Universidad del Valle

Facultad de ingeniería

Ingeniería en sistemas

Cristian David Pacheco Torres
2227437
Juan Sebastian Molina Cuellar
2224491

October 26, 2023

Taller 4: Colecciones y Expresiones For:
El problema de la subsecuencia incremental de longitud máxima

Contents

1 Vista general. Uso de colecciones y expresiones for.

El uso de colecciones y expresiones `for` en Scala representa una poderosa combinación que facilita la manipulación y transformación de datos. Las colecciones ofrecen una amplia gama de operaciones que permiten trabajar con conjuntos de datos de manera eficiente y expresiva. Por otro lado, las expresiones `for` proporcionan una sintaxis concisa para iterar y filtrar datos, haciendo que el código sea más legible y mantenible. Juntas, estas herramientas permiten a los desarrolladores escribir algoritmos complejos de manera más intuitiva, reduciendo la posibilidad de errores y mejorando la productividad.

2 Solución ingenua usando fuerza bruta

2.1 Generación de los índices asociados a todas las subsecuencias

2.1.1 Informe de uso de colecciones y expresiones for

```
def subindices(i: Int, n: Int): Set[Seq[Int]] = {
    val elements = (i until n).toSet
    (for {
        k <- 0 to elements.size
        combination <- elements.subsets(k)
    } yield combination.toSeq.sorted ).toSet
}
```

Listing 1: Código en Scala para la funcion subindices

Tabla $subindices(i : Int, n : nt)$		
Función	¿Se utilizó colecciones y expresiones for?	¿Razón?
$subindices(i : Int, n : Int)$...	Sí	**Colecciones en Scala:** • `Set` y `Seq` permiten representar y manipular conjuntos y secuencias de datos de manera eficiente. • `subsets(k)` es una función de las colecciones que facilita la generación de todas las combinaciones posibles de un conjunto. • `toSeq.sorted` convierte un conjunto en una secuencia ordenada, lo cual es útil para garantizar la consistencia en las combinaciones generadas. **Expresiones `for`:** • Facilitan la iteración sobre colecciones y la generación de nuevas colecciones. • Permiten combinar múltiples generadores y filtros en una sola expresión, simplificando el código y haciéndolo más legible.

Table 1: Tabla de uso de colecciones y expresiones for en la función $subindices(i : Int, n : Int)$

2.1.2 Informe de corrección

Argumentación sobre la corrección:

$$\text{Sea } i, n \in \mathbb{N}^+ \mid 0 \leq i \leq n;$$

$$\text{Sea } S = \{a_1, a_i, ..., a_n\} \text{ donde } a_{i-1} < a_i \text{ una secuencia denominada incremental;}$$

$$\text{Sea } P(S) \text{el conjunto potencia de } s_i, \text{es decir} s \in P(S) \text{ cuando } s_i \text{ esta contenido en} S$$

$$\text{Esto es } s_i \text{ es una subsecuencia incremental desde elemento } a_i \text{ de } S$$

$$\text{Sea } K \in \mathbb{N}^+\{0\}K \in [n; |S|]$$

$$P_k(S) \text{ representa los subconjuntos con tamaño } \in [0; k]$$

Estado inicial:

$$1)(0, n, \phi)$$

Estado:

$$2)(k, n, P_k(S))$$

$$\text{es final si } k == n + 1 \rightarrow s_f = (n + 1, n, P_n(S))$$

Invariante:

$$3)Inv(k, n, P_k(S)) \equiv k <= n + 1 \wedge |P_K(S) <= |2^k|$$

Transformar:

$$4)Tran(k, n, P_k(S)) = (k + 1, n, P_k(S) \cup P_{k+1}(S))$$

Demostracion:

1. $Inv(S_0)$ se cumple, pues $0 <= n + 1 \wedge P_0(S) = \{\phi\} \rightarrow |P_0(S)| = 1 <= 2^0 = 1$
2. $(s_k \neq s_f \wedge Inv(s_k)) \rightarrow Inv(tranformar(s_k))$

$$\equiv$$

$$k \neq n + 1 \wedge k <= k + 1 \wedge P_k(S) <= 2^k \rightarrow k + 1 <= n + 1 \wedge ...$$

$$...P_{k+1}(S) < 2^{k+1} \rightarrow P_{k+1}(S) = \bigcup_{0<i}^{k} P_k \cup P_{k+1}$$

Por tanto, $Inv(tranformar(s_k))$ se cumple.

3. $Inv(S_f) \rightarrow |P_n(S)| <= 2^n \equiv Inv(n + 1, n, P_n(S))$

$$\rightarrow n + 1 <= n + 1 \wedge |P_n(S)| <= 2^n \rightarrow P_n(S)$$

$$true \wedge true \rightarrow P_n(S)$$

4. En cada paso se aumenta k en una unidad hasta superar la condicion de n. por tanto cada vez estaria mas cerca de $n+1$ en cada iteracion se unen los subconjuntos tal que sus elementos son crecientes y acotados por una cantidad 2^k

Casos de prueba:

```
subindices(1, 5)
subindices(3, 6)
subindices(0, 4)
subindices(2, 7)
subindices(4, 8)
```

Listing 2: Casos de prueba para la función subindices

1. **Valor esperado**: HashSet(List(1), List(1, 2, 3), List(1, 3), List(3), List(), List(2, 3), List(1, 4), List(1, 3, 4), List(1, 2), List(2, 3, 4), List(3, 4), List(4), List(2), List(2, 4), List(1, 2, 3, 4), List(1, 2, 4))

2. **Valor esperado**: HashSet(List(4, 5), List(3), List(), List(3, 5), List(5), List(3, 4, 5), List(3, 4), List(4))

