Sorteeralgoritmes in Scala

De programmeertaal Scala is in 2004 door Martin Odersky uitgebracht. Scala is zowel een object-georiënteerde taal als een functionele taal. Het is een volledig object-georiënteerde taal op het vlak dat elke waarde een object is en het is functioneel omdat elke functie een waarde is.
Scala is volledig interoperabel met Java, het is dus geen extensie van Java. In de compilatie van Scala worden Scala bestanden omgezet naar Java bytecodes en wordt het uitgevoerd op een JVM[1]. Scala wordt door een indrukwekkend lijstje techgiganten gebruikt zoals Apple, LinkedIn, Twitter en Netflix[2].

Ik schrijf de blog om mijn onderzoeksdoel en het proces hiervan te delen. Het doel van mijn onderzoek is: "*Het ontwikkelen van de mergesort, insertionsort en quicksort algoritmes in Scala*". Dit doel is behaald als ik alle drie de sorteeralgoritmes heb uit geprogrammeerd voor een lijst van integers en dit heb getest is doormiddel van unittests waarbij er diverse combinaties van integers mogelijk is.

In de blog zal regelmatig gebruik worden gemaakt van code verwijzingen, hier zit altijd een bron bij met een linkje naar het desbetreffende bestand in mijn git repository.

Hello World

Om iets te leren, moet je beginnen. Klinkt als een simpele en vrije logische stap, maar wel een belangrijke stap. En waar begin je het beste? Bij de basis. Ik heb de Scala plugin geïnstalleerd in Intellij en heb een hello-world project gestart. Nou, een hello world stelt dus niks voor:

```
1| println("Hello world")
```

Het eerste wat mij opviel, de regel werd niet afgesloten met een puntkomma. Voor mij is dit geen vreemd fenomeen, omdat ik al ervaring heb met Kotlin waar dit ook niet gebeurd. Maar voor een taal die gebaseerd is op Java is dit wel opvallend. Ik heb een Scala Worksheet[3] gemaakt, dit is een omgeving waar je snel resultaat kan zien van je code zoals debugs en printlns. In dit worksheet ben ik begonnen met het uitzoeken naar de Scala syntax die ik nodig ga hebben voor de sorteeralgoritmes.

Iets wat vaak terugkomt in lijstjes, zijn for-loops. Eerst heb ik een array gemaakt. Als variabele keuze heb je de mogelijkheid om gebruik te maken van `var` en de read-only variant `val`. Door gebruik te maken van deze keywords hoef je geen telkens de datatypes te noteren. Nadat ik een lijstje van een paar integers had, ben ik begonnen aan de for-loop. "For" is een bestaand keyword binnen scala/intellij, maar daar blijft het dan ook wel bij. Na een snelle google kwam ik op een duidelijk uitleg over Scala's for-loop[4]. Deze pagina was erg behulpzaam voor al mijn vraagstukken betrekkend tot for-loops. Als ik het vergelijk met Java's for**i**-loop, hoef je in Scala geen index te initialiseren. Je hoeft dus ook geen index op te hogen. Om de lengte van de loop en de index te setten gebruik je de annotatie `INDEX <- START to END`. Het lijkt veel op de Java variant maar net wat compacter en minder code. Wat ook te vinden is op de pagina van geeksforgeeks, is het filteren van de for-loop. In Scala kan je nog voor het openen van de loop met de eerste accolade een if-statement aanroepen. Hier kan je een conditie neerzetten, als het niet aan de conditie voldoet, wordt de iteratie overgeslagen. De code van bovengenoemde notaties ziet als volgt uit:

```
3|  val numbers = Array(1, 48, 7, 98, 5)
4|  for (i <- 0 to 4) {
5|    println(numbers(i)) // 1, 48, 7, 98, 5
6|  }
7|
8|  for (i <- 0 to 4
9|       if numbers(i) < 42) {
10|   println(numbers(i)) // 1, 7, 5
11| }
```

Drie dingen die opvallen:

