Proyecto de curso

El problema de la Planificación de vuelos

Fundamentos de programación funcional y concurrente

Escuela de Ingeniería de Sistemas y Computación

Profesor Juan Francisco Díaz Frias Monitora Emily Núñez

Noviembre de 2025

1. Introducción

El presente proyecto tiene por objeto observar el logro de los siguientes resultados de aprendizaje del curso por parte de los estudiantes:

- Utiliza un lenguaje de programación funcional y/o concurrente, usando las técnicas adecuadas, para implementar soluciones a un problema dado.
- Aplica conceptos fundamentales de la programación funcional y concurrente, utilizando un lenguaje de programación adecuado como SCALA, para analizar un problema, modelar, diseñar y desarrollar su solución.
- Construye argumentaciones formalmente correctas, usando las técnicas de argumentación apropiadas, para sustentar la corrección de programas funcionales y para sustentar la mejoría en el desempeño de programas concurrentes frente a las soluciones secuenciales.
- Trabaja en equipo, desempeñando un rol específico y llevando a cabo un conjunto de actividades, usando mecanismos de comunicación efectivos con sus compañeros y el profesor, para desarrollar un proyecto de curso.

Para ello el estudiante:

- Desarrolla un programa funcional concurrente utilizando un lenguaje de programación adecuado como SCALA para resolver en grupo un proyecto de programación planteado por el profesor.
- Escribe un informe de proyecto, presentando los aspectos más relevantes del desarrollo realizado, para que un lector pueda evaluar el proyecto.
- Desarrolla una presentación digital, con los aspectos más relevantes del desarrollo realizado, para sustentar el trabajo ante los compañeros y el profesor.

Adicionalmente el estudiante:

- Desarrolla programas funcionales puros con estructuras de datos inmutables utilizando recursión, reconocimiento de patrones, mecanismos de encapsulación, funciones de alto orden e iteradores para resolver problemas de programación.
- Combina la programación funcional con la POO entendiendo las limitaciones y ventajas de cada enfoque para resolver problemas de programación.
- Razona sobre la estructura de programas funcionales utilizando la inducción como mecanismo de argumentación para demostrar propiedades de los programas que construye.
- Paraleliza algoritmos escritos en estilo funcional usando técnicas de paralelización de tareas y datos para lograr acelerar sus tiempos de ejecución.
- Razona sobre programas con paralelismo en datos y paralelismo en tareas en un contexto funcional para concluir sobre las ganancias en tiempo con respecto a las versiones secuenciales.
- Aplica técnicas de análisis de desempeño de programas paralelos a los programas funcionales paralelizados para concluir sobre el grado de aceleración de los tiempos de ejecución con respecto a las versiones secuenciales.
- Utiliza colecciones paralelas en la escritura de programas para lograr mejoras de desempeño en su ejecución.

2. El problema de la planificación de vuelos

Uno de los problemas más frecuentes que encuentran los viajeros aéreos (por lo general delegan el encontrar su solución a una agencia de viajes) es el de organizar la ruta de viaje más *adecuada* para ir de un sitio a otro. Por *adecuada* se entiende que satisfaga los requerimientos del viajero: Que en lo posible la ruta solo utilice un conjunto de aerolíneas de su gusto, que sea lo más corta posible, que solo utilice una aerolínea así sea un poco más larga la ruta o un poco más demorado el viaje, que haga un número mínimo de escalas, que en el camino haga escalas en algunos sitios determinados, etc ....

El objetivo de este proyecto es construir unos programas de ayuda a la organización de la ruta de viaje más adecuada para un viajero aéreo.

Los datos

Para modelar un aeropuerto y un vuelo se utilizan las clases case siguientes:

```
case class Aeropuerto(Cod: String, X: Int, Y: Int, GMT: Int)
case class Vuelo(Aln: String, Num: Int, Org: String, HS: Int, MS: Int, Dst: String, HL: Int, ML: Int, Esc: Int)
```

Los datos vienen dados por medio de dos listas: $aeropuertos : List[Aeropuerto]$ y $vuelos : List[Vuelo]$.

