INFORME PFC || TRABAJO FINAL

Tina María Torres 2259729, Juan José Gallego 2259433, Marlon Astudillo 202259462

Las estructuras de datos utilizadas

Soluciones secuenciales

★ prc_ingenuo

Esta es una función para generar una lista de combinaciones recurriendo a cada carácter del alfabeto, llamar recursivamente a **cadenas_candidatas** disminuyendo en 1 la longitud, genera combinaciones agregando el carácter actual a cada una de las combinaciones obtenidas en el llamado recursivo.

1. **Recursión:** Para generar todas las combinaciones posibles de caracteres del alfabeto para una longitud dada. La función `**cadenas_candidatas**` se llama a sí misma de forma recursiva con una longitud decreciente hasta llegar al caso base.
2. **Reconocimiento de Patrones(implícito):** Se utiliza el patrón común de división entre casos base y casos recursivos en la función `**cadenas_candidatas**`.
3. **Mecanismos de Encapsulación:** Al definir `**cadenas_candidatas**` dentro de `**prc_ingenuo**`.
4. **Funciones de Alto Orden:** `**flatMap**` y `**map**` sobre colecciones. `**flatMap**` aplica una función a cada elemento de la colección **'alfabeto'** y luego aplana los resultados en una sola colección. `**map**` aplica una función a cada elemento de la colección **'alfabeto.flatmap(...)'** y devuelve una nueva colección con los resultados.
5. **Colecciones:** En particular, el código trabaja con secuencias de caracteres (`Seq[Char]`) para representar el alfabeto y las cadenas generadas.

★ prc_mejorado

Esta es una función para generar una nueva lista de combinaciones tomando cada combinación actual, agregándole cada letra del alfabeto, filtrando según la función oráculo, con **subcaden_candidatas** se llama recursivamente aumentando la longitud en 1 y usando las combinaciones filtradas con el oráculo.

1. **Recursión:** La función `**subcaden_candidatas**` se llama a sí misma de forma recursiva para generar las combinaciones de cadenas.
2. **Mecanismos de Encapsulación:** Utiliza la encapsulación al definir `**subcaden_candidatas**` dentro de `**prc_mejorado**`
3. **Funciones de Alto Orden:** `**flatMap**`, `**map**` y `**filter**` son funciones de alto orden que se aplican sobre colecciones `**Seq**` para manipular los datos de forma más funcional.
4. **Iteradores y Colecciones:** Utiliza métodos sobre `**Seq**` como `**flatMap**`, `**map**` y `**filter**` para generar y manipular las combinaciones de cadenas. Además, hace uso de la recursión para iterar y construir las combinaciones requeridas.

★ prc_turbo

Esta es una función para generar combinaciones de cadenas concatenando cada combinación consigo misma, filtrando según una condición proporcionada, y duplicando la cantidad de combinaciones en cada iteración al multiplicar **k** por **2** en cada llamada recursiva.

1. **Recursión:** La función `**subcaden_candidatas**` se llama a sí misma de forma recursiva para generar cadenas concatenando todas las combinaciones de la secuencia de tamaño anterior **k/2**.
2. **Mecanismos de Encapsulación:** Al definir la función interna `**subcaden_candidatas**` dentro de la función externa `**prc_turbo**`.
3. **Funciones de Alto Orden:** `**flatMap**` y `**filter**` sobre `**Seq**`.
4. **Iteradores y Colecciones:** Hace uso de métodos sobre colecciones como `**flatMap**` y `**filter**` para generar, transformar y filtrar las combinaciones de cadenas.

★ prc_turboMejorado

Busca generar secuencias de caracteres concatenando cada combinación consigo misma, la función filtra las subcadenas en **cadenaActual** de modo que todas las subcadenas de longitud media estén presentes en **cadenaAnterior**. Esto ayuda a evitar las cadenas anteriormente filtradas que no deberían de ir. Tiene funciones de alto orden, iteradores, colecciones, mecanismos de encapsulación y manejo de recursión como **Prc_Turbo**.

