Proyecto Final **Programación Funcional y Concurrente**

Carlos Andrés Delgado

@Authors
Carlos Camacho Castaño – 2160331

Juan José Hernández Arenas - 2259500

Santiago Reyes Rodríguez - 2259738

Readme//

En el siguiente código se presentan algoritmos de reconocimiento de subcadenas con el fin de reconocer el patrón denominado genoma humano. El objetivo es lograr describir una pieza de DNA por medio de un patrón o cadena de caracteres compuestos por las letras (a, c, g, t). Con las diferentes versiones implementadas a continuación, cada una mejor que la anterior, se pretende mejorar cada vez más los tiempos de ejecución, el rendimiento y la complejidad de estas, además de poder comparar su desempeño con sus versiones paralelas y establecer si la paralelización es viable para este caso.

Funciones Auxiliares//

```
val alfabeto
```

Define un conjunto de caracteres que representa un alfabeto para secuencias de ADN. En este caso, el alfabeto está formado por las letras 'a', 'c', 'g' y 't'.

```
type Oraculo
```

Define un tipo llamado Oraculo que representa una función que toma una secuencia de caracteres y devuelve un valor booleano. Básicamente, el oráculo es una función que puede evaluar una secuencia de caracteres y dar como resultado un valor verdadero o falso, en caso de que sea subcadena o no.

```
def secuenciaaleatoria
```

Esta función genera una secuencia aleatoria de caracteres de longitud especificada (tamano). Utiliza el conjunto de caracteres definido en alfabeto y la clase Random para generar caracteres aleatorios. La función devuelve una cadena de caracteres que representa la secuencia aleatoria generada.

Implementaciones//

```
def ReconstruirCadenaIngenuo
```

La función ReconstruirCadenaIngenuo es una reconstrucción de una cadena de caracteres de una longitud específica, utilizando un conjunto de caracteres dado por **alfabeto** y un **oráculo (o)** que nos indica si su predecesor es subcadena o no.

La función toma como entrada un conjunto de caracteres (alfabeto), la longitud deseada de la cadena (longitud), y un oráculo (o) que determina si una secuencia de caracteres es válida.

```
def CadCandidatas
```

Esta es una función interna que genera todas las posibles combinaciones de secuencias de caracteres de la longitud especificada. Utiliza recursión para construir las combinaciones posibles.

```
val combinacionesPosibles
```

Se llama a la función **CadCandidatas** para obtener todas las combinaciones posibles de secuencias de caracteres de la longitud deseada, después, se aplica la función **flatMap** sobre las combinaciones posibles. Para cada secuencia seq, evalúa si es válida según el oráculo o. Si es válida, la secuencia se mantiene; de lo contrario, se descarta (None). El resultado final es una lista plana de las secuencias válidas.

Básicamente, la función reconstruye una cadena de caracteres de la longitud especificada, probando todas las combinaciones posibles y utilizando el oráculo para verificar la validez de cada combinación. La implementación se nombra ingenuo, ya que genera todas las combinaciones posibles, sin realizar optimizaciones más avanzadas.

```
def ReconstruirCadenaMejorado
```

La función **ReconstruirCadenaMejorado** es una versión mejorada de la anterior, utilizando un enfoque más eficiente al generar y filtrar subcadenas candidatas.

La función toma como entrada un conjunto de caracteres (alfabeto), la longitud deseada de la cadena (longitud), y un oráculo (o) que determina si una secuencia de caracteres es válida.

```
def subcadenas_candidatas
```

Esta es una función interna que genera subcadenas candidatas de longitud “m” a partir de las subcadenas candidatas existentes **SC**. Utiliza la función flatMap para agregar cada carácter del alfabeto a cada subcadena candidata y luego filtra aquellas que son válidas según el oráculo (o).

Posteriormente, se inicializa la lista de subcadenas candidatas (SC) llamando a la función subcadenas_candidatas con una subcadena inicial vacía de longitud 0.