La lista *aeropuertos* está compuesta por objetos de la forma $Aeropuerto(Cod\ X\ Y\ GMT)$ donde Cod es una cadena de 3 caracteres que representa el código del aeropuerto, X y Y son las coordenadas en el plano donde está ubicado el aeropuerto y GMT corresponde a un entero positivo o negativo representando la franja horaria con respecto al meridiano de greenwich (GMT). Por ejemplo $Aeropuerto(clo\ 100\ 300\ \ 8)$ es una tupla de este tipo.

La lista *vuelos* está compuesta por objetos de la forma $Vuelo(Aln\ Num\ Org\ HS\ MS\ Dst\ HL\ ML\ Esc)$ donde Aln corresponde a una cadena de caracteres que identifica la aerolínea, Num corresponde al número del vuelo, Org corresponde al código del aeropuerto de origen, HS:MS es la hora local de salida del vuelo, Dst corresponde al código del aeropuerto de llegada, HL:ML es la hora local de llegada del vuelo, y Esc es el número de escalas que tendrá el vuelo antes de llegar al destino final (sin cambiar de avión).

Por ejemplo $Vuelo(ava\ 92\ clo\ 11\ 50\ bog\ 12\ 45\ 0)$ representa el vuelo 92 de Cali a Bogotá que sale a las 11:50 de Cali y llega a las 12:45 a Bogotá (las horas se escriben en formato de 24 horas).

Para efectos de pruebas, se pondrá a disposición de los estudiantes un paquete con un repositorio de aeropuertos y vuelos suficientemente grandes.

Para los ejemplos que mencionaremos en este enunciado usaremos los siguientes datos:

```
  val aeropuertosCurso = List(

  Aeropuerto("CLO", 100, 200, -500), // Cali
  Aeropuerto("BOG", 300, 500, -500), // Bogota
  Aeropuerto("MDE", 200, 600, -500), // Medellin
  Aeropuerto("BAQ", 350, 850, -500), // Barranquilla
  Aeropuerto("SMR", 400, 950, -500), // Santa Marta
  Aeropuerto("CTG", 300, 800, -500), // Cartagena
  Aeropuerto("PTY", 400, 1000, -500), // Ciudad de Panama
  Aeropuerto("JFK", 2000, 2000, -400), // Nueva York
  Aeropuerto("MIA", 1000, 2000, -500), // Miami
  Aeropuerto("MEX", 1000, 1000, -600), // Ciudad de Mexico
  Aeropuerto("MAD", 5000, 5000, 100), // Madrid
  Aeropuerto("SVCS", 400, 1000, -600), // Caracas
  Aeropuerto("MID", 500, 100, -600), // Merida
  Aeropuerto("AUA", 500, 2000, -400), // Aruba
  Aeropuerto("IST", 9000, 9000, 300), // Estambul
  Aeropuerto("HND", 10000, 12000, 900), // Tokio
  Aeropuerto("DXB", 9500, 11500, 400), // Dubai
  Aeropuerto("SVO", 12500, 12500, 300) // Moscu
)
val vuelosCurso = List(
  Vuelo("AIRVZLA", 601, "MID", 5, 0, "SVCS", 6, 0, 0),
  Vuelo("AIRVZLA", 602, "SVCS", 6, 30, "MID", 7, 30, 0),
  Vuelo("AVA", 9432, "CLO", 7, 0, "SVO", 2, 20, 4),
  Vuelo("AVA", 9432, "CLO", 7, 0, "BOG", 8, 0, 0),
  Vuelo("IBERIA", 505, "BOG", 18, 0, "MAD", 12, 0, 0),
  Vuelo("IBERIA", 506, "MAD", 14, 0, "SVO", 23, 20, 0),
  Vuelo("IBERIA", 507, "MAD", 16, 0, "SVO", 1, 20, 0),
  Vuelo("LATAM", 787, "BOG", 17, 0, "MEX", 19, 0, 0),
  Vuelo("VIVA", 756, "BOG", 9, 0, "MDE", 10, 0, 0),
  Vuelo("VIVA", 769, "MDE", 11, 0, "BAQ", 12, 0, 0),
  Vuelo("AVA", 5643, "BAQ", 14, 0, "MEX", 16, 0, 0),
  Vuelo("COPA", 1234, "CTG", 10, 0, "PTY", 11, 30, 0),
  Vuelo("AVA", 4321, "CTG", 9, 30, "SMR", 10, 0, 0),
  Vuelo("COPA", 7631, "SMR", 10, 50, "PTY", 11, 50, 0),
  Vuelo("TURKISH", 7799, "CLO", 7, 0, "IST", 14, 0, 3),
  Vuelo("QATAR", 5566, "IST", 23, 0, "SVO", 2, 0, 0),
)
```

