```scala
package mastermind

import tinyscalautils.control.interruptibly

import scala.compiletime.uninitialized
import scala.math.Ordered.orderingToOrdered

object MasterMind:
   opaque type Pattern = Int
   opaque type ScoredPattern = Int
   opaque type Score = Int

   extension (pattern: Pattern)
      def colorAt(position: Int): Int = (pattern >> (3 * position)) & 0x7
      def + (color: Int): Pattern = (pattern << 3) | color
      def scored(score: Score): ScoredPattern = (pattern & 0x00FFFFFF) | (score << 24)

      def display(len: Int): String =
        val sb = new StringBuilder
        var i = len - 1
        while i >= 0 do
          sb.append(colorAt(i))
          i -= 1
        sb.toString

   end extension

   extension (pattern: ScoredPattern)
      def score: Score =  (pattern >>> 24)
      def unScored: Pattern = (pattern & 0x00FFFFFF)
   end extension

   extension (score: Score)
      def blacks: Int = (score >> 4) & 0xF
      def whites: Int = score & 0xF
```

```
36       def display: String = "B" * blacks + "W" * whites
37    end extension
38
39    def makeScore(blacks: Int, whites: Int): Score = ((
   blacks << 4) | whites)
40
41    def makePattern(colors: Int*): Pattern =
   interruptibly:
42       var pattern = 0
43       for color <- colors do pattern = MasterMind.+(
   pattern)(color)
44       pattern
45
46    given Ordering[Pattern] = Ordering.Int
47 end MasterMind
48
49 class MasterMind(val lengthOfSecret: Int, val
   numberOfColors: Int):
50    import MasterMind.*
51
52    require(lengthOfSecret > 0 && lengthOfSecret <= 8)
53    require(numberOfColors > 1 && numberOfColors <= 8)
54
55    private var startTable: Option[Map[Pattern, Seq[
   ScoredPattern]]] = None
56    private var firstGuess: Pattern
                                     = uninitialized
57    private var currentTable: Map[Pattern, Seq[
   ScoredPattern]]       = uninitialized
58    private var currentGuess: Pattern
                                   = uninitialized
59
60    // Sets initial guess to a "good" value
61    setFirstGuess(Inits.inits(lengthOfSecret,
   numberOfColors))
62
63    def checked(pattern: Pattern): Pattern =
64       for i <- 0 until 8 do
```

```
65          val color = pattern.colorAt(i)
66          if i < lengthOfSecret then require(color < numberOfColors, "invalid color: %d", color)
67          else require(color == 0, "too many colors")
68       pattern
69
70    def reset(): Unit =
71       currentTable = createTable
72       currentGuess = firstGuess
73
74    def fastReset(): Unit =
75       if startTable.isEmpty then startTable = Some(createTable)
76       currentTable = startTable.get
77       currentGuess = firstGuess
78
79    def setFirstGuess(guess: Pattern): Unit = firstGuess = checked(guess)
80
81    def guess: Pattern = currentGuess
82
83    def reply(score: Score): Pattern =
84       val remaining = possibles(currentTable, currentGuess, score)
85       currentTable = reduceTable(currentTable, remaining)
86       currentGuess = best(currentTable, remaining)(2)
87       currentGuess
88
89    ////////////////////////////////////
90    // Core implementation starts here //
91    ////////////////////////////////////
92
93    // calculateScore is the computation bottleneck, so it's written "C-style".
94    private val guessColors, secretColors = Array.ofDim[Int](numberOfColors)
95
```

```
 96    def calculateScore(guess: Pattern, secret: Pattern
    ): Score =
 97       java.util.Arrays.fill(guessColors, 0)
 98       java.util.Arrays.fill(secretColors, 0)
 99       var i, blacks, common = 0
100       while i < lengthOfSecret do
101          val g = guess.colorAt(i)
102          val s = secret.colorAt(i)
103          if g == s then blacks += 1
104          guessColors(g) += 1
105          secretColors(s) += 1
106          i += 1
107       end while
108       i = 0
109       while i < numberOfColors do
110          common += guessColors(i) min secretColors(i)
111          i += 1
112       end while
113       makeScore(blacks, whites = common - blacks)
114
115    def allPatterns: Seq[Pattern] = {
116      var patterns: Seq[Pattern] = Seq(makePattern())
117
118      for _ <- 0 until lengthOfSecret do
119        patterns =
120          for
121            p <- patterns
122            c <- 0 until numberOfColors
123          yield p + c
124
125      patterns
126    }
127
128
129    def createTable: Map[Pattern, Seq[ScoredPattern
    ]] = {
130      val patterns = allPatterns
131
```

```
132       (for (guess <- patterns) yield {
133         val scored =
134           for
135             secret <- patterns
136           yield secret.scored(calculateScore(guess,
    secret))
137         guess -> scored
138       }).toMap
139     }
140
141
142     def possibles(table: Map[Pattern, Seq[ScoredPattern
    ]], guess: Pattern, score: Score): Set[Pattern] = {
143       (for
144         sp <- table(guess)
145         if sp.score == score
146       yield sp.unScored).toSet
147     }
148
149     def reduceTable(
150         table: Map[Pattern, Seq[ScoredPattern]],
151         remaining: Set[Pattern]
152     ): Map[Pattern, Seq[ScoredPattern]] = {
153       (for
154         (guess, scored) <- table
155       yield
156         val filtered =
157           for
158             sp <- scored
159             if remaining.contains(sp.unScored)
160           yield sp
161         guess -> filtered
162         ).toMap
163     }
164
165     def groupByScore(patterns: Seq[ScoredPattern]): Map
    [Score, Seq[ScoredPattern]] =
166       patterns.groupBy(_.score)
```

```
167
168     def worst(groups: Map[Score, Seq[ScoredPattern]]):
    Int =
169       var max = 0
170       for (_, seq) <- groups do
171         if seq.length > max then max = seq.length
172       max
173
174     def best(
175         table: Map[Pattern, Seq[ScoredPattern]],
176         remaining: Set[Pattern]
177     ): (Int, Boolean, Pattern) =
178       var bestWorst = Int.MaxValue
179       var bestImpossible = true
180       var bestPattern: Pattern = makePattern()
181
182       for (pattern, scored) <- table do
183         val w = worst(groupByScore(scored))
184         val impossible = !remaining.contains(pattern)
185
186         val better =
187           (w < bestWorst) ||
188             (w == bestWorst && bestImpossible && !
    impossible) ||
189             (w == bestWorst && bestImpossible ==
    impossible && pattern < bestPattern)
190
191         if better then
192           bestWorst = w
193           bestImpossible = impossible
194           bestPattern = pattern
195
196       (bestWorst, bestImpossible, bestPattern)
197 end MasterMind
198
```