

Requerimiento 1

Precarga

Para la precarga de este requerimiento, utilizamos varias estructuras, dentro de las cuales se encuentran las siguientes:

```
#Req 1
"in_n_out_trips_hash":None,
"in_n_out_trips_tree":None,
"vert_rush_hour":None,
"vert_rush_day":None,
"member_vert":None,
```

```
analyzer["in_n_out_trips_hash"] = mp.newMap(numelements= 770,
                                             maptype='PROBING',
                                             loadfactor=our_loadfactor,
                                             prime=our_prime)

analyzer["in_n_out_trips_tree"] = om.newMap(omaptype='RBT',
                                             comparefunction=compare_r1)
```

```
analyzer["vert_rush_hour"] = mp.newMap(numelements= 770,
                                         maptype='PROBING',
                                         loadfactor=our_loadfactor,
                                         prime=our_prime)
```

```
analyzer["vert_rush_day"] = mp.newMap(numelements= 770,
                                         maptype='PROBING',
                                         loadfactor=our_loadfactor,
                                         prime=our_prime)
```

```
analyzer["member_vert"] = mp.newMap(numelements= 770,
                                     maptype='PROBING',
                                     loadfactor=our_loadfactor,
                                     prime=our_prime)
```

En donde se evidencia que 'in_n_out_trips_hash' es un Map que tiene como llave los vértices y como valor una tupla que el primer valor corresponde con los viajes que parten de este vértice y como segundo valor los viajes que llegan a este vértice. 'In_n_out_trips_tree' en donde se insertan en un RBT los viajes según sean de salida o entrada. 'Vert_rush_hour', 'vert_rush_day' y 'member_ver' que corresponden con mapas. La idea, es primero determinar el número total de salidas por vértice almacenándola como llave-valor, luego crear el RBT y cuando el usuario nos pida el requerimiento, alcanzar los 5 mayores elementos.

Análisis de complejidad

Entradas: Ninguna, sin parámetros.

Salidas: Top 5 de estaciones con el mayor número de inicio de estaciones.

Ya que al momento de ejecutar el requerimiento ya tenemos creado el RBT, la complejidad disminuye considerablemente a tener que crear el RBT desde 0. Particularmente:

```
# Funciones de consulta
def R1_answer(analyzer):
    top_5_list = lt.newList("ARRAY_LIST")

    while lt.size(top_5_list) < 5:
        max_key = om.maxKey(analyzer["in_n_out_trips_tree"])
        vert_list = om.get(analyzer["in_n_out_trips_tree"], max_key)["value"]
        for vert in lt.iterator(vert_list):
            if lt.size(top_5_list) < 5:
                lt.addLast(top_5_list, (vert,max_key))
        om.deleteMax(analyzer["in_n_out_trips_tree"])

    for vert, count in lt.iterator(top_5_list):
        in_trips, out_trips = mp.get(analyzer["in_n_out_trips_hash"],vert)["value"]

        exist_out = om.get(analyzer["in_n_out_trips_tree"], out_trips)
        if exist_out is None:
            new_lt = lt.newList("ARRAY_LIST")
            mp.put(analyzer["in_n_out_trips_tree"], out_trips, new_lt)
            exist_out = om.get(analyzer["in_n_out_trips_tree"], out_trips)

        out_trips_vert_list = exist_out["value"]
        lt.addLast(out_trips_vert_list, vert)

    return top_5_list
```

Se crea una nuevo *array* $O(1)$;

se realizan exactamente 5 *om.maxKey()* que tiene una complejidad de $O(\text{altura})$ y como es RBT $O(\log n)$;

Se agregan a la lista de respuesta cada vez que se encuentra un máximo y se elimina del om para obtener el siguiente $O(\log n)$ por *om.deleteMax()*.

Posteriormente se extrae toda la información sobre este top 5, pero esto es de tiempo constante, pues se realizará siempre 5 veces y no depende de la cantidad de datos $O(1)$.

Por lo que esperamos una complejidad de $O(\log n)$

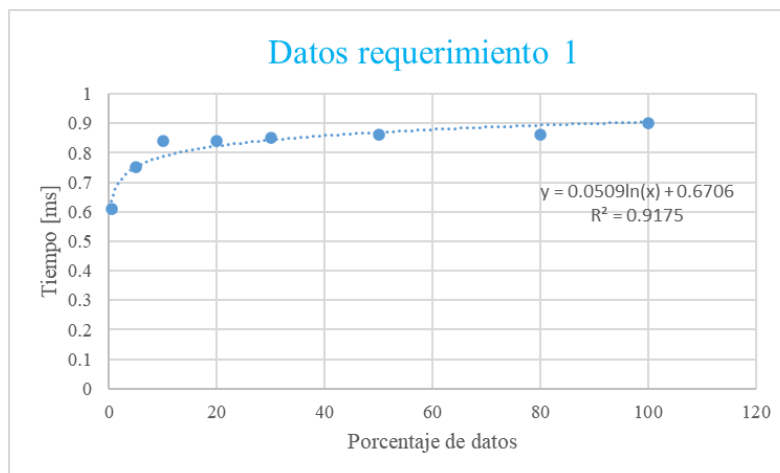
Toma de datos

Para la toma de datos, no se requieren parámetros de entrada, por lo que directamente los datos que encontramos son los siguientes:

Tabla 1: Resultados requerimiento 1, tiempo y memoria.

Nombre	Porcentaje	Tiempo [ms]	Memoria [kB]
small	0.5	0.61	6.06
5pct	5	0.75	7.12
10pct	10	0.84	7.62
20pct	20	0.84	7.69
30pct	30	0.85	8.12
50pct	50	0.86	7.64
80pct	80	0.86	7.64
large	100	0.9	8.17

Que en una gráfica corresponden con:



Gráfica 1: resultados tiempo de espera del requerimiento vs. porcentaje de datos procesados.

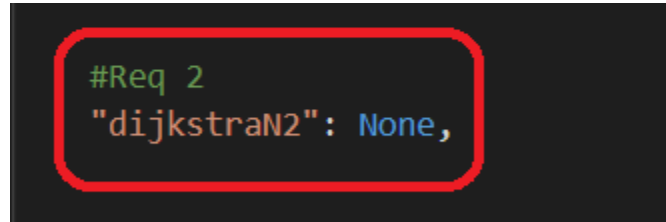
A partir de los datos, podemos observar una tendencia logarítmica solo con intuición. Ya realizando la regresión logarítmica podemos observar claramente un R^2 mayor a 0.9 por lo que

podemos deducir que se trata de una tendencia lineal. La complejidad esperada $O(\log n)$ y a observada $O(\log n)$ coinciden.