★ ArbolDeSufijos

Esta es una función para recibir a **secuenciaDeCadenas** que se ha filtrado con anterioridad en la función **prc_TurboMejorado**, esta secuencia se convertirá en un árbol, con el método adicionar mete una a una las subsecuencias de la lista recibida en un árbol vacío que se creó anteriormente.

1. **Recursión:** La función `**funcion_aux**` se utiliza de manera recursiva para construir un **Trie** a partir de una secuencia de secuencias de caracteres, `**secuenciaDeCadenas**`. Utiliza la recursión para procesar cada elemento de `**secuenciaDeCadenas**` y adicionarlo al **Trie** `**t**`.
2. **Reconocimiento de patrones:** El código hace uso de patrones en la función `**arbolSufijos**`. Utiliza la condición `**(secuenciaDeCadenas.isEmpty)**` para

comprobar si la secuencia de secuencias de caracteres está vacía, lo que marca el punto de parada para la recursión.

3. **Mecanismos de encapsulación:** La función interna `**funcion_aux**` encapsula la lógica de construcción del **Trie** a partir de las secuencias de caracteres. Toma como parámetros la secuencia de secuencias de caracteres y el **Trie** actual y, mediante la recursión, va construyendo el **Trie** resultante.
4. **Iteradores y colecciones:** Se utiliza la secuencia `**secuenciaDeCadenas**` que es una colección de secuencias de caracteres. La función `**funcion_aux**` utiliza `**secuenciaDeCadenas.tail**` para procesar recursivamente las secuencias restantes después de la cabeza de `**secuenciaDeCadenas**`.

➢ **Raíz**

Esta función auxiliar para **ArbolDeSufijos**, toma un nodo de tipo **Trie** y devuelve el carácter asociado a ese nodo. Si es un nodo interno **(Nodo)**, devuelve el carácter **c** asociado. Si es **'Hoja'**, devuelve el carácter **c,** el carácter **c** es el valor de una hoja o un nodo, este puede tener hijos.

➢ **Cabezas**

Esta función auxiliar para **ArbolDeSufijos**, devuelve una secuencia de caracteres correspondientes a las **"cabezas"** o caracteres iniciales de las ramas del **Trie**. Si hay un nodo, mapea sobre la lista de subnodos **(lT)** y aplica la función **raíz** a cada uno, devolviendo la lista de caracteres iniciales de esos nodos. Si el nodo es una **Hoja**, devuelve una secuencia que contiene solo ese carácter.

★ **Pertenece**

Esta función usa una restricción donde si la secuencia tiene un carácter, una cola y el nodo, verifica recursivamente si la cola pertenece a los hijos del nodo actual. Busca el siguiente nodo cuya raíz sea igual al primer carácter de la cola. Si no lo encuentra, asume una **Hoja** vacía y no marcada, pero, si la secuencia está vacía **(Nil)**, verifica si el nodo actual es un **Nodo** o una **Hoja** y devuelve el valor booleano asociado a su estado marcado. Si no coincide con ninguno de estos patrones, devuelve false.

1. **Recursión:** Para verificar la presencia de una secuencia de caracteres en el Trie. Se llama a sí misma de manera recursiva para explorar los nodos del **Trie** en busca de la coincidencia de caracteres.
2. **Reconocimiento de patrones:** Expresiones `**match**`. Todas las funciones proporcionadas (`**raiz**`, `**cabezas**`, y `**pertenece**`) utilizan coincidencia de patrones para hacer diferentes operaciones basadas en la estructura de los nodos del **Trie** (`**Nodo**` o `**Hoja**`).
3. **Mecanismos de encapsulación:** Los nodos (`**Nodo**` y `**Hoja**`) encapsulan la información sobre los caracteres y su relación en la estructura del **Trie**. Las funciones actúan sobre estos nodos, accediendo a sus datos internos a través de patrones de coincidencia.
4. **Funciones de alto orden:** `**map**` para aplicar la función `**raiz**` a cada elemento de la lista de subnodos en el caso de que el nodo sea un `**Nodo**`**.**