Después se busca la primera subcadena en SC que tiene la longitud deseada (longitud). Si se encuentra, se devuelve esa subcadena; de lo contrario, se devuelve una cadena vacía **(Seq()).**

Básicamente, la función **ReconstruirCadenaMejorado** utiliza un enfoque más eficiente al generar subcadenas candidatas, evitando generar todas las combinaciones posibles y filtrando directamente aquellas que son válidas según el oráculo. Esto puede mejorar significativamente el rendimiento en comparación con el enfoque ingenuo utilizado en la primera versión.

```
def reconstruirCadenaTurbo
```

La función **ReconstruirCadenaTurbo** es otra versión mejorada de las anteriores, utilizando un enfoque más avanzado al generar subcadenas candidatas.

La función toma como entrada un conjunto de caracteres (alfabeto), la magnitud deseada (magnitud) y un oráculo (o) que determina si una secuencia de caracteres es válida.

```
def subcadenas_candidatas
```

Como en la implementación mejorada, esta es una función interna genera subcadenas candidatas de longitud m a partir de las subcadenas candidatas existentes **SC**. La variable n se inicializa como el doble de m. Luego, se utiliza la función **flatMap** para agregar cada carácter del alfabeto a cada subcadena candidata y se **filtran** aquellas que son válidas según el oráculo. La función utiliza la recursión para generar subcadenas candidatas hasta que la magnitud m sea mayor que la magnitud deseada (magnitud). En cada iteración, n se actualiza como el doble de m y las subcadenas candidatas se filtran nuevamente.

Cuando se alcanza la magnitud deseada, la función devuelve la lista de subcadenas candidatas SC. Posteriormente se inicializa la lista de subcadenas candidatas (SC) llamando a la función subcadenas_candidatas con una subcadena inicial vacía de longitud 0 y con m y n inicializados en 1. Esto es la fase de inicialización de la búsqueda. El SC.finde lo que hace es buscar la primera subcadena en SC que tiene la magnitud deseada (magnitud). Si se encuentra, se devuelve esa subcadena; de lo contrario, se devuelve una cadena vacía (Seq()).

Básicamente, la función **ReconstruirCadenaTurbo** utiliza un enfoque más avanzado al generar subcadenas candidatas mediante el uso de la variable n y el filtrado continuo de subcadenas válidas. La fase de inicialización se realiza llamando a la función con una subcadena vacía y luego se busca la primera subcadena que tenga la magnitud deseada. Este enfoque puede ser más eficiente en términos de rendimiento en comparación con las implementaciones anteriores.

```
def reconstruirCadenaTurboMejorado
```

Esta es una versión mejorada, bastante similar a la versión anterior.

Los principales cambios fueron:

Condición de salida mejorada: Se ha cambiado la condición if (m <= magnitud) por if (m > magnitud). Ahora, la función deja de generar subcadenas candidatas cuando la magnitud actual (m) es mayor que la magnitud deseada, lo cual podría ser más eficiente en términos de rendimiento al evitar cálculos innecesarios.

Reemplazo de find por getOrElse: En lugar de utilizar find y luego manejar el caso en el que no se encuentre ninguna subcadena de la longitud deseada, se utiliza **getOrElse(Seq()).**

Esto simplifica el código y evita la necesidad de manejar explícitamente el caso de no encontrar ninguna subcadena de la longitud deseada. Estos cambios están orientados a mejorar la eficiencia y la claridad del código.

//Funciones necesarias para implementar reconstruirTurboAcelerada

//Clase Trie

Se define la clase abstracta llamada Trie (árbol de sufijos) ubicada en **/src/main/scala/Proyecto/Trie.scala.** Esta sirve como base para las clases específicas que representan nodos y hojas en el Trie, lo que sirve para buscar cadenas de manera eficiente.

```
case class Nodo
```

Representa un nodo interno en el Trie. Un nodo tiene un carácter (car), un indicador de si está marcado (marcada), y una lista de hijos (hijos), que son también instancias de Trie

```
case class Hoja
```

Representa una hoja del Trie. Al igual que los nodos, tiene un carácter y un indicador de si está marcada.

```
def raiz
```

Se define una función raiz que toma un objeto Trie y devuelve el carácter en la raíz del Trie. Si el objeto es un nodo, se devuelve el carácter del nodo; si es una hoja, se devuelve el carácter de la hoja.

```
def cabezas
```

Define una función cabezas que toma un objeto Trie y devuelve una secuencia de caracteres que representan las cabezas de los nodos en el Trie. Si el objeto es un nodo, se devuelve una secuencia de los caracteres en los hijos del nodo; si es una hoja, se devuelve una secuencia que contiene el carácter de la hoja.

```
object pre
```

Contiene versiones adicionales de las funciones raiz y cabezas. Puedes utilizar estas funciones a través del nombre de objeto pre.