1. Er wordt helemaal nergens over datatypes gesproken. De array wordt gevuld met integers (te herkennen aan de komma-lose en niet gequote cijfers). Door `val` te gebruiken, wordt dit geaccepteerd.
2. Om een punt in een array te bereiken gebruik je round brackets in plaats van square brackets. Niet heel veel geks aan de hand, maar wel even anders dan de annotatie in vele programmeertalen.
3. Ik vind dat de filter in de for-loop de code overzichtelijker maakt. Veelal zie je for-loops die beginnen met if zonder else. Dit kost weer een extra indent en extra nadenk moment. Door het in de for-loop te zetten, zal het mij ook dwingen om bij een te complexe conditie een functie aan te maken zodat je geen situatie krijgt van `(numbers(i) < 42 || numbers(i) > 50) && numbers(i) != 1`. Dit is natuurlijk nog steeds een simpele conditie, maar je begrijpt het idee.

Naast dat for-loops vaak terugkomen in sorteeralgoritmes, worden arrays ook vaak opgesplitst. Tijdens mijn zoektocht op het wereldwijde web kwam ik tegen dat een `List` eenvoudig kan worden opgesplitst Scala[5]. Scala's list bevat over de methodes `head` en `tail`. De head methode returned het eerste object in een array. De tail returned alles behalve de head. Bij meerdere sorteeralgoritmes kan dit zeer van pas komen. De implementatie is niet lastig. Over een list kan je met keyword `match` hier gebruik van maken. Simpel gezegd is match de switch van Scala. De match wordt geopend met accolades, er zijn meerdere cases. In Java kan je echter alleen over switchen over primitive types zoals integers, strings, enum etc. Door in Scala over een lijst te kunnen switchen kan de head en tail bemachtigen en gebruiken. Hier moet je wel eerst een empty check aan toevoegen om een error te voorkomen als de lijst leeg is. Dit kan eenvoudig met keyword `Nil`, wat leeg betekent. Hieronder een voorbeeld van een match:

```
13| List(1, 48, 7, 98, 5) match {
14|   case Nil => println("empty list")
15|   case (head :: tail) =>
16|     println(head) // 1
17|     println(tail) // List(48, 7, 98, 5)
18| }
19|
20| List() match {
21|   case Nil => println("empty list") // "empty list"
22|   case (head :: tail) =>
23|     println(head)
24|     println(tail)
25| }
```

Zoals ik al eerder zei, is Scala een combi van OO en functioneel. Nu ga ik niet echt inzomen op het OO binnen Scala, maar het is wel noemenswaardig. In Scala kan je zowel objecten als objecten maken. Dit

lijkt erg veel op de Java varianten. Een class kan een interface implementeren met keyword `with`, dit kunnen er net zoals bij Java meerdere zijn. Het extenden van een super class is ook mogelijk in Scala met keyword `extends`, dit kan net zoals bij Java slechts met één super class. OO in Scala zal nu ook verder niet echt meer aan bod komen omdat ik mij graag wil focussen op het functionele van Scala.

Functies in Scala zijn in de basis hetzelfde. De syntax is echter wel ietsjes anders. Java's `function` is vervangen door `def` gevolgd door een functie naam, hierin staan paramters. Opvallend is dat je hier `paramaternaar: Datatype` hebt. Vervolgens geef je met `: Dataype` de return waarde weer gevolgd met een `=` en de bekende accolades. Overigens, zijn de accolades overbodig in onderstaand voorbeeld. In Scala kan je een functie die bestaat uit één regel zonder accolades noteren. Scala's functiesyntax doet mij heel erg denken aan de syntax van Kotlin[6]. Kotlin heeft bijna gelijke opbouw aan een functie als Scala. `Def` is dan `fun` en in Scala is de return datatype verplicht. In Kotlin is dit alleen verplicht als je iets returned én geen one-liner (dus zonder accolades) iets retuned. Onderstaand is een voorbeeld van een simpele functie in Scala:

```
27| def myName(name: String): String = {
28|   "Mijn naam is " + name
29| }
30| println(myName("Thijs")) // "Mijn naam is Thijs"
```

Nu ik de basis syntax begrijp, ben ik door een aantal opdrachten van aperiodic[7] heen gegaan. Dit zijn vele Scala opdrachten die met uitleg en uitwerkingen te volgen zijn. Dit heeft mij erg geholpen in het leren van Scala. Nu ben ik klaar om aan mijn onderzoeksdoel te beginnen.

