Proyecto de curso

El problema de la reconstrucción de cadenas

Fundamentos de programación funcional y concurrente

Escuela de Ingeniería de Sistemas y Computación

Profesor Juan Francisco Díaz Frias Monitor Emily Núñez

Mayo 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 meca-

nismo 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 reconstrucción de cadenas: PRC

Investigadores en Biología han estado interesados, desde hace muchos años, en encontrar un patrón de nucleótidos que codifique toda la información necesaria para construir las proteínas de las cuales estamos hechos los humanos. A este patrón se le denomina *genoma humano*. El objetivo es lograr describir una pieza de DNA por medio de un patrón o a cadena de caracteres (a este proceso se le conoce como secuenciación de la pieza de DNA). Este patrón debe ser construido a partir de subpatrones que se sabe que aparecen en la pieza de DNA. Es decir, a partir de unos subpatrones, se necesita reconstruir el patrón escondido. Los "algoritmistas" han estado interesados en este proyecto por diferentes razones:

- Las secuencias de DNA pueden ser representadas acertadamente por cadenas de caracteres sobre el alfabeto $\{A, C, T, G\}$. Algunos problemas son, entonces, enunciados en términos de problemas con cadenas de caracteres.

- Las secuencias de DNA son cadenas MUY LARGAS. El genoma humano contiene aproximadamente 3 billones de caracteres. Por tanto se necesitan técnicas sofisticadas de computación para lograr manejarlo.

- Hay mucho dinero para estas investigaciones.

Como operación básica para la secuenciación se cuenta con la pregunta ¿s es subcadena de $\mathcal{S}$? (denotada ¿$s \preceq \mathcal{S}$?) que en la realidad es un experimento de laboratorio, el cual supondremos en este proyecto que es perfecto (siempre da la respuesta correcta). Para efectos de este proyecto supondremos que existe un oráculo $\Psi_{\mathcal{S}}$ tal que

$$\Psi_{\mathcal{S}}(s) \text{ sii } s \preceq \mathcal{S}.$$

Con el presente proyecto se pretende abordar a manera de ejercicio el problema de reconstruir una secuencia de DNA a partir de experimentos con un oráculo que nos permiten saber si una subsecuencia dada es o no subsecuencia de la cadena buscada.

2.1. Formalización

El problema se puede enunciar formalmente así:
PRC
Entrada: $\Sigma = \{\sigma_1, \ldots, \sigma_m\}, \quad N = |\mathcal{S}|, \Psi_{\mathcal{S}}$
Salida: $s \in \Sigma^* : |s| = N, \Psi_{\mathcal{S}}(s)$

2.2. ¿Entendimos el problema?

2.2.1. Ejemplo 1

- **Entrada:** : $\Sigma = \{a, c, g, t\}, \quad N = 4 = |\mathcal{S}|, \Psi_{\mathcal{S}}$

- **Salida:** $s \in \Sigma^* : |s| = 4, \Psi_{\mathcal{S}}(s)$

Nótese que $\mathcal{S}$ no se conoce a la entrada, pero sí se conoce su longitud (en este caso 4) y el oráculo $\Psi_{\mathcal{S}}$. Como la s de salida es una secuencia de Σ^* del mismo tamaño de $\mathcal{S}$ y tal que $\Psi_{\mathcal{S}}(s)$ es cierto, entonces $s = \mathcal{S}$, o sea, encontramos la cadena.

Pero, ¿cómo encontrar la cadena? **Haciendo preguntas al oráculo!!**

Por ejemplo, si $\mathcal{S} =< aaaa >$ y se le pregunta al oráculo $\Psi_{\mathcal{S}}(a), \Psi_{\mathcal{S}}(c), \Psi_{\mathcal{S}}(g)$, y $\Psi_{\mathcal{S}}(t)$, a partir de las respuestas podríamos producir como salida $s =< aaaa >$.