3. **Valor esperado**: HashSet(List(1, 2, 3), List(0, 1, 2, 3), List(0, 3), List(3), List(2, 3), List(0, 1), List(1, 2), List(0, 2), List(0), List(2), List(0, 1, 3), List(1), List(1, 3), List(0, 2, 3), List(0, 1, 2), List())

4. **Valor esperado**: HashSet(List(3, 5, 6), List(2, 3, 5, 6), List(5, 6), List(4, 5), List(), List(2, 3), List(2, 3, 5), List(3, 5), List(3, 6), List(3, 4), List(2, 3, 6), List(2), List(2, 4), List(4, 6), List(2, 4, 5), List(2, 3, 4, 5), List(2, 3, 4, 5, 6), List(3), List(3, 4, 5, 6), List(2, 3, 4, 6), List(2, 5, 6), List(4, 5, 6), List(5), List(6), List(2, 4, 6), List(3, 4, 6), List(3, 4, 5), List(2, 3, 4), List(2, 5), List(2, 6), List(2, 4, 5, 6), List(4))

5. **Valor esperado**: HashSet(List(4, 5, 7), List(4, 6, 7), List(5, 6), List(4, 5), List(), List(5, 7), List(4, 5, 6), List(6, 7), List(4), List(5, 6, 7), List(4, 7), List(4, 5, 6, 7), List(4, 6), List(7), List(5), List(6))

2.2 Generación de todas las subsecuencias de una secuencia

2.2.1 Informe de uso de colecciones y expresiones for

```scala
def subSecuenciaAsoc(s:Secuencia, inds:Seq[Int]): Subsecuencia
   =
    (for i <- 0 to inds.size-1 yield s(inds(i))).toList
```

Listing 3: Código en Scala para la funcion subSecuenciaAsoc

```scala
def subSecuenciasDe(s:Secuencia): Set[Subsecuencia] ={
  val combinationIndex = subindices(0, s.size)
  for index <- combinationIndex yield subSecuenciaAsoc(s, index
     )
}
```

Listing 4: Código en Scala para la funcion subSecuenciasDe

Tabla *subSecuenciaAsoc* y *subSecuenciasDe*		
Función	¿Se utilizó colecciones y expresiones for?	¿Razón?
subSecuenciaAsoc	Sí	**Colecciones en Scala:** • `Seq` representa una secuencia de elementos en Scala. En esta función, se utiliza para representar una secuencia de índices. • `toList` convierte una colección en una lista. Esto puede ser útil para garantizar un tipo de salida específico o para realizar operaciones específicas de las listas. **Expresiones `for`:** • La expresión `for` se utiliza para iterar sobre la secuencia de índices y extraer los elementos correspondientes de la secuencia `s`. • Facilita la generación de una nueva colección basada en otra, en este caso, una subsecuencia basada en índices específicos.
subSecuenciasDe	Sí	**Uso de Funciones Preexistentes:** • La función `subindices` se utiliza para obtener todas las combinaciones posibles de índices para una secuencia dada. Esto demuestra la reutilización de código y la composición de funciones en Scala. **Generación de Subsecuencias:** • La expresión `for` se utiliza para iterar sobre cada combinación de índices y generar la subsecuencia correspondiente utilizando la función `subSecuenciaAsoc`.

Table 2: Tabla de uso de colecciones y expresiones for en la función *subSecuenciaAsoc* y *subSecuenciasDe*

2.2.2 Informe de corrección

Argumentación sobre la corrección:
subSecuenciasAsoc:

Sea $S =< a_1, ..., a_i, ..., a_n >$ e $I =< b_1, ..., b_k, ..., b_r >$ tal que i,k,n,r $\in \mathbb{N}^+$ $\wedge 0 \leqslant i \leqslant n$ $0 \leqslant k \leqslant r \wedge$ $0 \leqslant r \leqslant$, dos secuencias de modo que I representa los indices ... de $a_i \in S\{a_n\}$ de los cuales se va a extraer sus respectivos valores.

Estado:

$$1)S = (k, n, r, S_k, I)$$

- Numero de indice actual en I tal que a_{bi} es el elemento a extraer.
- n,r tamanio de la secuencia de estados y de indices respectivamenteamentemente.

- S_k la secuencia tal $S_k =< a_b, ..., a_k >$

Estado inicial:

$$2) S_0 = (0, n, r, \{\phi\}, I) \qquad S_f = (r + 1, n, r, [a_{b1}, ..., a_{br}], I)$$

Invariante:

$$3)\text{Inv}(k, n, r, S_k, I) \equiv k \leq r + 1 \wedge r < n \wedge |S_k| = l \wedge S_k =< a_{b1}, ..., a_{bk} >$$

Transformar:

$$4)\text{Tran}(k, n, r, S_k, I) = (k + 1, n, r, S_k \cup \{a_{b_{k+1}}\}, I)$$

Demostración:

1. $Inv(S_0)$, se cumple. Se tiene $0 < r + 1 \wedge r < n \wedge |\{\phi\}| = 0$
2. $(S_k \neq S_f \wedge \text{Inv}(S_k)) \rightarrow \text{Inv}(\text{transformar}(S_k))$

$$\equiv$$

$$k \leq r + 1 \wedge r < n \wedge |S_k| = k \rightarrow k + 1 \leq r + 1 \wedge r < Mn \wedge |S_{k+1}| = k + 1$$

$$\text{Ya que } S_k =< a_{bi}, ..., a_{bk} >$$

$$\rightarrow$$

$$(S_k \neq S_f \wedge Inv(S_k)) \rightarrow Inv(Tran(S_k))$$

3. $\text{Inv}(S_f) \rightarrow S_r =< a_{b1}, ..., a_{br} > \wedge |Sr| = r$

$$\equiv$$

$$Inv(r + 1, n, r, S_r, I) \rightarrow r + 1 \leq r + 1 \wedge r < n \wedge |S_r| = r \wedge S_k =< a_{i1}, ..., a_r >$$

$$\equiv$$

$$true \wedge true \wedge true \wedge true \equiv true$$

4. Cada paso de iteracion converge a r, lo cual implica la extraccion de r elemento a_i de S, por tanto la subsequencia de r elementos incrementales de S es:

$$S_k =< a_{b1}, ..., a_{br} > |i \leq b_i \leq n - 1 \wedge a_{bi} \in S_i$$

subSecuenciasDe:

