

INFORME PROYECTO FINAL PROGRAMACIÓN FUNCIONAL Y CONCURRENTE - PROBLEMA DE LA RECONSTRUCCIÓN DE ADN

Víctor Manuel Hernández Ortiz
2259520
Universidad Del Valle

Jhon Alejandro Martinez Murillo
2259565
Universidad Del Valle

I. CORRECCIÓN DE LAS FUNCIONES IMPLEMENTADAS

A. Definición del Oráculo

```
type Oraculo = Seq[Char] => Boolean

def oraculoFunc(cadena: Seq[Char]): Oraculo = {
  (subcadena: Seq[Char]) =>
    cadena.mkString.contains(subcadena.mkString)
}
```

La función *oraculoFunc* actúa como un generador de oráculos que permiten verificar la presencia de subcadenas específicas en una cadena principal dada.

B. Solución Ingenua

Para la solución ingenua, se genera una lista de combinaciones posibles para cadenas del tamaño de la que se desea hallar, posteriormente se recorre cada cadena de las combinaciones posibles y se le pregunta al *oraculo* por esta, continua hasta finalizar y por ultimo se retorna el resultado.

```
def reconstruirCadenaIngenuo(n: Int, oraculo: Oraculo): Seq[Char] = {
  val combinaciones = generarCombinaciones(n)
  val cadenaEncontrada = for {
    cadena <- combinaciones
    if oraculo(cadena) == true
  } yield cadena.toSeq
  cadenaEncontrada.head
}
```

C. Solución Ingenua Paralela

Cabe aclarar que decidimos prescindir del uso de un umbral en nuestro análisis de paralelización para examinar detalladamente su comportamiento en los diversos tamaños estudiados.

Para la solución ingenua paralela, adoptamos dos enfoques: paralelización de datos y paralelización de tareas. Inicialmente, observamos que la instrucción *for* (que realiza el recorrido analizando con el *oraculo* cuál de las cadenas pertenecientes a la lista de todas las combinaciones posibles es la correcta) podía dividirse en varias partes. Si, por ejemplo, tenemos un tamaño de cadena $n = 4$, esto nos daría una lista de combinaciones de tamaño 4^n . Para este ejemplo, $4^4 = 256$ sería el tamaño de la lista. Entonces, podemos dividir esta

lista de combinaciones en 4 partes, cada una de tamaño 64, y con ello usar un *for* para cada sublista de tamaño 64 de forma paralela. De esta manera, cada instrucción *for* se ejecuta simultáneamente a las otras, lo que, en teoría, se traduce en menos tiempo para hallar la cadena correcta. Otra mejora es cambiar la instrucción *for* por un *filter* que realiza la misma acción de hallar la cadena mediante el *oraculo*, pero gracias al *filter*, podemos aplicar paralelización de datos a la secuencia que se filtrará. De esta forma, utilizamos una secuencia de datos paralela, aplicando paralelismo de datos, y cada comparación de las sublistas de combinaciones se ejecuta de forma paralela.

```
//tarea de recorrido, recibe una porcion de la lista de combinaciones
def tareaRecorrido(bloqueCombinaciones: Seq[String]) : Seq[Char] = {
  val combinacionesFiltradas = bloqueCombinaciones.par.
    filter(oraculo(_) == true)
  if (combinacionesFiltradas.isEmpty) {
    Seq.empty[Char]
  } else {
    val combinacion = combinacionesFiltradas.head
    combinacion.toSeq
  }
}

// separo en bloques
val (bloque1, bloque2, bloque3, bloque4) =
  separarCombinaciones(combinaciones)

// Ejecuto las tareas sobre cada bloque
val (resultado1, resultado2, resultado3, resultado4) =
  parallel(tareaRecorrido(bloque1), tareaRecorrido(
    bloque2), tareaRecorrido(bloque3), tareaRecorrido(
    bloque4))
```

Finalmente elijo el resultado que no se encuentre vacío, ya que solo una *tareaRecorrido* arrojará la cadena que se trata de reconstruir.