Dados los códigos de dos aeropuertos, cod_1 y cod_2, de define un *itinerario* de cod_1 a cod_2 como una lista de vuelos, donde el primero parte del aeropuerto de salida, los otros parten del aeropuerto de llegada del vuelo anterior, y el último vuelo llega al aeropuerto de llegada. Por defecto, un itinerario no visita dos veces el mismo aeropuerto.

```
type Itinerario = List[Vuelo]
```

2.1. Los requerimientos

Típicamente un usuario tiene claros los aeropuertos de salida cod_1 y de llegada cod_2, deseados, y expresa sus requerimientos de diferentes maneras, que se enuncian a continuación.

2.1.1. Encontrando itinerarios

La consulta más sencilla es simplemente la de hallar los itinerarios que hay para ir de cod_1 a cod_2.

2.1.2. Minimización de tiempo total de viaje

La primera prerrogativa de muchos viajeros aéreos es la de llegar lo más rápido posible a su destino. Su consulta consiste en hallar al menos tres itinerarios para ir de cod_1 a cod_2 (si los hay) que correspondan a los menores tiempos de viaje (contando tiempo de vuelo y tiempo de espera en tierra). Se debe tener en cuenta que la hora de salida de un vuelo es la hora local en el aeropuerto de salida, y la hora de llegada es la hora local en el aeropuerto de llegada.

2.1.3. Minimización de escalas

Hacer escalas muchas veces en un viaje es algo a lo que los viajeros aéreos le huyen. Su consulta consiste en hallar al menos tres itinerarios para ir de cod_1 a cod_2 (si los hay) que hagan el menor número de escalas, sin tener en cuenta el tiempo total de viaje (podría ser más rápido cambiar varias veces de avión, que esperar la salida de un vuelo que lo lleve al destino sin cambiar de avión). Nótese que hay una escala cuando en el itinerario es explícito que se cambia de un vuelo a otro, pero también hay vuelos que parecen directos entre un aeropuerto y otro, pero que tienen paradas o escalas técnicas (señaladas en el vuelo mismo).

2.1.4. Minimización del tiempo en aire

Muchos viajeros aéreos le temen a los viajes en avión y quisieran, siempre que viajan, minimizar el tiempo que están en el aire, el cual es proporcional a la distancia entre los aeropuertos. Su consulta consiste en hallar al menos tres itinerarios para ir de cod_1 a cod_2 (si los hay) que minimicen el tiempo de vuelo, sin tener en cuenta el tiempo total de viaje (podría ser mejor un menor tiempo de vuelo, aunque haya mayores tiempos de espera en tierra).

2.1.5. Optimización de la hora de salida

Muchos viajeros ejecutivos deben optimizar su tiempo de manera que lleguen a tiempo a sus citas, pero no quieren perder tiempo saliendo más temprano de lo necesario para llegar a las mismas. Estos viajeros, además de cod_1 y cod_2 especifican la hora a la que tienen la cita en la ciudad de destino: h:m. Su consulta consiste en determinar el itinerario tal que la hora de salida de cod_1 sea la hora más tarde posible para salir del aeropuerto cod_1 y llegar a tiempo a la cita (suponga que la cita es en el mismo aeropuerto de llegada).

3. Implementando soluciones funcionales al problema

3.1. Implementando el cálculo de itinerarios

Implemente la función `itinerarios` que dadas las listas de vuelos y de aeropuertos, devuelva una función que dados los códigos correspondientes a dos aeropuertos, $cod_1, cod_2 : String$, calcule los itinerarios enunciados en 2.1.1.

```
  def itinerarios((vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos  una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve todos los itinerarios entre esos dos aeropuertos

...
  }
```

