	***O(Nlog(n)) donde N es el input del usuario***

Pruebas Realizadas

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

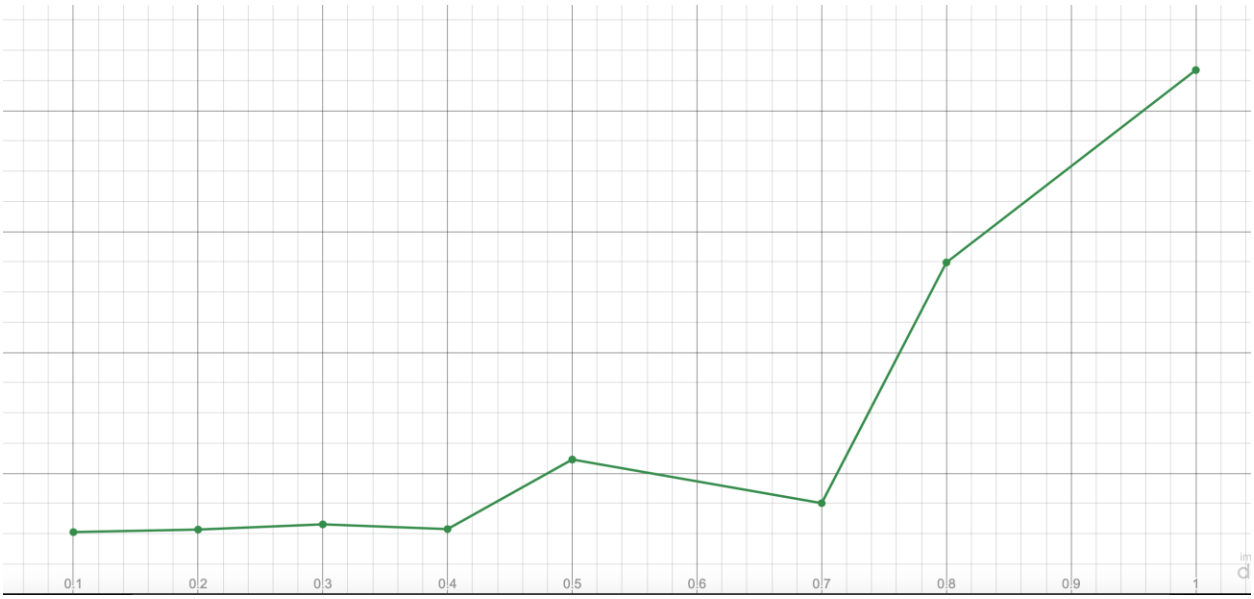
Procesadores	**1.1 GHz Dual-Core Intel Core i3**
Memoria RAM	8 GB
Sistema Operativo	Mac Os 14.4.1 (23E224)

Entrada	**Tiempo (ms)**
10 pct	0.001025
20 pct	0.001067
30 pct	0.001153
40 pct	0.001075
50 pct	0.002226
70 pct	0.001504
80 pct	0.005487
Large 100 pct	0.008671