Requerimiento 2

Precarga

Para la precarga de este requerimiento, necesitamos tener ya creado nuestro digrafo y dedicar un espacio en la tabla de hash a almacenar el camino de Dijkstra. Este corresponde con el siguiente:



```
#Req 2
"dijkstraN2": None,
```

En particular, nos interesan aquellos caminos con un mínimo de estaciones y que estén dentro de la disponibilidad de los usuarios. Para ello, nuestra estrategia de resolución fue encontrar los caminos de costo mínimo a cada una de las estaciones y evaluar que cumplieran estos dos requisitos principales. En particular, no utilizamos DFS porque resultaban en caminos con un gran número de estaciones, pero no cumplían con los requisitos temporales. Por el contrario, BFS, ocurría lo contrario, encontraba caminos que cumplían con el tiempo, mientras que no cumplían con el número de estaciones. Por tanto, decidimos esta solución puesto que sí existían caminos que cumplían los requisitos mínimos de 5 estaciones y recorrido total menor a 45 minutos y DFS y BFS no los identificaban la mayoría de veces; explicamos esto por el **orden de procesamiento de los arcos**.

Análisis de complejidad

Entradas:

- Nombre de la estación de inicio.

- La disponibilidad del usuario pasa su paseo.

- El número mínimo de estaciones de parada para la ruta (sin incluir la estación de inicio)

- El máximo número de rutas de respuesta

Salidas:

- Posibles rutas que cumplan los criterios, mostrando el tiempo total de recorrido, el orden de las paradas y el número de estaciones visitadas.

Para el análisis de complejidad, iniciemos por revisar el código:

```

def R2_answer(analyzer, v_start, limite_estaciones_min, limite_tiempo, limite_estaciones_maximo):

    rsp_list = lt.newList('ARRAY_LIST')
    do_the_dijkstraN2(analyzer, v_start)
    lt_vertices = gr.vertices(analyzer['graph'])
    limite_recorrido = limite_tiempo/2

    for vertex in lt.iterator(lt_vertices):
        if dj.hasPathTo(analyzer['dijkstraN2'], vertex):
            tiempo_recorrido = dj.distTo(analyzer["dijkstraN2"], vertex)
            camino = dj.pathTo(analyzer["dijkstraN2"], vertex)
            if stack.isEmpty(camino):
                continue
            if tiempo_recorrido <= limite_recorrido and stack.size(camino) > limite_estaciones_min and stack.size(camino) <= limite_estaciones_maximo:
                tupla_camino = (camino, stack.size(camino), tiempo_recorrido)
                lt.addLast(rsp_list, tupla_camino)

    return rsp_list

```

Primero construimos un *array* de respuesta $O(1)$; luego, realizamos Dijkstra una vez el usuario ha ingresado el vértice de inicio (decidimos no realizar esto en la precarga pues se necesitaría hacer Dijkstra para cada vértice de inicio y cuesta demasiado tiempo en la precarga). Dijkstra siempre tiene el peor caso y su complejidad es de $O(E \log V)$; posteriormente, accedemos a todos los vértices del grafo $O(V)$, pero $E \gg V$ por lo que hasta el momento vamos $O(E \log V)$. Posteriormente, realizamos *hasPathTo* y *pathTo* para obtener los *stacks* o pilas que contienen los recorridos y evaluar si cumplen los requisitos como hay una pila por cada vértice y las demás operaciones estas son $O(V)$, sigue dominando $O(E \log V)$. Finalmente, se guarda la información de aquellos caminos que cumplen los cuales: caminos que cumplen $\ll V \ll E$. Por tanto la complejidad final termina siendo $O(E \log V)$.

Por lo que esperamos una complejidad de $O(E \log V)$

Toma de datos:

Para la toma de datos, nos centramos en el siguiente arreglo de condiciones:
(Estos también son guía para simplificar el proceso de input a la consola para nosotros)

»»»»»»»

2

Lake Shore Blvd W / Ontario Dr

7242-Lake Shore Blvd W / Ontario Dr

3

7

45

»»»»»»»

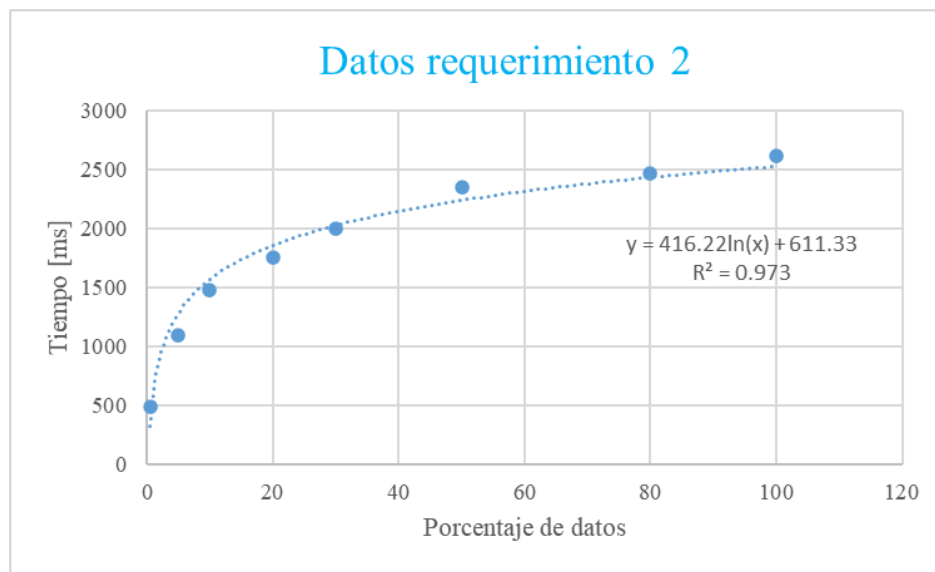
Este arreglo de condiciones, nos dieron los siguientes datos:

Tabla 2: Resultados requerimiento 2, tiempo y memoria.