Por ejemplo, usando los datos mencionados al principio del enunciado, una invocación a:

```
// Ejemplo
val itsCurso = itinerarios(vuelosCurso,aeropuertosCurso)
//2.1 Aeropuertos incomunicados
val its1 = itsCurso("MID", "SVCS")
val its2 = itsCurso("CLO", "SVCS")

// 4 itinerarios CLO-SVO

val its3 = itsCurso("CLO","SVO")

//2 itinerarios CLO-MEX

val its4 = itsCurso("CLO", "MEX")

//2 itinerarios CTG-PTY
val its5 = itsCurso("CTG","PTY")
```

dará como resultado:

```
val itsCurso: (String, String) => List[Datos.Itinerario] = <function>

val its1: List[Datos.Itinerario] = List(List(Vuelo(AIRVZLA,601,MID,5,0,SVCS,6,0,0)))
val its2: List[Datos.Itinerario] = List()



val its3: List[Datos.Itinerario] = List(List(Vuelo(AVA,9432,CLO,7,0,SVO,2,20,4)),
 List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(IBERIA,505,BOG,18,0,MAD,12,0,0), Vuelo(IBERIA,506,MAD,14,0,SVO,23,20,0)),
 List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(IBERIA,505,BOG,18,0,MAD,12,0,0), Vuelo(IBERIA,507,MAD,16,0,SVO,1,20,0)),
 List(Vuelo(TURKISH,7799,CLO,7,0,IST,14,0,3), Vuelo(QATAR,5566,IST,23,0,SVO,2,0,0)))



val its4: List[Datos.Itinerario] = List(List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(LATAM,787,BOG,17,0,MEX,19,0,0)),
List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(VIVA,756,BOG,9,0,MDE,10,0,0),
Vuelo(VIVA,769,MDE,11,0,BAQ,12,0,0), Vuelo(AVA,5643,BAQ,14,0,MEX,16,0,0)))


val its5: List[Datos.Itinerario] = List(List(Vuelo(COPA,1234,CTG,10,0,PTY,11,30,0)),
List(Vuelo(AVA,4321,CTG,9,30,SMR,10,0,0), Vuelo(COPA,7631,SMR,10,50,PTY,11,50,0)))
```

3.2. Implementando el cálculo de itinerarios que minimizan el tiempo total de viaje

Implemente la función `itinerariosTiempo` que dadas las listas de vuelos y de aeropuertos, devuelva una función que dados los códigos correspondientes a dos aeropuertos, $cod_1, cod_2 : String$, calcule los itinerarios enunciados en 2.1.2.

```
  def itinerariosTiempo(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve una funcion que devuelve los tres (si los hay) itinerarios que minimizan el tiempo de viaje entre esos dos aeropuertos
...
  }
```

Por ejemplo, usando los datos mencionados al principio del enunciado, una invocación a:

```
val itsTiempoCurso = itinerariosTiempo(vuelosCurso,aeropuertosCurso)

// prueba itinerariosTiempo
val itst1 = itsTiempoCurso("MID", "SVCS")
val itst2 = itsTiempoCurso("CLO", "SVCS")

// 4 itinerarios CLO-SVO

val itst3 = itsTiempoCurso("CLO","SVO")

//2 itinerarios CLO-MEX

val itst4 = itsTiempoCurso("CLO", "MEX")

//2 itinerarios CTG-PTY
val itst5 = itsTiempoCurso("CTG","PTY")
```

dará como resultado:

```
val itst1: List[Datos.Itinerario] = List(List(Vuelo(AIRVZLA,601,MID,5,0,SVCS,6,0,0)))
val itst2: List[Datos.Itinerario] = List()



val itst3: List[Datos.Itinerario] = List(List(Vuelo(AVA,9432,CLO,7,0,SVO,2,20,4)),
List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(IBERIA,505,BOG,18,0,MAD,12,0,0), Vuelo(IBERIA,506,MAD,14,0,SVO,23,20,0)),
List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(IBERIA,505,BOG,18,0,MAD,12,0,0), Vuelo(IBERIA,507,MAD,16,0,SVO,1,20,0)))



val itst4: List[Datos.Itinerario] = List(List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(VIVA,756,BOG,9,0,MDE,10,0,0),
Vuelo(VIVA,769,MDE,11,0,BAQ,12,0,0), Vuelo(AVA,5643,BAQ,14,0,MEX,16,0,0)),
List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(LATAM,787,BOG,17,0,MEX,19,0,0)))


val itst5: List[Datos.Itinerario] = List(List(Vuelo(COPA,1234,CTG,10,0,PTY,11,30,0)),
List(Vuelo(AVA,4321,CTG,9,30,SMR,10,0,0), Vuelo(COPA,7631,SMR,10,50,PTY,11,50,0)))
```

