

Análisis Reto-03

Juan Esteban Espitia Reyes - je.espitia@uniandes.edu.co - 202516958

Samuel Felipe Tovar - sf.tovarp1@uniandes.edu.co - 202211625

Fabio Andrés Salazar Ochoa - fa.salazar@uniandes.edu.co - 202511238

- Requerimiento <1>

```

121 def req_1(catalog, code, min_delay, max_delay):
122
123     inicio = get_time()
124
125     vuelos = catalog["flights"]
126     filtrados = al.new_list()
127
128     for i in range(vl.size(vuelos)):
129         v = vl.get_element(vuelos, i)
130
131         if v["carrier"] == code:
132             sched = v["sched dep time"]
133             real = v["dep time"]
134
135             sched_min = sched.hour * 60 + sched.minute
136             real_min = real.hour * 60 + real.minute
137
138             diff = real_min - sched_min
139
140             # Ajusta si cruza la medianoche
141             if diff < -720:
142                 diff += 1440
143             elif diff > 720:
144                 diff -= 1440
145
146             if min_delay <= diff <= max_delay:
147                 vuelo_con_retraso = v.copy()
148                 vuelo_con_retraso["retraso"] = round(diff, 2)
149                 al.add_last(filtrados, vuelo_con_retraso)
150
151     arbol = bst.new_map()
152
153     for i in range(al.size(filtrados)):
154         vuelo = al.get_element(filtrados, i)
155         key = (vuelo["retraso"], vuelo["date"], vuelo["dep time"])
156         bst.put(arbol, key, vuelo)
157
158
159     def another_rec(nodo, lista):
160         if nodo is not None:
161             another_rec(nodo["left"], lista)
162             al.add_list(lista, nodo["value"])
163             another_rec(nodo["right"], lista)
164
165     def inorder(tree):
166         lista = al.new_list()
167         if tree["root"] is not None:
168             inorder_rec(tree["root"], lista)
169
170         return lista
171
172     filtrados_ordenados = inorder(arbol)
173     total = al.size(filtrados_ordenados)
174
175     primeros = al.new_list()
176     ultimos = al.new_list()
177     limite = min(5, total)
178
179     for i in range(limite):
180         elem = al.get_element(filtrados_ordenados, i)
181         info = (
182             "id_vuelo": elem["id"],
183             "codigo_vuelo": elem["flight"],
184             "fecha": elem["date"].strftime("%Y-%m-%d"),
185             "hora": elem["dep time"].strftime("%H:%M"),
186             "numero_asolinea": elem["name"],
187             "codigo_asolinea": elem["carrier"],
188             "aeropuerto_origen": elem["origin"],
189             "aeropuerto_destino": elem["dest"],
190             "retraso_min": elem["retraso"]
191         )
192         al.add_last(primeros, info)
193
194     if total > 10:
195         for i in range(total - 5, total):
196             elem = al.get_element(filtrados_ordenados, i)
197             info = (
198                 "id_vuelo": elem["id"],
199                 "codigo_vuelo": elem["flight"],
200                 "fecha": elem["date"].strftime("%Y-%m-%d"),
201                 "hora": elem["dep time"].strftime("%H:%M"),
202                 "numero_asolinea": elem["name"],
203                 "codigo_asolinea": elem["carrier"],
204                 "aeropuerto_origen": elem["origin"],
205                 "aeropuerto_destino": elem["dest"],
206                 "retraso_min": elem["retraso"]
207             )
208             al.add_last(ultimos, info)
209
210     final = get_time()
211     tiempo = delta_time(inicio, final)
212
213     return tiempo, total, primeros, ultimos

```

Este requerimiento se encarga de filtrar vuelos según el código y un rango de retraso dentro de los vuelos.