Básicamente, esta implementación nos permite construir y manipular instancias de nodos internos (Nodo) y hojas (Hoja) y utilizar las funciones raiz y cabezas para acceder a la información contenida en el Trie.

//Auxiliares

```
def pertenece
```

Esta función verifica si una secuencia de caracteres (sec) pertenece al Trie proporcionado (trie).:

Si la secuencia sec está vacía, verifica si el nodo actual (trie) está marcado. Si la secuencia no está vacía y el nodo actual es un Nodo, verifica si alguno de los hijos tiene la misma raíz que el primer carácter de sec y recursivamente verifica el resto de la secuencia en ese hijo. Si la secuencia no está vacía y el nodo actual es una Hoja, devuelve false ya que una hoja no puede tener más hijos.

```
def adicionar
```

Esta función agrega una secuencia de caracteres (s) al Trie proporcionado (t).

La función interna nuevaRama crea una nueva rama (ya sea un nodo interno o una hoja) a partir de una secuencia de caracteres.

```
def adicionaraux
```

La función principal adicionaraux realiza la adición, considerando los casos específicos de los nodos internos y las hojas: Si el nodo actual es un Nodo y la secuencia no está vacía, agrega una nueva rama con el primer carácter de la secuencia y continúa el proceso de manera recursiva. Si el nodo actual es un Nodo y la secuencia está vacía, marca el nodo actual. Si el nodo actual es un Nodo y la secuencia no está vacía, pero la cabeza de la secuencia ya está presente en las cabezas de los hijos del nodo actual, actualiza los hijos de manera recursiva. Si el nodo actual es una Hoja, crea un nuevo nodo interno con la nueva rama y lo devuelve.

```
def arbolSufijos
```

Esta función crea un Trie a partir de una secuencia de secuencias de caracteres (sec).

Utiliza la función adicionar para agregar cada secuencia al Trie, comenzando con un nodo raíz vacío. La función foldLeft es utilizada para aplicar la operación de adición de manera acumulativa a lo largo de la secuencia.

Básicamente, estas funciones implementan la manipulación del Trie, permitiendo verificar la pertenencia de secuencias, agregar nuevas secuencias al Trie y construir un Trie a partir de un conjunto de secuencias de sufijos. Esto se hace útil para la búsqueda y manipulación de las cadenas.

//Implementación

```
def reconstruirCadenaTurboAcelerada
```

Esta función realiza una búsqueda de posibles subcadenas a partir del alfabeto proporcionado y utiliza el Oraculo (o) para filtrar las subcadenas que coinciden con el patrón.

Toma dos parámetros: n, para el número de iteraciones, y o, que es el oráculo utilizado para evaluar las subcadenas generadas.

```
def posiblesSucadenas
```

Es una función interna que genera posibles subcadenas concatenando elementos del alfabeto y las filtra utilizando la función filtrar.

```
def filtrar
```

Las subcadenas generadas se filtran utilizando la función filtrar. Se utiliza un árbol de sufijos (arbolDeSufijos) construido a partir de las subcadenas anteriores para filtrar las subcadenas actuales.

Recursión y Ampliación: El proceso se repite de manera recursiva, duplicando el tamaño (j * 2) de las subcadenas y filtrando nuevamente.

Resultado Final: El resultado final es la primera subcadena que cumple con las condiciones proporcionadas por el oráculo (o). Las demás cadenas son descartadas lo que hace este proceso apropiado para buscar de manera eficiente subcadenas que cumplen los criterios definidos por el oráculo, y el proceso se realiza de manera recursiva para ampliar la búsqueda.

Básicamente, esta función está diseñada para generar y filtrar subcadenas de manera eficiente utilizando recursión. Lo cual reduce los tiempos de ejecución y ser más eficiente dependiendo de la longitud.

//Implementaciones paralelas

```
def reconstruirCadenaIngenuoPar
```

Esta función es la implementación paralela de reconstruirCadenaIngenuo, es una implementación de la generación de todas las posibles combinaciones de secuencias de caracteres de una longitud específica utilizando un enfoque ingenuo, pero con una paralelización que utiliza la función parallel.

