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

```
1  def subindices(i: Int, n: Int): Set[Seq[Int]] = {
2      val elements = (i until n).toSet
3      (for {
4          k <- 0 to elements.size
5          combination <- elements.subsets(k)
6      } yield combination.toSeq.sorted ).toSet
7  }
```

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:
Casos de prueba:

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

2.1.3 Conclusiones

2.2 Generación de todas las subsecuencias de una secuencia

2.2.1 Informe de uso de colecciones y expresiones for

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

Listing 3: Código en Scala para la funcion subSecuenciaAsoc

```
1  def subSecuenciasDe(s:Secuencia): Set[Subsecuencia] ={
2    val combinationIndex = subindices(0, s.size)
3    for index <- combinationIndex yield subSecuenciaAsoc(s, index
       )
4  }
```

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:
Casos de prueba:

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

2.2.3 Conclusiones

2.3 Generación de todas las subsecuencias incrementales de una secuencia

2.3.1 Informe de uso de colecciones y expresiones for

```
1  def incremental(seq: Subsecuencia): Boolean = seq match {
2    case Nil => true
3    case _ => (for index <- 1 to (seq.size - 1) yield  seq(index
       - 1) < seq(index)) forall( x => x)
4  }
```

Listing 7: Código en Scala para la funcion incremental

```
1  def subSecuenciasInc(seq: Secuencia): Set[Subsecuencia] =
2    (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:
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))

2.3.3 Conclusiones

2.4 Hallar la subsecuencia incremental más larga

2.4.1 Informe de uso de colecciones y expresiones for

```
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:
Casos de prueba:

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

Listing 12: Casos de prueba para la función subsecuenciaIncrementalMasLarga

2.4.3 Conclusiones

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:
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

3.1.3 Conclusiones

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:
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

3.2.3 Conclusiones