D. Solución Mejorada

Esta solución utiliza un *oraculo* para reconstruir una cadena de longitud n . Si n es par, genera combinaciones de subcadenas de $n/2$ y las concatena para obtener las combinaciones válidas para el valor del *oraculo*. Si n es impar, realiza el mismo proceso, pero con dos subcadenas de longitudes diferentes, $n/2$ y $n - (n/2)$. Luego, se utiliza una función auxiliar, *reconstruirCadenaMejoradoAux*, que realiza una recursión de cola para generar y acumular combinaciones correctas para el *oraculo*. La recursión se detiene cuando se

alcanza una longitud de $n = 0$. En cada iteración, n se reduce en $n-1$ y se acumulan las combinaciones válidas en un vector.

```
def reconstruirCadenaMejoradoAux(cadena: Seq[Char],
  combinaciones: Seq[String], acumulador: Seq[String], n:
  Int): Seq[String] = {
  if (n == 0) {
    acumulador
  } else {
    val combinaciones = generarCombinaciones(n)

    val cadenaEncontrada = for {
      cadena <- combinaciones
    } if oraculo(cadena) == true
    } yield cadena

    val acumulacion = acumulador ++ cadenaEncontrada
    reconstruirCadenaMejoradoAux(combinaciones.head.
      toSeq, combinaciones.tail, acumulacion, n-1)
  }
}
```

Finalmente, se busca cual es la cadena que coincide 100% con el *oraculo* y se retorna dicha cadena.

E. Solución Mejorada Paralela

El añadido para la paralelización consiste en aplicar el mismo concepto utilizado en la reconstrucción ingenua paralela: dividir una tarea de recorrido en varias tareas y ejecutarlas en paralelo. En este caso, se adapta para su uso dentro de la función *reconstruirCadenaMejoradoAux*, en la cual se divide la lista de combinaciones en dos utilizando la siguiente función:

```
def separarCombinacion(listaCombinaciones: Seq[String]): (
  Seq[String], Seq[String]) = {
  val mitad = listaCombinaciones.size / 2
  (
    listaCombinaciones.slice(0, mitad),
    listaCombinaciones.slice(mitad, listaCombinaciones.
      size)
  )
}
```

Gracias a esto se generan dos bloques, los cuales se mandan a ejecutar paralelamente en la función *tareaPorRecorrer*

```
def tareaPorRecorrer(bloqueCombinaciones: Seq[String],
  oraculo: Oraculo) : Seq[String] = {
  val cadenaEncontrada = for {
    cadena <- bloqueCombinaciones
  } if oraculo(cadena) == true
  } yield cadena
  cadenaEncontrada
}
```

Se mandan los bloques a ejecutarse paralelamente en *tareaPorRecorrer* y finalmente se unen

```
// mando los dos bloques a ejecutarse paralelamente
val (cadenaEncontrada1, cadenaEncontrada2) = parallel(
  tareaPorRecorrer(bloque1, oraculo), tareaPorRecorrer(
    bloque2, oraculo))

val acumulacion = acumulador ++ (cadenaEncontrada1 ++
  cadenaEncontrada2)

//llamado recursivo
reconstruirCadenaMejoradoParallelAux(combinaciones.head.
  toSeq, combinaciones.tail, acumulacion, n-1)
```

También se implementa una paralelización en dos llamadas que anteriormente se ejecutaban secuencialmente. Observamos que podían ejecutarse de forma paralela, especialmente al

verificar el tamaño de la cadena. Si este es impar, se realiza lo siguiente:

```
val subCadenasCorrectas1 = reconstruirCadenaMejoradoAux(
  Seq(), Seq(), Seq(), n/2)
val subCadenasCorrectas2 = reconstruirCadenaMejoradoAux(
  Seq(), Seq(), Seq(), n-(n/2))
```

Por tanto esto se paraleliza de la siguiente manera

```
val (subCadenasCorrectas1, subCadenasCorrectas2) = parallel(
  reconstruirCadenaMejoradoParallelAux( Seq(), Seq(),
    Seq(), n/2), reconstruirCadenaMejoradoParallelAux( Seq
    (), Seq(), Seq(), n-(n/2)))
```

De esta forma se ejecutan paralelamente ahorrando algo de tiempo.