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

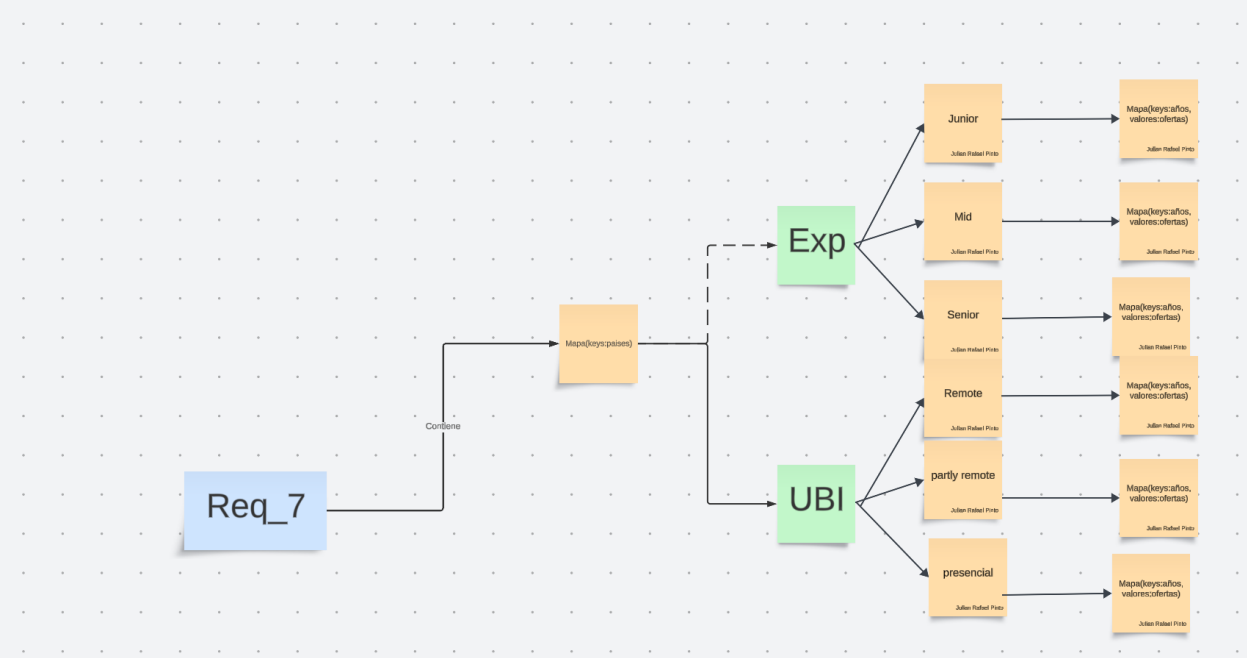
Los tiempos de este requerimiento son tan bajos que cualquier cosa los puede alterar significativamente, la gráfica no muestra una complejidad logarítmica pero si muestra un crecimiento parecido al que esperaríamos con esta complejidad.



Requerimiento 4 Julián Pinto

Descripción

Todas las ofertas están en listas




```python
def req7noskill(data_strcucts,anio,pais,crit):
    maxi=0
    mini=9999999999999
    ofertas=[]
    if crit=='workplace_type':
        key_word='ubi'
    else:
        key_word='exp'
    dicci=data_strcucts['model']['req_7']
    dicci=me.getValue(mp.get(dicci,pais))
    dicci=me.getValue(mp.get(dicci,key_word))
    tamanio=mp.size(dicci)
    headers=lt.newList(datastructure='ARRAY_LIST')
    for nombre in lt.iterator(mp.keySet(dicci)):
        encabezado=me.getKey(mp.get(dicci,nombre))
        lt.addLast(headers,encabezado)
    values=lt.newList(datastructure='ARRAY_LIST')
    for nombre in lt.iterator(mp.keySet(dicci)):
        valor=me.getValue(mp.get(dicci,nombre))
        valor=me.getValue(mp.get(valor,anio))
        valor=req7aux(valor,param=['published_at','title','company_name','country_code','city','company_size'],salario=True,data=data_strcucts)
        ofertas=ofertas+list(lt.iterator(valor))
        valor=lt.size(valor)
        if valor> maxi:
            maxi=valor
            maxim=me.getKey(mp.get(dicci,nombre))
        if valor<mini:
            mini=valor
            minim=me.getKey(mp.get(dicci,nombre))
        lt.addLast(values,valor)
    return headers,values,tamanio,maxi,maxim,mini,minim,ofertas


def req7skill(control,anio,pais):
    ofertas=[]
    data_structs=control['model']
    dicci0=data_structs['req_7']
    dicci0=me.getValue(mp.get(dicci0,pais))
    dicci0=me.getValue(mp.get(dicci0,'exp'))
    for oferta in lt.iterator(mp.keySet(dicci0)):
        valor=me.getValue(mp.get(dicci0,oferta))
        valor=me.getValue(mp.get(valor,anio))
        valor=req7aux(valor,param=['published_at','title','company_name','country_code','city','company_size'],salario=True,data=control)
        ofertas=ofertas+list(lt.iterator(valor))
    dicci=data_structs['req_7_skills']
    dicci=me.getValue(mp.get(dicci,pais))
    lista=me.getValue(mp.get(dicci,anio))
    tamanio=lt.size(lista)
    dictreto=mp.newMap(numelements=400)
    headers=lt.newList(datastructure="ARRAY_LIST")
    values=lt.newList(datastructure="ARRAY_LIST")
    maxi=0
    mini=9999999999999
    for i in range(1,lt.size(lista)):
        iD=lt.getElement(lista,i)
        lista_skill=me.getValue(mp.get(data_structs['skills_id'],iD))
        for j in range(1,lt.size(lista_skill)):
            skill=lt.getElement(lista_skill,j)
            skill=skill['id']
            if not mp.contains(dictreto,skill):
                mp.put(dictreto,skill,1)
            else:
                mp.put(dictreto,skill,(me.getValue(mp.get(dictreto,skill)))+1)
    for nombre in lt.iterator(mp.keySet(dictreto)):
        encabezado=me.getKey(mp.get(dictreto,nombre))
        lt.addLast(headers,encabezado)
    for nombre in lt.iterator(mp.keySet(dictreto)):
        valor=me.getValue(mp.get(dictreto,nombre))
        if valor> maxi:
            maxi=valor
            maxim=me.getKey(mp.get(dictreto,nombre))
        if valor<mini:
            mini=valor
            minim=me.getKey(mp.get(dictreto,nombre))
        lt.addLast(values,valor)
    return headers,values,tamanio,maxi,maxim,mini,minim,ofertas
```