División del Alfabeto: La función divide el alfabeto en dos partes (alfabeto1 y alfabeto2) para realizar la generación de subcadenas de manera paralela.

Generación Paralela de Subcadenas Candidatas: Utiliza la función parallel para generar subcadenas candidatas de manera paralela para cada mitad del alfabeto. Cada mitad realiza recursivamente la generación de subcadenas.

Combinación de Resultados Paralelos: Combina los resultados paralelos (p1 y p2) para obtener todas las combinaciones posibles de subcadenas.

Filtrado según el Oráculo: Filtra las combinaciones según el oráculo (o) y devuelve las secuencias que cumplen con la condición especificada.

Básicamente, la función utiliza paralelización para acelerar la generación de subcadenas candidatas, pero sigue utilizando un enfoque ingenuo para generar todas las combinaciones posibles.

```
def reconstruirCadenaMejoradoPar
```

Esta función es la paralelización de reconstruirCadenaMejorado, la cual implementa la paralelización para la creación de subcadenas candidatas.

Creación de Tareas Paralelas: Utiliza task para crear tareas paralelas que generan nuevas subcadenas candidatas. Cada tarea corresponde a agregar un carácter al final de una subcadena existente.

Ejecución y Filtrado Paralelo: Ejecuta las tareas en paralelo y filtra las nuevas subcadenas candidatas según el oráculo (o). Esto se hace mediante el uso de join() para obtener el resultado de cada tarea.

Llamada Recursiva con Nuevas Candidatas: Realiza una llamada recursiva con las nuevas subcadenas candidatas que han sido filtradas y generadas en paralelo.

Condición de Salida: Cuando la longitud deseada se alcanza (m > longitud), devuelve las subcadenas existentes. Si no se alcanza la longitud deseada, continúa generando nuevas subcadenas.

Resultado: Devuelve la primera subcadena de la longitud deseada encontrada, o una secuencia vacía si ninguna cumple con la condición.

Básicamente, esta versión utiliza paralelización para acelerar la generación y filtrado de subcadenas candidatas, mejorando potencialmente el rendimiento en comparación con la versión no paralelizada. La eficiencia dependerá de factores como la longitud y el hardware.

```
def reconstruirCadenaTurboPar
```

Esta función es la paralela de la función reconstruirCadenaTurbo, utiliza la biblioteca Fork-Join para aprovechar la concurrencia.

Clase Subcadenas: Una clase interna que extiende RecursiveTask y representa una tarea paralela para generar subcadenas candidatas.

La función compute realiza la lógica de generación y filtrado de subcadenas de manera similar a las versiones anteriores.

Generación y Paralelización de Tareas: La función genera subcadenas candidatas y luego crea tareas paralelas para procesar esas subcadenas. Las tareas paralelas se ejecutan en el pool Fork-Join.

Recopilación de Resultados Paralelos: Las tareas paralelas se ejecutan de manera concurrente, y los resultados se recopilan y se combinan para obtener el resultado final.

Fork-Join Pool: Se crea un nuevo pool de Fork-Join para gestionar las tareas paralelas.

Llamada Principal y Resultado: Se inicia la tarea principal y se invoca utilizando el pool Fork-Join. El resultado es una secuencia que contiene la primera subcadena de la longitud deseada encontrada, o una secuencia vacía si ninguna cumple con la condición.

Básicamente, esta versión utiliza Fork-Join para realizar la generación y filtrado de subcadenas candidatas de manera paralela, lo que puede mejorar el rendimiento en situaciones donde la concurrencia puede acelerar el proceso. La eficiencia dependerá de factores como la longitud y el hardware.

```
def reconstruirCadenaTurboMejoradoPar
```

Aquí se implementa una versión paralela de la función reconstruirCadenaTurboMejorado. En este caso, se utiliza el framework Fork-Join para realizar la ejecución paralela de tareas.