De manera similar, se aplica la paralelización a las soluciones Turbo, Turbo Mejorada y Turbo Acelerada, cada una con sus distintas particularidades, pero en general, el enfoque de paralelización es el mismo.

F. Turbo Solución

La lógica principal de esta solución es similar a la versión mejorada, para longitudes pares, se generan y concatenan subcadenas correctas de $n/2$, y para longitudes impares, se hacen dos llamados recursivos con longitudes $n/2$, $n-(n/2)$ y se combinan los resultados.

La mejora radica en la condición de avance en la recursión que ahora lo hace de $n-2$ en vez de $n-1$. Este cambio evita llamados recursivos innecesarios y mejora la eficiencia del algoritmo.

```
def reconstruirCadenaTurboAux(cadena: Seq[Char],
  combinaciones: Seq[String], acumulador: Seq[String], n:
  Int): Seq[String] = {
  if ((n == 0 || n == 1) && acumulador.length > 0) || n
    < 0) {
    acumulador
  } else {
    val combinaciones = generarCombinaciones(n)

    val cadenaEncontrada = for {
      cadena <- combinaciones
    } if oraculo(cadena) == true
    } yield cadena

    val acumulacion = acumulador ++ cadenaEncontrada
    reconstruirCadenaTurboAux(combinaciones.head.toSeq,
      combinaciones.tail, acumulacion, n-2)
  }
}
```

G. Turbo Mejorada

La lógica principal de esta solución es prácticamente igual a la versión Turbo. La mejora radica en la condición de avance en la recursión, que ahora se realiza de 2^n en lugar de $n-2$. Este cambio evita llamadas recursivas innecesarias y mejora la eficiencia del algoritmo.

```
def reconstruirCadenaTurboMejoradaAux(cadena: Seq[Char],
combinaciones: Seq[String], acumulador: Seq[String], n:
Int, baseInicial: Int, potencia: Int): Seq[String] = {
  if ( baseInicial >= n && acumulador.length > 0 ) {
    acumulador
  } else {
    val combinaciones = generarCombinaciones(n)

    val cadenaEncontrada = for {
      cadena <- combinaciones
      if oraculo(cadena) == true
    } yield cadena

    val acumulacion = acumulador ++ cadenaEncontrada
    val base = math.pow(2, potencia).toInt
    reconstruirCadenaTurboMejoradaAux(combinaciones,
      head.toSeq, combinaciones.tail, acumulacion, n-
      base, base, potencia+1)
  }
}
```

H. Turbo Acelerada

La lógica de esta solución sigue el mismo patrón que la versión Turbo Mejorada, realizando una recursión de 2^n para reconstruir las cadenas válidas y lograr un recorrido más eficiente. La diferencia radica en que utilizamos un Trie para almacenar las cadenas reconstruidas validadas, haciendo uso de la función *arbolDeSufijos* para convertir la lista de cadenas válidas en un árbol. Finalmente, se recorre el árbol en busca de la cadena 100% correcta.

La implementación de Trie que realizamos fue la siguiente:

- 1) Método *raiz*: La función *raiz* determina el carácter de la raíz de un Trie. Dependiendo del tipo de nodo, ya sea *Nodo* o *Hoja*, extrae el carácter correspondiente. Este método es esencial para identificar el inicio de cada rama del Trie.

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

- 2) Método *cabezas*: El método *cabezas* devuelve una secuencia de caracteres representando las cabezas de los nodos descendientes. En el caso de un *Nodo*, se obtienen los caracteres de los hijos; para una *Hoja*, se devuelve el carácter de la hoja en una secuencia. Esto facilita la exploración de las posibles continuaciones de una cadena en el Trie.

