Informe Proyecto final

Juan Miguel Posso Alvarado - 2259610
Esteban Revelo Salazar - 2067507

1.Estructuras de Datos

1.1 Implementación soluciones secuenciales

Para la implementación de los algoritmos definimos a alfabeto como una secuencia de caracteres “a, c, g, t” que serán los caracteres de los que pueden estar formado las cadenas
El oráculo es una parte muy importante en este programa, ya que el me decidirá si una subcadena enviada pertenece a la cadena principal, en nuestro proyecto es un object con una función que recibe una cadena y retorna un Oráculo, que es una secuencia de caracteres que me devolverá un valor booleano, la forma en que trabaja es haciendo uso de los métodos .mkString y .contains, que me convierten la cadena de caracteres en un String para después verificar que esta subcadena está contenida en la cadena enviada.
Antes de empezar la implementación de las soluciones creamos la función generarCadenas, que nos permite construir todas las posibles combinaciones dado un tamaño y el alfabeto definido anteriormente. Esta función hace uso de (.flatMap y .map) para trabajar por cada carácter del alfabeto recorrerlo e ir restando el tamaño para completar el tamaño original de la cadena, al final simplemente combinamos estos resultados nos dan una secuencia de Strings que contiene todas las posibles combinaciones.

- Solución Ingenua:
  reconstruirCadenaIngenuo recibe un entero que es el tamaño de la cadena y el oráculo, y me devolverá una secuencia de caracteres, hace uso de generarCombinaciones y hacemos uso del método (.find) encontrando que si la cadena encontrada en esas combinaciones da como resultado verdadero en el oráculo obtenemos ese String y lo convertimos en una secuencia de caracteres

- Solución Mejorado:
  La lógica de reconstruirCadenaMejorado se basa en tener 2 subcadenas de una subcadena de la principal, con esto tendremos que preguntarle menos verificaciones al oráculo y poder descubrir cual es la cadena que estamos buscando. Ya que vamos a tener dos subcadenas vamos a dividir el tamaño de la cadena en dos, entonces debemos hacer una implementación para los casos que sean par e impar, después de esto, generar las combinaciones a partir de la subcadena ya encontrada, al final juntamos las dos subcadenas y verificamos una vez más con el oráculo que coincide con la cadena a buscar.

- Turbo Solución:
  La lógica principal en la función reconstruirCadenaTurbo difiere dependiendo de si n es par o impar. Si n es par, se generan subcadenas buenas con la función reconstruirCadenaTurboAux usando n/2. Luego, se concatenan estas subcadenas de manera que, si la concatenación de dos subcadenas es aceptada por el oráculo y su longitud es igual a n, se añade a posiblesCadenas.
  En el caso de n impar, se generan dos conjuntos de subcadenas buenas usando reconstruirCadenaTurboAux con n/2 y n-(n/2) respectivamente. Después, se combinan estas subcadenas y se filtran de manera similar a la lógica utilizada cuando n es par.

- Turbo mejorada:
  Al igual que la versión anterior, esta función reconstruirCadenaTurboMejorada también se encarga de reconstruir una secuencia de caracteres con base en ciertas condiciones y un oráculo proporcionado.
  La lógica en la función reconstruir CadenaTurboMejoradaAux esta ajustada para generar cadenas de manera más eficiente. Se compara baseInicial con n para determinar si se ha alcanzado una condición de salida y se devuelve la secuencia acumulada hasta el momento.

1.2 Implementación soluciones paralelas

En las funciones paralelas se hizo uso de las colecciones “implementation group: 'org.scala-lang.modules', name: 'scala-parallel-collections_2.13', version:'1.0.4' ”

- Solución Ingenua paralela:
  En la reconstruircadenaingenuaPar tenemos dos argumentos que se envían, el tamaño de la cadena y el oráculo, vamos a utilizar casi la misma base de su función secuencial, pero utilizaremos los métodos (.par y .filter) para crear una filtración paralela en las combinaciones encontradas por la función generar cadenas, mejorando así un poco el tiempo de respuesta, pero siendo un algoritmo bastante ineficiente