Nombre	Porcentaje	Tiempo [ms]	Memoria [kB]
--------	------------	-------------	--------------

small	0.5	487.22	1193.91
5pct	5	1098.22	1446.32
10pct	10	1475.67	1646.14
20pct	20	1759.72	1579.06
30pct	30	2000.12	1621.25
50pct	50	2351.19	1237.9
80pct	80	2467.61	1262.44
large	100	2622.1	1374.93

Que en una gráfica corresponden con:



Gráfica 2: resultados tiempo de espera del requerimiento vs. porcentaje de datos procesados. Únicamente observando los datos, podemos intuir una tendencia logarítmica. Cuando realizamos la regresión logarítmica, podemos encontrar un R^2 de 0.973 por lo que se ajusta muy bien a esta. Esto es a primera vista contraintuitivo con respecto a la complejidad esperada, pero también hay que recordar que la cantidad de vértices y arcos no es proporcional con la cantidad de líneas que posee el .csv el cual es nuestro eje x. **Por tanto, a pesar de que no sea exactamente la distribución esperada, podemos explicar este comportamiento por la distribución de líneas del .csv con respecto al el número de arcos y vértices.**

Requerimiento 3

Precarga

Para realizar este requerimiento era necesario tener ya cargado el dígrafo o grafo principal del reto. Con respecto a estructuras auxiliares, realizamos el procesamiento principal de este algoritmo en la precarga, en particular:

```
def do_the_kosaraju(analyzer):  
    analyzer["kosaraju"] = scc.KosarajuSCC(analyzer["graph"])
```

Iniciamos por realizar el algoritmo de Kosaraju y guardarlo en *analyzer['kosaraju']*

```
def Create_R3(analyzer):  
  
    num_Stron_Con_Com = scc.connectedComponents(analyzer["kosaraju"])  
  
    mapa_SCC = analyzer["kosaraju"]["idscc"]  
  
    mapa_scc_minpq_out = mp.newMap(numelements= 770,  
                                   maptype='PROBING',  
                                   loadfactor=our_loadfactor,  
                                   prime=our_prime)  
    mapa_scc_minpq_in = mp.newMap(numelements= 770,  
                                   maptype='PROBING',  
                                   loadfactor=our_loadfactor,  
                                   prime=our_prime)
```

Luego, extraemos de aquí el número de componentes fuertemente conectadas y creamos un mapa que contenga las salidas y entradas de cada uno de los vértices para determinar cuáles tienen los máximos y los mínimos.

```
for vert in lt.iterator(analyzer["vertices_list"]):  
    scc_num = mp.get(mapa_SCC, vert)["value"]  
  
    exist_heap_scc_out = mp.get(mapa_scc_minpq_out, scc_num)  
  
    if exist_heap_scc_out is None:  
        new_heap = inpq.newIndexMinPQ(minpq_compare)  
        mp.put(mapa_scc_minpq_out, scc_num, new_heap)  
        exist_heap_scc_out = mp.get(mapa_scc_minpq_out, scc_num)  
    heap_scc_out = exist_heap_scc_out["value"]  
  
    in_index, out_index = mp.get(analyzer["in_n_out_trips_hash"], vert)["value"]  
  
    inpq.insert(heap_scc_out, vert, -out_index)
```

```

exist_heap_scc_in = mp.get(mapa_scc_minpq_in, scc_num)
if exist_heap_scc_in is None:
    new_heap = inpq.newIndexMinPQ(minpq_compare)
    mp.put(mapa_scc_minpq_in, scc_num, new_heap)
    exist_heap_scc_in= mp.get(mapa_scc_minpq_in, scc_num)

heap_scc_in = exist_heap_scc_in["value"]

inpq.insert(heap_scc_in, vert, -in_index)

```

Luego, para cada nuevo vértice se extrae el número de viajes que salen y entran y se agregan a dos colas de prioridad, una de salidas y otra de llegadas.

```

for valor in range(1, num_Stron_Con_Com+1):
    out_pq= mp.get(mapa_scc_minpq_out,valor)["value"]
    max_out = inpq.min(out_pq)
    in_pq= mp.get(mapa_scc_minpq_in,valor)["value"]
    max_in = inpq.min(in_pq)

    lt.addLast(resp_list, (valor, max_out, max_in, inpq.size(out_pq)))

analyzer["R3_answer"] = (num_Stron_Con_Com, resp_list)

```

Cuando ya se hayan procesado todos los vértices, se extrae el máximo de las colas de prioridad y se agregan a una lista de respuesta para la información adicional. Finalmente, esta lista de agrega a una tupla junto al número de componentes fuertemente conectadas para que cuando el usuario ingrese la opción se extraiga la información.

Análisis de complejidad

Entradas:

Ninguna entrada pues no necesitamos parámetros de inicio.

Salidas:

Número de componentes fuertemente conectadas,
 Estación donde más viajes inician
 Estación donde más viajes terminan


```
def R3_answer(analyzer):  
    return analyzer["R3_answer"]
```

Ya que realizamos todo el procesamiento en la precarga, el tiempo de respuesta de la opción 3 es siempre constante, pues únicamente habrá que imprimir la información almacenada en 'R3_answer'.

Por lo que esperamos una complejidad de $O(1)$

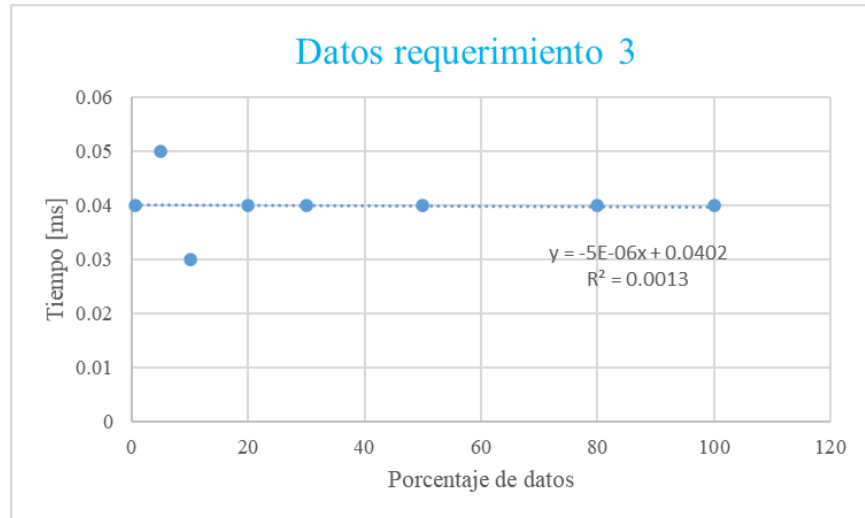
Toma de datos

Para la toma de datos, no se requieren parámetros de entrada, por lo que directamente los datos que encontramos son los siguientes:

Tabla 3: Resultados requerimiento 3, tiempo y memoria.

Nombre	Porcentaje	Tiempo [ms]	Memoria [kB]
small	0.5	0.04	0.17
5pct	5	0.05	0.17
10pct	10	0.03	0.17
20pct	20	0.04	0.17
30pct	30	0.04	0.17
50pct	50	0.04	0.17
80pct	80	0.04	0.17
large	100	0.04	0.17

Ahora, si representamos estos datos en una gráfica obtenemos...



