

Análisis Reto-03

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

Samuel Felipe Tovar - sf.tovarpl@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(al.size(vuelos)):
129         v = al.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             # Ajuste 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     def inorder_rec(nodo, lista):
159         if nodo is not None:
160             inorder_rec(nodo["left"], lista)
161             al.add_last(lista, nodo["value"])
162             inorder_rec(nodo["right"], lista)
163
164     def inorder(tree):
165         lista = al.new_list()
166         if tree["root"] is not None:
167             inorder_rec(tree["root"], lista)
168         return lista
169
170     filtrados_ordenados = inorder(arbol)
171     total = al.size(filtrados_ordenados)
172
173     primeros = al.new_list()
174     ultimos = al.new_list()
175     limite = min(5, total)
176
177     for i in range(limite):
178         elem = al.get_element(filtrados_ordenados, i)
179         info = {
180             "id_vuelo": elem["id"],
181             "codigo_vuelo": elem["flight"],
182             "fecha": elem["date"].strftime("%Y-%m-%d"),
183             "hora_salida": elem["dep_time"].strftime("%H:%M"),
184             "nombre_avion": elem["name"],
185             "codigo_avion": elem["carrier"],
186             "aeropuerto_origen": elem["origin"],
187             "aeropuerto_destino": elem["dest"],
188             "retraso_min": elem["retraso"]
189         }
190         al.add_last(primeros, info)
191
192     if total > 10:
193         for i in range(total - 5, total):
194             elem = al.get_element(filtrados_ordenados, i)
195             info = {
196                 "id_vuelo": elem["id"],
197                 "codigo_vuelo": elem["flight"],
198                 "fecha": elem["date"].strftime("%Y-%m-%d"),
199                 "hora_salida": elem["dep_time"].strftime("%H:%M"),
200                 "nombre_avion": elem["name"],
201                 "codigo_avion": elem["carrier"],
202                 "aeropuerto_origen": elem["origin"],
203                 "aeropuerto_destino": elem["dest"],
204                 "retraso_min": elem["retraso"]
205             }
206             al.add_last(ultimos, info)
207
208     final = get_time()
209     tiempo = delta_time(inicio, final)
210
211     return tiempo, total, primeros, ultimos
212
```

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

Entrada	Catálogo, código, 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(al.size(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["data"], vuelo["arr_time"])
        bst.put(arbol, key, vuelo)

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

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

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

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

    for i in range(limitas):
        f = al.get_element(filtrados_ordenados, i)
        info = {
            "id": f["id"],
            "flight": f["flight"],
            "data": f["data"].strftime("%Y-%m-%d"),
            "hora_llegada_real": f["arr_time"].strftime("%H:%M"),
            "airline_name": f["name"],
            "airline_code": f["carrier"],
            "origin": f["origin"],
            "dest": f["dest"],
            "anticipation_min": f["anticipation"]
        }
        al.add_last(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"],
                "data": f["data"].strftime("%Y-%m-%d"),
                "hora_llegada_real": f["arr_time"].strftime("%H:%M"),
                "airline_name": f["name"],
                "airline_code": f["carrier"],
                "origin": f["origin"],
                "dest": f["dest"],
                "anticipation_min": f["anticipation"]
            }
            al.add_last(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
    """
    # por lo tanto 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["flight"]
    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:
            # time["distance"] ya es float
            distancia = temp["distance"]

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

    # usar NBT para ordenar por (distancia, fecha, hora_llegada_real)
    arbol = nbt.new_map()

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

    # Recorrido inorden del NBT para obtener vuelos ordenados
    def inorder_rec(nodo, lista):
        if nodo is not None:
            inorder_rec(nodo["left"], lista)
            al.add_last(lista, nodo["value"])
            inorder_rec(nodo["right"], lista)

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

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

    # Extraer primeros y últimos 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"],
                "distancia": round(vuelo["distance"], 2)
            }
            al.add_last(primeros, info)

        # Últimos 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"],
                "distancia": 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"],
                    "distancia": 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)	Si. 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 req_4(catalog, t_inicial, t_final, h_inicial, h_final, n):
    inicio = get_time()
    flights = catalog["flights"]

    fecha_ini = datetime.strptime(t_inicial, "%Y-%m-%d")
    fecha_fin = datetime.strptime(t_final, "%Y-%m-%d")

    t_ini = datetime.strptime(h_inicial, "%H:%M").time()
    t_fin = datetime.strptime(h_final, "%H:%M").time()

    def es_franja(t):
        # Hora normal y la que cruza medianoche
        if t_inicial < t < t_final:
            return (t_inicial < t < t_final)
        else:
            # Cruza medianoche: válido si t es inicio o t es final
            return (t == t_inicial) or (t == t_final)

    por_aerolineas = nlp.new_map(nlp.size(flights), 0.7)

    for i in range(nlp.size(flights)):
        v = nlp.get_element(flights, i)

        # v = v["data"]
        if (fecha_ini < v["scheduled_time"] < fecha_fin):
            if es_franja(v["time"]):
                # code = v["carrier"]
                reg = nlp.get(por_aerolineas, code)
                if reg is None:
                    reg = {
                        "code": code,
                        "name": v["name"],
                        "count": 0,
                        "sum_air": 0.0,
                        "sum_dist": 0.0,
                        "best_dir_prog": None,  # mejor dirección (float)
                        "best_dir_prog": None,  # dirección de fecha-hora programada (para desempate)
