```scala
// Branch and bound knapsack problem
import collection.mutable.PriorityQueue

/** This class represents a thing we can put in the napsack.
  */
class Thing(wIn: Int, vIn: Int) extends Ordered[Thing] {
  val w = wIn
  val v = vIn
  def v_w = v / w
  override def toString = "w=%d, v=%d, v/w=%d".format(w, v, v_w)
  override def compare(that: Thing) = {
    v_w compare that.v_w
  }
}

/** This is a node in the state-space tree
  */
class Node(wIn: Int, vIn: Int, ubIn: Int, levelIn: Int) extends Ordered[Node] {
  val w = wIn
  val v = vIn
  val ub = ubIn
  val level = levelIn
  var left: Node = null
  var right: Node = null
  var use = false

  override def compare(that: Node) = {
    ub compare that.ub
  }

  override def toString = {
    "---\nw=%s\nv=%s\nub=%s\nlevel=%s\nuse=%s\n---".format(
      w, v, ub, level,
      use match {
        case true => "true"
        case false => "false"

      }
    )
  }
}

/** This class finds the optimal solution to the given napsack
  * problem using branch and bound.
  *
  * items: the array of things that we can put in the napsack.
  * size: the size of the knapsack.
  */
class Knapsack(itemsIn: Array[Thing], size: Int) {
  // Sort all the lists by v/w and put everything into a new object.
  val items = itemsIn.sorted.reverse

  /** This function gives the upper bound on an item, i
    */
  def upperBound(w: Int, v: Int, v_w: Int) = {
    //println("%d + (%d - %d)(%d)".format(v, size, w, items(i).v_w))
    v + ((size - w) * v_w)
  }

  /** Finds the optimal items to put in the knapsack. Returns
    * a list of the indeces of the items to use in the list.
    */
  def findOptimal: List[Int] = {
    var result: List[Int] = Nil
```

```scala
    // the main part - build the state-space tree
    var foundOptimal = false;
    // A priority queue to keep track of the leaves so we know which node
    // to work on next
    var leaves = new PriorityQueue[Node]
    val root = new Node(0, 0, upperBound(0, 0, 0), 0)
    leaves += root

    while (!foundOptimal) {
      // Find out which parent node to work from
      val parent = leaves.dequeue
      val level  = parent.level

      // Find the v_w value
      val v_w = if (level + 1 < items.length) {
        items(level + 1).v_w
      } else 0

      // If we're using this node then add it to the result
      if (parent.use) {
        result ::= level - 1
      }

      // Terminating case
      if (level + 1 > items.length) {
        foundOptimal = true
      // Otherwise find the children nodes.
      } else {
        // Add the left node
        val l_w = parent.w + items(level).w
        // If this node is feasable and we haven't yet reached the
        // end of the nodes
        if (l_w <= size && !foundOptimal) {
          val l_v = parent.v + items(level).v
          val l_ub = upperBound(l_w, l_v, v_w)
          parent.left = new Node(l_w, l_v, l_ub, level + 1)
          parent.left.use = true
          leaves += parent.left
        }

        // Add the right node
        val r_w = parent.w
        // If this node is feasable
        if (r_w <= size) {
          val r_v = parent.v
          val r_ub = upperBound(r_w, r_v, v_w)
          parent.right = new Node(r_w, r_v, r_ub, level + 1)
          leaves += parent.right
        }
      }
    }

    result.reverse
  }
}

object Main {
  def main(args: Array[String]) = {
    // This is the sample problem defined in the Levitin book.

    val things = Array(
      new Thing(7, 42),
      new Thing(5, 25),
```

```
      new Thing(3, 12)
      //new Thing(4, 40)
    )
    val k = new Knapsack(things, 10)
    val optimal = k.findOptimal

    // val things = Array(
    //   new Thing(2, 40),
    //   new Thing(1, 1)
    // )
    // val k = new Knapsack(things, 4)
    // val optimal = k.findOptimal

    optimal.foreach(x =>
      println("Take item: " + things(x)).toString
    )
  }
}
```