Sea $S =< a_1, ..., a_i, ..., a_n >$ tal que k,n,i $\in \mathbb{N}^+$

Sea SA la funcion demostrada anteriormente (subsecuenciasAsoc)

Sea $Si(i, n)$ la funcion demostrada anteriormente (subindices)

Sea $D_s(S)$ como la funcion que devuelve todas las posibles subsecuencias de S

Primero se define la aplicacion de s_i sobre los extremos $1-n$ de la secuencia S

$SI(1,n) = \{e|e \in D(n)\}$, donde $P(n)$ define el conjunto potencia sobre los n primero numeros N, es decir

$$i \leq n \wedge |P(n)| = m$$

Llamemos $SI_k \in SI(1,n)|1 \leq l \leq m$, el $k-esimo$ elemento de SI_k

Estado:

$$1)S = (k, m, SI_k, LC, S)$$

- S La secuencia original que genera todas sus posibles subsecuencias.
- Donde k representa el $k-esimo$ elemento asociado a SI.
- m el tamaño de SI.
- SI_k representa el $k-esimo$ conjunto de indices posibles de SI.
- LC representa el resultado de la aplicacion $SA(S, SI_k)$ para extraer los elementos correspondientes a los SI_k indices.

Estado inicial:

$$2)S_0 = (0, m, SI_0, \{\phi\}, S) \qquad S_f = (m+1, m, SI, [SA(S, SI_1), ..., SA(S, SI_{m-1})], S)$$

Invariante:

$$3)\text{Inv}(k, m, SI_k, LC, S) \equiv k \leq m+1 \wedge LC = [SA(S, SI_1), ..., SA(S, SI_{m-1})]$$

Transformar:

$$4)\text{Tran}(k, m, SI_k, LC, S) \rightarrow (k+1, m, SI_{k+1}, LC \cup SA(S, SI_{k+1}), S)$$

Demostración:

1. $Inv(S_0)$, se cumple, puesto que $(0, m, SI_0, [\phi])$
2. $(S_k \neq S_f \wedge \text{Inv}(S_k)) \rightarrow \text{Inv}(\text{transformar}(S_k))$

$$\equiv$$

$$k \leq m+1 \wedge LC = [SA(S, SI_1), ..., SA(S, SI_k)]$$

$$\rightarrow$$

$$k+1 \leq m+1 \wedge LC = [SA(S, SI_1), ..., SA(S, SI_{k+1})]$$

Puesto que LC representa un subconjunto asociado a las k posibilidades de seleccion de indices, se tiene:

$$(S_k \neq S_f \wedge Inv(S_k)) \rightarrow Inv(Tran(S_k))$$

3. $\text{Inv}(S_f) \rightarrow m + 1 \leq m + 1 \wedge LC = [SA(S, SI_1), ..., SA(S, SI_{m-1})]$

$$\equiv$$

$$true \rightarrow true \wedge true$$

$$\equiv true$$

Casos de prueba:

```
val s1 = Seq(5, 25, 35, 45, 55, 65, 75)
subSecuenciaAsoc(s1, Seq())
subSecuenciaAsoc(s1, Seq(0, 2, 4))
subSecuenciaAsoc(s1, Seq(1, 2, 4, 6))
subSecuenciaAsoc(s1, Seq(0, 3, 5))
subSecuenciaAsoc(s1, Seq(2, 3, 4, 5))
```

Listing 5: Casos de prueba para la función subSecuenciaAsoc

1. **Valor esperado**: List()
2. **Valor esperado**: List(5, 35, 55)
3. **Valor esperado**: List(25, 35, 55, 75)
4. **Valor esperado**: List(5, 45, 65)
5. **Valor esperado**: List(35, 45, 55, 65)

```
1 val s2 = Seq(20, 30, 10)
2 subSecuenciasDe(s2)
3 val s3 = Seq(10, 20)
4 subSecuenciasDe(s3)
5 val s4 = Seq(5, 15, 25, 35)
6 subSecuenciasDe(s4)
7 val s5 = Seq(1, 2, 3, 4, 5)
8 subSecuenciasDe(s5)
9 val s6 = Seq(50, 60, 70, 80, 90, 100)
10 subSecuenciasDe(s6)
```

Listing 6: Casos de prueba para la función subSecuenciasDe

1. **Valor esperado** (line 2): HashSet(List(30), List(20, 30, 10), List(30, 10), List(20), List(10), List(20, 30), List(20, 10), List())
2. **Valor esperado** (line 4): Set(List(), List(10), List(20), List(10, 20))
3. **Valor esperado** (line 6): HashSet(List(5, 15, 25, 35), List(), List(5, 15, 35), List(15, 35), List(5, 25, 35), List(15, 25, 35), List(25), List(35), List(15, 25), List(15), List(5, 35), List(5, 15), List(5), List(25, 35), List(5, 25), List(5, 15, 25))