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

- 3) Método *arbolDeSufijos*: La función *arbolDeSufijos* crea un Trie a partir de una secuencia de sufijos. Utiliza la función *adicionar* para construir gradualmente la estructura del Trie. Esta estrategia de construcción incremental permite manejar eficientemente grandes cantidades de datos.

```
def arbolDeSufijos(sufijos: Seq[String]): Trie = {
  sufijos.foldLeft(Nodo(' ', false, List.empty[Trie]):
    Trie) { (trie, sufijo) =>
    adicionar(trie, sufijo)
  }
}
```

- 4) Método *adicionar*: El método *adicionar* agrega un sufijo al Trie. Hace uso de la recursión para manejar casos específicos de nodos, permitiendo así la construcción eficiente de la estructura del Trie. La función es fundamental para la creación dinámica del Trie a medida que se adicionan nuevos sufijos.

```
def adicionar(t: Trie, sufijo: String): Trie = sufijo.
  toList match {
    case Nil => t
    case x :: xs =>
      val nuevoHijo = Nodo(x, true, List.empty[Trie])
      t match {
        case Hoja(c, marcada) =>
          Nodo(c, marcada, List(adicionar(nuevoHijo, xs.
            mkString)))
        case Nodo(c, marcada, hijos) =>
          val hijoExistente = hijos.find(h => raiz(h) ==
            x)
          val nuevosHijos = hijoExistente match {
            case Some(h) =>
              hijos.updated(hijos.indexOf(h), adicionar(h
                , xs.mkString))
            case None =>
              hijos :+ adicionar(nuevoHijo, xs.mkString)
          }
          Nodo(c, marcada, nuevosHijos)
      }
  }
```

- 5) Método *generarPosibilidades*: La función *generarPosibilidades* devuelve todas las cadenas posibles representadas por el Trie. Hace uso de la función auxiliar *reconstruyendoPosibilidades*, que explora recursivamente los caminos del Trie para generar combinaciones de caracteres. Este método proporciona una visión global de las cadenas almacenadas en el Trie.

```
def generarPosibilidades(t: Trie): Seq[String] = {
  def reconstruyendoPosibilidades(t: Trie, prefijo:
    String): Seq[String] = t match {
    case Hoja( _, _ ) => Seq(prefijo)
    case Nodo( _, _, hijos ) =>
      if (hijos.isEmpty) {
        Seq(prefijo)
      } else {
        hijos.flatMap(h =>
          reconstruyendoPosibilidades(h, prefijo +
            raiz(h)))
      }
  }
  reconstruyendoPosibilidades(t, "")
}
```

- 6) Método *pertenece*: La función *pertenece* verifica si una cadena pertenece al Trie. Utiliza recursión para recorrer el Trie y determinar si la cadena se encuentra en la estructura.

```
def pertenece(s: String, t: Trie): Boolean = {
  t match {
    case Nodo( c , m , lt ) => {
      if ( s.isEmpty ) {
        m
      } else {
        val ( hijos , resto ) = lt.partition( t
          => raiz( t ) == s.head )
        if ( hijos.isEmpty ) {
          false
        } else {
          pertenece( s.tail,hijos.head )
        }
      }
    }
    case Hoja ( c , m ) => {
      if ( s.isEmpty ) {
        m
      } else {
        false
      }
    }
  }
}
```

- 7) Clase *Hoja*: Representa un nodo final en el Trie que almacena un carácter y una marca booleana indicando si la cadena hasta ese punto es completa.

```
case class Hoja ( car : Char , marcada : Boolean )
extends Trie
```

- 8) Clase *Nodo*: Representa un nodo interno en el Trie. Contiene un carácter, una marca booleana y una lista de hijos. La lista de hijos permite representar la estructura jerárquica del Trie.