Entrada	Catálogo, codigo, delay min, delay max
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Sí. Implementado por Samuel

Análisis de la complejidad:

Pasos	Complejidad
Recorrido de todos los vuelos para filtrar	$O(n)$
Inserción en BST (Desordenado)	$O(h)$, pero en promedio $\log N$
En el peor caso si h es igual a los filtrados	$O(n^2)$
Complejidad de la función (peor caso):	$O(n^2)$
Complejidad de la función (caso promedio):	$N \log N$

Tiene esta complejidad ya que en el peor de los casos en un bst sin ordenar si o si la complejidad es de N , sin embargo, en casos promedio normalmente es $\log N$, y al integrar las inserciones a la función esto escala a $N \log N$, en el peor caso siendo n^2

• Requerimiento <2>

```

def req_2(catalog, dest, min_anticipation, max_anticipation):
    inicio = get_time()

    flights = catalog["Flights"]
    filtrados = al.new_list()

    for i in range(len(flights)):
        vuelo = al.get_element(flights, i)

        if vuelo["dest"] == dest:
            sched_arr = vuelo["sched_arr_time"]
            real_arr = vuelo["arr_time"]

            sched_min = sched_arr.hour * 60 + sched_arr.minute
            real_min = real_arr.hour * 60 + real_arr.minute

            diff = real_min - sched_min

            if diff < -720:
                diff += 1440
            elif diff > 720:
                diff -= 1440

            if diff < 0:
                anticipo = abs(diff)
                if min_anticipation < anticipo <= max_anticipation:
                    vuelo_copia = dict(vuelo)
                    vuelo_copia["anticipation"] = round(anticipo, 2)
                    al.add_last(filtrados, vuelo_copia)

    arbol = bst.new_map()

    for i in range(al.size(filtrados)):
        vuelo = al.get_element(filtrados, i)
        key = (vuelo["anticipation"], vuelo["date"], vuelo["arr_time"])
        bst.put(arbol, key, vuelo)

    def inorden_rec(nodo, lista):
        if nodo is not None:
            inorden_rec(nodo["left"], lista)
            al.add_list(lista, nodo["value"])
            inorden_rec(nodo["right"], lista)

    def inorden(tree):
        lista = al.new_list()
        if tree["root"] is not None:
            inorden_rec(tree["root"], lista)
        return lista

    filtrados_ordenados = inorden(arbol)
    total = al.size(filtrados_ordenados)

    primeros = al.new_list()
    ultimos = al.new_list()
    limite = min(5, total)

    for i in range(limite):
        f = al.get_element(filtrados_ordenados, i)
        info = {
            "id": f["id"],
            "flight": f["flight"],
            "date": f["date"].strftime("%Y-%m-%d"),
            "hora_llegada_real": f["arr_time"].strftime("%H:%M"),
            "hora_llegada_sched": f["sched_arr_time"],
            "airline_name": f["airline"],
            "airline_code": f["carrier"],
            "origin": f["origin"],
            "dest": f["dest"],
            "anticipation_min": f["anticipation"]
        }
        al.add_list(primeros, info)

    if total > 10:
        for i in range(total - 5, total):
            f = al.get_element(filtrados_ordenados, i)
            info = {
                "id": f["id"],
                "flight": f["flight"],
                "date": f["date"].strftime("%Y-%m-%d"),
                "hora_llegada_real": f["arr_time"].strftime("%H:%M"),
                "hora_llegada_sched": f["sched_arr_time"],
                "airline_name": f["airline"],
                "airline_code": f["carrier"],
                "origin": f["origin"],
                "dest": f["dest"],
                "anticipation_max": f["anticipation"]
            }
            al.add_list(ultimos, info)

    final = get_time()
    tiempo = delta_time(inicio, final)

    return tiempo, total, primeros, ultimos

```

Esta función filtra los datos según el destino, el mínimo de anticipación y el máximo según los vuelos.