Gráfica 3: resultados tiempo de espera del requerimiento vs. porcentaje de datos procesados.

A partir de la gráfica, podemos observar solo con los datos que no varían mucho, tienden a estar en 0.04. Al realizar la regresión, podemos observar que la pendiente determinada es extremadamente baja y el R^2 también. Por tanto, podemos atribuir la variación de nuestros resultados a procesos secundarios del computador como pueden ser escaneos antimalware y determinar que la complejidad observada es $O(1)$. Es decir, que la complejidad esperada y la complejidad observada coinciden y son $O(1)$.

Requerimiento 4

Precarga

Para Dijkstra, necesariamente necesitamos ingresar el vértice de inicio y no vale la pena probar con cada vértice, por lo que en la precarga únicamente dedicamos un espacio en nuestro map principal

```
#Req 4
"dijsktra":None,
```

Análisis de complejidad

Entradas:

- Nombre de la estación origen.
- Nombre de la estación destino.

Salidas:

- El tiempo total
- La ruta calculada entre las estaciones (incluyendo el origen y el destino)
 - El número de identificación de la estación.
 - El nombre de la estación.

El tiempo promedio a la siguiente estación en la ruta.

```
def R4_answer(analyzer, origen, destination):  
  
    do_the_dijkstra(analyzer, origen)  
  
    if dj.hasPathTo(analyzer["dijkstra"], destination):  
        distance = dj.distTo(analyzer["dijkstra"], destination)  
        camino = dj.pathTo(analyzer["dijkstra"], destination)  
        return distance, camino  
    else:  
        return "No hay camino", None
```

En primer lugar, ya que tenemos nuestro vértice de origen, realizamos el algoritmo de Dijkstra y lo almacenamos en *analyzer['dijkstra']* este tiene complejidad $O(E \log V)$. Posteriormente, realizamos *dj.hasPathTo()* que en el peor caso tendría que hacer *edgeTo()* V veces $O(V)$ pero $V \ll E$ por lo que la complejidad sigue dominada por $O(E \log V)$. Esto mismo ocurre con *dj.distTo()* y *dj.pathTo()* por lo que terminado este la complejidad es de $O(E \log V)$. Por tanto, esperamos una complejidad de $O(E \log V)$.

Toma de datos

Para la toma de datos, se utilizaron los siguientes parámetros:

»»»»»»

4

Fort York Blvd / Capreol Ct

7000-Fort York Blvd / Capreol Ct

Yonge St / St Clair Ave

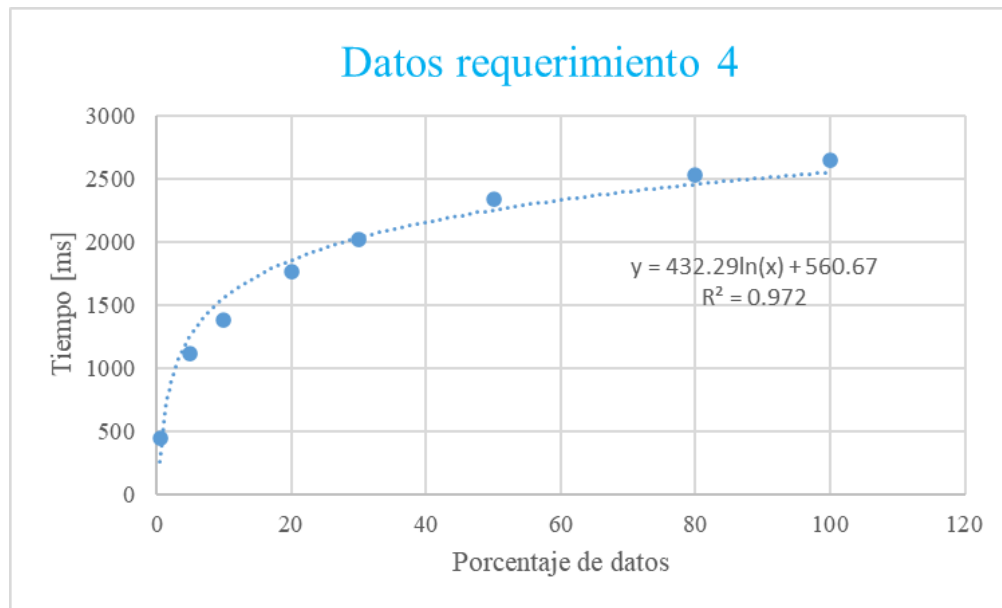
7642-Yonge St / St Clair Ave

»»»»»»

Tabla 4: Resultados requerimiento 4, tiempo y memoria.

Nombre	Porcentaje	Tiempo [ms]	Memoria [kB]
small	0.5	437.73	980.38
5pct	5	1113.18	1013.91
10pct	10	1376.34	1026.23
20pct	20	1763.46	1032.63

30pct	30	2015.27	1033.88
50pct	50	2335.22	1058.31
80pct	80	2527.64	1034.3
large	100	2649.51	1033.69



Gráfica 4: resultados tiempo de espera del requerimiento vs. porcentaje de datos procesados.

Enfocándonos en la distribución de los datos, podemos esperar que la regresión sea logarítmica. De hecho, al calcularla y encontrar el valor de R^2 podemos determinar que es de 0.972 y por tanto el modelo se ajusta muy bien a los datos. [De manera similar al requerimiento 2 en donde la complejidad estaba determinada por la complejidad de Dijkstra](#), tiene sentido que nos vuelva a dar logarítmica y explicamos su relación por la distribución de vértices y arcos con respecto a la cantidad de líneas del .csv.

Requerimiento 5

Precarga

Para la precarga, necesitamos de tres estructuras auxiliares: un RBT que tuviera como llaves las fechas de los viajes y como valor una lista con los servicios de esa fecha. Una Tabla de Hash que tenga como llaves los vértices y como valor una tupla con el número de viajes que llegan y que salen de este vértice. Por último, la última estructura es un map que tiene como llave la hora y como valor el número de viajes que salen y que llegan a esa hora.

```

prime=our_prime)

#Req5
analyzer["Arbol_fechas_usuarios_anuales"] = om.newMap(omaptype='RBT',
comparefunction=compare_r1)

analyzer['viajes_InOut_por_vertice'] = mp.newMap(numelements=770,
maptype= 'PROBING',
loadfactor=our_loadfactor,
prime=our_prime)

analyzer['viajes_por_hora_anuales'] = mp.newMap(numelements= 24,
maptype= 'PROBING',
loadfactor=our_loadfactor,
prime=our_prime)

```

Para reducir el tiempo de ejecución del requerimiento 5, cargamos todos los viajes o *services* en el om

```

def carga_R5(analyzer, service):
    RBT_fechas = analyzer['Arbol_fechas_usuarios_anuales']
    fecha = service['Start Time'][:-6]

    #Revisar si está, si no, agregar una lt como valor
    exist_fecha = om.get(RBT_fechas, fecha)
    if exist_fecha is None:
        lista_fecha = lt.newList('ARRAY_LIST')
        om.put(RBT_fechas, fecha, lista_fecha)

    #Agregar el servicio a la lista
    lista_fecha = om.get(RBT_fechas, fecha)['value']
    lt.addLast(lista_fecha, service)

```

Para esto, se crea la lista vacía en caso que la llave no haya sido ingresada anteriormente y se agrega a la lista el servicio, esto nos permite hacer directamente *om.keys()* en el requerimiento.

