Scala Interview Handbook — Batch 5

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).
```

041. LowestCommonAncestor

Problem Overview & Strategy

LowestCommonAncestor — 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 Problem041LowestCommonAncestor {
  final class Node(var v:Int, var l:Node, var r:Node)
  def lca(root: Node, p: Node, q: Node): Node = {
    if (root==null || root eq p || root eq q) return root
    val L = lca(root.l, p, q); val R = lca(root.r, p, q)
    if (L!=null && R!=null) root else if (L!=null) L else R
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem041LowestCommonAncestor

class Problem041LowestCommonAncestorSpec extends AnyFunSuite {
  test("lca") {
    val r = new Problem041LowestCommonAncestor.Node(3, null, null)
    r.l = new Problem041LowestCommonAncestor.Node(5, null, null); r.r = new Problem041LowestCommonAncestor.Node(1, null
    assert(Problem041LowestCommonAncestor.lca(r, r.l, r.r).v == 3)
  }
}
```

042. TrieInsertSearch

Problem Overview & Strategy

TrieInsertSearch — Detailed Explanation Approach: Classic binary search over a sorted array (or search-space). Correctness: Midpoint halving preserves invariant that target lies in [lo,hi]. Complexity: O(log n) time, O(1) space.

Scala Solution

```
package problems
import scala.collection.mutable

object Problem042TrieInsertSearch {
  final class TrieNode { val c = mutable.HashMap.empty[Char, TrieNode]; var end=false }
  final class Trie {
    private val root = new TrieNode
    def insert(w:String): Unit = {
      var n = root
      w.foreach { ch => n = n.c.getOrElseUpdate(ch, new TrieNode) }
      n.end = true
    }
    def search(w:String): Boolean = {
      var n = root
      for (ch <- w) {
        if (!n.c.contains(ch)) return false
        n = n.c(ch)
      }
      n.end
    }
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem042TrieInsertSearch

class Problem042TrieInsertSearchSpec extends AnyFunSuite {
  test("trie") {
    val t = new Problem042TrieInsertSearch.Trie
    t.insert("hi")
    assert(t.search("hi") && !t.search("h"))
  }
}
```

043. DijkstraSimple

Problem Overview & Strategy

DijkstraSimple — Detailed Explanation Approach: BFS/DFS for traversal & cycle detection; Kahn for topological order; Dijkstra with a min-heap. Correctness: Standard graph invariants (visited sets, indegrees, relaxation) guarantee optimality. Complexity: O(V+E) for BFS/DFS/Topo; O((V+E) log V) for Dijkstra.

Scala Solution

```
package problems
import scala.collection.mutable

object Problem043DijkstraSimple {
  def dijkstra(g: Map[String, List[(String, Int)]], src:String): Map[String, Int] = {
    val dist = mutable.Map[String, Int]().withDefaultValue(Int.MaxValue/4)
    dist(src)=0
    val pq = mutable.PriorityQueue.empty[(Int,String)](Ordering.by[(Int,String),Int](-_._1))
    pq.enqueue((0,src))
    while (pq.nonEmpty) {
      val (d,u) = pq.dequeue()
      if (d <= dist(u)) {
        for ((v,w) <- g.getOrElse(u, Nil)) {
          val nd = d + w
          if (nd < dist(v)) { dist(v)=nd; pq.enqueue((nd,v)) }
        }
      }
    }
    dist.toMap
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem043DijkstraSimple

class Problem043DijkstraSimpleSpec extends AnyFunSuite {
  test("dijkstra") {
    val g = Map("A" -> List(("B",1)), "B" -> List(("C",2)), "C" -> Nil)
    assert(Problem043DijkstraSimple.dijkstra(g,"A")("C") === 3)
  }
}
```

044. BfsGraph

Problem Overview & Strategy

BfsGraph — Detailed Explanation Approach: BFS/DFS for traversal & cycle detection; Kahn for topological order; Dijkstra with a min-heap. Correctness: Standard graph invariants (visited sets, indegrees, relaxation) guarantee optimality. Complexity: O(V+E) for BFS/DFS/Topo; O((V+E) log V) for Dijkstra.

Scala Solution

```
package problems
import scala.collection.mutable

object Problem044BfsGraph {
  def bfs(g: Map[Int, List[Int]], s:Int): List[Int] = {
    val seen = mutable.LinkedHashSet[Int](s)
    val q = mutable.Queue[Int](s)
    while (q.nonEmpty) {
      val u = q.dequeue()
      for (v <- g.getOrElse(u, Nil) if !seen.contains(v)) {
        seen += v; q.enqueue(v)
      }
    }
    seen.toList
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem044BfsGraph

class Problem044BfsGraphSpec extends AnyFunSuite {
  test("bfs") {
    val g = Map(1->List(2,3), 2->List(4), 3->Nil, 4->Nil)
    assert(Problem044BfsGraph.bfs(g,1) == List(1,2,3,4))
  }
}
```

045. DfsGraphRecursive

Problem Overview & Strategy

DfsGraphRecursive — Detailed Explanation Approach: BFS/DFS for traversal & cycle detection; Kahn for topological order; Dijkstra with a min-heap. Correctness: Standard graph invariants (visited sets, indegrees, relaxation) guarantee optimality. Complexity: O(V+E) for BFS/DFS/Topo; O((V+E) log V) for Dijkstra.