All code is guilty, until proven innocent.

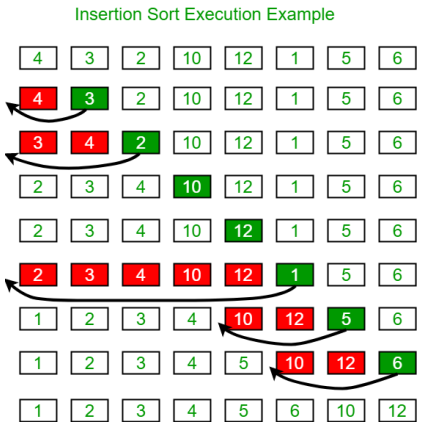
En zo is het ook. De quote, die helaas bronloos is, geeft een goede reden waarom testen belangrijk is. Elke regel code is fout totdat het bewezen dat hij niet fout is. Het bewijzen dat een regel code niet fout is, kan doormiddel van unittesten. Ook in Scala kan dit. Met behulp van ScalaTest dependency[8] kan er geunit test worden. Het installeren van de dependency en het werkend krijgen van de tests is appeltje eitje. Het is een hele eenvoudige manier om code te testen en onschuldig te kunnen verklaren. Unittesten lijkt heel erg op het testen zoals bij Java.
Om te valideren of mijn onderzoeksdoel is behaald, zal ik mijn sorteeralgoritmes door verschillende testscenario's halen. Ik heb ervoor gekozen om dit te doen aan de hand van vier scenario's:

1. Happy flow, een niet-gesorteerde lijst van meer dan 5 integers.
2. Empty list, een lege lijst die een lege lijst hoort terug te krijgen zonder foutmeldingen.
3. Sorted list, een al gesorteerde lijst die ook na het uitvoeren nog steeds gesorteerd is.
4. Same numbers, een lijst met alleen maar dezelfde nummers.

Met deze vier scenario's heb ik kunnen aantonen dat mijn onderzoeksdoel gehaald is. Dit zal ik dan ook voor elk sorteeralgoritme done.

Sorteeralgoritme #1: Insertionsort

Het principe van de insertionsort is als het goed is bekend. Korte geheugensteun: een lijst van nummers moet gesorteerd worden. Je kijkt bij telkens of het eerste nog-niet-gesorteerde nummer kleiner is dan de al gesorteerde nummers. Visueel ziet dat er uit als de afbeelding rechts[9]:



Om dit te realiseren ben ik begonnen met een object te maken `InsertSort`. In dit object heb ik de functie `insertSort` gemaakt. Deze krijgt een `List` van integers mee en returned ook weer een lijst van integers. Allereerst ga ik kijken of de lijst uberhaupt gevuld is. Dit doe ik behulp van de `isEmpty()` functie die ik op een list kan aanroepen. Als dit zo is, return ik de lege lijst. Als de lijst niet leeg is, dan ga ik de functie insert aanroepen. Deze functie heeft parameters. De `head` van de lijst en de uitkomst van `insertSort(numbers.tail)`, de body:

```
1| object InsertSort {
2|   def insertSort(numbers: List[Int]): List[Int] = {
3|     if (numbers.isEmpty) List()
4|     else insert(numbers.head, insertSort(numbers.tail))
5|   }
6| }
```