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

II. DESEMPEÑO Y COMPARACIÓN DE LAS SOLUCIONES SECUENCIALES Y PARALELAS

Para las pruebas de desempeño, usamos la función *desempenoFunciones*, la cual prueba cada algoritmo 100 veces con cadenas distintas y para distintos tamaños, al final se promedian los resultados de tiempo arrojando el resultado que se verá a continuación.

A. Desempeño de todas las soluciones

Estas cuatro tablas representan una en realidad, pero por efectos visuales decidimos anexarla de esta manera

Tamaño	Ingenua	Ingenua Paralela	Mejorada
2	0.07099	0.397834003	0.046989003
3	0.0409939975	0.221954004	0.0461529986
4	0.256975	0.4479849986	0.052617
5	0.925090993	0.714916002	0.116211004
6	4.205704997	2.559324994	0.164369002
7	19.173248005	10.388582	0.567024
8	89.955017003	50.073333996	0.673484
9	432.899327997	265.497611998	2.430250997
10	1819.61834	1079.72203004	2.613432992

Tamaño	Mejorada Paralela	Turbo Solución	Turbo Solución Paralela
2	0.130685	0.0282669983	0.093362
3	0.110056997	0.020805004	0.099302003
4	0.159283998	0.016641	0.071262002
5	0.210248	0.065309003	0.131954007
6	0.3265710017	0.0573550024	0.101322995
7	0.65730499	0.3327909984	0.337009006
8	0.767055002	0.3542710017	0.402883996
9	2.073047003	1.895841006	1.472026003
10	2.3463179983	1.555699008	1.162396002

Tamaño	Turbo Mejorada	Turbo Mejorada Paralela	Turbo Acelerada
2	0.0207229988	0.093035002	0.04697
3	0.021505007	0.100872998	0.037825004
4	0.017485001	0.084095002	0.039702
5	0.068525998	0.141456997	0.10093
6	0.059777	0.101933998	0.108831004
7	0.292408006	0.287755	0.375736007
8	0.293187	0.24520998	0.4029919985
9	1.917477	1.321653994	1.774043997
10	1.58790995	1.181311	1.848526006

Tamaño	Turbo Acelerada Paralela
2	0.141435006
3	0.188628996
4	0.145149
5	0.197650994
6	0.202372005
7	0.394096001
8	0.42397099
9	1.623637002
10	1.512335002

De esta manera se calcularon los desempeños

```
// Primera parte de la funcion
def desempenoDeFunciones(tamanoCadena: Int): Vector[Double] = {
    println("Tamaño: " + tamanoCadena)

    // repido 100 veces una reconstruccion para una cadena
    // distinta y al final promedio los resultados, esto
    // se repite para todas las funciones
    val tiemposIngenua = (1 to 100).map(_ => 0.0).toArray
    for (i <- 0 until 100) {
        val time = withWarmer(new Warmer.Default) measure {
            val cadenaAleatoria = crearADN(tamanoCadena)
            val oraculo = oraculoFunc(cadenaAleatoria)
            reconstruirCadenaIngenuo(cadenaAleatoria.length,
                                    oraculo)
        }
        tiemposIngenua(i) = time.value
    }
    ...
    ...

    // resultado, promedios de todas las funciones
    Vector(tiemposIngenua.sum / 100, tiemposIngenuaPar.sum
           / 100, tiempoMejorado.sum / 100,
           tiempoMejoradoPar.sum / 100, tiempoTurbo.sum /
           100, tiempoTurboPar.sum / 100, tiempoTurboMejorada
           .sum / 100, tiempoTurboMejoradaPar.sum / 100,
           tiempoTurboAcelerada.sum / 100,
           tiempoTurboAceleradaPar.sum / 100)
}
```

