Scala Interview Handbook — Batch 7

Generated: 2025-09-13 02:45:31Z (UTC)

Scala Theory & Cheatsheet

```
SCALA THEORY & CHEATSHEET (Quick Ref)
-------------------------------------
- Syntax: vals vs vars; methods; case classes; pattern matching.
- Collections: immutable List/Vector/Map/Set; mutable variants.
- Functional: map/flatMap/filter/fold; Options/Eithers; for-comprehensions.
- Performance: prefer immutable + structural sharing; use arrays for tight loops.
- Testing: ScalaTest AnyFunSuite; assertions; property-based (optional).
```

061. HeapSort

Problem Overview & Strategy

HeapSort — Detailed Explanation Approach: Use appropriate sorting—Merge/Quick/Heap—or selection (min-heap/quickselect). Correctness: Follows standard algorithms with proven invariants. Complexity: typically O(n log n) sort; quickselect average O(n).

Scala Solution

```
package problems

object Problem061HeapSort {
  def sort(a:Array[Int]): Unit = {
    val n=a.length
    var i=n/2-1; while (i>=0) { heapify(a, n, i); i-=1 }
    i=n-1; while (i>=0) {
      val t=a(0); a(0)=a(i); a(i)=t
      heapify(a, i, 0); i-=1
    }
  }
  private def heapify(a:Array[Int], n:Int, i:Int): Unit = {
    var largest=i; val l=2*i+1; val r=2*i+2
    if (l<n && a(l)>a(largest)) largest=l
    if (r<n && a(r)>a(largest)) largest=r
    if (largest!=i) { val t=a(i); a(i)=a(largest); a(largest)=t; heapify(a,n,largest) }
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem061HeapSort

class Problem061HeapSortSpec extends AnyFunSuite {
  test("heap sort") {
    val a = Array(4,1,3,2); Problem061HeapSort.sort(a)
    assert(a.toList == List(1,2,3,4))
  }
}
```

062. BucketSortSimple

Problem Overview & Strategy

BucketSortSimple — Detailed Explanation Approach: Use appropriate sorting—Merge/Quick/Heap—or selection (min-heap/quickselect). Correctness: Follows standard algorithms with proven invariants. Complexity: typically O(n log n) sort; quickselect average O(n).

Scala Solution

```
package problems


object Problem062BucketSortSimple {
  def sort(a:Array[Double]): Array[Double] = {
    val n = a.length
    val buckets = Array.fill(n)(List[Double]())
    for (x <- a) {
      val idx = math.min((x * n).toInt, n-1).max(0)
      buckets(idx) = (x :: buckets(idx)).sorted
    }
    buckets.flatten
  }
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem062BucketSortSimple

class Problem062BucketSortSimpleSpec extends AnyFunSuite {
  test("bucket sort [0,1)") { val r=Problem062BucketSortSimple.sort(Array(0.78,0.17,0.39,0.26,0.72,0.94,0.21,0.12,0.23,
}
```

063. CountingSortSimple

Problem Overview & Strategy

CountingSortSimple — Detailed Explanation Approach: Use appropriate sorting—Merge/Quick/Heap—or selection (min-heap/quickselect). Correctness: Follows standard algorithms with proven invariants. Complexity: typically O(n log n) sort; quickselect average O(n).

Scala Solution

```
package problems

object Problem063CountingSortSimple {
  def sort(a:Array[Int], maxVal:Int): Array[Int] = {
    val count = Array.fill(maxVal+1)(0)
    a.foreach(x => count(x) += 1)
    var idx=0; var v=0
    while (v<=maxVal) { while (count(v) > 0) { a(idx)=v; idx+=1; count(v)-=1 }; v+=1 }
    a
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem063CountingSortSimple

class Problem063CountingSortSimpleSpec extends AnyFunSuite {
  test("counting sort") {
    val a = Array(3,1,2,1,0)
    assert(Problem063CountingSortSimple.sort(a,3).toList == List(0,1,1,2,3))
  }
}
```

064. RadixSortSimple

Problem Overview & Strategy

RadixSortSimple — Detailed Explanation Approach: Use appropriate sorting—Merge/Quick/Heap—or selection (min-heap/quickselect). Correctness: Follows standard algorithms with proven invariants. Complexity: typically O(n log n) sort; quickselect average O(n).

Scala Solution

```
package problems

object Problem064RadixSortSimple {
  def sort(a:Array[Int]): Array[Int] = {
    var max = 0; a.foreach(x => if (x>max) max=x)
    val n = a.length; val out = Array.ofDim[Int](n)
    var exp = 1
    while (max/exp > 0) {
      val cnt = Array.fill(10)(0)
      var i=0; while (i<n) { cnt((a(i)/exp)%10) += 1; i+=1 }
      var d=1; while (d<10) { cnt(d) += cnt(d-1); d+=1 }
      i=n-1; while (i>=0) { val dgt=(a(i)/exp)%10; cnt(dgt)-=1; out(cnt(dgt))=a(i); i-=1 }
      System.arraycopy(out, 0, a, 0, n)
      exp *= 10
    }
    a
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem064RadixSortSimple

class Problem064RadixSortSimpleSpec extends AnyFunSuite {
  test("radix sort") {
    val a = Array(170,45,75,90,802,24,2,66)
    assert(Problem064RadixSortSimple.sort(a).toList == List(2,24,45,66,75,90,170,802))
  }
}
```

065. FlattenNestedList

Problem Overview & Strategy

FlattenNestedList — Detailed Explanation Approach: Table or memoized recursion; define states and transitions (e.g., LIS with patience sorting tails). Correctness: Optimal substructure and overlapping subproblems. Complexity: Typically O(n^2) or O(n log n) depending on optimization.