3.3. Implementando el cálculo de itinerarios que minimizan las escalas

Implemente la función `itinerariosEscalas` que dadas las listas de vuelos y de aeropuertos, devuelva una función que dados los códigos correspondientes a dos aeropuertos, $cod_1, cod_2 : String$, calcule los itinerarios enunciados en 2.1.3.

```
  def itinerariosEscalas(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve los tres (si los hay) itinerarios que minimizan el numero de cambios de avion entre esos dos aeropuertos
...
  }
```

Por ejemplo, usando los datos mencionados al principio del enunciado, una invocación a:

```
val itsEscalasCurso = itinerariosEscalas(vuelosCurso,aeropuertosCurso)

val itsc1 = itsEscalasCurso("MID", "SVCS")
val itsc2 = itsEscalasCurso("CLO", "SVCS")

// 4 itinerarios CLO-SVO

val itsc3 = itsEscalasCurso("CLO","SVO")

//2 itinerarios CLO-MEX

val itsc4 = itsEscalasCurso("CLO", "MEX")

//2 itinerarios CTG-PTY
val itsc5 = itsEscalasCurso("CTG","PTY")
```

dará como resultado:

```
val itsc1: List[Datos.Itinerario] = List(List(Vuelo(AIRVZLA,601,MID,5,0,SVCS,6,0,0)))
val itsc2: List[Datos.Itinerario] = List()



val itsc3: List[Datos.Itinerario] = List(List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(IBERIA,505,BOG,18,0,MAD,12,0,0),
Vuelo(IBERIA,506,MAD,14,0,SVO,23,20,0)),
List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(IBERIA,505,BOG,18,0,MAD,12,0,0), Vuelo(IBERIA,507,MAD,16,0,SVO,1,20,0)),
List(Vuelo(AVA,9432,CLO,7,0,SVO,2,20,4)))



val itsc4: List[Datos.Itinerario] = List(List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(LATAM,787,BOG,17,0,MEX,19,0,0)),
List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(VIVA,756,BOG,9,0,MDE,10,0,0),
Vuelo(VIVA,769,MDE,11,0,BAQ,12,0,0), Vuelo(AVA,5643,BAQ,14,0,MEX,16,0,0)))


val itsc5: List[Datos.Itinerario] = List(List(Vuelo(COPA,1234,CTG,10,0,PTY,11,30,0)),
List(Vuelo(AVA,4321,CTG,9,30,SMR,10,0,0), Vuelo(COPA,7631,SMR,10,50,PTY,11,50,0)))
```

3.4. Implementando el cálculo de itinerarios que minimizan el tiempo en aire entre esos dos aeropuertos

Implemente la función `itinerariosAire` que dadas las listas de vuelos y de aeropuertos, devuelva una función que dados los códigos correspondientes a dos aeropuertos, $cod_1, cod_2 : String$, calcule los itinerarios enunciados en 2.1.4.

```
  def itinerariosAire(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve los tres (si los hay) itinerarios que minimizan el tiempo en el aire entre esos dos aeropuertos
...
  }
```

Por ejemplo, usando los datos mencionados al principio del enunciado, una invocación a:

```
val itsAireCurso = itinerariosAire(vuelosCurso,aeropuertosCurso)

val itsa1 = itsAireCurso("MID", "SVCS")
val itsa2 = itsAireCurso("CLO", "SVCS")
```

```
// 4 itinerarios CLO-SVO

val itsa3 = itsAireCurso("CLO","SVO")

//2 itinerarios CLO-MEX

val itsa4 = itsAireCurso("CLO", "MEX")

//2 itinerarios CTG-PTY
val itsa5 = itsAireCurso("CTG","PTY")
```