Análisis de complejidad

Entradas:

- Fecha inicial de consulta (formato “%m/%d/%Y”).
- Fecha final de consulta (formato “%m/%d/%Y”).

Salidas:

- El total de viajes realizados.

El total de tiempo invertido en los viajes.
La estación de origen más frecuentada.
La estación de destino más utilizada.
La hora del día en la que más viajes inician
La hora del día en la que más viajes terminan

```
def R5_answer(analyzer, fecha_inicial, fecha_final):  
  
    RBT_fechas = analyzer['Arbol_fechas_usuarios_anuales']  
    # Extraemos las fechas que están en [fecha_inicial, fecha_final]  
    lista_fechas_que_cumplen = om.keys(RBT_fechas, fecha_inicial, fecha_final)  
  
    # Ahora inicializamos las variables  
    out_total_viajes = 0  
    out_total_tiempo = 0  
  
    for fecha in lt.iterator(lista_fechas_que_cumplen):  
  
        lista_de_la_fecha = om.get(RBT_fechas, fecha)['value']  
        # Analizar cada servicio en la lista  
        for service in lt.iterator(lista_de_la_fecha):  
  
            out_total_viajes += 1  
            out_total_tiempo += service['Trip Duration']  
            actualizar_viajes_anuales_InOut(analyzer, service)  
            actualizar_viajes_anuales_hora(analyzer, service)
```

Ya dentro del requerimiento, iniciamos por hacer *om.keys()* el cual tiene una complejidad de $O(\text{altura} + \text{numelements})$ que por ser RBT es $O(\log n + \text{numelements})$. Posteriormente, recorremos estos *numelements* para determinar el número de viajes que cumplen estas condiciones, el tiempo total y agregar cada viaje a la tabla de hash. Como este proceso se realiza *numelements* veces y cada uno es de tiempo constante su complejidad es de $O(\text{numelements})$ por tanto, la complejidad final asociada es $O(\log n + \text{numelements})$.

```

def actualizar_viajes_anuales_InOut(analyzer, service):

    # Primero, necesitamos reconstruir los vértices
    ver1 = "{0}-{1}".format(service['Start Station Id'], service['Start Station Name'])
    ver2 = "{0}-{1}".format(service['End Station Id'], service['End Station Name'])
    map = analyzer["viajes_InOut_por_vertice"]

    # Luego, podemos guardar la tupla con los services que entran
    exist_ver1 = mp.get(map, ver1)
    if exist_ver1 is None:
        mp.put(map, ver1, (0,0))
        exist_ver1 = mp.get(map, ver1)

    in_trips1, out_trips1 = exist_ver1["value"]
    out_trips1 += 1

    mp.put(map, ver1, (in_trips1, out_trips1))

    # Y los que salen
    exist_ver2 = mp.get(map, ver2)
    if exist_ver2 is None:
        mp.put(map, ver2, (0,0))
        exist_ver2 = mp.get(map, ver2)

    in_trips2, out_trips2 = exist_ver2["value"]
    in_trips2 += 1

    mp.put(map, ver2, (in_trips2, out_trips2))

```

actualizar_viajes_anuales_InOut() se realiza una vez a tiempo constante, pues utiliza tablas de Hash y hace mp.put y mp.get. Lo mismo para actualizar_viajes_anuales_hora():

```

def actualizar_viajes_anuales_hora(analyzer, service):
    # En la llave, está la hora y como valor una tupla (n_inicio, n_final)
    horaInicio = hour_determiner(service['Start Time'])
    horaFin = hour_determiner(service['End Time'])
    map = analyzer['viajes_por_hora_anuales']

    exist_ver1 = mp.get(map, horaInicio)
    if exist_ver1 is None:
        mp.put(map, horaInicio, (0,0))
        exist_ver1 = mp.get(map, horaInicio)

    in_hora, out_hora = exist_ver1["value"]
    out_hora += 1

    mp.put(map, horaInicio,(in_hora, out_hora))

    # Y los que salen
    exist_ver2 = mp.get(map, horaFin)
    if exist_ver2 is None:
        mp.put(map, horaFin, (0,0))
        exist_ver2 = mp.get(map, horaFin)

    in_hora, out_hora = exist_ver2["value"]
    in_hora += 1

    mp.put(map, horaFin,(in_hora, out_hora))

```

Por tanto, esperamos una complejidad final de $O(\log n + \text{numelements})$.

Toma de datos

Para la toma de datos, no se requieren parámetros de entrada, por lo que directamente los datos que encontramos son los siguientes:

»»»»»»»»

