Scala Interview Handbook — Batch 4

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).
```

031. EditDistance

Problem Overview & Strategy

EditDistance — 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 Problem031EditDistance {
  def edit(a: String, b: String): Int = {
    val m=a.length; val n=b.length
    val dp = Array.ofDim[Int](m+1, n+1)
    for (i <- 0 to m) dp(i)(0) = i
    for (j <- 0 to n) dp(0)(j) = j
    for (i <- 1 to m; j <- 1 to n) {
      dp(i)(j) =
        if (a(i-1)==b(j-1)) dp(i-1)(j-1)
        else 1 + math.min(dp(i-1)(j-1), math.min(dp(i-1)(j), dp(i)(j-1)))
    }
    dp(m)(n)
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem031EditDistance

class Problem031EditDistanceSpec extends AnyFunSuite {
  test("edit distance") {
    assert(Problem031EditDistance.edit("kitten","sitting") === 3)
  }
}
```

032. UniquePathsGrid

Problem Overview & Strategy

UniquePathsGrid — 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 Problem032UniquePathsGrid {
  def uniquePaths(m:Int, n:Int): Int = {
    val dp = Array.fill(m,n)(0)
    for (i <- 0 until m) dp(i)(0)=1
    for (j <- 0 until n) dp(0)(j)=1
    for (i <- 1 until m; j <- 1 until n) dp(i)(j) = dp(i-1)(j)+dp(i)(j-1)
    dp(m-1)(n-1)
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem032UniquePathsGrid

class Problem032UniquePathsGridSpec extends AnyFunSuite {
  test("unique paths") {
    assert(Problem032UniquePathsGrid.uniquePaths(3,3) === 6)
  }
}
```

033. CoinChangeMin

Problem Overview & Strategy

CoinChangeMin — 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 Problem033CoinChangeMin {
  def coinChange(coins:Array[Int], amount:Int): Int = {
    val INF = 1_000_000
    val dp = Array.fill(amount+1)(INF)
    dp(0)=0
    for (a <- 1 to amount; c <- coins if c<=a) dp[a] = math.min(dp[a], dp[a-c]+1)
    if (dp(amount) >= INF) -1 else dp(amount)
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem033CoinChangeMin

class Problem033CoinChangeMinSpec extends AnyFunSuite {
  test("coin change") {
    assert(Problem033CoinChangeMin.coinChange(Array(1,2,5),11) === 3)
  }
}
```

034. WordBreak

Problem Overview & Strategy

WordBreak — 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
import scala.collection.mutable

object Problem034WordBreak {
  def wordBreak(s:String, dict:Set[String]): Boolean = {
    val dp = Array.fill(s.length+1)(false)
    dp(0)=true
    for (i <- 1 to s.length) {
      var ok=false
      var j=0
      while (j<i && !ok) {
        if (dp(j) && dict.contains(s.substring(j,i))) ok=true
        j+=1
      }
      dp(i)=ok
    }
    dp(s.length)
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem034WordBreak

class Problem034WordBreakSpec extends AnyFunSuite {
  test("word break") {
    assert(Problem034WordBreak.wordBreak("leetcode", Set("leet","code")))
  }
}
```

035. LongestPalindromicSubstring

Problem Overview & Strategy

LongestPalindromicSubstring — Detailed Explanation Approach: Use hash maps / frequency arrays and linear scans. Correctness: Frequencies capture necessary character counts; single pass maintains invariants. Complexity: O(n) time, O(Σ) space.

Scala Solution

```
package problems

object Problem035LongestPalindromicSubstring {
  def lps(s: String): String = {
    if (s==null || s.isEmpty) return ""
    var best=(0,0)
    def expand(L:Int,R:Int): (Int,Int) = {
      var l=L; var r=R
      while (l>=0 && r<s.length && s(l)==s(r)) { l-=1; r+=1 }
      (l+1,r-1)
    }
    for (i <- s.indices) {
      val a = expand(i,i); val b = expand(i,i+1)
      val pick = if (a._2-a._1 >= b._2-b._1) a else b
      if (pick._2-pick._1 > best._2-best._1) best = pick
    }
    s.substring(best._1, best._2+1)
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem035LongestPalindromicSubstring

class Problem035LongestPalindromicSubstringSpec extends AnyFunSuite {
  test("lps") {
    val r = Problem035LongestPalindromicSubstring.lps("babad")
    assert(r == "bab" || r == "aba")
  }
}
```

036. SerializeDeserializeTree

Problem Overview & Strategy

SerializeDeserializeTree — Detailed Explanation Approach: Recursive traversals (in/pre/post), stack-based inorder for BST k-th, LCA via divide-and-conquer, Trie with hash children. Correctness: Traversal properties and BST invariants ensure correctness. Complexity: O(n) time, O(h) stack.

Scala Solution