4. **Valor esperado** (line 8): HashSet(List(1), List(1, 2, 3), List(1, 3), List(3, 4), List(4), List(2), List(1, 2, 3, 4), List(1, 2, 4), List(1, 2, 3, 4, 5), List(1, 2, 3, 5), List(1, 4, 5), List(4, 5), List(1, 3, 4, 5), List(2, 4, 5), List(2, 3, 4, 5), List(3), List(1, 5), List(), List(1, 2, 4, 5), List(2, 3), List(2, 3, 5), List(1, 4), List(1, 3, 4), List(3, 5), List(1, 2), List(1, 2, 5), List(5), List(3, 4, 5), List(2, 3, 4), List(2, 5), List(1, 3, 5), List(2, 4))

5. **Valor esperado** (line 10): HashSet(List(50, 60, 70), List(70), List(60, 80, 90), List(50, 60, 80, 90, 100), List(50, 70), List(60, 70, 80), List(100), List(50, 90, 100), List(50, 60, 90), List(90), List(60, 100), List(80, 90, 100), List(60, 70, 100), List(50, 80, 100), List(70, 100), List(50, 60, 70, 90, 100), List(50, 80, 90, 100), List(50, 60), List(50, 100), List(50, 70, 80, 90), List(50, 60, 100), List(50, 70, 90, 100), List(50, 60, 70, 80, 100), List(60, 70, 80, 90), List(60, 80), List(80, 100), List(70, 90, 100), List(50, 70, 80, 100), List(60, 70, 90, 100), List(60, 90), List(60, 80, 90, 100), List(50, 70, 100), List(80), List(60, 90, 100), List(), List(50, 60, 70, 90), List(50, 60, 70, 80, 90, 100), List(70, 80, 90, 100), List(50, 60, 80, 100), List(50, 60, 80), List(80, 90), List(60, 70, 80, 100), List(60, 70), List(50, 70, 80, 90, 100), List(90, 100), List(70, 80, 100), List(60, 80, 100), List(50, 70, 80), List(50, 80), List(60, 70, 90), List(50, 60, 70, 80, 90), List(70, 90), List(60, 70, 80, 90, 100), List(50, 80, 90), List(50), List(60), List(50, 60, 80, 90), List(70, 80), List(50, 90), List(50, 60, 70, 80), List(50, 60, 70, 100), List(50, 60, 90, 100), List(50, 70, 90), List(70, 80, 90))

2.3 Generación de todas las subsecuencias incrementales de una secuencia

2.3.1 Informe de uso de colecciones y expresiones for

```
def incremental(seq: Subsecuencia): Boolean = seq match {
  case Nil => true
  case _ => (for index <- 1 to (seq.size - 1) yield  seq(index
     - 1) < seq(index)) forall( x => x)
}
```

Listing 7: Código en Scala para la funcion incremental

```
def subSecuenciasInc(seq: Secuencia): Set[Subsecuencia] =
  (for subsequence <- subSecuenciasDe(seq) if incremental(
     subsequence) yield subsequence).toSet
```

Listing 8: Código en Scala para la funcion subSecuenciasInc

Tabla *incremental* y *subSecuenciasInc*		
Función	¿Se utilizó colecciones y expresiones for?	¿Razón?
incremental	Sí	**Pattern Matching:** • El pattern matching es una característica poderosa de Scala que permite descomponer y verificar estructuras de datos. En este caso, se utiliza para manejar dos escenarios: cuando la subsecuencia es vacía (representada por 'Nil') y cuando no lo es. **Verificación Incremental:** • La expresión `for` se utiliza para iterar sobre la subsecuencia y verificar si cada elemento es menor que el siguiente. Esto genera una colección de valores booleanos. • La función `forall` se utiliza para verificar que todos los valores en la colección booleana sean `true`, lo que indica que la subsecuencia es incremental.
subSecuenciasInc	Sí	**Generación y Filtrado de Subsecuencias:** • La función `subSecuenciasDe` se utiliza para generar todas las posibles subsecuencias de la secuencia dada. • La cláusula `if incremental(subsequence)` dentro de la expresión `for` filtra las subsecuencias, conservando solo aquellas que son incrementales. Esto demuestra cómo Scala permite combinar la generación y el filtrado de colecciones de manera concisa. **Conversión a Conjunto:** • `toSet` convierte la colección resultante en un conjunto, eliminando posibles duplicados y garantizando la unicidad de las subsecuencias.

Table 3: Tabla de uso de colecciones y expresiones for en la función *incremental* y *subSecuenciasInc*

2.3.2 Informe de corrección

Argumentación sobre la corrección:
incremental:

Sea *seq* una subsecuencia de números naturales.

Sea $S = \{a_1, a_2, ..., a_n\}$ donde $n = seq.size$ y a_i es el i-ésimo elemento de seq.

El objetivo del algoritmo es determinar si la subsecuencia es incremental, es decir,

$$\text{si } a_i < a_{i+1} \text{ para todo } i \text{ en } [1, n-1].$$

Estado inicial:

$$1)(1, \text{seq}, \text{true})$$

Estado:

$$2)(i, \text{seq}, \text{true/false})$$

$$\text{es final si } i = n \to s_f = (n, \text{seq}, \text{true/false})$$

Invariante:

$$3)\text{Inv}(i, \text{seq}, \text{true/false}) \equiv i \leq n \wedge (a_{i-1} < a_i \text{ implica true, de lo contrario false})$$

Transformar:

$$4)\text{Tran}(i, \text{seq}, \text{true/false}) = (i+1, \text{seq}, \text{true/false según la comparación de } a_i \text{ y } a_{i+1})$$

Demostración:

1. $\text{Inv}(S_0)$ se cumple, pues $1 \leq n$ y no hay elementos anteriores a a_1 para comparar, por lo que el valor inicial es verdadero.
2. $(s_k \neq s_f \wedge \text{Inv}(s_k)) \to \text{Inv}(\text{transformar}(s_k))$

   $$\equiv$$

   $i \neq n \wedge i \leq n \wedge (a_{i-1} < a_i$ implica true, de lo contrario false$) \to i+1 \leq n \wedge (a_i < a_{i+1}$ implica true, de lo contrario false$)$

   $$\text{Por tanto, Inv(transformar}(s_k)) \text{ se cumple.}$$

3. $\text{Inv}(S_f) \to (a_{n-1} < a_n$ implica true, de lo contrario false$)$

   $$\to n \leq n \wedge (a_{n-1} < a_n \text{ implica true, de lo contrario false})$$

   $$\text{true} \wedge \text{true/false según la comparación de } a_{n-1} \text{ y } a_n$$

4. En cada paso, se aumenta i en una unidad hasta alcanzar n. En cada iteración, se verifica si el elemento actual es mayor que el anterior y se actualiza el valor de verdad según el resultado de la comparación.

subSecuenciasInc:

Sea seq una secuencia de números naturales.