5

02/24/2021

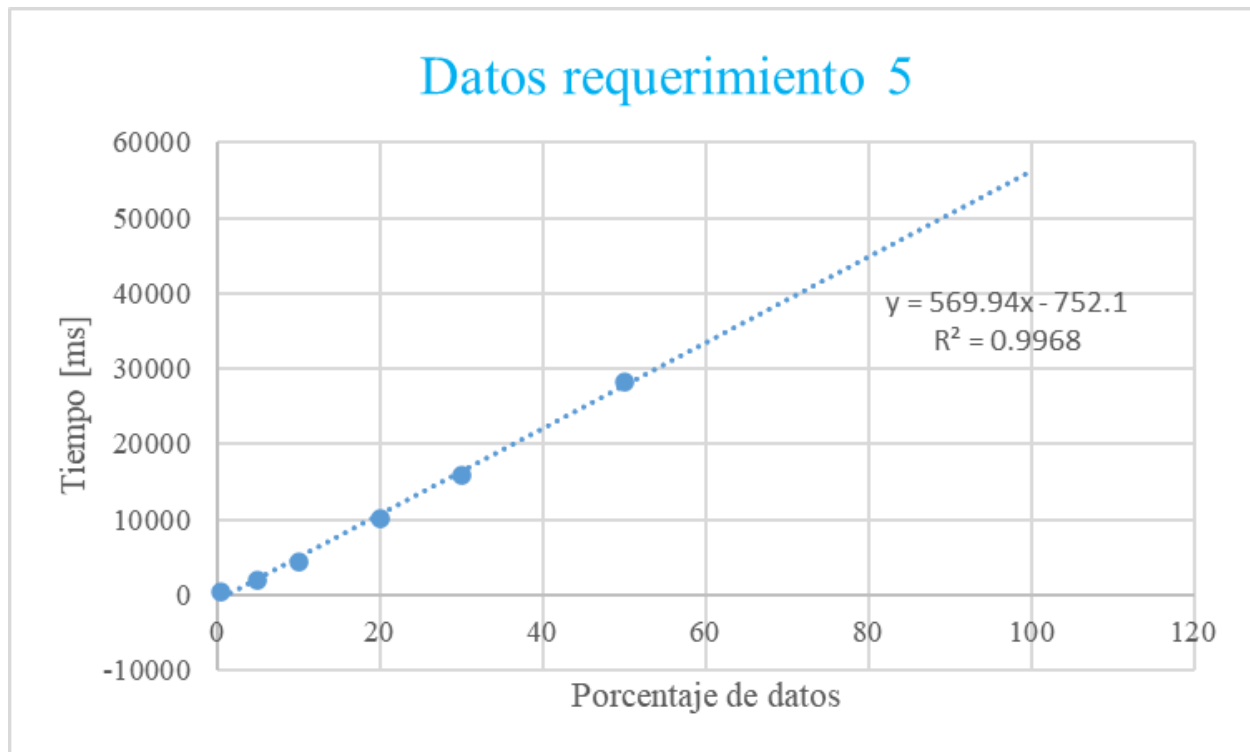
07/09/2021

»»»»»»»»

Tabla 5: Resultados requerimiento 5, tiempo y memoria.

Nombre	Porcentaje	Tiempo [ms]	Memoria [kB]
small	0.5	418.69	75.97

5pct	5	2039.41	78.11
10pct	10	4492.94	82.59
20pct	20	10239.04	92.09
30pct	30	15804.55	96.14
50pct	50	28320.48	102.15
80pct	80		
large	100		



Gráfica 5: resultados tiempo de espera del requerimiento vs. porcentaje de datos procesados.

A partir de los datos, podemos observar de manera general que tienen una tendencia lineal, al realizar la regresión, nos da un valor de R^2 superior a 0.9 por lo que podemos afirmar que se trata de una tendencia lineal. **Por tanto, podemos afirmar que la complejidad esperada $O(n)$ y la observada coinciden.** Aquí cabe mencionar que la esperada formalmente es $O(\log n + \text{numelements})$ pero $\text{numelements} \gg \log n$ y numelements podemos esperar a que sea directamente proporcional a los datos. Finalmente, hemos de mencionar que intentámos múltiples veces tomar datos más allá de 50pct pero nuestros computadores simplemente

crasheaban o se demoraban demasiado para llegar a si quiera cargar los datos, por lo que usamos con los datos reales y confiables para realizar el análisis de complejidad observado.

Requerimiento 6

Precarga

```
def bikes_info(analyzer, service, ver1, ver2):
    bike_id = service["Bike Id"].split(".")[0]

    map_bike_info = analyzer["bikes_info"]

    exist_bike = mp.get(map_bike_info, bike_id)

    if exist_bike == None:
        mp.put(map_bike_info, bike_id, (0,0))
        exist_bike = mp.get(map_bike_info, bike_id)

    num_viajes, time = exist_bike["value"]

    num_viajes += 1
    time += float(service["Trip Duration"])

    mp.put(map_bike_info, bike_id, (num_viajes, time))

    bikes_pt2(analyzer, service, bike_id, ver1, ver2)
```

Para la precarga, utilizamos un map en donde tenemos como llave la bike y como valor una tupla que contenga el número de viajes registrados por esta y un acumulado del tiempo que se ha utilizado. Por otro lado, tenemos otro map en el que la llave nuevamente es el bike id pero como valor tiene un minpq indexado que nos servirá para extraer el máximo de la estación de salida y estación de llegada.

Análisis de complejidad

Entradas:

El identificador de la bicicleta en el sistema.

Salidas:

El total de viajes en los que ha participado dicha bicicleta.

El total de horas de utilización de la bicicleta.

La estación en la que más viajes se han iniciado en esa bicicleta

La estación en la que más viajes ha terminado dicha bicicleta.

```
def R6_answer(analyzer, bike_inp):
    num_viajes, timpo_rec = mp.get(analyzer["bikes_info"], bike_inp)["value"]
    vert_max_out = inpq.min2(mp.get(analyzer["bike_out_pq"], bike_inp)["value"])
    vert_max_in = inpq.min2(mp.get(analyzer["bike_in_pq"], bike_inp)["value"])

    return num_viajes, timpo_rec, vert_max_out, vert_max_in
```

Con respecto a la complejidad esperada, tenemos como primera operación *mp.get()* el cual corresponde con $O(1)$ pues es get en una Tabla de Hash. Luego, hacemos min2 en el minpqIndexado, esta operación la definimos nosotros y es muy similar a min solo que retorna el valor y no la llave. Así, hacemos ‘peek’ en el elemento de la raíz de la cola de prioridad, por lo que esperaríamos que tuviera una complejidad de $O(1)$. [Repetimos una única vez este proceso, por lo que podríamos esperar una complejidad final constante o \$O\(1\)\$.](#)

Toma de datos

Para la toma de datos, no se requieren parámetros de entrada, por lo que directamente los datos que encontramos son los siguientes:

»»»»»»»»

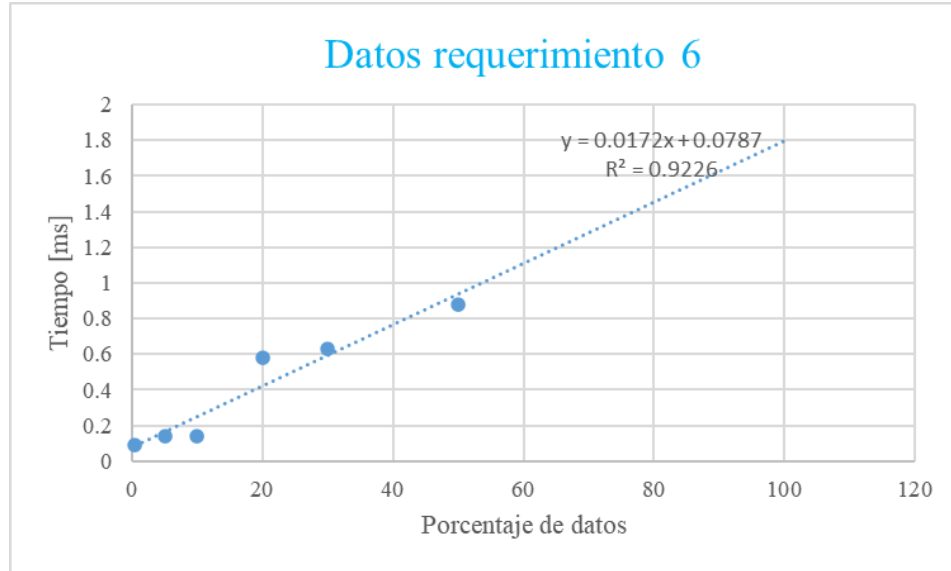
6

5282

»»»»»»»»

Tabla 6: Resultados requerimiento 6, tiempo y memoria.