Scala Solution

```
package problems


object Problem065FlattenNestedList {
  def flatten(lst: List[Any]): List[Any] = lst.flatMap {
    case l: List[_] => flatten(l.asInstanceOf[List[Any]])
    case x => List(x)
  }
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem065FlattenNestedList

class Problem065FlattenNestedListSpec extends AnyFunSuite {
  test("flatten nested") { assert(Problem065FlattenNestedList.flatten(List(1,List(2,List(3)),4)) == List(1,2,3,4)) }
}
```

066. FlattenDictKeys

Problem Overview & Strategy

FlattenDictKeys — Detailed Explanation Approach: Idiomatic Scala solution with clear invariants and tested behavior. Correctness: Proven by invariant reasoning and unit tests. Complexity: See code comments.

Scala Solution

```
package problems


object Problem066FlattenDictKeys {
  def flatten(m: Map[String, Any], prefix:String=""): Map[String,Any] = {
    m.flatMap { case (k,v) =>
      val nk = if (prefix.isEmpty) k else s"$prefix.$k"
      v match {
        case mm: Map[_,_] => flatten(mm.asInstanceOf[Map[String,Any]], nk)
        case other => Map(nk -> other)
      }
    }
  }
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem066FlattenDictKeys

class Problem066FlattenDictKeysSpec extends AnyFunSuite {
  test("flatten dict keys") { val r=Problem066FlattenDictKeys.flatten(Map("a"->Map("b"->1))); assert(r.contains("a.b"))
}
```

067. ZipTwoLists

Problem Overview & Strategy

ZipTwoLists — Detailed Explanation Approach: Sort and sweep with two pointers to shrink/grow sum. Correctness: Sorting allows monotonic movement to approach target sum. Complexity: O(n log n) for sort + O(n) scan.

Scala Solution

```scala
package problems


object Problem067ZipTwoLists {
  def zip2[A,B](a: List[A], b: List[B]): List[(A,B)] = a.zip(b)
}
```

ScalaTest

```scala
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem067ZipTwoLists

class Problem067ZipTwoListsSpec extends AnyFunSuite {
  test("zip") { assert(Problem067ZipTwoLists.zip2(List(1,2), List("a","b")) == List((1,"a"),(2,"b"))) }
}
```

068. GroupByKey

Problem Overview & Strategy

GroupByKey — Detailed Explanation Approach: Idiomatic Scala solution with clear invariants and tested behavior. Correctness: Proven by invariant reasoning and unit tests. Complexity: See code comments.

Scala Solution

```scala
package problems


object Problem068GroupByKey {
  def groupByKey[K,V](pairs: List[(K,V)]): Map[K,List[V]] =
    pairs.groupBy(_._1).view.mapValues(_.map(_._2)).toMap
}
```

ScalaTest

```scala
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem068GroupByKey

class Problem068GroupByKeySpec extends AnyFunSuite {
  test("groupByKey") { val r=Problem068GroupByKey.groupByKey(List(("a",1),("a",2))); assert(r("a")==List(1,2)) }
}
```

069. MapReduceWordCount

Problem Overview & Strategy

MapReduceWordCount — Detailed Explanation Approach: Idiomatic Scala solution with clear invariants and tested behavior. Correctness: Proven by invariant reasoning and unit tests. Complexity: See code comments.

Scala Solution

```
package problems


object Problem069MapReduceWordCount {
  def wordCount(lines: List[String]): Map[String, Int] =
    lines.flatMap(_.toLowerCase.split("\W+").filter(_.nonEmpty))
         .groupBy(identity).view.mapValues(_.size).toMap
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem069MapReduceWordCount

class Problem069MapReduceWordCountSpec extends AnyFunSuite {
  test("word count") { val r=Problem069MapReduceWordCount.wordCount(List("A a", "a b")); assert(r("a")==3 && r("b")==1)
}
```

070. TailFLIke

Problem Overview & Strategy

TailFLIke — Detailed Explanation Approach: Idiomatic Scala solution with clear invariants and tested behavior. Correctness: Proven by invariant reasoning and unit tests. Complexity: See code comments.

Scala Solution

```
package problems


object Problem070TailFLIke {
  def tailLike(lines: List[String], n:Int): List[String] = lines.takeRight(n)
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem070TailFLIke

class Problem070TailFLIkeSpec extends AnyFunSuite {
  test("tailLike") { assert(Problem070TailFLIke.tailLike(List("1","2","3"),2) == List("2","3")) }
}
```