2.2.2. Ejemplo 2

Otro ejemplo, si $\mathcal{S} =< agga >$ y se le pregunta al oráculo $\Psi_{\mathcal{S}}(a), \Psi_{\mathcal{S}}(c), \Psi_{\mathcal{S}}(g), \Psi_{\mathcal{S}}(t)$, $\Psi_{\mathcal{S}}(aa), \Psi_{\mathcal{S}}(ag), \Psi_{\mathcal{S}}(ga), \Psi_{\mathcal{S}}(gg), \Psi_{\mathcal{S}}(aga), \Psi_{\mathcal{S}}(agg)$, $\Psi_{\mathcal{S}}(gga), \Psi_{\mathcal{S}}(ggg), \Psi_{\mathcal{S}}(gaa), \Psi_{\mathcal{S}}(gag), \Psi_{\mathcal{S}}(agga)$, $\Psi_{\mathcal{S}}(aggg), \Psi_{\mathcal{S}}(ggaa)$ y $\Psi_{\mathcal{S}}(ggag)$, y a partir de las respuestas podríamos producir como salida $s =< agga >$.

2.2.3. Conclusión

El quid del asunto es: ¿Qué preguntas le hacemos al oráculo? A continuación, se presentarán varias alternativas.

2.3. Alternativas de solución

A continuación se enumerarán, en forma de algoritmos iterativos, varias alternativas de solución al problema.

2.3.1. Solución Ingenua

Una primera solución para el problema es la siguiente:

```
PRC_Ingenuo(Σ, N, Ψ_S)
1 foreach w ∈ Σ^N
2         do if Ψ_S(w)
3               then return w
```

2.3.2. Solución Mejorada

Una primera mejora en el algoritmo se puede dar generando las preguntas más inteligentemente. Una idea interesante nace de la siguiente propiedad:

Propiedad: Sea $s = s_1 \cdot s_2$. Si $\Psi_{\mathcal{S}}(s)$ entonces $\Psi_{\mathcal{S}}(s_1)$ y $\Psi_{\mathcal{S}}(s_2)$.

En otras palabras, toda subcadena de $\mathcal{S}$ está formada por concatenación de subcadenas de $\mathcal{S}$. Sin embargo si s_1, s_2 son subcadenas de $\mathcal{S}$ no se puede asegurar que s también lo sea.

La idea entonces consiste en generar para cada $k \leq n$ un conjunto SC_k de cadenas de longitud k candidatas a ser subcadenas de $\mathcal{S}$. Solo se consulta el oráculo para las cadenas candidatas. Inicialmente sólo hay una cadenas candidata de longitud 0: λ (la cadena vacía "")

```
PRC_Mejorado(Σ, N, Ψ_S)
1 SC_0 ← {λ}
2 for k = 1 to N
3     do SC_k ← SC_{k-1} • Σ
4        foreach w ∈ SC_k
5                do if ¬Ψ_S(w)
6                      then SC_k ← SC_k − {w}
7                      else if |w| = N
8                             then return w
```

2.3.3. Turbo Solución

Podemos aún mejorar más el tiempo en la reconstrucción de $\mathcal{S}$:

```
PRC_Turbo(Σ, N, Ψ_S)
1 SC_1 ← Σ, k ← 2
2 while k ≤ N
3       do SC_k ← SC_{k/2} • SC_{k/2}
4          foreach w ∈ SC_k
5                  do if ¬Ψ_S(w)
6                        then SC_k ← SC_k − {w}
7                        else if |w| = N
8                               then return w
9          k ← 2 * k
```

2.3.4. Turbo mejorada

Note que si se sabe que $SC_2 = \{AC, CA, CC\}$ la cadena $CAAC \in SC_2 \bullet SC_2$ será calculada en la línea 3 e insertada en SC_4 a pesar de que $AA \preceq CAAC$ pero $AA \notin SC_2$. Por ello se va a remplazar la línea 3 del algoritmo PRC_Turbo por la línea :

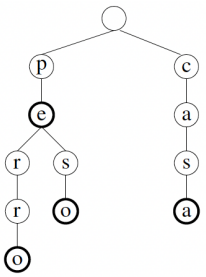
$$SC_k \leftarrow Filtrar(SC_{k/2}, k/2).$$

$Filtrar$ recibe un conjunto de cadenas SC de longitud k y devuelve el conjunto de cadenas de longitud $2k$ en $SC \bullet SC$ que no tengan el problema indicado. De esta manera solo se consultará al oráculo sobre cadenas que el algoritmo no haya podido descartar sin consultarlo. Tenga en cuenta que aunque el oráculo es una operación básica, en la realidad es un experimento y eso es costoso (o sea, constante pero caro).

