Scala Interview Handbook — Batch 2

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).
```

011. BinarySearch

Problem Overview & Strategy

BinarySearch — 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

object Problem011BinarySearch {
  def search(a: Array[Int], target: Int): Int = {
    var lo = 0; var hi = a.length - 1
    while (lo <= hi) {
      val mid = (lo + hi) >>> 1
      if (a(mid) == target) return mid
      else if (a(mid) < target) lo = mid + 1
      else hi = mid - 1
    }
    -1
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem011BinarySearch

class Problem011BinarySearchSpec extends AnyFunSuite {
  test("search") {
    assert(Problem011BinarySearch.search(Array(1,2,3,4), 3) === 2)
  }
}
```

012. MergeSort

Problem Overview & Strategy

MergeSort — 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 Problem012MergeSort {
  def sort(a: Array[Int]): Array[Int] = {
    if (a.length <= 1) return a.clone
    val m = a.length / 2
    val left = sort(a.slice(0, m))
    val right = sort(a.slice(m, a.length))
    merge(left, right)
  }
  private def merge(l: Array[Int], r: Array[Int]): Array[Int] = {
    val res = Array.ofDim[Int](l.length + r.length)
    var i=0; var j=0; var k=0
    while (i<l.length && j<r.length) {
      if (l(i) <= r(j)) { res(k)=l(i); i+=1 }
      else { res(k)=r(j); j+=1 }
      k+=1
    }
    while (i<l.length) { res(k)=l(i); i+=1; k+=1 }
    while (j<r.length) { res(k)=r(j); j+=1; k+=1 }
    res
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem012MergeSort

class Problem012MergeSortSpec extends AnyFunSuite {
  test("mergesort") {
    assert(Problem012MergeSort.sort(Array(3,1,2)).sameElements(Array(1,2,3)))
  }
}
```

013. QuickSort

Problem Overview & Strategy

QuickSort — 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 Problem013QuickSort {
  def sort(a: Array[Int]): Unit = qs(a, 0, a.length-1)
  private def qs(a: Array[Int], l: Int, r: Int): Unit = {
    if (l >= r) return
    val p = a((l+r)/2)
    var i = l; var j = r
    while (i <= j) {
      while (a(i) < p) i += 1
      while (a(j) > p) j -= 1
      if (i <= j) { val t=a(i); a(i)=a(j); a(j)=t; i+=1; j-=1 }
    }
    qs(a, l, j); qs(a, i, r)
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem013QuickSort

class Problem013QuickSortSpec extends AnyFunSuite {
  test("quicksort") {
    val a = Array(3,1,2); Problem013QuickSort.sort(a); assert(a.sameElements(Array(1,2,3)))
  }
}
```

014. KthLargest

Problem Overview & Strategy

KthLargest — 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 Problem014KthLargest {
  def kthLargest(a:Array[Int], k:Int): Int = {
    val pq = mutable.PriorityQueue.empty[Int](Ordering.Int.reverse)
    a.foreach { x =>
      pq.enqueue(x)
      if (pq.size > k) pq.dequeue()
    }
    pq.head
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem014KthLargest

class Problem014KthLargestSpec extends AnyFunSuite {
  test("kth largest") {
    assert(Problem014KthLargest.kthLargest(Array(3,2,1,5,6,4),3) === 3)
  }
}
```

015. KadaneMaxSubarray

Problem Overview & Strategy

KadaneMaxSubarray — 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 Problem015KadaneMaxSubarray {
  def maxSubarray(a:Array[Int]): Int = {
    var best=a(0); var cur=a(0)
    for (i <- 1 until a.length) {
      cur = math.max(a(i), cur + a(i))
      best = math.max(best, cur)
    }
    best
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem015KadaneMaxSubarray

class Problem015KadaneMaxSubarraySpec extends AnyFunSuite {
  test("kadane") {
    assert(Problem015KadaneMaxSubarray.maxSubarray(Array(-2,1,-3,4,-1,2,1,-5,4)) === 6)
  }
}
```

016. Permutations

Problem Overview & Strategy

Permutations — 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 Problem016Permutations {
  def permute(nums:Array[Int]): List[List[Int]] = {
    val res = ArrayBuffer[List[Int]]()
    val used = Array.fill(nums.length)(false)
    def bt(cur: List[Int]): Unit = {
      if (cur.length==nums.length) res += cur
      else {
        for (i <- nums.indices if !used[i]) {
          used[i]=true; bt(cur :+ nums(i)); used[i]=false
        }
      }
    }
    bt(Nil); res.toList
  }
}
```

ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem016Permutations

class Problem016PermutationsSpec extends AnyFunSuite {
  test("permute size") {
    assert(Problem016Permutations.permute(Array(1,2,3)).size === 6)
  }
}
```

017. Combinations

Problem Overview & Strategy

Combinations — 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
import scala.collection.mutable.ArrayBuffer

object Problem017Combinations {
  def combine(n:Int, k:Int): List[List[Int]] = {
    val res = ArrayBuffer[List[Int]]()
    def bt(start:Int, cur: List[Int]): Unit = {
      if (cur.length==k) res += cur
      else for (i <- start to n) bt(i+1, cur :+ i)
    }
    bt(1, Nil); res.toList
  }
}
```

ScalaTest

```scala
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem017Combinations

class Problem017CombinationsSpec extends AnyFunSuite {
  test("combine size") {
    assert(Problem017Combinations.combine(4,2).size === 6)
  }
}
```

018. PowerSet

Problem Overview & Strategy

PowerSet — 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 Problem018PowerSet {
  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.Problem018PowerSet

class Problem018PowerSetSpec extends AnyFunSuite {
  test("power set size") {
    assert(Problem018PowerSet.powerSet(Array(1,2,3)).size === 8)
  }
}
```

019. FibMemo

Problem Overview & Strategy

FibMemo — 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 Problem019FibMemo {
  private val memo = scala.collection.mutable.HashMap[Int,Long]()
  def fib(n:Int): Long = memo.getOrElseUpdate(n, if (n<2) n else fib(n-1)+fib(n-2))
}
```

ScalaTest

```scala
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem019FibMemo

class Problem019FibMemoSpec extends AnyFunSuite {
  test("fib memo") { assert(Problem019FibMemo.fib(10) == 55L) }
}
```

020. NthFibonacciIterative

Problem Overview & Strategy

NthFibonacciIterative — 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 Problem020NthFibonacciIterative {
  def fibN(n:Int): Long = { if (n<2) n else (2 to n).foldLeft((0L,1L)){case ((a,b),_) => (b,a+b)}._2 }
}
```

ScalaTest

```scala
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem020NthFibonacciIterative

class Problem020NthFibonacciIterativeSpec extends AnyFunSuite {
  test("fib iterative") { assert(Problem020NthFibonacciIterative.fibN(10) == 55L) }
}
```