B. Desempeño de las soluciones secuenciales

Tamaño	Ingenua	Mejorada	Turbo	Turbo Mejo- rada	Turbo Aceler- ada
2	0.15595	0.07877	0.07428	0.02492	0.05777
3	0.07714	0.04482	0.02592	0.03250	0.05864
4	0.22028	0.04200	0.04750	0.07834	0.03039
5	0.83654	0.09354	0.06095	0.05700	0.08043
6	3.78186	0.13751	0.05629	0.05711	0.11032
7	17.65640	0.47610	0.28831	0.24989	0.30307
8	80.99626	0.57314	0.32838	0.22031	0.34147
9	373.4264	2.01825	1.46584	1.19072	1.38513
10	1640.309	2.16946	1.31452	1.26137	1.33956

C. Desempeño de las soluciones paralelas

Tamaño	Ingenua Paralela	Mejorada Paralela	Turbo Paralela	Turbo Mejo- rada Paralela	Turbo Aceler- ada Paralela
2	0.33280	0.10646	0.10809	0.08803	0.17371
3	0.18621	0.11593	0.10569	0.10524	0.28257
4	0.35297	0.13164	0.08734	0.07610	0.13447
5	0.63948	0.18769	0.12617	0.14752	0.22089
6	2.34667	0.29456	0.10350	0.10099	0.19439
7	9.77154	0.54370	0.30917	0.26295	0.38850
8	44.77282	0.82634	0.42969	0.28062	0.45692
9	206.9668	1.96083	1.34753	1.26533	1.43333
10	948.4382	1.30567	0.62431	0.62999	0.74529

De esta manera se calcularon los desempeños para las algoritmos secuenciales y paralelos:

```
def desempenoDeFuncionesSecuenciales(tamanoCadena: Int):
    Vector[Double] = {
        println("Tamaño: " + tamanoCadena)

        //Se repite este mismo patron para cada algoritmo tanto
        //secuencial como paralelo
        val tiempoMejorado = (1 to 100).map(_ => 0.0).toArray
        for (i <- 0 until 100) {
            val cadenaAleatoria = crearADN(tamanoCadena)
            val oraculo = oraculoFunc(cadenaAleatoria)
            val time = withWarmer(new Warmer.Default) measure {
                reconstruirCadenaMejorado(cadenaAleatoria.length,
                                          oraculo)
            }
            tiempoMejorado(i) = time.value
        }
        Vector(tiempoIngenua.sum / 100, tiempoMejorado.sum /
              100, tiempoTurbo.sum / 100,
              tiempoTurboMejorada.sum / 100, tiempoTurboAcelerada.
              sum / 100)
    }
```

III. COMPARACIÓN DE LAS DISTINTAS SOLUCIONES Y SUS PARALELAS

A. Comparación de la solución ingenua y la ingenua Paralela

Tamaño	Ingenua	Ingenua Paralela	Aceleración
2	0.5254	0.9547	0.5503
3	0.2235	1.123	0.1990
4	0.2843	3.2886	0.0865
5	2.1254	1.4338	1.4824
6	1.8316	2.0491	0.8939
7	8.1376	6.0126	1.3534
8	50.5376	24.8344	2.0350
9	221.771	147.2517	1.5061
10	998.7833	599.0677	1.6672
11	5651.5319	3632.5512	1.5558
12	42989.9452	33009.7467	1.3023

B. Comparación de la solución Mejorada y la Mejorada Paralela

Tamaño	Mejorada	Mejorada Paralela	Aceleración
2	0.1611	0.2378	0.6775
3	0.2323	0.764	0.3041
4	0.1069	0.3939	0.2714
5	0.4804	0.4968	0.9670
6	0.1839	0.3513	0.5235
7	0.9965	1.0721	0.9295
8	0.6191	0.9048	0.6842
9	1.1074	1.1217	0.9873
10	1.5747	1.588	0.9916
11	4.9572	2.9537	1.6783
12	8.8451	3.5376	2.5003
13	36.2634	11.9666	3.0304
14	17.0906	12.7356	1.3420
15	86.8269	53.1912	1.6324