El objetivo del algoritmo es determinar todas las subsecuencias incrementales de seq.

Estado inicial:

$$1)(0, \text{seq}, \{\phi\})$$

Estado:

$$2)(i, \text{seq}, \text{subSecuencias})$$

$$\text{es final si } i = \text{seq.size} \to s_f = (\text{seq.size}, \text{seq}, \text{subSecuencias})$$

Invariante:

$$3)\text{Inv}(i, \text{seq}, \text{subSecuencias}) \equiv i \leq \text{seq.size} \wedge \text{subSecuencias contiene todas las subsecuencias incrementales de seq hasta el índice } i$$

Transformar:

$$4)\text{Tran}(i, \text{seq}, \text{subSecuencias}) = (i{+}1, \text{seq}, \text{subSecuencias} \cup \text{subSecuencia si es incremental})$$

Demostración:

1. $\text{Inv}(S_0)$ se cumple, pues $0 \leq \text{seq.size}$ y no hay subsecuencias para considerar en el índice 0.
2. $(s_k \neq s_f \wedge \text{Inv}(s_k)) \to \text{Inv}(\text{transformar}(s_k))$

   $$\equiv$$

   $i \neq \text{seq.size} \wedge i \leq \text{seq.size} \wedge$subSecuencias contiene todas las subsecuencias incrementales de seq hasta el índice $i \to i{+}1 \leq \text{seq.size} \wedge$subSecuencias contiene todas las subsecuencias incrementales de seq hasta el índice $i+1$

   Por tanto, $\text{Inv}(\text{transformar}(s_k))$ se cumple.
3. $\text{Inv}(S_f) \to$ subSecuencias contiene todas las subsecuencias incrementales de seq

   $\to \text{seq.size} \leq \text{seq.size} \wedge$subSecuencias contiene todas las subsecuencias incrementales de seq

   $$\text{true} \wedge \text{true}$$
4. En cada paso, se aumenta i en una unidad hasta alcanzar seq.size. En cada iteración, se verifica si la subsecuencia actual es incremental y, si es así, se agrega a la colección de subsecuencias incrementales.

Casos de prueba:

```
val s7 = Seq(1, 2, 3, 4, 5, 6, 7)
incremental(s7)// true
val s8 = Seq()
incremental(s8)//true
val s9 = Seq(1, 1, 1, 1, 1, 1, 1)
incremental(s9)//false
val s10 = Seq(1, 2, 3, 5, 4, 6, 7)
incremental(s10)//false
val s11 = Seq(7, 6, 5, 4, 3, 2, 1)
incremental(s11)//false
```

Listing 9: Casos de prueba para la función incremental

```
subSecuenciasInc(Seq(1, 2))
subSecuenciasInc(Seq(5, 7, 9))
subSecuenciasInc(Seq(2, 4, 8, 16))
subSecuenciasInc(Seq(0, 1, 2))
subSecuenciasInc(Seq(10, 20, 30, 40))
```

Listing 10: Casos de prueba para la función subSecuenciasInc

1. **Valor esperado**: Set(List(), List(1), List(2), List(1, 2))
2. **Valor esperado**: HashSet(List(5, 9), List(9), List(7), List(), List(5, 7), List(7, 9), List(5, 7, 9), List(5))
3. **Valor esperado**: HashSet(List(8), List(16), List(4, 16), List(2, 8), List(2, 4, 16), List(4), List(2, 16), List(2, 8, 16), List(8, 16), List(4, 8, 16), List(), List(2), List(2, 4, 8, 16), List(4, 8), List(2, 4), List(2, 4, 8))
4. **Valor esperado**: HashSet(List(1), List(0, 1), List(1, 2), List(0, 2), List(0), List(2), List(0, 1, 2), List())
5. **Valor esperado**: HashSet(List(30, 40), List(10, 20), List(10, 40), List(20, 40), List(30), List(10, 30, 40), List(10), List(20, 30, 40), List(20, 30), List(10, 20, 30), List(10, 30), List(), List(20), List(40), List(10, 20, 40), List(10, 20, 30, 40))

2.4 Hallar la subsecuencia incremental más larga

2.4.1 Informe de uso de colecciones y expresiones for

```scala
def subsecuenciaIncrementalMasLarga(seq: Secuencia): Subsecuencia
   = {
  val subsequences = (for subsequence <- subSecuenciasInc(seq) if
      incremental(subsequence) yield subsequence).toList
  val subsequencesSizes = subsequences.map(_.size)
  val indexOfLargestSubsequence = (subsequencesSizes.indexWhere(x
      => x == subsequencesSizes.max))
  indexOfLargestSubsequence match {
    case x if x < 0 => List()
    case x => subsequences(x)
  }
}
```

Listing 11: Código en Scala para la funcion subsecuenciaIncrementalMasLarga

Tabla base $subsecuenciaIncrementalMasLarga$		
Función	¿Se utilizó colecciones y expresiones for?	¿Razón?
subsecuenciaIncremental-MasLarga	Sí	**Generación y Filtrado de Subsecuencias Incrementales:** • Se utiliza la función `subSecuenciasInc` para obtener todas las subsecuencias incrementales de la secuencia dada. • Aunque `subSecuenciasInc` ya filtra las subsecuencias incrementales, la cláusula `if incremental(subsequence)` se mantiene por claridad y robustez. **Determinación de la Subsecuencia Más Larga:** • `map` se utiliza para transformar la lista de subsecuencias en una lista de sus tamaños. • `find` se utiliza para obtener el índice de la subsecuencia más larga. **Pattern Matching para la Salida:** • Se utiliza pattern matching para manejar diferentes escenarios: cuando no se encuentra ninguna subsecuencia, cuando se encuentra una subsecuencia y cualquier otro caso.