Entrada	Catálogo, destino, min anticipación, max anticipación
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Sí. Implementado por Fabio

Análisis de la complejidad temporal:

Pasos	Complejidad
Recorrido de todos los vuelos para filtrar	$O(n)$
Inserción en BST (Desordenado)	$O(h)$, pero en promedio $\log N$
En el peor caso si h es igual a los filtrados	$O(n^2)$
Complejidad de la función (peor caso):	$O(n^2)$
Complejidad de la función (caso promedio):	$N \log N$

Como en el anterior, tiene esta complejidad ya que en el peor de los casos en un bst sin ordenar si o si la complejidad es de N , sin embargo, en casos promedio normalmente es $\log N$, y al integrar las inserciones a la función esto escala a $N \log N$ debido al bucle para hacer la inserción, en el peor caso siendo n^2

• Requerimiento <3>

```

def req_3(catalog, c_carrier, c_destino, rango_d):
    """
    Retorna el resultado del requerimiento 3
    """
    # posicas rango_d es un list de dos elementos [min, max]
    inicio = get_time()
    rango_ini = rango_d[0]
    rango_fin = rango_d[1]
    vuelos = catalog["flights"]
    filtrados = al.new_list()

    for i in range(al.size(vuelos)):
        temp = al.get_element(vuelos, i)
        if temp["carrier"] == c_carrier and temp["dest"] == c_destino:
            # temp["distance"] ya es float
            distancia = temp["distance"]

            if rango_ini <= distancia <= rango_fin:
                al.add_last(filtrados, temp)

    # Usar RBT para ordenar por (distancia, fecha, hora_llegada_real)
    arbol = rbt.new_map()

    for i in range(al.size(filtrados)):
        vuelo = al.get_element(filtrados, i)
        key = vuelo["distance"], vuelo["date"], vuelo["arr_time"]
        rbt.put(arbol, key, vuelo)

    # Recorrer inorden del RBT para obtener vuelos ordenados
    def inorden_rec(nodo, lista):
        if nodo is not None:
            inorden_rec(nodo["left"], lista)
            al.add_list(lista, nodo["value"])
            inorden_rec(nodo["right"], lista)

    def inorden(tree):
        lista = al.new_list()
        if tree["root"] is not None:
            inorden_rec(tree["root"], lista)
        return lista

    filtrados_ordenados = inorden(arbol)
    total = al.size(filtrados_ordenados)

    # Extraer primeros y ultimos 5
    primeros = al.new_list()
    ultimos = al.new_list()

    if total > 10:
        # Primeros 5
        for i in range(5):
            vuelo = al.get_element(filtrados_ordenados, i)
            info = {
                "id": vuelo["id"],
                "flight": vuelo["flight"],
                "date": vuelo["date"].strftime("%Y-%m-%d"),
                "hora_llegada_real": vuelo["arr_time"].strftime("%H:%M"),
                "carrier": vuelo["carrier"],
                "name": vuelo["name"],
                "origin": vuelo["origin"],
                "dest": vuelo["dest"],
                "distance": round(vuelo["distance"], 2)
            }
            al.add_last(primeros, info)

        # Ultimos 5
        for i in range(total - 5, total):
            vuelo = al.get_element(filtrados_ordenados, i)
            info = {
                "id": vuelo["id"],
                "flight": vuelo["flight"],
                "date": vuelo["date"].strftime("%Y-%m-%d"),
                "hora_llegada_real": vuelo["arr_time"].strftime("%H:%M"),
                "carrier": vuelo["carrier"],
                "name": vuelo["name"],
                "origin": vuelo["origin"],
                "dest": vuelo["dest"],
                "distance": round(vuelo["distance"], 2)
            }
            al.add_last(ultimos, info)
    else:
        # Si hay 10 o menos, mostrar todos en primeros
        for i in range(total):
            vuelo = al.get_element(filtrados_ordenados, i)
            info = {
                "id": vuelo["id"],
                "flight": vuelo["flight"],
                "date": vuelo["date"].strftime("%Y-%m-%d"),
                "hora_llegada_real": vuelo["arr_time"].strftime("%H:%M"),
                "carrier": vuelo["carrier"],
                "name": vuelo["name"],
                "origin": vuelo["origin"],
                "dest": vuelo["dest"],
                "distance": round(vuelo["distance"], 2)
            }
            al.add_last(primeros, info)

    final = get_time()
    tiempo = delta_time(inicio, final)
    return tiempo, total, primeros, ultimos