- Solución Mejorada paralela:
  Para la solucion de la funcion reconstruirCadenaMejoradoPar utilizaremos los metodos task y parallel para su implementacion, creamos la tarea paralela en caso de que la cadena sea de tamaño par, y a la hora de juntar las dos subcadenas las unimos con ayuda de .join() volviendolos a el mismo hilo lanzado anteriormente. Para tamaño impar vamos a lanzar un parallel con las subcadenas que vamos a encontrar, logrando que estas se realicen en simultaneo ayudandonos a la eficiencia de tiempo del programa. Esta funcion tiene la misma implementación de su forma secuencial, solo logra que las dos subcadenas se hagan en simultaneo.

- Turbo Solución paralela:
  La función toma tres parámetros: umbral, que parece ser un límite (aunque no se utiliza en el código); n, que representa la longitud de las cadenas a generar; y o, un oráculo que devuelve un booleano indicando si una cadena cumple ciertas condiciones.
  Al igual que en la versión secuencial evaluamos si n es un número par, la función ejecuta reconstruirCadenaTurboParAux en paralelo con n / 2 como parámetro utilizando task. Posteriormente, espera a que esta tarea finalice y utiliza sus resultados (subCadenasCorrectas) para generar las posibles combinaciones de cadenas. Si n es un número impar, la función ejecuta dos llamadas a reconstruirCadenaTurboParAux en paralelo: una con n / 2 y otra con n - (n / 2) mediante parallel. Espera simultáneamente ambos resultados en paralelo.
  En ambos casos, se utilizan las cadenas resultantes para generar posibles combinaciones y se filtran aquellas que cumplan con las condiciones del oráculo o. Estas combinaciones válidas se concatenan y se retorna una cadena que representa las posibles combinaciones resultantes.

- Turbo mejorada paralela:
  En esta función se reciben los mismos 3 parámetros umbral (que no se usa), n que es la longitud de la cadena y o que es el oráculo.
  En esta función como en la anterior verificamos que el tamaño de la cadena sea par o impar, si n es un número par, la función ejecuta reconstruirCadenaTurboMejoradaParAux en paralelo con n / 2 como parámetro utilizando task. Luego, espera a que esta tarea finalice y utiliza los resultados obtenidos (subCadenasCorrectas) para generar posibles combinaciones de cadenas. Posteriormente, filtra aquellas que cumplan con las condiciones del oráculo.
  En caso de que n sea impar, se ejecutan dos llamadas a reconstruirCadenaTurboMejoradaParAux en paralelo: una con n / 2 y otra con n - (n / 2) mediante parallel. Se esperan ambos conjuntos de resultados en paralelo.
  En ambas situaciones, las combinaciones válidas obtenidas se utilizan para generar posibles combinaciones de cadenas, filtrando aquellas que cumplan con las condiciones del oráculo o. Finalmente, se retorna una cadena que representa las posibles combinaciones resultantes, concatenadas bajo las condiciones especificadas.

2. Comparación de algoritmos.

2.1 Evaluación comparativa

| Tam | IngenuaSec | IngenuaPar | Aceleracion |
|---|---|---|---|
| 4 | 0,1885 | 0,7314 | 0,2577249 |
| 5 | 0,5543 | 0,6855 | 0,8086069 |
| 7 | 3,1399 | 4,5464 | 0,6906343 |
| 8 | 9,9374 | 10,875 | 0,9138091 |
| 10 | 266,78 | 577,84 | 0,4616932 |
| 16 | OutOfMemory | OutOfMemory | OutOfMemor |
| 20 | OutOfMemory | OutOfMemory | OutOfMemor |

| Tam | MejoradaSec | MejoradaPar | Aceleracion2 |
|---|---|---|---|
| 4 | 0,2116 | 0,2147 | 0,9855612 |
| 5 | 0,3111 | 0,2202 | 1,4128065 |
| 7 | 0,6258 | 0,5100 | 1,2270588 |
| 8 | 0,8514 | 0,5833 | 1,4596263 |
| 10 | 3,8811 | 1,0531 | 3,6854050 |
| 16 | 26,396 | 26,193 | 1,0077463 |
| 20 | 708,31 | 583,27 | 1,2143774 |