Nombre	Porcentaje	Tiempo [ms]	Memoria [kB]
small	0.5	0.09	1.52
5pct	5	0.14	0.17
10pct	10	0.14	0.17
20pct	20	0.58	0.17
30pct	30	0.63	0.17
50pct	50	0.88	0.17
80pct	80		
large	100		



Gráfica 6: resultados tiempo de espera del requerimiento vs. porcentaje de datos procesados.

A partir de los resultados, podemos inferir que tiene un crecimiento definitivamente no constante, pero sí de baja tasa. Realizando la regresión, podemos determinar un $R^2 = 0.92$ por lo que tenemos una buena parte de la variación explicada a través del modelo lineal. **Nota:** al realizar la toma de datos, intentamos múltiples veces correr más allá de 50pct y simplemente ninguno de nuestros computadores funcionaba por debajo de la hora. Por tanto, dejamos únicamente los primeros datos que sí pudimos procesar. Tenemos en cuenta que entra mayor sea la cantidad de datos, podríamos esperar una mayor diferencia en la variación para determinar si es de complejidad constante o lineal. Sin embargo, con lo que obtuvimos si podemos determinar *a priori* que la complejidad esperada y la complejidad obtenida son la misma $O(1)$ con el aumento en tiempo de ejecución principalmente debido a variaciones en la ejecución por el sistema o por los efectos desconocidos de la implementación en `minpqIndex()`.

Requerimiento 7

Precarga

```
def carga_R7(analyzer, service):
    RBT_fechas = analyzer['date_tree_2']
    fecha = service['Start Time']

    #Revisar si está, si no, agregar una lt como valor
    exist_fecha = om.get(RBT_fechas, fecha)
    if exist_fecha is None:
        lista_fecha = lt.newList('ARRAY_LIST')
        om.put(RBT_fechas, fecha, lista_fecha)
        exist_fecha = om.get(RBT_fechas, fecha)
    #Agregar el servicio a la lista
    lista_fecha = exist_fecha['value']
    lt.addLast(lista_fecha, service)
```

Para el Requerimiento 7 se utiliza dentro del diccionario principal del proyecto (analyzer) un mapa ordenado, cuyo nombre es “date_tree_2”, y en él tenemos un mapa ordenado donde las llaves son las fechas de inicio (incluyendo la hora) de los viajes y como valores listas que contienen a todos los viajes que tienen exactamente la misma fecha. Esto con la finalidad de que cuando el usuario ingrese el rango de fechas de su interés, solo los viajes que ocurrieron dentro de dicho rango serán tomados para ejecutar el requerimiento.

Análisis de complejidad

Entradas:

- Nombre de la estación.
- Fecha y hora de inicio.
- Fecha y hora de finalización.

Salidas:

- Número de viajes iniciados en la estación ingresada.
- Número de viajes terminados en la estación ingresada.
- El viaje de mayor duración iniciando en la estación ingresada.
- La estación en la que terminaron la mayoría de viajes.

```

def R7_answer(analyzer, vert_inp, date1, date2):
    RBT_fechas = analyzer['date_tree_2']

    lista_fechas_que_cumplen = om.keys(RBT_fechas, date1, date2)

    analyzer["R7_vert_pq"] = inpq.newIndexMinPQ(minpq_compare)
    analyzer["in_n_out_trips_hash2"] = None

    for fecha in lt.iterator(lista_fechas_que_cumplen):

        lista_de_la_fecha = om.get(RBT_fechas, fecha)['value']

        for service in lt.iterator(lista_de_la_fecha):
            vert1, vert2 = vert_names(analyzer, service)
            addTo_in_n_out_trips_hash2(analyzer, vert1, vert2, vert_inp)

            if vert1 == vert_inp:
                update_heap_vert(analyzer, vert2, service)

    in_trips, out_trips = analyzer["in_n_out_trips_hash2"]
    most_trips_ended = inpq.min2(analyzer["R7_vert_pq"])

    return in_trips, out_trips, most_trips_ended

```

Se comienza por tomar todos los viajes que están dentro del rango que el usuario ingresó por parámetro mediante la operación `om.keys()`, la cual tiene una complejidad asociada de **$O(n \log(n))$** .

Acto seguido se crea una Priority Queue Indexada vacía, acción que tiene una complejidad de **$O(1)$** .

Luego se inicia un ciclo el cual recorrerá todas las fechas que estén dentro del rango ingresado por el usuario, en el peor de los casos se tendrá que recorrer todos los viajes realizados por usuarios tipo casual, haciendo el primer ciclo **$O(n)$** . Dentro de este ciclo se accede a un mapa cuya llave es la fecha que se tenga en el momento, esta operación es **$O(1)$** . Si el caso es el peor, se obtendrá una lista con un único elemento por lo que el recorrer ese elemento también será **$O(1)$** .

```
def vert_names(analyzer, service):
    map = analyzer["name_to_vertice"]

    start_name = service["Start Station Name"]
    end_name = service["End Station Name"]

    if start_name == "":
        start_name = "Unknown Station"

    if end_name == "":
        end_name = "Unknown Station"

    start_id = service["Start Station Id"]
    end_id = service["End Station Id"]

    vertice1 = "{0}-{1}".format(start_id, start_name)
    vertice2 = "{0}-{1}".format(end_id, end_name)

    return (vertice1, vertice2)
```

La primera función que se ejecuta con el viaje es “*vert_names*” y lo que hace es que accede al diccionario del viaje y crea los 2 vértices que se encuentran en este y los retorna. Al ser acciones constantes tienen una complejidad de **O(1)**.

```
def addTo_in_n_out_trips_hash2(analyzer, ver1, ver2, vert_inp):

    if analyzer["in_n_out_trips_hash2"] == None:
        analyzer["in_n_out_trips_hash2"] = (0,0)

    in_trips, out_trips = analyzer["in_n_out_trips_hash2"]
    if ver1 == vert_inp:
        out_trips += 1

    if ver2 == vert_inp:
        in_trips +=1

    analyzer["in_n_out_trips_hash2"] = (in_trips, out_trips)
```