- **SubcadenasTask:** Se define una clase interna que extiende RecursiveTask[Seq[Seq[Char]]]. Esta clase representa una tarea que se divide recursivamente para generar subcadenas candidatas en paralelo.
- **compute():** El método compute implementa la lógica de la tarea. Si la magnitud actual (m) es mayor que la magnitud deseada, la tarea devuelve la lista de subcadenas actual (SC). En caso contrario, genera nuevas subcadenas candidatas y las filtra utilizando el oráculo (oraculo)
- **Creación de tareas paralelas:** Para cada nueva subcadena candidata, se crea una nueva instancia de SubcadenasTask con m + 1 y la secuencia de esa subcadena. Estas tareas se inician en paralelo utilizando el método fork.
- **Recopilación de resultados paralelos:** Se espera la finalización de todas las tareas paralelas mediante el método join. Los resultados se recopilan en la lista results. Devuelve el resultado combinado: El resultado final de la tarea principal es la concatenación de todos los resultados recopilados.
- **Creación del Fork-Join Pool:** Se crea una instancia de ForkJoinPool, que es el pool de hilos utilizado para ejecutar tareas Fork-Join. Inicia la tarea principal y obtiene el resultado: Se crea una instancia de SubcadenasTask para la tarea principal con la magnitud inicial y la secuencia vacía. Luego, se invoca la tarea principal utilizando invoke en el Fork-Join Pool, obteniendo así el resultado final. Devuelve la primera subcadena de la longitud deseada encontrada: Finalmente, se

utiliza el método find para buscar la primera subcadena en el resultado final que tenga la longitud deseada. Si se encuentra, se devuelve esa subcadena; de lo contrario, se devuelve una secuencia vacía.

Esta implementación paralela debería aprovechar la capacidad de procesamiento paralelo para mejorar el rendimiento al generar subcadenas candidatas en paralelo. Se debe tener en cuenta que la eficacia de la paralelización puede depender de la longitud y del hardware.

```
def reconstruirCadenaTurboAceleradaPar
```

La función **reconstruirCadenaTurboAceleradaPar** es la versión paralela de reconstruirCadenaTurboAcelerada, la cual reconstruye una cadena a partir de un conjunto de secuencias candidatas.

- **posiblesSucadenas:** Esta función genera posibles subcadenas candidatas de longitud creciente hasta alcanzar el tamaño deseado (tamano). Se divide la secuencia de subcadenas en dos partes (sc1 y sc2) y se realiza la generación en paralelo. Luego, se filtran las subcadenas de acuerdo con ciertos criterios utilizando la función filtrar. La función es recursiva y se llama a sí misma con el doble de la longitud actual (n * 2).

```
def filtrar
```

Verifica si la longitud de las subcadenas actuales es mayor a 2. Si es así, calcula los sufijos a partir de las subcadenas pasadas (cadPasada). Divide las subcadenas actuales en dos partes (cad1 y cad2). Utiliza paralelismo para filtrar las subcadenas de manera eficiente, aplicando condiciones específicas basadas en la función pertenece.

Generación del **Alfabeto** y **Subcadenas** Iniciales, por medio de la variable Alfabeto2 se crea una lista de secuencias individuales del alfabeto y filtra según el oráculo (o). Por medio de la variable SS se inicia el proceso de generación de subcadenas llamando a posiblesSucadenas con la mitad inicial del alfabeto.

El resultado final devuelve la primera subcadena que cumple con las condiciones filtradas, que podría considerarse como la mejor candidata según el proceso de generación y filtrado.

Básicamente, la función genera de manera eficiente posibles subcadenas de ADN y las filtra en paralelo, utilizando condiciones específicas basadas en sufijos y la función pertenece. El resultado final es la primera subcadena que cumple con los criterios establecidos. Se usa la recursión de cola, haciendo que el programa sea más eficiente gracias a la concurrencia aprendida en clase.

//Colecciones empleadas

Se emplearon las colecciones **common.parallel y java.util.concurrent.{ForkJoinPool, RecursiveTask}**

- **import common.parallel:** Este import está relacionado con un paquete common que contiene funcionalidades relacionadas con la programación paralela.
- **import java.util.concurrent.{ForkJoinPool, RecursiveTask}:** Estas importaciones provienen del paquete java.util.concurrent. En este caso, se están importando dos clases específicas: ForkJoinPool: Se utiliza para administrar pools de hilos para tareas Fork-Join. La clase ForkJoinPool proporciona métodos para iniciar y gestionar tareas paralelas. RecursiveTask: También parte de java.util.concurrent, RecursiveTask es una clase abstracta que se utiliza para representar tareas que pueden dividirse en sub-tareas más pequeñas y ejecutarse en paralelo. La tarea principal debe extender esta clase e implementar el método compute.