dará como resultado:

```
val itsa1: List[Datos.Itinerario] = List(List(Vuelo(AIRVZLA,601,MID,5,0,SVCS,6,0,0)))
val itsa2: List[Datos.Itinerario] = List()



val itsa3: List[Datos.Itinerario] = List(List(Vuelo(AVA,9432,CLO,7,0,SVO,2,20,4)),
List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(IBERIA,505,BOG,18,0,MAD,12,0,0), Vuelo(IBERIA,506,MAD,14,0,SVO,23,20,0)),
List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(IBERIA,505,BOG,18,0,MAD,12,0,0), Vuelo(IBERIA,507,MAD,16,0,SVO,1,20,0)))



val itsa4: List[Datos.Itinerario] = List(List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(LATAM,787,BOG,17,0,MEX,19,0,0)),
List(Vuelo(AVA,9432,CLO,7,0,BOG,8,0,0), Vuelo(VIVA,756,BOG,9,0,MDE,10,0,0),
Vuelo(VIVA,769,MDE,11,0,BAQ,12,0,0), Vuelo(AVA,5643,BAQ,14,0,MEX,16,0,0)))


val itsa5: List[Datos.Itinerario] = List(List(Vuelo(COPA,1234,CTG,10,0,PTY,11,30,0)),
List(Vuelo(AVA,4321,CTG,9,30,SMR,10,0,0), Vuelo(COPA,7631,SMR,10,50,PTY,11,50,0)))
```

3.5. Implementando el cálculo del itinerario que optimiza la hora de salida

Implemente la función `itinerarioSalida` que dadas las listas de vuelos y de aeropuertos, devuelva una función que dados los códigos correspondientes a dos aeropuertos, $cod_1, cod_2 : String$, y la hora de la cita, $h{:}m$ en $c2$,calcule el itinerario enunciado en 2.1.5.

```
  def itinerarioSalida(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String, String, Int, Int)=>Itinerario={
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos, y h:m una hora de la cita en c2
    // y devuelve el itinerario que optimiza la hora de salida para llegar a tiempo a la cita
...
  }
```

Por ejemplo, usando los datos mencionados al principio del enunciado, una invocación a:

```
val itSalidaCurso = itinerarioSalida(vuelosCurso,aeropuertosCurso)

val itsal1 = itSalidaCurso("CTG","PTY",11, 40)
val itsal2 = itSalidaCurso("CTG","PTY",11, 55)
val itsal3 = itSalidaCurso("CTG","PTY",10,30)
```

dará como resultado:

```
val itsal1: Datos.Itinerario = List(Vuelo(COPA,1234,CTG,10,0,PTY,11,30,0))
val itsal2: Datos.Itinerario = List(Vuelo(COPA,1234,CTG,10,0,PTY,11,30,0))
val itsal3: Datos.Itinerario = List(Vuelo(COPA,1234,CTG,10,0,PTY,11,30,0))
```

4. Acelerando los cálculos con paralelismo de tareas y de datos

Una vez terminada esta etapa del proyecto, donde usted ya ha programado las versiones secuenciales `itinerarios`, `itinerariosTiempo`, `itinerariosEscalas`, `itinerariosAire`, y `itinerarioSalida`, queremos implementar las versiones paralelas de ellas para ver si se logran tiempos mucho mejores para el cálculo.

Para el ejercicio de paralelización use paralelismo de tareas (principalmente) y de datos (si fuese útil) donde lo vea pertinente. Implemente las versiones paralelas `itinerariosPar`, `itinerariosTiempoPar`,

`itinerariosEscalasPar`, `itinerariosAirePar`, y `itinerarioSalidaPar` donde estas funciones hacen esencialmente lo mismo que las versiones secuenciales.

```
  def itinerariosPar(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos  una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve todos los itinerarios entre esos dos aeropuertos
...
  }

  def itinerariosTiempoPar(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos  una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve  los tres (si los hay) itinerarios que minimizan el tiempo de viaje entre esos dos aeropuertos
...
  }

  def itinerariosEscalasPar(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos  una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve  los tres (si los hay) itinerarios que minimizan el numero de cambios de avion  entre esos dos aeropuertos
...
  }

  def itinerariosAirePar(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos  una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve  los tres (si los hay) itinerarios que minimizan el tiempo en el aire entre esos dos aeropuertos
...
  }

  def itinerarioSalidaPar(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String, String, Int, Int)=>Itinerario= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos  una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos y h:m la hora de la cita,
    // y devuelve  el itinerario que optimiza la hora de salida para llegar a tiempo a la cita
...
  }
```

5. Produciendo datos para hacer la evaluación comparativa de las versiones secuencial y concurrente

Un punto esencial en estos proyectos de paralelización, es realizar el análisis comparativo y concluir en qué casos es beneficioso usar la paralelización y en qué casos no. Para ellos es importante hacer muchas tablas con los tiempos que toman los dos tipos de versiones (secuencial y paralela).