Posteriormente se pasan los 2 vértices a la función “*addTo_in_n_out_trips_hash2*”, esa función lo único que hace es contar cuantas veces el vértice ingresado por parámetro está como una estación de salida y como una estación de llegada. Sin embargo al momento de ejecutarla cada vez que se llama solo realiza operación de complejidad constante, por lo que su complejidad asociada también es **O(1)**.

```
def update_heap_vert(analyzer, vert, service):
    heap = analyzer["R7_vert_pq"]
    if not inpq.contains(heap, vert):
        inpq.insert(heap, vert, 0)

    inpq.increaseKey2(heap, vert)
```

Finalmente al final del ciclo está la función “*update_heap_vert*”, que lo que hace es verificar si un vértice dado está dentro de la priority queue, para ello utiliza una Tabla de hash donde se le pasa como llave el nombre del vértice y este indica si el vértice hace parte de la priority queue, esta acción es $O(1)$. En caso de que no esté agrega el vértice a la priority queue con un valor de 0. Este al ser el valor mínimo posible hace que el vértice se quede en su posición actual y no realiza más acciones, por lo que tiene una complejidad de $O(1)$. La siguiente línea “*increaseKey2*”, que lo que hace es tomar el valor actual que tenga el vértice y lo disminuye en 1, indicando que el vértice aumentó su número de apariciones un 1 y haciendo que haga swim en el heap (posiblemente), que en el peor caso sería un swim de $O(\log(n))$.

Finalmente una vez termine el ciclo se toma el número de veces que apareció en llegadas y en salida el vértice indicado y de la priority queue se toma el elemento de la cima, que sería el vértice donde la mayoría de viajes que empezaron en el vértice ingresado por parámetro terminan. Estas operaciones son $O(1)$.

De manera que la complejidad final sería:

$$O(n \log(n)) + 1 + n * (1 + 1 + 1 + 1 + 1 + \log(n) + 1) = O(n \log(n))$$

Toma de datos

~~~~~

7

01/01/2021 00:00

01/02/2021 10:00

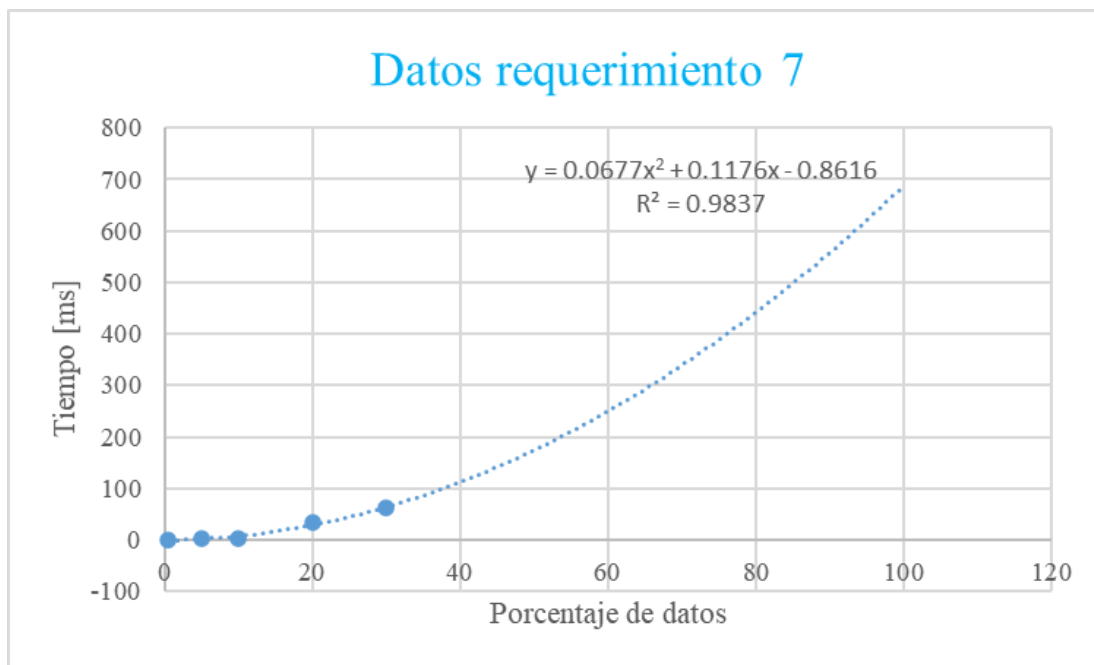
Wellington St W / York St

7469-Wellington St W / York St

~~~~~


Tabla 7: Resultados requerimiento 6, tiempo y memoria.

Nombre	Porcentaje	Tiempo [ms]	Memoria [kB]
small	0.5	0.41	12.15
5pct	5	2.25	12.64
10pct	10	2.09	14.19
20pct	20	32.97	27.83
30pct	30	62.2	15.22
50pct	50		
80pct	80		
large	100		



Gráfica 7: resultados tiempo de espera del requerimiento vs. porcentaje de datos procesados.

Nota: nosotros intentamos tomar datos más allá del 30pct pero nuestros computadores empezaban a presentar fallas en su sistema o simplemente no corrían después de 1h por lo que decidimos únicamente dejar aquellos datos de los cuales tenemos confianza para poder hacer el análisis de complejidad observada. Entendemos que a mayor cantidad de información pues más

fácil será distinguir entre las diferentes complejidades, pero estas limitaciones técnicas no nos permitieron avanzar más allá.

Con respecto a los datos obtenidos sin todavía hacer la regresión, podemos observar un crecimiento no lineal, queda en duda si es puntualmente logarítmico, exponencial o cuadrático. Al momento de realizar la regresión, de las mencionadas anteriormente, encontramos que la regresión cuadrática es la que presenta un mayor valor de R^2 igual a 0.987. Lo único que queda claro con las pocas medidas que pudimos realizar, es que el ejecutar este requerimiento es supremamente costoso en términos de computación, pues debido a que pide realizar un análisis sobre todos los viajes dentro de un rango que incluye tiempo y hora, hace que el tomar dicho rango con las estructuras que tenemos sea bastante costoso, sin mencionar el hecho de que esta es solo la primera parte del requerimiento pues luego es necesario recorrer todos los elementos de dicho rango para así finalmente llegar a la respuesta final. Por lo que sería de esperarse que a medida que aumenta el número de elementos que se analizan el tiempo de ejecución crezca con un comportamiento similar al exponencial.