```

El requerimiento filtra los vuelos por código de Carrier, ciudad de destino y rangos de distancia, el cual es una lista que tiene el rango deseado para que la función compare

Entrada	Catálogo, rangos distancia, ciudad destino, código carrier
---------	--

Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Sí. Implementado por Juan Esteban

Análisis de la complejidad temporal:

Pasos	Complejidad
Recorrido de todos los vuelos para filtrar	$O(n)$
Inserción en RBT (Ordenado)	$\log N$
En el peor caso si h es igual a los filtrados en su iteración para la inserción	$O(N \log N)$
Complejidad de la función (peor caso):	$N \log N$
Complejidad de la función (caso promedio):	$N \log N$

Esta función al usar estructuras de datos ordenadas como RBT, reduce la complejidad temporal a todos los casos a $N \log N$, a diferencia de las anteriores que tienen $N \log N$ en el caso promedio, pero en la posibilidad del peor caso de las otras es N cuadrado.

• Requerimiento <4>

```

def run_4(catalog, t_inicial, t_final, h_inicial, h_final, n):
    inicio = get_time()
    flighs = catalog['flights']
    fecha_inic = datetime.strptime(t_inicial, "%Y-%m-%d")
    fecha_fin = datetime.strptime(t_final, "%Y-%m-%d")
    t_inic = datetime.strptime(h_inicial, "%H:%M").time()
    t_fin = datetime.strptime(h_final, "%H:%M").time()

    def en_franja():
        # Permite tratar normal y la que cruza medianoche
        if t_inic < t_fin:
            return (t_inic < t) & (t < t_fin)
        else:
            # cruza medianoche; válido si t > inicio o t < final
            return (t > t_inic) | (t < t_fin)

    por_aerolineas = alp.new_map(al.size(flighs), 0.7)

    for i in range(al.size(flighs)):
        v = al.get_element(i)
        if (fecha_inic < v['fly']) & (v['fly'] < fecha_fin):
            t_prog = v['sched.dep.time']
            if en_franja(t_prog):
                code = v['carrier']
                reg = alp.get(por_aerolineas, code)
                if reg is None:
                    reg = {}
                    reg['code': code,
                     'name': v['name'],
                     'count': 1,
                     'sum_air': 0.0,
                     'sum_dist': 0.0,
                     'best_air': None,           # menor duración (flight)
                     'best_dt_prog': None,       # datetime de fecha-hora programada (para desempate)
                     'best_flight': None,        # dict con datos del vuelo ganador
                     ]
                else:
                    air = v['air_time']
                    dist = v['distance']
                    reg['count'] += 1
                    reg['sum_air'] += air
                    reg['sum_dist'] += dist
                    # Consolidar a vuelo de menor duración
                    # Combinar fecha y hora para comparación cronológica
                    dt_prog = datetime.combine(v['date'], v['sched.dep_time'])

                    if (reg['best_air'] is None) or (air < reg['best_air']) or \
                       (air == reg['best_air']) and dt_prog < reg['best_dt_prog']:
                        reg['best_air'] = air
                        reg['best_dt_prog'] = dt_prog
                        reg['best_flight'] = v

                reg['best_flight'] = {
                    'id': v['id'],
                    'origin': v['origin'],
                    'dest': v['dest'],
                    'date': v['date'].strftime("%Y-%m-%d"),
                    'sched.dep.time': v['sched.dep_time'].strftime("%H:%M"),
                    'origin': v['origin'],
                    'dest': v['dest'],
                    'air': air
                }
            alp.put(por_aerolineas, code, reg)

    heap = pq.new_heap(is_min_pq=True)
    keys = alp.keys(por_aerolineas)

    for i in range(al.size(keys)):
        code = al.get_element(keys[i])
        reg = alp.get(por_aerolineas, code)
        if reg['best_flight'] is not None:
            prom_air = reg['sum_air'] / reg['count']
            prom_dist = reg['sum_dist'] / reg['count']

            bf = reg['best_flight']
            resumen = {
                'codigo_aerolinea': reg['code'],
                'nombre_aerolinea': reg['name'],
                'count': reg['count'],
                'sum_air': reg['sum_air'],
                'duration_promedio_min': round(prom_air, 2),
                'distancia_promedio_millas': round(prom_dist, 2),
                'vuelos_resumen': [
                    {
                        'bf': bf,
                        'codigo_vuelo': bf['flight'],
                        'fecha_hora_valide_programada': v.strftime("%Y-%m-%d %H:%M"),
                        'origen': bf['origin'],
                        'destino': bf['dest'],
                        'duracion_min': round(bf['air_time'], 2)
                    }
                ]
            }

            priorizado = (-reg['count'], reg['code']) # min-heap = mayor count primero; empate por código asc
            pq.insert(heap, priorizado, resumen)

    seleccion = al.pq.list()
    total_heap = pq.size(heap)
    extreme = n if total_heap > n else total_heap

    for j in range(extreme):
        val = pq.remove(heap)
        al.add_list(seleccion, val)

    final = get_time()
    tiempo = delta_time(inicio, final)
    return tiempo, seleccion