★ Adicionar

Esta es una función para agregar una secuencia de caracteres **`s`** a un **Trie `t`**. Hace uso de una función interna llamada **`agregarRama`**, la cual realiza la mayor parte del trabajo, **`agregarRama`** toma dos parámetros: el **Trie** actual **(`arbolActual`)** y la secuencia de caracteres restantes **(`remaining`)**, utiliza patrones de coincidencia para manejar diferentes situaciones según la estructura del **Trie** y la secuencia de caracteres, los casos principales en el **`match`** son los siguientes:

- ➢ Cuando el **`arbolActual`** es un **`Nodo`** y aún quedan caracteres en la secuencia **(`remaining`),** y la cabeza de la secuencia coincide con algún hijo del nodo actual, se realiza una actualización recursiva de los hijos que coinciden con el camino proporcionado por la secuencia **`s`**.
- ➢ Cuando el **`arbolActual`** es una **`Hoja`** y aún quedan caracteres en la secuencia, se convierte la **`Hoja`** en un nuevo **`Nodo`** con la secuencia restante como un nuevo hijo.
- ➢ Cuando el **`arbolActual`** es un **`Nodo`,** pero el camino se detiene (la secuencia está vacía), se agrega un nuevo nodo a la lista de hijos con el carácter restante de la secuencia **`s`**.
- ➢ Cuando el **`arbolActual`** es un **`Nodo`** sin marca y la secuencia está vacía, se modifica el valor de **`marcada`** a **`true`** para indicar que la cadena representada por esa rama está presente en el **Trie**.
- ➢ Otros casos no coincidentes simplemente devuelven el **`arbolActual`** sin cambios.

La función **`adicionar`** invoca **`agregarRama`** con el **Trie** original **`t`** y la secuencia de caracteres **`s`**, lo que efectivamente agrega la secuencia **`s`** al **Trie**, extiende el **Trie** existente con una nueva secuencia de caracteres, manteniendo la estructura del **Trie** y marcando la cadena representada por esa secuencia como presente en el **Trie**.

1. **Recursión:** Para recorrer y actualizar el **Trie**. Se llama a sí misma para seguir navegando a través de los nodos del **Trie** hasta encontrar la posición correcta para agregar la nueva secuencia de caracteres.
2. **Reconocimiento de patrones:** mediante el uso de coincidencias de patrones **(`match`)** en el código. Los patrones son verificados para diferentes casos de los nodos del **Trie (`Nodo`** y **`Hoja`)** y para las condiciones de la secuencia de caracteres.
3. **Mecanismos de encapsulación:** La lógica principal está encapsulada en la función **`agregarRama`**, la cual maneja la lógica detallada de cómo agregar una secuencia al **Trie**.
4. **Funciones de alto orden:** La función **`map`** se usa dentro de la lógica de **`agregarRama`**. Se utiliza para modificar la lista de hijos de un nodo **(`hijosActualizados = hijos.map {...}`)**, aplicando transformaciones a cada hijo del nodo.
5. **Iteradores y colecciones:** Se utilizan colecciones **(`Seq`, `List`)** para manejar y procesar los nodos y los caracteres en la secuencia que se agrega al **Trie**. Además, se

utiliza el operador `:+` para agregar elementos a una lista **(`hijos :+ adicionar(tail, Nodo(head, false, List()))`)**.

Estos elementos se utilizan para manejar la estructura recursiva del **Trie**, reconocer diferentes patrones en los nodos y en la secuencia de caracteres, encapsular la lógica de actualización del **Trie**, emplear funciones de alto orden para transformar la estructura de datos y trabajar con iteradores, colecciones y operaciones de colección para agregar y manipular elementos en el **Trie**.

