README Exercise 2

This exercise session includes some more tips about working with SBT followed by Scala programming exercises.

Run SBT with Arguments

Running SBT one time with arguments "arg0" "arg1" "arg2":

```
sbt "run arg0 arg1 arg2"
```

Another way is to first open the SBT shell and then pass the run command including the arguments to the executor:

```
> sbt
> run arg0 arg1 arg2
```

This is a better option if you are running several successive commands.

Execute SBT Command on Save

By adding ~ before an SBT command, the command will be executed every time a file in the project is saved. For instance, execute the run command on save:

```
sbt "~run arg0 arg1 arg2"
```

SBT Dependencies

SBT defines something called "Managed Dependencies" which enables it to download the specified libraries. Last week we had no library dependencies, but if you look in the SBT file this week, it has the keyword "libraryDependencies". Adding library dependencies has the following format:

```
libraryDependencies += groupID % artifactID % revision
```

The names groupID, artifactID and revision are all defined as strings meaning that they are written with "". An example of how to add the scalactic library is seen in the SBT file:

```
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.8"
```

See this link for more information about library dependencies with SBT: https://www.scala-sbt.org/1.x/docs/Library-Dependencies.html

SBT Clean

It is possible to delete all generated files by running:

```
sbt clean
```

SBT Define Main Function

There are many ways to run your program, we suggest four different ways that depending on what you want to achieve can be used.

Options are as follows:

1. Run From IntelliJ, this is done by using the arrow next to a method inside the editor
2. From console, if there are no main method specified you will be given a list of options to chose from. This can be tried using the terminal on this project by typing ("run") in the SBT console.
3. From console where you in the console specify which main method to use. An example of this is:

´´´sbt runMain BIDMTMath.SimpleMath ´´´ 4. It is possible to define which main function is used when "sbt run" is executed. This is done by adding the following to the SBT file:

```
mainClass in (Compile, run) := Some("path")
```

Here, "path" needs to be replaced with the path to the package where the Scala file with the main function is. For instance, choosing the main function in SimpleMath.scala:

```
mainClass in (Compile, run) := Some("BIDMTMath.SimpleMath")
```

Try running the main method in SimpleMath.scala. Change the main method so that it adds the numbers given as argument and test it by running the method with different arguments.

Remember that you can also import a package and run the methods in the SBT console:

```
sbt console
```

```
// Import the package:
import package_name._
// Call the function:
Program_name.function_name
```

Problems

Add a new package to this project and a new file within that package. In each problem, you need to implement a function in your new file that takes a certain input and returns a certain output. Use recursion and pattern matching whenever possible to practice using them.

You will possibly need to use the following list functions to solve the problems: reverse, span, length, head, tail, last, init, map, flatMap, and flatten. More information about these function can be found @ https://www.scala-lang.org/api/current/scala/collection/immutable/List.html

The problems are specified as follows:

1. P05 - Reverse a list (using two approaches: one that exploits the list "reverse" function and another recursive function that does not use "reverse")

```
def reverse(l:List[Int]):List[Int] = ???
reverse(List(1, 1, 2, 3, 5, 8))
res0: List[Int] = List(8, 5, 3, 2, 1, 1)
```

2. P06 - Find out whether a list is a palindrome

```
def isPalindrome(list: List[Int]):Boolean = ???
isPalindrome(List(1, 2, 3, 2, 1))
res0: Boolean = true
```

3. P09 - Pack consecutive duplicates of list elements into sublists

If a list contains repeated elements they should be placed in separate sublists.

```
def pack[A](ls: List[A]): List[List[A]] = ???
pack(List('a', 'a', 'a', 'a', 'b', 'c', 'c', 'a', 'a', 'd', 'e', 'e', 'e', 'e'))
res0: List[List[Char]] = List(List('a', 'a', 'a', 'a'), List('b'), List('c', 'c'),
List('a', 'a'), List('d'), List('e', 'e', 'e', 'e'))
```

4. P10 - Run-length encoding of a list

Use the result of problem P09 to implement the so-called run-length encoding data compression method. Consecutive duplicates of elements are encoded as tuples (N, E) where N is the number of duplicates of the element E. You need to also try writing a function similar to pack that returns the required encoding.

```
def encode[A](ls: List[A]):List[(Int, A)] = ???
encode(List('a', 'a', 'a', 'a', 'b', 'c', 'c', 'a', 'a', 'd', 'e', 'e', 'e', 'e'))
res0: List[(Int, Char)] = List((4,'a'), (1,'b'), (2,'c'), (2,'a'), (1,'d'),
(4,'e'))
```

5. P12 - Decode a run-length encoded list

Given a run-length code list generated as specified in problem P10, construct its uncompressed version. Try to explore using flatMap, map and flatten, List.fill, and writing your own makeList that has the same effect of List.fill.

```
def decode[A](ls: List[(Int,A)]) = ???
decode(List((4, 'a'), (1, 'b'), (2, 'c'), (2, 'a'), (1, 'd'), (4, 'e')))
res0: List[Char] = List('a', 'a', 'a', 'a', 'b', 'c', 'c', 'a', 'a', 'd', 'e',
'e', 'e', 'e')
```

The problems are p05, p06, p09, p10 and p12 from the following link: http://aperiodic.net/phil/scala/s-99/