Scala Interview Handbook — Batch 11

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).
```

100. EmailValidationRegex

Problem Overview & Strategy

EmailValidationRegex — Detailed Explanation Approach: Use precompiled regex patterns with sane anchors and character classes. Correctness: Patterns encode structural rules; not a full RFC check but covers common cases. Complexity: O(n) matching.

Scala Solution

```
package problems
import java.util.regex.Pattern

object Problem100EmailValidationRegex {
  private val P = Pattern.compile("^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$")
  def isValid(email:String): Boolean = email!=null && P.matcher(email).matches()
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem100EmailValidationRegex

class Problem100EmailValidationRegexSpec extends AnyFunSuite {
  test("email validation") {
    assert(Problem100EmailValidationRegex.isValid("a@b.com"))
    assert(!Problem100EmailValidationRegex.isValid("not-an-email"))
  }
}
```

101. FindKthSmallestBst

Problem Overview & Strategy

FindKthSmallestBst — 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
import scala.collection.mutable

object Problem101FindKthSmallestBst {
  final class Node(var v:Int, var l:Node, var r:Node)
  def kthSmallest(root: Node, k:Int): Int = {
    val st = new mutable.ArrayDeque[Node]()
    var cur: Node = root; var count = 0
    while (cur!=null || st.nonEmpty) {
      while (cur!=null) { st.append(cur); cur = cur.l }
      cur = st.removeLast()
      count += 1
      if (count==k) return cur.v
      cur = cur.r
    }
    -1
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem101FindKthSmallestBst

class Problem101FindKthSmallestBstSpec extends AnyFunSuite {
  test("kth smallest BST") {
    val r = new Problem101FindKthSmallestBst.Node(2, new Problem101FindKthSmallestBst.Node(1,null,null), new Problem101
    assert(Problem101FindKthSmallestBst.kthSmallest(r,2) === 2)
  }
}
```

102. CountSetBits

Problem Overview & Strategy

CountSetBits — Detailed Explanation Approach: Kernighan’s trick for bit counts; XOR & popcount for Hamming; reflect-and-append for Gray code. Correctness: Bit identities and definitions. Complexity: O(#set-bits) or O(2^n) for sequence generation.

Scala Solution

```
package problems

object Problem102CountSetBits {
  def countBits(x:Int): Int = {
    var n=x; var c=0
    while (n!=0) { n &= (n-1); c += 1 }
    c
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem102CountSetBits

class Problem102CountSetBitsSpec extends AnyFunSuite {
  test("count bits") {
    assert(Problem102CountSetBits.countBits(0b1011) === 3)
  }
}
```

103. HammingDistance

Problem Overview & Strategy

HammingDistance — Detailed Explanation Approach: Kernighan’s trick for bit counts; XOR & popcount for Hamming; reflect-and-append for Gray code. Correctness: Bit identities and definitions. Complexity: O(#set-bits) or O(2^n) for sequence generation.

Scala Solution

```
package problems

object Problem103HammingDistance {
  def hamming(a:Int, b:Int): Int = {
    var x = a ^ b; var c=0
    while (x!=0) { x &= (x-1); c += 1 }
    c
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem103HammingDistance

class Problem103HammingDistanceSpec extends AnyFunSuite {
  test("hamming") {
    assert(Problem103HammingDistance.hamming(1,4) === 2)
  }
}
```

104. GrayCode

Problem Overview & Strategy

GrayCode — Detailed Explanation Approach: Kernighan’s trick for bit counts; XOR & popcount for Hamming; reflect-and-append for Gray code. Correctness: Bit identities and definitions. Complexity: O(#set-bits) or O(2^n) for sequence generation.

Scala Solution

```
package problems
import scala.collection.mutable.ArrayBuffer

object Problem104GrayCode {
  def grayCode(n:Int): List[Int] = {
    val res = ArrayBuffer[Int](0)
    var i=0; while (i<n) {
      val add = 1<<i
      var j=res.size-1; while (j>=0) { res += (res(j) + add); j-=1 }
      i+=1
    }
    res.toList
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem104GrayCode

class Problem104GrayCodeSpec extends AnyFunSuite {
  test("gray code 2") {
    assert(Problem104GrayCode.grayCode(2) == List(0,1,3,2))
  }
}
```

105. PowerSetIterative

Problem Overview & Strategy

PowerSetIterative — 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.ArrayBuffer

object Problem105PowerSetIterative {
  def powerSet(nums:Array[Int]): List[List[Int]] = {
    val res = ArrayBuffer[List[Int]](Nil)
    for (x <- nums) {
      val cur = res.toList
      cur.foreach(s => res += (s :+ x))
    }
    res.toList
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem105PowerSetIterative

class Problem105PowerSetIterativeSpec extends AnyFunSuite {
  test("ps iterative size") {
    assert(Problem105PowerSetIterative.powerSet(Array(1,2,3)).size === 8)
  }
}
```