★ prc_turboacelerada (trie)

esta función realiza la misma gestión que la versión anterior ("**prc_turboMejorado** ") pero en la función filtrar primero se crea un árbol con la función **ArbolDeSufijos** que contendrá la lista de subcadenas de tamaño **k/2** para luego preguntar si cada una de las porciones de las cadenas de **cadenasActual** de tamaño **k/2** pertenecen al árbol anteriormente creado, esta función al estar en un **.filter** descartó las subcadenas de tamaño **k** que contengan subcadenas de tamaño **k/2** que no estén en la lista anterior, evitando así generar combinaciones de tamaño **k/2** que ya habíamos descartado.
Es importante resaltar el uso de **.slice**. En primer lugar, tenemos el uso de una variable **i** que va iterando en un rango que irá aumentando gracias a la función de alto orden **"forall"** para permitirnos pasarle a **.slice** los índices de donde debe empezar a tomar la porción y donde debe terminar **(i+k)** en la secuencia de tamaño **k**, permitiéndonos así, pasarle la porción a evaluar a la función perteneciente.

Soluciones paralelas

★ length

Todos los métodos paralelos usan este método no funcional.

★ Prc_ingenuoPar

Esta función se separa en dos tareas (split-task) para generar las cadenas candidatas, en **cadenas_candidatas**, a partir de ello hace una sola colección concatenando las dos mitades y luego paralelamente se generan otras dos tareas que van a tener la mitad de esas combinaciones, estas van a preguntar independientemente la existencia de la secuencia al oráculo, en cuanto una de ellas lo encuentre, retornará el resultado.

★ prc_mejoradoPar

Se separan dos tareas **(split-task)** y cada una hace la mitad de las candidatas de tamaño **k** y en el llamado recursivo, en **subcadena_candidatas**, se manda la concatenación de ambas.

★ prc_turboPar

Dentro de la función **subcaden_candidatas** se parte en dos la lista de subcadenas de tamaño **k** con el método **splitAt** y se le asigna una mitad a cada una de las dos tareas paralelas, las cuales tomarán correspondientemente su mitad, esta se concatena con todas y cada una de las secuencias de las subcadenas **k** (concatenar **sck/2** con **sck**).

★ prc_turboMejoradoPar

Crea dos tareas, las cuales hacen la concatenación de la mitad de las combinaciones posibles de las cadenas de tamaño **k/2** y llama la función filtrar que luego será filtrada también por el oráculo **".filter(o)"**

★ **prc_turboaceleradaPar**

se hacen dos tareas en paralelo las cuales cada una tendrá que concatenar la mitad de las cadenas de tamaño anterior, formando así las cadenas del tamaño **k** actual, luego se pasa dicha mitad a la función filtrar y luego se filtra con el oráculo las combinaciones que no están en la cadena a encontrar

Corrección de las funciones desarrolladas

Las funciones más relevantes

prc_mejorado

Este código mejora el enfoque inicial de la prc_ingenua de realizar una lista de todas las posibles combinaciones de tamaño n con el alfabeto y preguntar al oráculo cuál es, al filtrar las combinaciones (agregando una letra a la vez) directamente durante la generación, reduciendo la cantidad de combinaciones generadas y optimizando el proceso en función de la condición proporcionada por el oráculo.

prc_turbo

No hace cadenas agregando una letra a la vez, sino que concatena todas las subcadenas de la secuencia anterior **SCK-1** combinándola y agregando todas las nuevas cadenas de tamaño k a la subcadena **SCk**, luego de esto se filtra las cadenas que el oráculo dictamina que no están contenidas por la secuencia a encontrar.

prc_turboMejorado

Su mejora se implementa agregando una función filtrar que se va a llamar justo después de hacer todas las concatenaciones de **SCk-1**, consecuentemente, con la lista de cadenas de tamaño k, verificar que no se hayan generado combinaciones de tamaño **k/2** dentro de SCk que sean inválidas, cadenas que ya se hayan descartado en el llamado recursivo anterior por medio del oráculo, evitando así tener cadenas inválidas, optimizando la función.