Un posible algoritmo $Filtrar$ es el siguiente:

```
Filtrar(SC, k)
1 F ← ∅
2 foreach s_1, s_2 ∈ SC
3         do s ← s_1 · s_2
4            if {w : w ⪯ s, |w| = k} ⊆ SC
5               then F ← F ⋃ {s}
6 return F
```

Hasta este momento SC es visto como una secuencia de cadenas de caracteres, donde saber si una secuencia está o no en SC es lineal en el tamaño de SC.



2.3.5. Turbo acelerada

Para terminar de acelerar la solución, se pretende guardar SC en una estructura de datos mas práctica para el procesamiento de cadenas: *árbol de sufijos*.

Un *trie* es un árbol donde cada nodo representa un caracter y la raíz representa la cadena nula. Así, cada camino que parta de la raíz representa la cadena resultante de concatenar los nodos atravesados por el camino. Cualquier conjunto finito de cadenas de caracteres definen un *trie*. Por ejemplo las cadenas {pe,perro,peso,casa} definen el *trie* mostrado en la figura siguiente:

Dado un *trie* que representa un conjunto de palabras, es fácil decidir si una cadena dada s está o no en el conjunto. Simplemente se recorre el árbol según indiquen los caracteres de s. Si al final se llega a un nodo marcado, la cadena estaba en el conjunto inicial. Sino, o si no existe un nodo etiquetado con el caracter respectivo, la cadena no estaba.

Un *árbol de sufijos* es un *trie* que contiene todos los sufijos de un conjunto dado de cadenas. Por ejemplo, el árbol de sufijos de $\{ACAC, CACT\}$ será el *trie* que contiene las cadenas $\{ACAC, CAC, AC, C, CACT, ACT, CT, T\}$.

Suponga que, en el algoritmo $Filtrar$, SC se guarda como un árbol de sufijos.

3. Implementando soluciones funcionales al problema

Para implementar los algoritmos presentados de manera funcional, se definen el alfabeto y el tipo de dato *Oraculo*:

```
val alfabeto=Seq('a', 'c', 'g', 't')
type Oraculo = Seq[Char] => Boolean
```

3.1. Implementando la solución ingenua

Implemente la función `reconstruirCadenaIngenuo` que dados n, la longitud de la cadena a reconstruir y o, un oráculo asociado a esa cadena, devuelva la cadena que representa el oráculo, de acuerdo al algoritmo presentado en 2.3.1.

```
def reconstruirCadenaIngenuo(n:Int,o:Oraculo):Seq[Char]= {
  // recibe la longitud de la secuecnia que hay que reconstruir (n), y un oraculo para esa secuencia
  // y devuelve la secuencia reconstruida
...
}
```

Por ejemplo,

3.2. Implementando la solución mejorada

Implemente la función `reconstruirCadenaMejorado` que dados n, la longitud de la cadena a reconstruir y o, un oráculo asociado a esa cadena, devuelva la cadena que representa el oráculo, de acuerdo al algoritmo presentado en 2.3.2.

```
  def reconstruirCadenaMejorado(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuecnia que hay que reconstruir (n), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
...
  }
```

3.3. Implementando la solución turbo

Implemente la función `reconstruirCadenaTurbo` que dados n, la longitud de la cadena a reconstruir y o, un oráculo asociado a esa cadena, devuelva la cadena que representa el oráculo, de acuerdo al algoritmo presentado en 2.3.3.

```
  def reconstruirCadenaTurbo(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuencia que hay que reconstruir (n, potencia de 2), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
   // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
...
  }
```

3.4. Implementando la solución turbo mejorada

Implemente la función `reconstruirCadenaTurboMejorada` que dados n, la longitud de la cadena a reconstruir y o, un oráculo asociado a esa cadena, devuelva la cadena que representa el oráculo, de acuerdo al algoritmo presentado en 2.3.4.

```
  def reconstruirCadenaTurboMejorada(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuencia que hay que reconstruir (n, potencia de 2), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
...
  }
```

3.5. Implementando la solución turbo acelerada

Para implementar esta solución es necesario implementar primero los árboles de sufijos.