Table 4: Tabla de uso de colecciones y expresiones for en la función $subsecuenciaIncrementalMasLarga$

2.4.2 Informe de corrección

Argumentación sobre la corrección:

$$\text{Sea } S =< a_1, ..., a_i, ..., a_n > |1 \leq i \leq n \in \mathbb{N}^+ \text{una secuencia}$$
$$\text{Sea } Inc(S) \text{ una funcion predicado que determina si } S \text{ es incremental}$$
$$\text{Sea } SInc(S)) \text{ la funcion que retorna el conjunto de las subsequiencias incrementales de } S$$
$$\text{Sea } k|k \leq m \text{ donde } |SInc(S)| = m, \text{ es la cantidad de subsecuencias de } S$$
$$\text{Sea } SInc_k \text{ la } k - esima \text{ subsecuencia incremental de } SInc$$

Estado:

$$1)S = (k, m, SInc_k, Inc, S)$$

- k elemento de $SInc$.
- m el tamaño de conjunto $SInc$.
- $SInc_k$ la $k - esima$ subsecuencia incremental de $SInc_k$.
- Inc conjunto que colecciona las k subsecuencias determinadas hasta entonces.

- S la secuencia original.

Estado inicial:

$$2)S_0 = (0, m, \{\phi\}, \{\phi\}, S) \qquad S_f = (m+1, m, \{\phi\}, Inc_n, S)|Inc_n = [SInc_1, ..., SInc_m]$$

Invariante:

$$3)\text{Inv}(S_k) \equiv k \leq m+1 \wedge Inc_k = [SInc_1, ..., SInc_k]$$

Transformar:

$$4)\text{Tran}(k, m, SInc_k, Inc, S) \rightarrow (k+1, m, SInc_{k+1}, Inc \cup SInc_k, S)$$

Demostración:

1. $Inv(S_0)$, se cumple, puesto que $(0, m, [\phi], [\phi], S)$
2. $(S_k \neq S_f \wedge \text{Inv}(S_k)) \rightarrow \text{Inv(transformar}(S_k))$

$$\equiv$$

$$k \leq m+1 \wedge Inc_k = [SInc_1, ..., SInc_k]$$

$$\rightarrow$$

$$k+1 \leq m+1 \wedge Inc \cup SInc_{k+1}$$

$$\text{Puesto que } Inc_{k+1} = \bigcup_{1 \leq i \leq k} SInc_i \cup Inc_{k+1}$$

Se tiene:

$$(S_k \neq S_f \wedge Inv(S_k)) \rightarrow Inv(Tran(S_k))$$

3. $\text{Inv}(S_f) \rightarrow m+1 \leq m+1 \wedge Inc_n \equiv Inc$

donde Inc representa todas las posibles combinaciones incrementales de S

$$\equiv$$

$$true \rightarrow true \wedge true$$

$$\equiv true$$

Hasta aqui, se ha determinado todas las posibles subsequiencias de S, ahora la subsecuencia mas larga. Sea $Size(z)$ una funcion que aplica una transformacion de cada $SInc_k$ a su respectivo tamaño. $Size(Inc) = \{x | x = |Inc_k|\}$.
Sea la funcion $max(S)$ que determina el mayor valor del elemento de S y sea $find(e, N)$ una funcion que devuelve un resultado opcional si se cumple l predicado p sobre un elemento $n \in N$ secuencia.
$\rightarrow find(max(size(Inc)), Inc)$ da la subsecuencia mas larga en los elementos de Inc.

Casos de prueba:

```
  val s14 = Seq(1, 2, 3, 4, 5)
  subsecuenciaIncrementalMasLarga(s14) //List(1, 2, 3, 4, 5)
  val s15 = Seq(5, 10, 15, 14, 13, 12)
  subsecuenciaIncrementalMasLarga(s15) //List(5, 10, 14)
  val s16 = Seq(2, 4, 8, 7, 6, 5)
  subsecuenciaIncrementalMasLarga(s16) //List(2, 4, 7)
  val s17 = Seq(0, 1, 1, 2, 3, 5, 4)
  subsecuenciaIncrementalMasLarga(s17) //List(0, 1, 2, 3, 5)
  val s18 = Seq(10, 20, 30, 25, 35, 45)
  subsecuenciaIncrementalMasLarga(s18) //List(10, 20, 30, 35, 45)
```