```

Se filtran los elementos según la diferencia de fechas, luego la diferencia de horas y extrae n muestras según la entrada

Entrada	Catálogo, fecha ini, fecha final, hora ini, hora final, n
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Sí

Análisis de la complejidad temporal:

Pasos	Complejidad
Recorrido de todos los vuelos	$O(N)$ Siendo n el tamaño del catalogo
Put y Get dentro de la función (x iteracion)	$O(N)$
Key set por aerolínea	$O(z)$
Insertar el Key set en el heap	$O(z \log z)$
Extraer del heap los datos según n	$O(n \log z)$
Complejidad de la función:	$Z \log Z$

A diferencia que las anteriores, se integra heap. Queda como complejidad predominante la inserción de los keyset en el heap debido a que z es mucho más pesado de lo que podría ser n al ser n una muestra.

- **Requerimiento <5>**

```

def resample_catalog(f_inicial, f_final, muestra, n):
    inicio = get_time()
    flights = catalog["flights"]
    por_muestras = np.floor(muestra * len(flights)) / 0.7

    fecha_ini = datetime.strptime(f_inicial, "%Y-%m-%d")
    fecha_fin = datetime.strptime(f_final, "%Y-%m-%d")

    for i in range(al.size(flights)):
        v = al.get_element(flights, i)
        f_v = v["date"]

        if v["dest"] == destino and (fecha_ini <= f_v <= fecha_fin):
            t_sched = v["sched.arr_time"]
            t_real = v["arr.time"]

            sched_min = t_sched.hour * 60 + t_sched.minute
            real_min = t_real.hour * 60 + t_real.minute
            punt = real_min - sched_min

            if punt < -720:
                punt += 24480
            elif punt > 720:
                punt -= 24480

            code = v["carrier"]
            reg = np.get(por_muestras, code)

            if reg is None:
                reg = {
                    "code": code,
                    "name": v["name"],
                    "sun_punt": 0.0,
                    "sun_air": 0.0,
                    "sun_dist": 0.0,
                    "max_dist": -1.0,
                    "max_flight": None
                }

            reg["sun_punt"] += punt
            reg["count"] += 1

            reg["sun_air"] += v["distance"]
            dist_v = v["distance"]
            reg["sun_dist"] += dist_v

            if dist_v > reg["max_dist"]:
                reg["max_dist"] = dist_v
                reg["max_flight"] = {
                    "id": id,
                    "flight": v["flight"],
                    "date": v["date"].strftime("%Y-%m-%d"),
                    "arr_time": v["arr.time"].strftime("%H:%M"),
                    "origin": v["origin"],
                    "dest": v["dest"],
                    "air_time": v["air_time"]
                }

            np.put(por_muestras, code, reg)

    heap = np.zeros(len(heap), dtype=[('x', int), ('y', float)])
    keys = np.zeros(len(por_muestras), dtype=[('x', int), ('y', float)])

    for i in range(al.size(keys)):
        code = al.get_element(keys, i)
        reg = np.get(por_muestras, code)

        if reg["count"] == 0 and reg["max_flight"] is not None:
            prom_punt = reg["sun_punt"] / reg["count"]