Scala Solution

```
package problems
object Problem045DfsGraphRecursive {
  def dfs(g: Map[Int,List[Int]], start:Int): List[Int] = {
    val seen = scala.collection.mutable.LinkedHashSet[Int]()
    def rec(u:Int): Unit = if (!seen(u)) { seen += u; g.getOrElse(u,Nil).foreach(rec) }
    rec(start); seen.toList
  }
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem045DfsGraphRecursive

class Problem045DfsGraphRecursiveSpec extends AnyFunSuite {
  test("dfs") { val g=Map(1->List(2,3),2->List(4),3->Nil,4->Nil); assert(Problem045DfsGraphRecursive.dfs(g,1)==List(1,2
}
```

046. SieveEratosthenes

Problem Overview & Strategy

SieveEratosthenes — 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 Problem046SieveEratosthenes {
  def sieve(n:Int): List[Int] = {
    val p = Array.fill(n+1)(true)
    if (n>=0) p(0)=false; if (n>=1) p(1)=false
    var i=2; while (i*i<=n) {
      if (p(i)) { var j=i*i; while (j<=n) { p(j)=false; j+=i } }
      i+=1
    }
    (2 to n).filter(p).toList
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem046SieveEratosthenes

class Problem046SieveEratosthenesSpec extends AnyFunSuite {
  test("sieve 10") {
    assert(Problem046SieveEratosthenes.sieve(10) === List(2,3,5,7))
  }
}
```

047. NQueensCount

Problem Overview & Strategy

NQueensCount — 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 Problem047NQueensCount {
  def count(n:Int): Int = {
    val cols = new Array[Boolean](n)
    val d1 = new Array[Boolean](2*n)
    val d2 = new Array[Boolean](2*n)
    def bt(r:Int): Int = {
      if (r==n) 1
      else (0 until n).filter(c => !cols(c) && !d1(r+c) && !d2(r-c+n)).map { c =>
        cols(c)=true; d1(r+c)=true; d2(r-c+n)=true
        val v=bt(r+1)
        cols(c)=false; d1(r+c)=false; d2(r-c+n)=false
        v
      }.sum
    }
    bt(0)
  }
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem047NQueensCount

class Problem047NQueensCountSpec extends AnyFunSuite {
  test("n-queens 4 -> 2") { assert(Problem047NQueensCount.count(4) == 2) }
}
```

048. SudokuSolver

Problem Overview & Strategy

SudokuSolver — 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 Problem048SudokuSolver {
  def solve(board:Array[Array[Char]]): Boolean = {
    def ok(r:Int,c:Int,ch:Char): Boolean = {
      for(i<-0 until 9) {
        if(board(r)(i)==ch || board(i)(c)==ch) return false
      }
      val br=(r/3)*3; val bc=(c/3)*3
      for(i<-0 until 3; j<-0 until 3) if(board(br+i)(bc+j)==ch) return false
      true
    }
    def backtrack(pos:Int): Boolean = {
      if(pos==81) return true
      val r=pos/9; val c=pos%9
      if(board(r)(c)!='.') return backtrack(pos+1)
      for(ch <- "123456789") {
        if(ok(r,c,ch)) { board(r)(c)=ch; if(backtrack(pos+1)) return true; board(r)(c)='.' }
      }
      false
    }
    backtrack(0)
  }
}
```

ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem048SudokuSolver

class Problem048SudokuSolverSpec extends AnyFunSuite {
  test("sudoku solvable") { val b=Array.fill(9)(Array.fill(9)('.')); assert(Problem048SudokuSolver.solve(b)) }
}
```

049. RotateMatrix90

Problem Overview & Strategy

RotateMatrix90 — Detailed Explanation Approach: Bounds pointers (top/bottom/left/right) for spiral; transpose+reverse for rotation; classic i-k-j loops for multiplication. Correctness: Each layer/element is moved/visited exactly once. Complexity: O(mn) time.

Scala Solution

```
package problems

object Problem049RotateMatrix90 {
  def rotate(m:Array[Array[Int]]): Unit = {
    val n = m.length
    var i=0; while (i<n) {
      var j=i; while (j<n) {
        val t = m(i)(j); m(i)(j)=m(j)(i); m(j)(i)=t
        j+=1
      }
      i+=1
    }
    var r=0; while (r<n) {
      var l=0; var h=n-1
      while (l<h) { val t=m(r)(l); m(r)(l)=m(r)(h); m(r)(h)=t; l+=1; h-=1 }
      r+=1
    }
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem049RotateMatrix90

class Problem049RotateMatrix90Spec extends AnyFunSuite {
  test("rotate 2x2") {
    val m = Array(Array(1,2), Array(3,4))
    Problem049RotateMatrix90.rotate(m)
    assert(m.map(_.toList).toList == List(List(3,1), List(4,2)))
  }
}
```

050. MatrixMultiply

Problem Overview & Strategy

MatrixMultiply — Detailed Explanation Approach: Bounds pointers (top/bottom/left/right) for spiral; transpose+reverse for rotation; classic i-k-j loops for multiplication. Correctness: Each layer/element is moved/visited exactly once. Complexity: O(mn) time.

Scala Solution

```
package problems

object Problem050MatrixMultiply {
  def multiply(A:Array[Array[Int]], B:Array[Array[Int]]): Array[Array[Int]] = {
    val m=A.length; val n=A(0).length; val p=B(0).length
    val C = Array.ofDim[Int](m, p)
    var i=0; while (i<m) {
      var k=0; while (k<n) {
        if (A[i][k] != 0) {
          var j=0; while (j<p) { C[i][j] += A[i][k] * B[k][j]; j+=1 }
        }
        k+=1
      }
      i+=1
    }
    C
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem050MatrixMultiply

class Problem050MatrixMultiplySpec extends AnyFunSuite {
  test("multiply 2x2") {
    val A = Array(Array(1,2), Array(3,4))
    val B = Array(Array(5,6), Array(7,8))
    val R = Problem050MatrixMultiply.multiply(A,B)
    assert(R(0).toList == List(19,22))
    assert(R(1).toList == List(43,50))
  }
}
```