Listing 12: Casos de prueba para la función subsecuenciaIncrementalMasLarga

3 Hacia una solución más eficiente

3.1 Calculando $SIML_i(S)$

3.1.1 Informe de uso de colecciones y expresiones for

```
def sssimlComenzandoEn(i: Int, seq: Secuencia): Subsecuencia = {

  def sssimlHelper(r: Int, seq: Secuencia, subsequence:
     Subsecuencia, maxValueOfSubsequence: Int): Subsecuencia = {
    r match {
        case r if seq.size == r => subsequence
        case r => {
          val isLargestValue = seq(r) > maxValueOfSubsequence
          sssimlHelper(
          r + 1,
          seq,
          if (isLargestValue) subsequence ++ List(seq(r)) else
             subsequence,
          if (isLargestValue) seq(r) else maxValueOfSubsequence)
        }
    }
  }

  val subsequences = (
    for{ k <- i until seq.size
          j <- k  until seq.size
          subsequence = sssimlHelper(j, seq, List(seq(k)), seq(k)
             )
  } yield {
    subsequence}).toList
  val subsequencesMaxSize = subsequences.map(_.size).max
  subsequences.find(x => x.size == subsequencesMaxSize) match {
    case None => List()
    case Some(x) => x
```

```
    }
}
```

Listing 13: Código en Scala para la funcion sssimlComenzandoEn

Tabla base completar		
Función	¿Se utilizó colecciones y expresiones for?	¿Razón?
ssimlComenzandoEn	Si	**Generación de Subsecuencias**: La expresión `for` permite iterar sobre la secuencia de entrada y generar todas las posibles subsecuencias incrementales que comienzan en el índice *i*. Esta estructura proporciona una forma concisa y eficiente de generar subsecuencias. **Filtrado y Búsqueda**: Las colecciones en Scala ofrecen métodos como `map`, `max` y `find` que se utilizan en el algoritmo para filtrar y buscar la subsecuencia deseada. **Recursividad y Colecciones**: La función auxiliar `sssimlHelper` utiliza recursividad para construir la subsecuencia incremental. Las colecciones facilitan la construcción y el paso de subsecuencias a través de llamadas recursivas.

Table 5: Tabla de uso de colecciones y expresiones for en la función $ssimlComenzandoEn$

3.1.2 Informe de corrección

Argumentación sobre la corrección:
la funcion *sssimlComenzandoEn* devuelve la secuencia S_i mas larga de la secuencia S desde el indice i; hace uso de una expresion for la cual genera un producto cartesiano kxj, donde k representa las lista original sin los k primeros elementos, y j el recorrido desde k hasta el tamaño n de la subsecuencia de tamaño $n-k$. Dentro de la expresion for se llama para cada combinacion (k, j) la funcion $sssimlHelper$, que recibe el parametro j actual, la secuencia original, la subsecuencia de los valores que han sido incrementales en sus anteriores iteraciones, y el ultimo valor mas grande e incremental encontrado. Dentro, tiene las condicion de parada si se ha alcanzado el tamaño de la secuencia original; de lo contrario, se aplica el predicado que determina si el valor actual de la secuencia $seq(r)$ en r supera en valor númerico al ultimo valor incremental $maxValueOfSubsequence$; si ello se cumple, el proximo llamado recursivo a $sssimlHelper$ sera con este nuevo valor determinado addicionado a *subsecuencia* y $maxValueOfSubsequence$ actualizado a dicho valor con la logica de detrminacion de termino incrementales de la subsecuencia implicada para el indice t. Si el predicado no se cumple, se incrementa r en uno para que tienda a *seq.size* y seguir el proceso sin modificar la *subsecuence* hasta el paso el paso r

Este proceso lo hace iterativamente para cada k donde $i \leq k \leq n$. Al final del paso anterior, donde se cumple las invariantes de $k \leq seq.size \wedge k \leq j \leq seq.size$, se tiene que *subsequences* es el conjunto de todas las posibles subsecuencias incrementales de *seq* desde i hasta *seq.size*

Al anterior resultado le aaplicamos un transformacion $Size(subsequences)$ a cada elemento de *subsequences* para obtener el tamaño de cada subsecuencia incrementales de *seq*. Entonces se usa la funcion $max(Size(subsequences))$ para determinar cual de estas secuencias incrementales tiene mayor numero de elementos, que es el interes ultimo de este algoritmo. El resultado de *max* devuelve un valor que a posteriori sera utilizazo como algumento de la funcion $find$, la cual toma como argumento un predicado p y una secuencia a la cual aplicar el predicado. En particular, el predicado determina el primer elemento dentro de *subsequences* tal que su valor se corresponda al valor maximo determinado. La ultima instruccion es un match pattern sobre el resultado de *find*, funcion que retorna el resultado en un contexto computacional, la monada *Optional*, el cual permite manejar la ausencia y presencia de un valor de forma natural y sin errores inesperados. Por tanto, la expresion de match pattern permite desenvolver el valor correspondiente a la subsecuencia mas larga dentor de *seq* desde *i*, y retornala como el valor retorno de *sssimlComenzandoEn*. Esto demuestra, la correctitud de la implementacion de la funcion.

Casos de prueba:

```
  sssimlComenzandoEn(0, Seq(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 22,
     21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11))
  // List(10, 22)
  sssimlComenzandoEn(5, Seq(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
  //List(6, 7, 8, 9, 10)
  sssimlComenzandoEn(3, Seq(5, 6, 7, 1, 2, 3, 4, 8, 9, 10))
  //List(1, 2, 3, 4, 8, 9, 10)
  sssimlComenzandoEn(2, Seq(10, 20, 5, 15, 25, 35, 45))
  //List(5, 15, 25, 35, 45)
  sssimlComenzandoEn(4, Seq(2, 4, 6, 8, 1, 3, 5, 7, 9))
  //List(1, 3, 5, 7, 9)
  sssimlComenzandoEn(1, Seq(5, 1, 2, 3, 4, 6, 7, 8, 9, 10))
  //List(1, 2, 3, 4, 6, 7, 8, 9, 10)
```