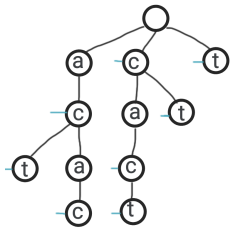
3.5.1. Árboles de sufijos

Considere un *trie* definido con la siguiente clase case:

```
  abstract class Trie
  case class Nodo (car:Char, marcada:Boolean,
                   hijos: List[Trie]) extends Trie
  case class Hoja(car: Char, marcada:Boolean) extends Trie

  def raiz(t:Trie):Char = {
    t match {
      case Nodo(c,_,_) => c
      case Hoja(c,_) => c
    }
  }
  def cabezas(t:Trie):Seq[Char] = {
    t match {
      case Nodo(_,_,lt) => lt.map(t=>raiz(t))
      case Hoja(c,_) => Seq[Char](c)
    }
  }
```

El trie de la figura,



correspondiente al árbol de sufijos de $\{ACAC, CACT\}$ será el *trie*:

```
val t = Nodo('_', false, List(
  Nodo('a', false, List(
    Nodo('c', true, List(
      Nodo('a', false, List(
        Hoja('c', true)
      )),
      Hoja('t', true)
    ))
  )),
  Nodo('c', true, List(
    Nodo('a', false, List(
      Nodo('c', true, List(
        Hoja('t', true)
      ))
    )),
    Hoja('t', true)
  )),
  Hoja('t',true)
))
```

Implemente las funciones `pertenece, adicionar,` y `arbolDeSufijos` tales que:

- `pertenece` recibe una secuencia s y un trie t y devuelve un booleano que responde si $s \in t$.
- `adicionar` recibe una secuencia s y un trie t y devuelve el trie correspondiente a adicionar s a t.
- `arbolDeSufijos` que recibe una secuencia de secuencias ss y devuelve el trie correspondiente al árbol de sufijos de ss.

3.5.2. Solución turbo acelerada

Implemente la función `reconstruirCadenaTurboAcelerada` que dados n, la longitud de la cadena a reconstruir y o, un oráculo asociado a esa cadena, devuelva la cadena que representa el oráculo, de acuerdo al algoritmo presentado en 2.3.5.

```
def reconstruirCadenaTurboAcelerada(n:Int,o:Oraculo):Seq[Char]= {
  // recibe la longitud de la secuencia que hay que reconstruir (n, potencia de 2), y un oraculo para esa secuencia
  // y devuelve la secuencia reconstruida
  // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
  // Usa arboles de sufijos para guardar Seq[Seq[Char]]...
}
```

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 `reconstruirCadenaIngenuo`, `reconstruirCadenaMejorado`,

`reconstruirCadenaTurbo`, `reconstruirCadenaTurboMejorada`, y `reconstruirCadenaTurboAcelerada`, queremos implementar sus versiones paralelas de algunas de ellas para ver si se logran tiempos mucho mejores para la reconstrucción.

Para el ejercicio de paralelización use paralelismo de datos y de tareas donde lo vea pertinente. Implemente las versiones paralelas `reconstruirCadenaIngenuoPar`, `reconstruirCadenaMejoradoPar`,

`reconstruirCadenaTurboPar`, `reconstruirCadenaTurboMejoradaPar`, y `reconstruirCadenaTurboAceleradaPar`, donde estas funciones hacen esencialmente lo mismo que las versiones secuenciales, sólo que reciben un parámetro adicional denominado *umbral* y que lo pueden usar para decidir si usar paralelismo de tareas o no donde lo vean pertinente. Este *umbral* está asociado al tamaño del conjunto SC que es el conjunto que va cambiando en cada iteración de las soluciones mencionadas.

```
  def reconstruirCadenaIngenuoPar(umbral:Int)(n:Int,o:Oraculo):Seq[Char]= {
...
  }

  def reconstruirCadenaMejoradoPar(umbral:Int)(n:Int,o:Oraculo):Seq[Char]= {
...
  }

  def reconstruirCadenaTurboPar(umbral:Int)(n:Int,o:Oraculo):Seq[Char]= {
...
  }

  def reconstruirCadenaTurboMejoradaPar(umbral:Int)(n:Int,o:Oraculo):Seq[Char]= {
...
  }

  def reconstruirCadenaTurboAceleradaPar(umbral:Int)(n:Int,o:Oraculo):Seq[Char]= {
...
  }
```

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).