C. Comparación de la solución Turbo con la Turbo Paralela

Tamaño	Turbo	Turbo Paralela	Aceleración
2	0.1796	0.2413	0.7443
3	0.1242	0.2947	0.4214
4	3.6878	0.1483	24.8672
5	0.2011	0.4194	0.4795
6	0.1364	0.2140	0.6374
7	0.6585	0.5886	1.1188
8	0.3279	0.5852	0.5603
9	0.9744	0.6955	1.4010
10	0.6055	0.6846	0.8845
11	2.9213	2.6010	1.1231
12	2.7400	2.6862	1.0200
13	11.5387	8.3907	1.3752
14	11.1461	6.9173	1.6113
15	65.6961	35.8842	1.8308

D. Comparación de la solución Turbo Mejorada con la Turbo Mejorada Paralela

Tamaño	Turbo Mejorada	Turbo Mejorada Paralela	Aceleración
2	0.1777	0.3736	0.4756
3	0.1099	0.3417	0.3216
4	0.1304	0.2494	0.5229
5	0.2226	0.3977	0.5597
6	0.1266	0.2803	0.4517
7	0.5331	0.5740	0.9287
8	0.4141	0.4059	1.0202
9	1.4004	0.7065	1.9822
10	0.8512	0.9600	0.8867
11	3.2410	2.6681	1.2147
12	2.8437	2.0619	1.3792
13	13.4945	8.8758	1.5204
14	11.2194	7.3774	1.5208
15	66.7083	37.8467	1.7626

E. Comparación de la solución Turbo Acelerada con la Turbo Acelerada Paralela

Tamaño	Turbo Acelerada	Turbo Acelerada Paralela	Aceleración
2	0.2388	0.4078	0.5856
3	0.2308	0.6855	0.3367
4	0.9576	0.4174	2.2942
5	0.3942	0.4943	0.7975
6	0.5213	2.2681	0.2298
7	0.8283	0.7241	1.1439
8	0.6568	0.8154	0.8055
9	0.9013	0.9068	0.9939
10	1.2216	1.0947	1.1159
11	3.9154	2.9863	1.3111
12	5.7006	2.3085	2.4694
13	16.5676	8.7531	1.8928
14	13.3986	9.4514	1.4176
15	78.0346	44.7768	1.7427

De esta manera se calcularon las comparaciones

```
def compararAlgoritmos(Funcion1: (Int, Oraculo) => Seq[Char],
    Funcion2: (Int, Oraculo) => Seq[Char]) (n: Int, oraculo:
    Oraculo): (Double, Double, Double) = {
    val timeF1 = withWarmer(new Warmer.Default) measure {
        Funcion1(n, oraculo)
    }
    val timeF2 = withWarmer(new Warmer.Default) measure {
        Funcion2(n, oraculo)
    }
    val promedio = timeF1.value / timeF2.value
    (timeF1.value, timeF2.value, promedio)
}
```

IV. ANÁLISIS COMPARATIVO DE LAS DIFERENTES SOLUCIONES Y CONCLUSIONES

Análisis comparativo de la solución ingenua secuencial VS su versión paralela

Al analizar la tabla de desempeño, se puede notar que a partir de un tamaño de 5, la versión paralela comienza a superar consistentemente a la versión normal en términos de tiempos. Sin embargo, en tamaños menores a 5, la versión normal resulta más eficiente. Este fenómeno probablemente se deba al costo asociado con la paralelización de las 4 tareas, ya que en tamaños pequeños, la paralelización tiene un efecto contrario, aumentando los tiempos.

Al observar la tabla de comparación, se evidencia el mismo comportamiento: la solución paralela muestra una aceleración con respecto a la solución secuencial, más notoria a partir de tamaños 5-6, y desde ese punto en adelante, siempre presenta una ventaja sobre la versión secuencial.