| Tam | TurboSec | TurboPar | Aceleracion3 |
|---|---|---|---|
| 4 | 0,1609 | 0,3321 | 0,484493 |
| 5 | 0,2530 | 0,3738 | 0,676833 |
| 7 | 0,4404 | 0,3006 | 1,465070 |
| 8 | 0,5506 | 0,3275 | 1,681221 |
| 10 | 1,5775 | 0,4389 | 3,594213 |
| 16 | 21,461 | 20,588 | 1,042419 |
| 20 | 632,50 | 429,56 | 1,472440 |

| Tam | TurboMejorad | TurboMejoradaP | Aceleracion4 |
|---|---|---|---|
| 4 | 0,1594 | 0,1608 | 0,99129 |
| 5 | 0,2231 | 0,1702 | 1,31081 |
| 7 | 0,4517 | 0,3442 | 1,31232 |
| 8 | 0,3739 | 0,2784 | 1,34303 |
| 10 | 1,1065 | 0,6938 | 1,59484 |
| 16 | 21,169 | 20,110 | 1,05268 |
| 20 | 533,87 | 524,39 | 1,01809 |

Como podemos observar en los resultados puestos a continuación observamos que la implementación más rápida fue la versión paralela esto queda demostrado gracias a las

aceleraciones, Además de que en el caso de la solución ingenua que al ser tan poco efectiva necesitaba un alto costo de memoria por lo que era muy difícil que la solución ingenua fuera recorriendo las cadenas una por una y de un tamaño tan grande como lo fue $4^{16}$ o $4^{20}$. Algo que podemos notar en las tablas es que la aceleración en las soluciones ingenuas es menos ya que la implementación paralela tardaba mas en tiempo de ejecución esto se puede deber a que es más rápido contar uno por uno en un solo hilo que poner a muchos hilos a contar y esperar a que los otros terminen, en cuanto a las otras soluciones demostraron tener una buena aceleración ya que sus soluciones paralelas eran mucho mejor que las versiones secuenciales en términos de tiempo de ejecución.

## 2.2 Desempeño de las funciones secuenciales y paralelas

Funciones Secuenciales:

| Tam | IngenuaSec | MejoradaSec | TurboSec | TurboMejoradaSec |
|---|---|---|---|---|
| 4 | 0,1885 | 0,2116 | 0,1609 | 0,1594 |
| 5 | 0,5543 | 0,3111 | 0,2530 | 0,2231 |
| 7 | 3,1399 | 0,6258 | 0,4404 | 0,4517 |
| 8 | 9,9374 | 0,8514 | 0,5506 | 0,3739 |
| 10 | 266,78 | 3,8811 | 29,837 | 1,1065 |
| 16 | OutOfMemory | 26,396 | 21,461 | 21,169 |
| 20 | OutOfMemory | 708,31 | 632,50 | 533,87 |

Funciones Paralelas:

| Tam | IngenuaPar | MejoradaPar | TurboPar | TurboMejoradaPar |
|---|---|---|---|---|
| 4 | 0,7314 | 0,2147 | 0,3321 | 0,1608 |
| 5 | 0,6855 | 0,2202 | 0,3738 | 0,1702 |
| 7 | 4,5464 | 0,5100 | 0,3006 | 0,3442 |
| 8 | 10,875 | 0,5833 | 0,3275 | 0,2784 |
| 10 | 577,84 | 1,0531 | 0,6265 | 0,6938 |
| 16 | OutOfMemory | 26,193 | 20,588 | 20,110 |
| 20 | OutOfMemory | 583,27 | 429,56 | 524,39 |

En cada función se puede evidenciar una mejoría en cuestión de tiempo, en estos casos las soluciones que seguían a las primeras sacaban mejores tiempos siendo la turboMejorada la que más rápido iba en nuestro caso, y entre la secuencial y la paralela es claro que las soluciones paralelas al dividir el trabajo lograban mejores resultados en ejecución.