Utilice *org.scalameter* y *expresiones for* para generar datos 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.

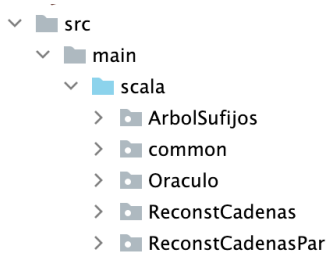
7. Entrega

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

1. Definir e implementar las estructuras de datos para manipularlos de la manera más adecuada posible, teniendo en cuenta el objetivo de encontrar soluciones lo más eficientemente posible.
2. Construir una solución secuencial funcional, y argumentar sobre su corrección.
3. A partir de esa solución, construir una solución concurrente, y argumentar sobre su corrección.
4. Hacer un análisis comparativo de las dos soluciones (la secuencial y la paralela) para concluir sobre la pertinencia o no de paralelizar la solución.

5. Escribir un informe que dé cuenta de los aspectos que se quieren evidenciar como resultados de aprendizaje.

Usted deberá entregar tres paquetes `ReconstCadenas`, `ReconstCadenasPar` y `ArbolSufijos` los cuales harán parte de su estructura de proyecto `IntellijIdea`, como se muestra en la figura a continuación, junto con los paquetes `common` y `Oraculo` provistos para el proyecto.



Paquete *Oraculo*

Es un paquete muy sencillo provisto por el proyecto:

```
package object Oraculo {
  val alfabeto=Seq('a', 'c', 'g', 't')
  type Oraculo = Seq[Char] => Boolean

  def crearOraculo(delay:Int)(c:Seq[Char]):Oraculo ={
    def esSubcadena(s:Seq[Char]):Boolean = {
      Thread.sleep(delay)
      c.containsSlice(s)
    }
    esSubcadena
  }
}
```

Se dfine el *alfabeto* a utilizar, el tipo *Oraculo* y la función `crearOraculo`, currificada, que recibe el costo en milisegundos del experimento (sugiero utilizar 1, porque con valores más grandes los experimentos se pueden demorar mucho), y luego recibe la secuencia que hay que adivinar, y devuelve el oráculo asociado a esa secuencia. Este paquete sirve para crear los oráculos para las pruebas. Por ejemplo, si se quiere crear un oráculo para la cadena "gaatccagat" y llamar uno de los algoritmos de reconstrucción de cadenas `reconstruirCadenaXXX`, se hace lo siguiente:

```
scala> val sec= List('g', 'a', 'a', 't', 'c', 'c', 'a', 'g', 'a', 't')
val sec: Seq[Char] = List(g, a, a, t, c, c, a, g, a, t)
scala> val or=crearOraculo(1)(sec)
val or: Oraculo.Oraculo = Oraculo.packageLambda$6211/0x0000000801b74040@55ad47ff
scala> reconstruirCadenaXXX(sec.length, or)
val res0: Seq[Char] = List(g, a, a, t, c, c, a, g, a, t)
```

7.1. Paquete *ArbolSufijos*

Las funciones correspondientes a los árboles de sufijos, `pertenece`, `adicionar`, y `arbolDeSufijos` deben ser implementadas en un paquete de Scala denominado `ArbolSufijos`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
package object ArbolSufijos {
  // Definiendo otra estructura para manipular Seq[Seq[Char]]
  abstract class Trie
  case class Nodo (car:Char, marcada:Boolean,
                   hijos: List[Trie]) extends Trie
  case class Hoja(car: Char, marcada:Boolean) extends Trie

  def raiz(t:Trie):Char = {
    t match {
      case Nodo(c,_,_) => c
      case Hoja(c,_) => c
    }
```