Las implementaciones emplean la función parallel para realizar operaciones en paralelo. Esto puede ser beneficioso en sistemas multi-core, pero hay que tener en cuenta que la sobrecarga de la creación y gestión de hilos puede superar los beneficios de paralelismo en problemas pequeños o si el tiempo de ejecución paralelo es menor que el tiempo de ejecución secuencial.

Se debe tener en cuenta que la programación paralela puede ofrecer mejoras significativas en ciertos casos, pero también introduce complejidad y posibles problemas de concurrencia.

//Comparación de implementaciones

Para la comparación entre las dos implementaciones, secuenciales y paralelas, se crea el Benchmarking

La función compararAlgoritmos realiza una comparación de rendimiento entre dos funciones Funcion1 y Funcion2 que toman tres argumentos (Seq[Char], Int, Oraculo) y devuelven Seq[Char].

La comparación se realiza midiendo el tiempo de ejecución de cada función con un conjunto dado de parámetros (alfabeto, magnitud, o) y calculando el promedio del tiempo de ejecución de la primera función respecto a la segunda.

- **Parámetros de Entrada:** Funcion1: La primera función que se va a comparar. Funcion2: La segunda función que se va a comparar.
- **alfabeto:** Una secuencia de caracteres (a, c, g, t) utilizada como entrada para las funciones.
- **magnitud:** EL tamaño de la secuencia.
- **o:** El oráculo, es decir, una función Seq[Char] => Boolean.

- **Medición del Tiempo de Ejecución:** Se utiliza la función withWarmer del de benchmarking para medir el tiempo de ejecución de cada función. timeF1 y timeF2 contienen los resultados de estas mediciones.
- **Cálculo del Promedio:** scala Copy code val promedio = timeF1.value / timeF2.value Se calcula el promedio del tiempo de ejecución de la primera función respecto a la segunda. Este valor puede proporcionar una indicación de cuánto más rápido o más lento es un algoritmo en comparación con el otro.
- **Resultados Devueltos:** scala Copy code (timeF1.value, timeF2.value, promedio) La función devuelve una tupla que contiene: El tiempo de ejecución de la primera función (timeF1.value). El

tiempo de ejecución de la segunda función (timeF2.value). El promedio de los tiempos de ejecución (promedio).

Esto nos permite comparar fácilmente el rendimiento relativo de dos las dos implementaciones bajo las mismas condiciones de entrada.

//Clase Comparar

Se implementa la clase Comparar ubicada en app/src/main/scala/Proyecto/Comparar.scala, que tiene como objetivo realizar comparaciones entre las diferentes implementaciones de algoritmos de reconstrucción de secuencias de ADN, tanto en versiones secuenciales como paralelas.

```
def comparaciones
```

Esta función toma dos funciones de reconstrucción de cadena (funcion1 y funcion2), un nombre (name) y un umbral (Umbral). Itera sobre tamaños de secuencia desde 1 hasta el umbral especificado. Posteriormente dentro del bucle, crea una secuencia aleatoria (sec) de tamaño 2^i. Crea un oráculo (m) que verifica si una secuencia aleatoria contiene una subcadena dada. Mide el tiempo secuencial y paralelo utilizando la función Benchmark.compararAlgoritmos, e imprime los resultados en cada iteración permitiendo observar los diferentes resultados de las versiones secuenciales y paralelas, además de su aceleración, para así poder hacer una comparación directa y establecer conclusiones pertinentes.

```
def main
```

Calienta la JVM con una operación simple (crear un array de 100,000,000 elementos). Llama a comparaciones para comparar diferentes implementaciones de algoritmos de reconstrucción de cadenas. Después, compara las versiones secuenciales con sus contrapartes paralelas utilizando la función Benchmark.compararAlgoritmos.

En resumen, la clase Comparar proporciona un marco para comparar nuestras implementaciones secuenciales y paralelas de algoritmos de reconstrucción de secuencias de ADN bajo diferentes tamaños de secuencia y condiciones de oráculo, permitiendo hacer un análisis y establecer si la paralelización es pertinente.