Listing 14: Casos de prueba para la función sssimlComenzandoEn

3.2 Calculando una subsecuencia incremental más larga, versión 2

3.2.1 Informe de uso de colecciones y expresiones for

```
def subSecIncMasLargaV2(sequence: Secuencia) =
  val si = (for i <- 0 until sequence.size yield
     sssimlComenzandoEn(i, sequence))
  val siSizes = for j <- 0 until si.size yield si(j).size
  (si.find(x => x.size == siSizes.max)) match {
      case None => List()
      case Some(x) => x
  }
```

Listing 15: Código en Scala para la funcion subSecIncMasLargaV2

Tabla base completar		
Función	¿Se utilizó colecciones y expresiones for?	¿Razón?
$subSecIncMasLargaV2$	Si	**Generación de Subsecuencias**: La primera expresión `for` itera sobre cada índice de la secuencia y utiliza la función `sssimlComenzandoEn` para generar la subsecuencia incremental más larga que comienza en ese índice. Esto resulta en una colección de subsecuencias. **Obtención de Tamaños**: La segunda expresión `for` itera sobre la colección de subsecuencias generadas y obtiene el tamaño de cada una. Esto facilita la identificación de la subsecuencia más larga en pasos posteriores. **Búsqueda de la Subsecuencia Más Larga**: Se utiliza el método `find` de las colecciones en Scala para buscar la subsecuencia con el tamaño máximo. Esta operación es eficiente y concisa gracias a las capacidades de las colecciones en Scala. **Manejo de Casos**: El uso del emparejamiento de patrones (pattern matching) con `match` permite manejar diferentes casos, como cuando no se encuentra ninguna subsecuencia o cuando se encuentra la subsecuencia deseada. Esto proporciona una forma estructurada y legible de manejar diferentes escenarios de salida.

Table 6: Tabla de uso de colecciones y expresiones for en la función subSecIncMasLargaV2

3.2.2 Informe de corrección

Argumentación sobre la corrección:

$$\text{Sea sequence una secuencia de números naturales.}$$
$$si \text{ contiene todas las subsecuencias incrementales de sequence hasta el índice } i$$

El objetivo del algoritmo es determinar la subsecuencia incremental más larga de sequence.

Estado inicial:

$$1)(0, \text{sequence}, \{\phi\}, \{\phi\})$$

Estado:

$$2)(i, \text{sequence}, si, \text{siSizes})$$
$$\text{es final si } i = \text{sequence.size} \to s_f = (\text{sequence.size}, \text{sequence}, si, \text{siSizes})$$

Invariante:

$$3)\text{Inv}(i, \text{sequence}, si, \text{siSizes}) \equiv i \leq \text{sequence.size} \land si \text{ contiene todas las subsecuencias}$$
$$\text{incrementales de sequence hasta el índice } i$$

Transformar:

$$4)\text{Tran}(i, \text{sequence}, si, \text{siSizes}) = (i+1, \text{sequence}, si \cup \text{subSecuencia si es incremental},$$
$$\text{siSizes} \cup \text{tamaño de subSecuencia})$$

Demostración:

1. $\text{Inv}(S_0)$ se cumple, pues $0 \leq \text{sequence.size}$ y no hay subsecuencias para considerar en el índice 0.

2. $(s_k \neq s_f \wedge \text{Inv}(s_k)) \rightarrow \text{Inv}(\text{transformar}(s_k))$

$$\equiv$$

$$i \neq \text{sequence.size} \wedge i \leq \text{sequence.size} \wedge si$$

$$i \rightarrow i + 1 \leq \text{sequence.size} \wedge si$$

Por tanto, $\text{Inv}(\text{transformar}(s_k))$ se cumple.

3. $$\text{Inv}(s_f) \rightarrow si$$

$$\rightarrow \text{sequence.size} \leq \text{sequence.size} \wedge si$$

$$\text{true} \wedge \text{true}$$

4. En cada paso, se aumenta i en una unidad hasta alcanzar sequence.size. En cada iteración, se verifica si la subsecuencia actual es incremental y, si es así, se agrega a la colección de subsecuencias incrementales y se actualiza el tamaño de la subsecuencia.

Casos de prueba:

```
subSecIncMasLargaV2(Seq())
// List()
subSecIncMasLargaV2(Seq(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
// List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
subSecIncMasLargaV2(Seq(5, 6, 7, 1, 2, 3, 4, 8, 9, 10))
// List(1, 2, 3, 4, 8, 9, 10)
subSecIncMasLargaV2(Seq(10, 20, 5, 15, 25, 35, 45))
// List(10, 20, 25, 35, 45)
subSecIncMasLargaV2(Seq(2, 4, 6, 8, 1, 3, 5, 7, 9))
// List(2, 4, 6, 8, 9)
subSecIncMasLargaV2(Seq(5, 1, 2, 3, 4, 6, 7, 8, 9, 10))
// List(1, 2, 3, 4, 6, 7, 8, 9, 10)
```

Listing 16: Casos de prueba para la función subSecIncMasLargaV2

4 Conclusiones

Los casos de prueba actuales son bastante robustos y cubren una amplia variedad de escenarios que el programa podría encontrar en la práctica. Se trata de abordar diferentes escenarios para que mientras se va desarrollando el programa/funcion se puedan encontrar errores en el proceso de solucion de este. Como bien sabemos la mejor manera de demostrar la correctitud de este tipo de algoritmos, es a través de una demostracion formal. Para estas 8 funciones que se desarrollaron en este taller, la modalidad de demostracion que se uso fue $Induccion\ Estructural$, creemos que es pertinente porque todas las funciones utilizan expresiones for y colecciones de $scala$. A pesar que algunos casos de prueba nos ayudaron en el proceso de la creacion o solucion de las funciones, es practicamente imposible hacer suficientes casos de prueba para abarcar todas las posibles combinaciones de listas o sequencias que podrian entrar por parametro para estas funciones. Por lo tanto se acude a una demostracion formal para verificar su correctitud en todos los puntos del taller.