Para efectos de contar con datos para pruebas, nosotros les proveeremos unos datos de prueba de "juguete" para hacer pruebas de funcionalidad y corrección sencillas, y unos datos de prueba reales de aeropuertos y vuelos en los Estados Unidos, para que hagan pruebas con datos de tamaño muy grande, y miren qué es lo más lejos que logran llegar.

Utilice *org.scalameter* para hacer los análisis mencionados. Comente claramente cómo genera los datos de análisis, y preséntelos en el informe tabulados en tablas, de manera que después pueda analizarlas y sacar conclusiones relevantes.

6. Técnicas utilizadas

Todo grupo de trabajo es libre de diseñar las funciones de la manera que considere más adecuada e implementar dicho modelo con el algoritmo que desee, siempre y cuando se ajuste al estilo de programación del paradigma de programación funcional y a las técnicas de programación vistas en clase.

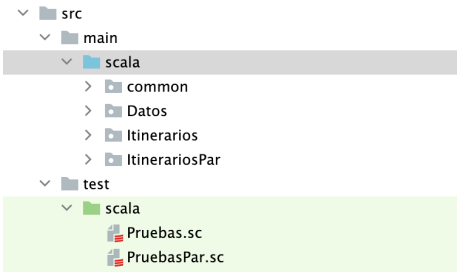
Para lograr soluciones más eficientes en tiempo, deberán usar las técnicas de paralelización vistas en clase y hacer las evaluaciones comparativas correspondientes.

7. Entrega

Para el desarrollo del proyecto, se sugiere trabajar en el siguiente orden:

1. Construir una solución secuencial funcional de cada una de las funciones especificadas, y argumentar sobre su corrección.
2. A partir de esas soluciones, construir las soluciones concurrentes, y argumentar sobre su corrección.
3. Hacer un análisis comparativo de las dos soluciones (las secuenciales y las paralelas) para concluir sobre la pertinencia o no de paralelizar la solución.
4. Escribir un informe que dé cuenta de los aspectos que se quieren evidenciar como resultados de aprendizaje.

Usted deberá entregar dos paquetes `Itinerarios` e `ItinerariosPar`, los cuales harán parte de su estructura de proyecto `IntellijIdea`, como se muestra en la figura a continuación, junto con los paquetes `common` y `Datos` provistos para el proyecto.



7.1. Paquete *Itinerarios*

Las funciones correspondientes a cada ejercicio, `itinerarios`, `itinerariosTiempo`, `itinerariosEscalas`, `itinerariosAire`, y `itinerarioSalida`, deben ser implementadas en un paquete de Scala denominado `Itinerarios`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
import Datos._
package object Itinerarios {
  def itinerarios(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve todos los itinerarios entre esos dos aeropuertos
...
  }

  def itinerariosTiempo(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve una funcion que devuelve los tres (si los hay) itinerarios que minimizan el tiempo de viaje entre esos dos aeropuertos
...
  }

  def itinerariosEscalas(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve los tres (si los hay) itinerarios que minimizan el numero de cambios de avion entre esos dos aeropuertos
...
  }

  def itinerariosAire(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve los tres (si los hay) itinerarios que minimizan el tiempo en el aire entre esos dos aeropuertos
...
  }
```

```
  def itinerarioSalida(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String, String, Int, Int)=>Itinerario= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos  una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos, y h:m una hora de la cita en c2
    // y  devuelve  el itinerario que optimiza la hora de salida para llegar a tiempo a la cita
...
}
```

7.2. Paquete *ItinerariosPar*

Las funciones correspondientes a cada ejercicio, `itinerariosPar`, `itinerariosTiempoPar`, `itinerariosEscalasPar`, `itinerariosAirePar`, y `itinerarioSalidaPar` deben ser implementadas en un paquete de Scala denominado `ItinerariosPar`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
import Datos._
import common._
import Itinerarios._


import scala.collection.parallel.CollectionConverters._
import scala.collection.parallel.ParSeq

package object ItinerariosPar {
  def itinerariosPar(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // Recibe vuelos, una lista de todos los vuelos disponibles y
    // aeropuertos  una lista de todos los aeropuertos
    // y devuelve una funcion que recibe c1 y c2, codigos de aeropuertos
    // y devuelve todos los itinerarios entre esos dos aeropuertos
...
  }


  def itinerariosTiempoPar(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // recibe c1 y c2, codigos de aeropuertos
    // y devuelve  los tres (si los hay) itinerarios que minimizan el tiempo de viaje entre esos dos aeropuertos
...
  }

  def itinerariosEscalasPar(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // recibe c1 y c2, codigos de aeropuertos
    // y devuelve  los tres (si los hay) itinerarios que minimizan el numero de cambios de avion  entre esos dos aeropuertos
...
  }


  def itinerariosAirePar(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String,String)=>List[Itinerario]= {
    // recibe c1 y c2, codigos de aeropuertos
    // y devuelve  los tres (si los hay) itinerarios que minimizan el tiempo en el aire entre esos dos aeropuertos
...
  }

  def itinerarioSalidaPar(vuelos:List[Vuelo], aeropuertos:List[Aeropuerto]):(String, String, Int, Int)=>Itinerario= {
    // recibe c1 y c2, codigos de aeropuertos
    // y devuelve  el itinerario que optimiza la hora de salida para llegar a tiempo a la cita
...
}
```