//Clase ComparарX

Se implementa la clase ComparарX ubicada en app/src/main/scala/Proyecto/Comparar.scala. Se implementa para realizar comparaciones entre distintas las diferentes versiones de algoritmos de reconstrucción de secuencias, tanto secuenciales como paralelos, utilizando el ScalaMeter para medir el rendimiento.

```
def main
```

Calienta la **JVM** con una operación simple (crear un array de 100,000,000 elementos). Crea una secuencia aleatoria de tamaño magnitud. Define un oráculo (o) que verifica si la secuencia aleatoria contiene una subcadena dada.

Se imprimen el nombre de la comparación y la secuencia aleatoria generada. Utilizamos la función Benchmark.compararAlgoritmos para comparar el rendimiento de las implementaciones secuenciales y paralelas de cada algoritmo. Imprime los resultados de las comparaciones.

Muy parecida a la Comparar, la clase ComparарX organiza y ejecuta comparaciones entre las diferentes versiones de algoritmos de reconstrucción de cadenas. A diferencia de la clase Comparar, esta clase mide su rendimiento con diferentes tamaños de secuencia utilizando el ScalaMeter.

Estos nos permite generar un análisis adicional manipulando la longitud de la cadena a conveniencia y así establecer qué versiones se comportan mejor en los diferentes tamaños de cadena que se van probando, y establecer si los tiempos son proporcionales a la longitud de la cadena y a la versión del algoritmo, ya sea secuencial o paralela.

//Comparación de rendimiento

Cadena ingenuo e ingenuoPar			
Tamaño	Tiempo secuencial	Tiempo paralelo	Aceleración
2	0,2687	0,4629	0,580470944048390
4	0,833101	1,351799	0,616290587579958
8	113,169901	105,6615	1,071060897299390

Al comparar los tiempos de ejecución de cada algoritmo, concluimos que la versión secuencial tiene mejor tiempo que la versión paralela en las secuencias de tamaños 2 y 4, pero en el tamaño 8, la versión paralela es más veloz que la secuencial, con una aceleración de 1,071. No comparamos cadenas con más tamaño, ya que el algoritmo de ingenuo e ingenuoPar se quiebra al hacerlos con secuencias desde el tamaño 16.

Cadena mejorado y mejoradoPar			
Tamaño	Tiempo secuencial	Tiempo paralelo	Aceleración
2	1,2944	0,6668	1,94121175764847
4	0,1611	0,634	0,254100946372239
8	0,5426	0,7488	0,724626068376068
16	1,0785	1,612	0,669044665012406
32	4,4849	4,738	0,946640775059628
64	17,3001	19,2063	0,900751315974446
128	123,8459	114,3085	1,08343561502425
256	839,6672	881,1475	0,952924680601147
512	6.492,4713	6.150,6615	1,05557285179813
1024	28.129,1645	27.777,8971	1,01264557207968

Se observa que la aceleración es significativa para tamaños de entrada pequeños (2, 4), pero disminuye a medida que el tamaño de entrada aumenta, por ejemplo, que la versión paralela tiene un tiempo de casi el doble de la versión secuencial. La aceleración inicialmente mejora con tamaños pequeños, pero a medida que aumenta el tamaño de entrada, la mejora se reduce. En algunos casos (por ejemplo, tamaño 64), la ejecución paralela puede ser ligeramente más lenta que la secuencial.

Cadena turbo y turboPar			
Tamaño	Tiempo secuencial	Tiempo paralelo	Aceleración
2	0,391	0,8576	0,456273320895522
4	0,1227	0,6461	0,189908682866429
8	0,3675	1,4274	0,257461118116855
16	0,9314	1,7639	0,52803446907421
32	1,4204	2,5546	0,556016597510373
64	7,0363	3,7799	1,8615042725998
128	64,4361	13,5068	4,77064145467468
256	445,6504	58,8209	7,57639546487727
512	3.725,9451	588,5650	6,33055839202127
1024	27.537,997	4.338,9278	6,34672856275690

Al comparar las funciones turbo y turboPar, observamos que la versión secuencial es más rápida que la versión paralela desde el tamaño 2 al 32, desde el 64 la comparación hace un cambio asombroso, ya que se cambian los papeles, haciendo que la versión paralela gane, teniendo aceleraciones de más del doble. Por ejemplo, en la comparación del tamaño 256, la versión paralela es siete veces más rápida que la versión secuencial.