Análisis comparativo de la solución mejorada secuencial VS su versión paralela

El comportamiento de la paralelización aquí resulta interesante, ya que al paralelizar de la forma en que lo hicimos, no se observa una mejora significativa hasta tamaños a partir de 9-10 en adelante. Por lo tanto, se puede concluir que vale la pena paralelizar solo en casos en los que los

tamaños son mayores a 9; en tamaños menores, la versión secuencial funciona mejor. Esto puede deberse a que la implementación de la paralelización, dividiendo en dos partes la tarea del bucle for, funciona mejor en tamaños mucho más grandes, donde las dos partes resultan más equilibradas y aprovechan mejor los recursos paralelos disponibles. Para tamaños de datos más pequeños, la versión secuencial del algoritmo puede superar en rendimiento a la paralelización debido a la sobrecarga asociada con la gestión de hilos y la división de tareas en conjuntos de datos más pequeños.

Análisis comparativo de la solución turbo secuencial VS su versión paralela

Al observar la tabla de comparaciones, notamos que para ciertos tamaños menores a 9-10, la paralelización tiene efectos positivos. Sin embargo, al repetir varias veces las comparaciones, nos dimos cuenta de que esto resulta ser un poco aleatorio; a veces mejora, a veces no mejora, y en ocasiones es en tamaño 4, otras veces en otros tamaños. Esta variabilidad sugiere cierta aleatoriedad. Para tamaños mayores a 10, la aceleración se estabiliza, dando como resultado que la versión paralela funcione mejor que la secuencial.

Análisis comparativo de la solución turbo mejorada secuencial VS su versión paralela

En esta comparación, observamos que ocurre algo similar a la comparación de la solución mejorada. En este caso, a partir de tamaños 8-9, la paralelización comienza a mejorar los tiempos.

Las tres comparaciones anteriores (Mejorada, Turbo y Turbo Mejorada) son muy similares, ya que se implementó prácticamente la misma paralelización en las tres.

Análisis comparativo de la solución turbo acelerada secuencial VS su versión paralela

Se observa un patrón evidente en esta comparación, ya que desde el tamaño 2 hasta el tamaño 9, siempre se muestra una superioridad de la versión secuencial sobre la paralela. Sin embargo, al llegar al tamaño 10, comienza a notarse una diferencia cada vez más favorable en la versión paralela. Esto se debe principalmente al enfoque de árbol del algoritmo, donde al tener varios procesos independientes, solo se aprecia una mejora en la paralelización cuando el volumen de datos es lo suficientemente extenso. De lo contrario, en lugar de mejorar en el tiempo de ejecución, empeorará, como se pudo apreciar.

Análisis comparativo de los algoritmos secuenciales

Al observar los resultados, notamos que, en general, la versión Turbo Mejorada tiene los tiempos más bajos en comparación con las otras versiones, indicando de esta manera que la Turbo Mejorada es la mejor opción para realizar la reconstrucción de cadenas.

Análisis comparativo de los algoritmos paralelos

Teniendo en cuenta los resultados obtenidos, notamos que se repitió el mismo patrón: en general, la versión Turbo Mejorada Paralela es la más eficiente en comparación con las otras versiones. Además, cabe destacar que cada versión mejoró su tiempo de ejecución.

¿Las paralelizaciones sirvieron?

En general, se puede decir que las paralelizaciones lograron mejorar el tiempo de ejecución de las soluciones. Sin embargo, es importante tener en cuenta que esta mejora solo se empieza a notar a partir de una cadena de tamaño 10, ya que a partir de ese tamaño la cantidad de datos es lo suficientemente extensa para apreciar una gran diferencia en el tiempo de ejecución.

V. PRUEBAS DE SOFTWARE

[Link del código Fuente](#)

Las pruebas de software que realizamos fueron las siguientes:

- A. Comprobación que los algoritmos reconstruyeran las cadenas de manera correcta*
- B. Comparación de cada algoritmo secuencial con su versión paralela*
- C. Evaluación de desempeño de todos los algoritmos*
- D. Evaluación de desempeño de todos los algoritmos secuenciales*
- E. Evaluación de desempeño de todos los algoritmos paralelos*