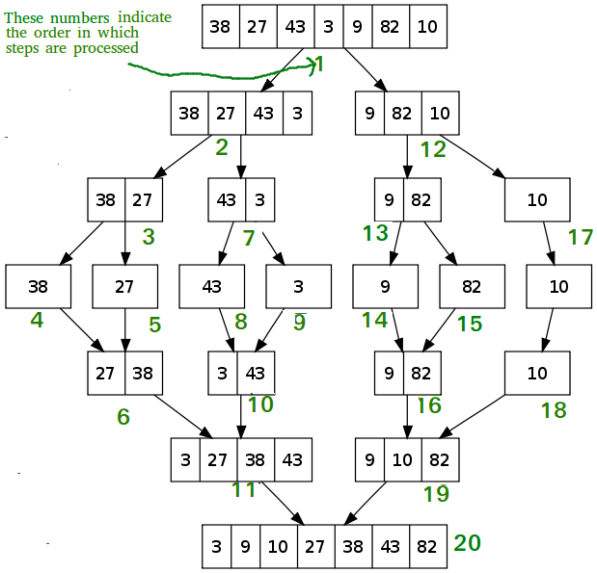
Nu moet ik `insert()` gaan uitwerken. Deze functie moet gaan kijken of de head kleiner of gelijk is aan de body. Als dit zo is, return dan een lijst die begint met de head gevolgd door de body. Als dit niet het geval is en de head dus groter is dan de bodyhead. Dan moet wordt de bodyhead opgeslagen als eerste, en roept de functie zichzelf aan met de head die hij als eerste meekreeg en de tail van de body (dus min het kleinere getal). Als je dit in code uitwerkt krijg je dit:

```
1|  object InsertSort {
2|    def insertSort(numbers: List[Int]): List[Int] = {
3|      if (numbers.isEmpty) List()
4|      else insert(numbers.head, insertSort(numbers.tail))
5|    }
6|
7|  private def insert(head: Int, body: List[Int]): List[Int] =
8|    body match {
9|      case Nil => List(head)
10|     case bodyHead :: bodyTail =>
11|      if (head <= bodyHead) head :: body
12|      else bodyHead :: insert(head, bodyTail)
13|  }
14|
15| }
```

Bovenstaand voorbeeld is net iets anders dan ik eerder beschreef. Ik kwam er namelijk achter dat er ook een check moet komen of er überhaupt wel een body is. Als dit er niet is, return dan een lijst van de head.
De insert sort is af. Tijd om het te valideren. Ik heb mijn van tevoren bedachte testscenario's uitgewerkt in ScalaTest. Voor inzicht is dit te vinden in de git repository[10]. Het resultaat van de tests is positief. De insertion sort werkt! Het eerste deel van het onderzoeksdoel is succesvol afgerond. Op naar algoritme #2.

Sorteeralgoritme #2: MergeSort

Een mergesort splitst een lijst van nummers continu op totdat het nog maar uit één nummer bestaat. Dan gaat de lijst weer per opsplitsing samenvoegen. Rechts staat het het idee visueel weergegeven[11]:



Zogezegd zo gedaan … uhm geprobeerd. Het eerste gedeelte van het realiseren ging goed. Ik heb een functie `mergeSort()` gemaakt die een lijst van numbers meekrijgt. Als allereerste wordt er gekeken of de lijst wel meerdere nummers bevat, dan is de lijst namelijk helemaal opgesplitst. Als dit niet het gehval is, dan moet de lijst worden opgedeeld in links en rechts. Ik kwam achter nog een mooie functie van Scala, het opsplitsen van een lijst. Door de functie op de lijst aan te roepen met de mediaan meegeven als parameter krijg je twee variabelen terug, de linkerkant en de rechterkant. Met deze twee zijdes kan je de functie `merge()` aanroepen met de parameters `mergeSort(left)` en `mergeSort(right)`, om zo de lijsten op te splitsen.

Ik kwam achter nog een interessante functionaliteit in Scala. Er kan een `match` worden gemaakt over meerdere lijstjes. Dit scheelt dubbele code, wat zorgt dat het beter te onderhouden is. Net zoals bij de insertion sort is het belangrijk om te checken of de lijst überhaupt nummers bevatten. Als dit in beide gevallen zo is, dan worden de lijstjes opgesplitst met head en tail. Vervolgens wordt er gekeken of de linkerhead kleiner is dan de rechter head, hier wordt dan de lijst op samengevoegd. De code ziet er als volgt uit:

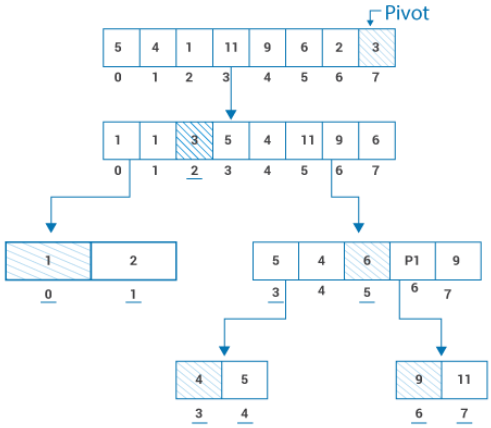
```
1|  object MergeSort {
2|    def mergeSort(numbers: List[Int]): List[Int] = {
3|      if (numbers.length <= 1) numbers
4|      else {
5|        val (left, right) = numbers.splitAt(numbers.length / 2)
6|        merge(mergeSort(left), mergeSort(right))
7|      }
8|    }
9|
10|    private def merge(left: List[Int], right: List[Int]): List[Int] =
11|      (left, right) match {
12|        case (_, Nil) => left
13|        case (Nil, _) => right
14|        case (leftHead :: leftTail, rightHead :: rightTail) =>
15|          if (leftHead < rightHead) leftHead :: merge(leftTail, right)
16|          else rightHead :: merge(left, rightTail)
17|      }
18|  }
```

Het opsplitsen van een lijst met head en tail maakt de code stukken leesbaarder dan in Java. In java zou je d.m.v. van indexen lijstjes moeten maken en kopieren om te kunnen mergen. De volgende is het schrijven van de tests. Hier kan bijna helemaal gebruikt worden gemaakt van de tests van de InsertSort,

de scenario's blijven immers hetzelfde. De tests zijn te vinden in de git repository[12]. Ook hier slagen alle testen. Onderzoeksdoel 2 ook aftekenen, door naar sorteeralgoritme #3.

Sorteeralgoritme #3: QuickSort

Quicksort heeft dezelfde strategie als mergesort: opsplitsen en sorteren. De keuze waar je de pivot legt kan verschillen, maar ik kies er voor om altijd het laatste item uit de lijst te pakken zoals in het voorbeeld rechts[13]:



Ik ben begonnen om de `quickSort()` functie uit te werken. De functie krijgt een array van integers, de laagste index en de hoogste index. Als de laagste index niet kleiner is dan de hoogste index, dan is de lijst gesorteerd. Dit zal ook de failsafe zijn. Vervolgens wordt de partition index opgehaald door `partitition()` aan te roepen met alle nummers en de laagste en hoogste index. Zoals benoemd wordt de pivot de hoogste index. De partition index wordt op de laagste index gezet. Vervolgens wordt er doormiddel van een for-loop waar een if-statement in zit gekeken of het nummer kleiner is dan het nummer van de pivot. Voor elke nummer waarvoor dit geldt, wordt er een swap uitgevoerd tussen de current index en de partition index. Tot slot wordt er nogmaals een swap uitgevoerd met de partition index en de pivot en wordt de index gereturnd. Deze index wordt in de `quickSort()` gebruikt om zichzelf recursief aan te roepen. Dit verhaaltje in code ziet er als volgt uit:

```
1| object QuickSort {
2|   def quickSort(numbers: Array[Int], low: Int, high: Int) {
3|     if (low < high) {
4|       val pi = partition(numbers, low, high)
5|       quickSort(numbers, low, pi - 1)
6|       quickSort(numbers, pi + 1, high)
7|     }
8|   }
9|
10|  private def partition(numbers: Array[Int], low: Int, high: Int): Int = {
11|    val pivot = high
12|    var i = low
13|    for (
14|      j <- low to high
15|      if numbers(j) < numbers(pivot)
16|    ) {
17|      swap(numbers, i, j)
18|      i += 1
19|    }
20|
21|    swap(numbers, i, pivot)
22|    i
23|  }
24|
25|  private def swap(numbers: Array[Int], pos1: Int, pos2: Int) {
```

```
26|     val stash = numbers(pos1)
27|     numbers(pos1) = numbers(pos2)
28|     numbers(pos2) = stash
29|   }
30|}
```