Cadena turboMejorado y turboMejoradoPar			
Tamaño	Tiempo secuencial	Tiempo paralelo	Aceleración
2	0,444	1,0278	0,431990659661412
4	0,2384	0,923	0,258288190682556
8	0,1961	1,3918	0,140896680557551
16	0,4863	2,3201	0,096030343519676
32	1,7677	2,6587	0,664873810508895
64	11,5606	3,6916	3,13159605591071
128	61,0745	11,8996	5,13248344482167
256	444,3220	72,9530	6,09052403602319
512	3.424,2001	518,1295	6,60877271029732
1024	25.847,4149	4.180,2705	6,18319194894206

En la comparación turboMejorado y turboMejoradoPar, observamos un comportamiento idéntico a la comparación anterior, la versión secuencial tiene mejor tiempo en las secuencias de tamaños pequeños, mientras que la versión paralela es mejor desde el tamaño 64 y tiene tiempos de más del triple que el tiempo de su versión secuencial. Por ejemplo, en el tamaño 16 la versión secuencial gana por 4 veces que la versión paralela; mientras que en el tamaño 512, la versión paralela tiene un tiempo de más de **6,6 veces**.

Cadena turboAcelerada y turboAceleradaPar			
Tamaño	Tiempo secuencial	Tiempo paralelo	Aceleración
2	0,1792	0,2312	0,775086505190311
4	0,3577	0,7058	0,506800793425899
8	0,736	1,0066	0,731174249950327
16	1,6105	3,071	0,524422012373819
32	6,8844	5,2118	1,32092559192601
64	36,5103	38,7347	0,942573454809253
128	560,7011	525,377	1,06723571835082
256	6.842,7983	6.501,706	1,05246196921238
512	96.764,4051	70.580,8187	1,37097311822482

Observamos que es más viable realizar la cadena turbo acelerada de forma secuencial cuando se trata de cadenas de tamaño pequeño,ya que tienen un tiempo menor a comparación de la forma paralela, por ejemplo en el tamaño 4, la versión secuencial es dos veces más rápida que la versión paralela. En cambio cuando se tratan de magnitudes grandes, es más sencillo realizar la cadena con la versión paralela, por ejemplo la del tamaño 512, que la versión paralela tiene un tiempo de 70580,81 con una aceleración de **1,371**.

//Conclusiones generales

Para terminar, cerraremos con unas conclusiones generales en donde podemos destacar los siguientes aspectos identificados y observados en este proyecto:

- El algoritmo de cadenaIngenuo y su version paralela no resisten a cadenas con un tamaño mayor a 8, es un proceso muy demorado ya que multiplica todas las opciones posibles al ser producto cartesiano, dependiendo de los componentes de la computadora el proceso se podría ver abortado, como en este caso. Por lo general no alcanzan longitudes altas.
- El oráculo fue muy necesario para el proceso de la creación de las funciones, ya que con este identificar las cadenas fué más práctico y sencillo.
- Las versiones paralelas de las funciones, tienen mejor tiempo si se tratan de secuencias con tamaños grandes(aprox 2^4 en adelante), mientras que la versión secuencial es más veloz con cadenas de tamaños pequeños.
- La paralelización de datos no siempre es la mejor opción, ya que al ser un proceso más complejo que el secuencial hay que tener en cuenta la naturaleza del problema, la escalabilidad y hasta el hardware para establecer si es pertinente.
- Y finalmente notamos como apropiando todos los recursos brindados para este proyecto (Abstracción, Colecciones, Concurrencia, Listas,entre otros) se puede lograr cambiar nuestra orientación sobre la programación y construir nuevos sistemas y algoritmos con nuevas características y capacidades, en pro de nuestro desarrollo y futuro.
- La concurrencia fue importante en los algoritmos cuando se buscaba mejorar el rendimiento y la eficiencia en la ejecución de tareas, reduciendo el tiempo de ejecución de las mismas.
- En general, es importante tener en cuenta los distintos aspectos que pueden influir en el desempeño de un algoritmo ya que las diferentes implementaciones son apropiadas o no, dependiendo las condiciones dadas.