prc_turboacelerada: este enfoque mejora la eficiencia de la función filtrar al guardar la secuencia generada de tamaño **k** en un árbol, para luego recorriendo el **trie "t"** buscar si está compuesto por solo subcadenas válidas de tamaño **k/2**, cuidando así que la concatenación hecha anteriormente no haya generado cadenas inválidas. Es muy importante resaltar que la eficiencia en la búsqueda del árbol es mucho más grande que su versión anterior, aunque debamos hacer una conversión antes de filtrar las cadenas.

Evaluaciones comparativas

secuencial | paralela

Solución ingenua

Tamaño de la Secuencia	Tiempo de la función -	Tiempo de la función -

	prc_ingenua	prc_ingenuaPar
4	2.481	2.0516
8	203.1188	133.5879

Solución mejorada

Tamaño de la Secuencia	Tiempo de la función-prc_mejorado	Tiempo de la función-prc_mejoradoPar
4	1.1017	0.8553
8	0.9001	1.2443
16	4.0325	3.5058
32	23.4096	14.063
64	358.8443	206.7327
128	4499.609	2450.117

Solución turbo

Tamaño de la Secuencia	Tiempo de la función prc_turbo	Tiempo de la función prc_turboPar
4	1.2728	0.577
8	1.2147	0.7375
16	3.5331	1.3693
32	28.3797	14.0553
64	800.2105	425.3863
128	8818.43	4779.4317
256	179149.1655	78571.368799

Solución turbo mejorada

Tamaño de la Secuencia	Tiempo de la función prc_turbomejorado	Tiempo de la función prc_turbomejoradopar

4	0.2445	0.1676
8	0.2217	0.2836
16	0.4758	0.4458
32	2.5286	1.3773
64	10.0981	5.7267
128	85.4297	46.5283
256	803.0549	447.2471
512	7942.7571	4927.7358

Solución turbo acelerada

Tamaño de la Secuencia	**Tiempo de la función prc_turboacelerada**	**Tiempo de la función prc_turboaceleradaPar**
4	0.2269	0.2145
8	0.3055	0.2472
16	0.3897	0.3018
32	1.6009	0.6052
64	4.896	3.1592
128	82.8252	25.0551
256	330.4075	197.4646
512	2917.4615	2047.4754

Conclusión…

Solución ingenua: en el caso de la solución ingenua podemos concluir dos cosas, primero que la versión paralela ganó siempre y el crecimiento de la velocidad fue abismal, haciendo imposible realizar pruebas por encima de tamaño 8

Solución mejorada: en esta ocasión ganó la paralela, llegando a tener una aceleración de 1.837, en promedio ganando en todas la versión paralela.

Solución turbo: en esta ocasión ganó la paralela, llegando a tener una aceleración de 2.281, ganando exactamente en todas la versión paralela

Solución turbomejorada: en promedio podemos concluir que ganó siempre la paralela, incluso en los tamaños pequeños, con una aceleración máxima de 1.835

Solución turboacelerada: desde los tamaños pequeños, la paralela demostró mayor aceleración, ganando siempre en retornar el valor esperado, llegando a tener aceleraciones de 3.305.

Conclusiones generales

En el tamaño de 256 tenemos a la versión paralela de la **turboacelerada** con una velocidad de 197.4646 ms con una aceleración con respecto a la **turbomejoradaPar** de 2.270, más, sin embargo, la versión secuencial de la turbo mejorada es la mejor opción en tamaños pequeños, con el mejor tiempo de 0.2217. Esto se esperaba debido a que la asignación y creación de un árbol en la última versión podría ser más lenta que la operación misma, haciendo que en tamaños muy pequeños en promedio, sea mejor la solución turbo mejorada.