```
package problems

object Problem036SerializeDeserializeTree {
  final class Node(var v:Int, var l:Node, var r:Node)
  object Node { def apply(v:Int): Node = new Node(v, null, null) }

  def serialize(root: Node): String = {
    val sb = new StringBuilder
    def ser(n:Node): Unit = {
      if (n==null) { sb.append("#,"); return }
      sb.append(n.v).append(','); ser(n.l); ser(n.r)
    }
    ser(root); sb.toString
  }
  def deserialize(data:String): Node = {
    val t = data.split(",")
    val idx = Array(0)
    def des(): Node = {
      if (idx(0) >= t.length) return null
      val v = t(idx(0)); idx(0) += 1
      if (v=="" || v=="#") return null
      val n = Node(v.toInt); n.l = des(); n.r = des(); n
    }
    des()
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem036SerializeDeserializeTree

class Problem036SerializeDeserializeTreeSpec extends AnyFunSuite {
  test("serdes") {
    val r = Problem036SerializeDeserializeTree.Node(1); r.l=Problem036SerializeDeserializeTree.Node(2); r.r=Problem036S
    val s = Problem036SerializeDeserializeTree.serialize(r)
    assert(Problem036SerializeDeserializeTree.deserialize(s) ne null)
  }
}
```

037. BinaryTreeInorder

Problem Overview & Strategy

BinaryTreeInorder — Detailed Explanation Approach: Recursive traversals (in/pre/post), stack-based inorder for BST k-th, LCA via divide-and-conquer, Trie with hash children. Correctness: Traversal properties and BST invariants ensure correctness. Complexity: O(n) time, O(h) stack.

Scala Solution

```
package problems
object Problem037BinaryTreeInorder {
  final class Node(var v:Int, var l:Node, var r:Node)
  def traverse(n:Node): List[Int] = if (n==null) Nil else traverse(n.l) ::: List(n.v) ::: traverse(n.r)
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem037BinaryTreeInorder

class Problem037BinaryTreeInorderSpec extends AnyFunSuite {
  test("inorder") { val r=new Problem037BinaryTreeInorder.Node(2,new Problem037BinaryTreeInorder.Node(1,null,null),new
}
```

038. BinaryTreePreorder

Problem Overview & Strategy

BinaryTreePreorder — Detailed Explanation Approach: Recursive traversals (in/pre/post), stack-based inorder for BST k-th, LCA via divide-and-conquer, Trie with hash children. Correctness: Traversal properties and BST invariants ensure correctness. Complexity: O(n) time, O(h) stack.

Scala Solution

```
package problems
object Problem038BinaryTreePreorder {
  final class Node(var v:Int, var l:Node, var r:Node)
  def traverse(n:Node): List[Int] = if (n==null) Nil else List(n.v) ::: traverse(n.l) ::: traverse(n.r)
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem038BinaryTreePreorder

class Problem038BinaryTreePreorderSpec extends AnyFunSuite {
  test("preorder") { val r=new Problem038BinaryTreePreorder.Node(2,new Problem038BinaryTreePreorder.Node(1,null,null),n
}
```

039. BinaryTreePostorder

Problem Overview & Strategy

BinaryTreePostorder — Detailed Explanation Approach: Recursive traversals (in/pre/post), stack-based inorder for BST k-th, LCA via divide-and-conquer, Trie with hash children. Correctness: Traversal properties and BST invariants ensure correctness. Complexity: O(n) time, O(h) stack.

Scala Solution

```
package problems
object Problem039BinaryTreePostorder {
  final class Node(var v:Int, var l:Node, var r:Node)
  def traverse(n:Node): List[Int] = if (n==null) Nil else traverse(n.l) ::: traverse(n.r) ::: List(n.v)
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem039BinaryTreePostorder

class Problem039BinaryTreePostorderSpec extends AnyFunSuite {
  test("postorder") { val r=new Problem039BinaryTreePostorder.Node(2,new Problem039BinaryTreePostorder.Node(1,null,null
}
```

040. IsBalancedTree

Problem Overview & Strategy

IsBalancedTree — Detailed Explanation Approach: Recursive traversals (in/pre/post), stack-based inorder for BST k-th, LCA via divide-and-conquer, Trie with hash children. Correctness: Traversal properties and BST invariants ensure correctness. Complexity: O(n) time, O(h) stack.

Scala Solution

```
package problems

object Problem040IsBalancedTree {
  final class Node(var v:Int, var l:Node, var r:Node)
  def isBalanced(n: Node): Boolean = height(n) != -1
  private def height(n: Node): Int = {
    if (n == null) return 0
    val lh = height(n.l); if (lh == -1) return -1
    val rh = height(n.r); if (rh == -1) return -1
    if (math.abs(lh - rh) > 1) -1 else math.max(lh, rh) + 1
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem040IsBalancedTree

class Problem040IsBalancedTreeSpec extends AnyFunSuite {
  test("balanced") {
    val r = new Problem040IsBalancedTree.Node(1, new Problem040IsBalancedTree.Node(2,null,null), null)
    assert(Problem040IsBalancedTree.isBalanced(r))
  }
}
```