                        "best_flight": None  # dict con datos del vuelo ganador
                    }

                air = v["air_time"]
                dist = v["distance"]

                reg["count"] += 1
                reg["sum_air"] += air
                reg["sum_dist"] += dist

                # Candidato a vuelo de menor duración
                # Comparar fecha y hora para comparación cronológica
                dt_prog = datetime.combine(v["date"], v["sched_dep_time"])

                if (reg["best_dir"] is None) or (air < reg["best_dir"]) or \
                    (air == reg["best_dir"] and dt_prog < reg["best_dir_prog"]):
                    reg["best_dir"] = air
                    reg["best_dir_prog"] = dt_prog
                    reg["best_flight"] = {
                        "air": v["air"],
                        "flight": v["flight"],
                        "date": v["date"].strftime("%Y-%m-%d"),
                        "sched_dep_time": v["sched_dep_time"].strftime("%H:%M"),
                        "origin": v["origin"],
                        "dest": v["dest"],
                        "air_time": air
                    }

                nlp.put(por_aerolineas, code, reg)

    heap = nlp.new_heap(nlp.size(por_aerolineas))
    keys = nlp.keys(por_aerolineas)

    for i in range(nlp.size(keys)):
        code = nlp.get_element(keys, i)
        reg = nlp.get(por_aerolineas, code)
        if reg["count"] > 0 and 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_aerolineas": reg["code"],
                "nombre_aerolineas": reg["name"],
                "vuelos_programados": reg["count"],
                "direccion_promedio_miles": round(prom_dist, 2),
                "distancia_promedio_millas": round(prom_dist, 2),
                "vuelos_menor_duracion": {
                    "air": bf["air"],
                    "codigo_vuelo": bf["flight"],
                    "fecha_hora_salida_programada": f"({bf['date']}) ({bf['sched_dep_time']})",
                    "origen": bf["origin"],
                    "destino": bf["dest"],
                    "duracion_min": round(bf["air_time"], 2)
                }
            }

            prioridad = (-reg["count"], reg["code"])  # min-heap = mayor count primero; empate por código asc
            nlp.insert(heap, prioridad, resumen)

    seleccion = nlp.new_list()
    total_heap = nlp.size(heap)
    extrasear = 0
    while total_heap > 0 and total_heap > n:
        for i in range(extrasear):
            val = nlp.remove(heap)
            nlp.add_last(seleccion, val)

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

        return tiempo, extrasear, 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 funcion (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 req_5(conting, t_inicial, t_final, destino, n):
    inicio = get_time()
    flights = catalog("flights")
    nro_serie = len(conting) * n
    nro_serie = min(nro_serie, 0.7)

    fecha_ini = datetime.strptime(t_inicial, "%Y-%m-%d")
    fecha_fin = datetime.strptime(t_final, "%Y-%m-%d")

    for i in range(1, nro_serie + 1):
        v = al_get_element(flights, i)

        t_v = v["date"]

        if v["dest"] == destino and (fecha_ini <= t_v <= fecha_fin):
            t_scheduled = v["scheduled_time"]
            t_real = v["real_time"]

            sched_min = t_scheduled.hour * 60 + t_scheduled.minute
            real_min = t_real.hour * 60 + t_real.minute

            punt = real_min - sched_min

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

            code = v["carrier"]
            req = nro_get_por_serie(code)

            if req is None:
                req = {
                    "code": code,
                    "name": v["name"],
                    "num_punt": 0.0,
                    "count": 0,
                    "num_sir": 0.0,
                    "num_dist": 0.0,
                    "num_dist": 0.0,
                    "num_flight": None
                }

            req["num_punt"] += punt
            req["count"] += 1

            req["num_sir"] += v["sir_time"]
            dist_v = v["distance"]
            req["num_dist"] += dist_v

            if dist_v > req["num_dist"]:
                req["num_dist"] = dist_v
                req["num_flight"] = {
                    "date": v["date"],
                    "date": v["date"].strftime("%Y-%m-%d"),
                    "sir_time": v["sir_time"].strftime("%H:%M"),
                    "origin": v["origin"],
                    "dest": v["dest"],
                    "sir_time": v["sir_time"]
                }

            nro_get_por_serie(code, req)

    heap = pq_new_heap(is_min_req=True)
    keys = nro_get_por_serie(keys)

    for i in range(1, nro_serie + 1):
        code = al_get_element(keys, i)
        req = nro_get_por_serie(code)

        if req["count"] > 0 and req["num_flight"] is not None:
            prom_punt = req["num_punt"] / req["count"]
            prom_sir = req["num_sir"] / req["count"] if req["count"] > 0 else 0.0
            prom_dist = req["num_dist"] / req["count"] if req["count"] > 0 else 0.0

            of = req["num_flight"]
            resumen = {
                "codigo_serie": req["code"],
                "nombre_serie": req["name"],
                "num_serie": req["count"],
                "duracion_promedio_min": round(prom_sir, 2),
                "distancia_promedio_millas": round(prom_dist, 2),
                "puntaje_promedio_min": round(prom_punt, 2),
                "codigo_serie": {
                    "date": of["date"],
                    "date": of["date"].strftime("%Y-%m-%d"),
                    "sir_time": of["sir_time"],
                    "origin": of["origin"],
                    "dest": of["dest"],
                    "sir_time": of["sir_time"]
                }
            }

            # Prioridad: ((promedio puntaje), code) = mas cercano a 0 primero; empate por código
            prioridad = (abs(prom_punt), req["code"])
            pq_insert(heap, prioridad, resumen)

    seleccion = al_new_list()
    total_heap = pq_size(heap)
    extraer = min(heap, total_heap)

    for i in range(extraer):
        val = pq_remove(heap)
        al_add_list(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

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)	$O(N)$
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>**

[illegible]

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)	Si

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.