```
  }
  def cabezas(t:Trie):Seq[Char] = {
    t match {
      case Nodo(_,_,lt) => lt.map(t=>raiz(t))
      case Hoja(c,_) => Seq[Char](c)
    }
  }
  def pertenece(s:Seq[Char], t:Trie): Boolean = {
  // Devuelve true si la secuencia s es reconocida por el trie t, y false si no.
...
  }
  def adicionar(s:Seq[Char], t:Trie):Trie ={
    // Adiciona una secuencia de uno o mas caracteres a un trie
...
  }

  def arbolDeSufijos(ss:Seq[Seq[Char]]):Trie = {
    // dada una secuencia no vacia de secuencias devuelve el arbol de sufijos asociado a esas secuencias
...
  }

}
```

7.2. Paquete *ReconstCadenas*

Las funciones correspondientes a cada ejercicio, `reconstruirCadenaIngenuo`, `reconstruirCadenaMejorado`, `reconstruirCadenaTurbo`, `reconstruirCadenaTurboMejorada` y `reconstruirCadenaTurboAcelerada` deben ser implementadas en un paquete de Scala denominado `ReconstCadenas`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
import ArbolSufijos._
import Oraculo._

package object ReconstCadenas {

  def reconstruirCadenaIngenuo(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuencia que hay que reconstruir (n), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
...
  }

  def reconstruirCadenaMejorado(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuencia que hay que reconstruir (n), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
...  }

  def reconstruirCadenaTurbo(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuencia que hay que reconstruir (n, potencia de 2), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
...  }

  def reconstruirCadenaTurboMejorada(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuencia que hay que reconstruir (n, potencia de 2), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
    // Usa el filtro para ir mas rapido
...
  }

  def reconstruirCadenaTurboAcelerada(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuencia que hay que reconstruir (n, potencia de 2), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
    // Usa el filtro para ir mas rapido
    // Usa arboles de sufijos para guardar Seq[Seq[Char]]
...
  }
}
```

7.3. Paquete *ReconstCadenasPar*

Las funciones correspondientes a cada ejercicio, `reconstruirCadenaIngenuoPar`, `reconstruirCadenaMejoradoPar`, `reconstruirCadenaTurboPar`, `reconstruirCadenaTurboMejoradaPar` y `reconstruirCadenaTurboAceleradaPar` deben ser implementadas en un paquete de Scala denominado `ReconstCadenasPar`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
import common._
import scala.collection.parallel.CollectionConverters._
```

```
import Oraculo._
import ArbolSufijos._

package object ReconstCadenasPar {
  // Ahora versiones paralelas

  def reconstruirCadenaIngenuoPar(umbral:Int)(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuencia que hay que reconstruir (n), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa paralelismo de tareas
...
  }

  def reconstruirCadenaMejoradoPar(umbral:Int)(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuencia que hay que reconstruir (n), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
    // Usa paralelismo de tareas y/o datos
...
  }

  def reconstruirCadenaTurboPar(umbral:Int)(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuencia que hay que reconstruir (n, potencia de 2), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
    // Usa paralelismo de tareas y/o datos
...
  }

  def reconstruirCadenaTurboMejoradaPar(umbral:Int)(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuecnia que hay que reconstruir (n, potencia de 2), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
    // Usa paralelismo de tareas y/o datos
...
  }

  def reconstruirCadenaTurboAceleradaPar(umbral:Int)(n:Int,o:Oraculo):Seq[Char]= {
    // recibe la longitud de la secuecnia que hay que reconstruir (n, potencia de 2), y un oraculo para esa secuencia
    // y devuelve la secuencia reconstruida
    // Usa la propiedad de que si s=s1++s2 entonces s1 y s2 tambien son subsecuencias de s
    // Usa arboles de sufijos para guardar Seq[Seq[Char]]
    // Usa paralelismo de tareas y/o datos
...
  }
}
```

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

La fecha de entrega límite es el 12 de junio de 2025 a las 23:59. La sustentación será el 19 de junio 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. Particularmente debe describir su implementación de *trie* y por qué son correctas las funciones `pertenece`, `adicionar` y `arbolDeSufijos`. 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.
- 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!!!