            prom_air = reg["sun_air"] / reg["count"] if reg["count"] > 0 else 0.0
            prom_dist = reg["sun_dist"] / reg["count"] if reg["count"] > 0 else 0.0

            af = reg["max_flight"]
            resumen = {
                "codigo_aerolinea": reg["code"],
                "nombre_aerolinea": reg["name"],
                "distancia_promedio": reg["sun_dist"],
                "duracion_promedio_min": round(prom_air, 2),
                "duracion_promedio_max": round(prom_dist, 2),
                "particularidad_promedio_min": round(prom_punt, 2),
                "velocidad_promedio": (
                    "({} km/h)".format(prom_air) if prom_air > 0 else "0 km/h"),
                "codigos_vuelos": af["flight"],
                "fecha_hora_llegada": af["date"] + " " + af["arr_time"],
                "origen": af["origin"],
                "destino": af["dest"],
                "duracion_min": round(af["air_time"], 2)
            }

            a_resumido = ((prom_dist / total_dist), code) + mas_cercano_a_0_primeros; empate por codigo
            prioridad = (abs(prom_punt), reg["code"])
            pq.insert(heap, prioridad, resumen)

    seleccion = al.new_list()
    total_heap = pq.list(heap)
    extraer = min(n, total_heap)

    for i in range(extraer):
        val = pq.remove(heap)
        al.add_last(seleccion, val)

    final = get_time()
    tiempo = delta_time(inicio, final)

    return tiempo, extraer, seleccion

```

Esta función filtra según las fechas, el destino y se selecciona la muestra de datos que se quiere tener al final por primeros y últimos.

Entrada	Catálogo, destino, fecha ini, fecha fin, tamaño de la muestra
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Sí

Pasos	Complejidad
Recorrido de todos los vuelos	$O(N)$ Siendo n el tamaño del catalogo
Put y Get dentro de la función (x iteración)	$O(N)$
Key set por aerolínea	$O(z)$
Insertar el Key set en el heap	$O(z \log z)$
Extraer del heap los datos según n	$O(n \log z)$
Complejidad de la función:	$Z \log Z$

En esta función es bastante similar que la anterior, ya que usa heap y tablas de hash que en este caso se usa la estructura de linear probing, haciendo que el mapa sea eficiente.

- **Requerimiento <6>**

Estructuras de Datos y Algoritmos

ISIS-1225

Este requerimiento filtra entre distancias y fechas.

Entrada	Catálogo, distancia inicial, distancia final, fecha inicial, fecha final, m
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Sí

Análisis de la complejidad temporal:

Pasos	Complejidad
Recorrido de todos los vuelos	$O(N)$ Siendo n el tamaño del catalogo
Put y Get dentro de la función (x iteración)	$O(N)$
Cálculo de promedio de cada aerolínea	$O(N)$
Cálculo de desviación estándar	$O(N)$
Búsqueda del vuelo más cercano	$O(N)$
Key set por aerolínea	$O(z)$
Insertar el Key set en el heap	$O(z \log z)$
Extraer del heap los datos según m	$O(m \log z)$
Complejidad de la función:	$Z \log Z$

Igual que en la 5, 4. Solo que en esta se toma el turno de sacar más datos como el promedio, la desviación y la búsqueda del vuelo más cercano. Se tiene también en cuenta el retraso de los vuelos.