Este requerimiento se encarga de graficar la cantidad de ofertas en un año y en un país separado por un criterio dado por el usuario. Esto es hecho con 2 funciones separadas, si el usuario quiere evaluar experiencia o ubicación entra a una función y si es skills entra a otra. Ambas usan estructuras diferentes, pero de acceso directo. Este requerimiento no tiene una complejidad alta.

Entrada	Pais,añoy criterio
Salidas	Una grafica, cantidad de ofertas,mayor criterio,menor criterio
Implementado (Sí/No)	Si. Implementado por Julian Pinto

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Extraer las listas	O(1)
Poner las llaves y el count de los valores en lista	O (3)
Req7aux	O(3n) tamaño de las listas es n
TOTAL	***O(3n)***

Crit no es skill

Pasos	**Complejidad**
Extraer las listas	O(1)
Poner las llaves y el count de los valores en lista	O (n) donde n es la cantidad de skills
Req7aux	O(m*n) tamaño de las listas es n cantidad de skills es m
TOTAL	O(m*n)

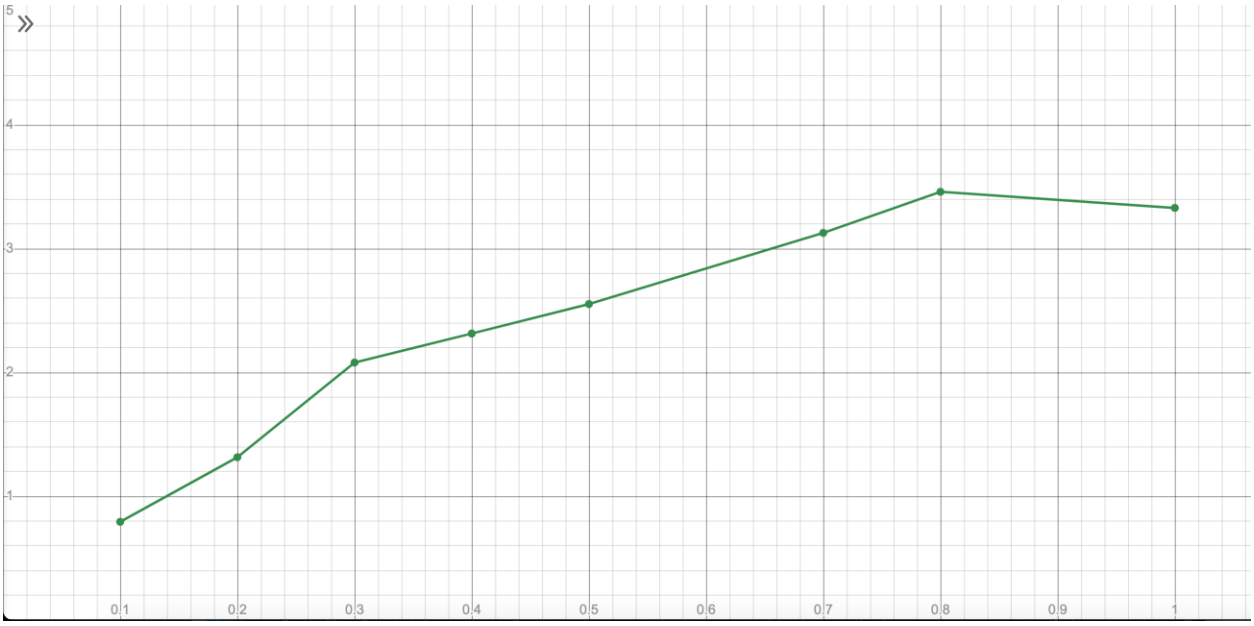
Crit es skill

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron 2022, PL, 2

Procesadores	**1.1 GHz Dual-Core Intel Core i3**
Memoria RAM	8 GB
Sistema Operativo	Mac Os 14.4.1 (23E224)

Entrada	**Tiempo (ms)**
10 pct	0.793250
20 pct	0.315065
30 pct	2.079891
40 pct	1.314706
50 pct	4.552341
70 pct	0.001504
80 pct	0.005487
Large 100 pct	0.008671



Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Los tiempos de este requerimiento son tan bajos que cualquier cosa los puede alterar significativamente, la gráfica muestra que al principio el tiempo es muy sensible a cambios en información, después se va estabilizando, esto es debido a que despues de cierto punto la cantidad de listas empieza a ser iguales.

Requerimiento Ejemplo

Descripción

```python
def get_data(data_structs, id):
    """
    Retorna un dato a partir de su ID
    """
    pos_data = lt.isPresent(data_structs["data"], id)
    if pos_data > 0:
        data = lt.getElement(data_structs["data"], pos_data)
        return data
    return None
```

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

Entrada	Estructuras de datos del modelo, ID.
Salidas	El elemento con el ID dado, si no existe se retorna None
Implementado (Sí/No)	Si. Implementado por Juan Andrés Ariza

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Buscar si el elemento existe (isPresent)	O(n)
Obtener el elemento (getElement)	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 ID 1.

Procesadores	**AMD Ryzen 7 4800HS with Radeon Graphics**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**

small	0.05
5 pct	0.33
10 pct	1.28
20 pct	2.54
30 pct	4.98
50 pct	7.51
80 pct	13.81
large	25.97

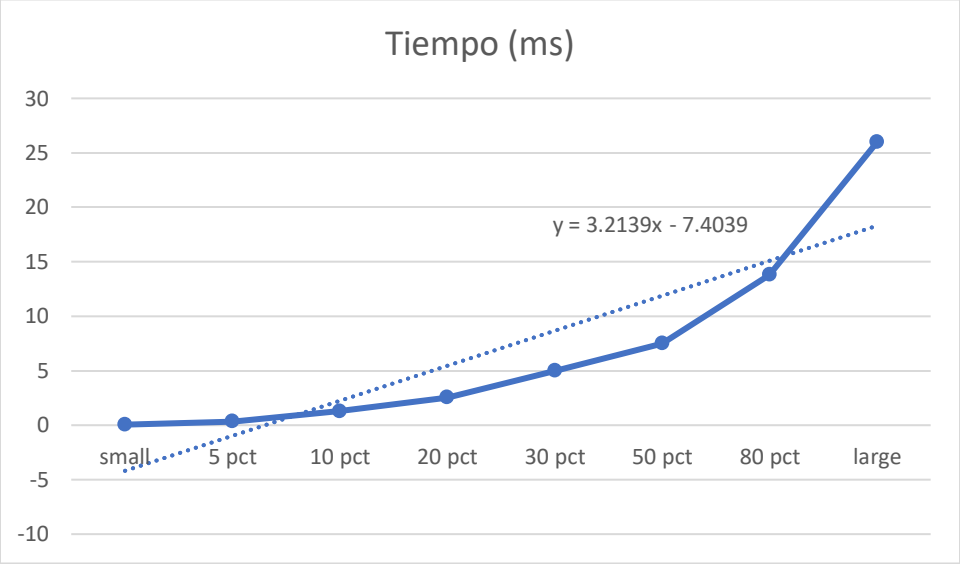
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Dato1	0.05
5 pct	Dato2	0.33
10 pct	Dato3	1.28
20 pct	Dato4	2.54
30 pct	Dato5	4.98
50 pct	Dato6	7.51
80 pct	Dato7	13.81
large	Dato8	25.97

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

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

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