Estos paquetes deberán correr en conjunto. Las pruebas las haremos importando esos paquetes.

La fecha de entrega límite es el 4 de diciembre de 2025 a las 23:59. La sustentación será el 11 de diciembre de 2025.

Debe entregar un informe en formato pdf, los paquetes mencionados, un archivo Readme.txt que describa todos los archivos entregados y las instrucciones para ejecutar los programas. Todo lo anterior en un solo archivo empaquetado cuyo nombre contiene los apellidos de los autores y cuya extensión corresponde al modo de compresión. Por ejemplo Diaz.zip o Diaz.rar, o Diaz.tgz o ...

8. Sustentación y calificación

El trabajo debe ser sustentado por los autores en día y hora especificados. La calificación del proyecto se hará teniendo en cuenta los siguientes criterios:

1. Informe (1/3)

- Debe describir claramente las estructuras de datos utilizadas, tanto para las soluciones secuenciales como para las soluciones paralelas. No olvide señalar también qué colecciones paralelas fueron utilizadas.
- Las funciones desarrolladas deben responder a programas funcionales puros. Las que no deben ser justificadas, por qué no.
- En general, el código debe evidenciar el manejo de la recursión, del reconocimiento de patrones, de mecanismos de encapsulación, de funciones de alto orden, de iteradores, de colecciones y de expresiones for. El informe debe señalar dónde se usaron estos elementos.
- El informe debe traer las argumentaciones sobre la corrección de las funciones desarrolladas. Por lo menos de las más relevantes (la función `itinerarios`, una de las funciones `itinerariosTiempo`, `itinerariosEscalas` o `itinerariosAire`, y la función `itinerarioSalida`.
- El informe debe señalar dónde se usaron técnicas de paralelización de tareas y de datos, cuáles se usaron y qué impacto tuvieron en el desempeño del programa.
- El informe debe traer las argumentaciones sobre las razones que permiten asegurar que las técnicas de paralelización usadas deberían tener un impacto positivo en el desempeño del programa.
- El informe también debe evidenciar, con tablas, las evaluaciones comparativas realizadas entre las pruebas realizadas a las soluciones secuenciales y las paralelas, y concluir sobre el grado de aceleración logrado.

2. Implementación (1/2). Esto quiere decir que el código entregado funciona por lo menos para las diferentes pruebas que tendremos para evaluarlo.

3. Desempeño grupal en la sustentación (1/6), lo cual incluye la capacidad del grupo de navegar en el código y realizar cambios rápidamente en él, así como la capacidad de responder con solvencia a las preguntas que se le realicen.

Todo lo anterior está condensado en la rúbrica que se les comparte con el enunciado del proyecto.

En todos los casos la sustentación será pilar fundamental de la nota asignada. Cada persona de cada grupo, después de la sustentación, tendrá asignado un número real (el factor de multiplicación) entre 0 y 1, correspondiente al grado de calidad de su sustentación. Su nota definitiva será la nota del proyecto, multiplicada por ese valor. Si su asignación es 1, su nota será la del proyecto. Pero si su asignación es 0.9, su nota será 0.9 por la nota del proyecto. La no asistencia a la sustentación tendrá como resultado una asignación de un factor de 0.

La idea es que lo que no sea debidamente sustentado no vale así funcione muy bien!!!

Éxitos!!!