De Scala code komt heel erg veel overeen met de Java code. Syntactisch is er ook weinig verschil. De `if` zit in de for-loop en je ziet nergens `return` staan, hiervoor is de `=`. Verder heb ik eigenlijk weinig te opmerken over de code. Het schrijven hiervan ging eenvoudig, dit door de overeenkomsten met Java en dankzij mijn van tevoren opgedane kennis. Het testen hiervan was net zoals bij de eerdere algoritmes eenvoudig en na het slagen van de tests heb ik mijn onderzoeksdoel behaald!

All adventures, especially into new territory, are scary.

Een ander paradigma, geen OO. Had hij dat goed gezegd? Ja toch echt… Programmeren in een taal waar je niks doet met objecten. Dat klonk wel een beetje eng. Scala is een taal die wel OO is, maar grotendeels functioneel. Tijdens mijn avontuur met Scala heb ik geprobeerd om zo min mogelijk met objecten te werken. Ik ben van mening dat dit gelukt is. Van tevoren heb ik het onderzoeksdoel gesteld dat ik drie sorteeralgoritmes wil realiseren in Scala en dit kunnen testen. Dit alles is gelukt. Ik heb de algoritmes geschreven met functionaliteiten die in een Java niet aanwezig zijn.

In het begin van mijn tijd met Scala had ik wat moeite om de juiste stappen te zetten om de taal te leren. Gelukkig kwam ik enkele websites tegen die mij konden helpen om Scala beter te beheersen. Mijn playground was ook echt nuttig om de sorteeralgoritmes met zo weinig mogelijk tegenstoot te realiseren.

Scala lijkt veel op Java, maar ik vond het ook veel op Kotlin lijken. Van de notaties van datatypes in de parameters als de oneliners. Persoonlijk ben ik een fan van de compacte stijl van Kotlin waarbij alle overbodige dingen niet worden weergegeven. In Scala komt dit terug, het is compact en efficiënt. Dit vind ik fijn om mee te werken. Dit bracht mij meer vertrouwen in Scala en dat ik mijn onderzoeksdoel ging behalen. Ik kan mij goed voorstellen als je geen andere programmeerervaring dan Java hebt, dan zal er nog best wel het een en ander gek en eng zijn. De ervaring met Kotlin helpt om de stap naar Scala eenvoudiger te zetten.

Doordat Scala ook een objecten ondersteund, kan het potentie hebben om samen te werken met andere talen en te praten naar endpoints. In de toekomst ga ik dit meenemen in de overwegingen om andere mogelijkheden te bieden dan een object georiënteerde taal.

Ik heb met vallen en opstaan Scala geleerd en mijn eerste project succesvol er mee afgerond. De ervaring neem ik mee en zoals ik al zei zal ik de mogelijkheden van Scala meenemen naar toekomstige keuzes. Alle code van dit onderzoek is te vinden in mijn git repository[14].

Bronnen

[1] https://www.javatpoint.com/history-of-scala
[2] https://alvinalexander.com/scala/whos-using-scala-akka-play-framework/
[3] https://github.com/ThijsBaan/scala_blog/blob/master/src/Playground.sc
[4] https://www.geeksforgeeks.org/for-loop-in-scala/
[5] https://www.scala-lang.org/api/2.7.0/scala/List.html
[6] https://kotlinlang.org/
[7] http://aperiodic.net/phil/scala/s-99/
[8] https://docs.scala-lang.org/getting-started/intellij-track/testing-scala-in-intellij-with-scalatest.html
[9] https://gaebster.ch/insertionsort-algorithm/
[10] https://github.com/ThijsBaan/scala_blog/blob/master/src/test/scala/InsertSortTest.scala
[11] https://www.geeksforgeeks.org/merge-sort/
[12] https://github.com/ThijsBaan/scala_blog/blob/master/src/test/scala/MergeSortTest.scala
[13] https://www.edureka.co/blog/quick-sort-in-cpp
[14] https://github.com/ThijsBaan/scala_blog