```scala
// P01 (*) Find the last element of a list.
//     Example:
//     scala> last(List(1, 1, 2, 3, 5, 8))
//     res0: Int = 8

// The start of the definition of last should be
//     def last[A](l: List[A]): A = ...
// The `[A]` allows us to handle lists of any type.

object P01 {
  // There are several ways to solve this problem.  If we use builtins, it's very
  // easy.
  def lastBuiltin[A](ls: List[A]): A = ls.last

  // The standard functional approach is to recurse down the list until we hit
  // the end.  Scala's pattern matching makes this easy.
  def lastRecursive[A](ls: List[A]): A = ls match {
    case h :: Nil  => h
    case _ :: tail => lastRecursive(tail)
    case _         => throw new NoSuchElementException
  }
}
// P02 (*) Find the last but one element of a list.
//     Example:
//     scala> penultimate(List(1, 1, 2, 3, 5, 8))
//     res0: Int = 5

object P02 {
  // Again, with builtins this is easy.
  def penultimateBuiltin[A](ls: List[A]): A =
    if (ls.isEmpty) throw new NoSuchElementException
    else ls.init.last

  // But pattern matching also makes it easy.
  def penultimateRecursive[A](ls: List[A]): A = ls match {
    case h :: _ :: Nil => h
    case _ :: tail     => penultimateRecursive(tail)
    case _             => throw new NoSuchElementException
  }


  // Just for fun, let's look at making a generic lastNth function.

  // An obvious modification of the builtin solution works.
  def lastNthBuiltin[A](n: Int, ls: List[A]): A = {
    if (n <= 0) throw new IllegalArgumentException
    if (ls.length < n) throw new NoSuchElementException
    ls.takeRight(n).head
  }

  // Here's one approach to a non-builtin solution.
  def lastNthRecursive[A](n: Int, ls: List[A]): A = {
    def lastNthR(count: Int, resultList: List[A], curList: List[A]): A =
      curList match {
        case Nil if count > 0 => throw new NoSuchElementException
        case Nil              => resultList.head
        case _ :: tail        =>
          lastNthR(count - 1,
                   if (count > 0) resultList else resultList.tail,
                   tail)
      }
    if (n <= 0) throw new IllegalArgumentException
    else lastNthR(n, ls, ls)
  }
}
// P03 (*) Find the Kth element of a list.
//     By convention, the first element in the list is element 0.
//
//     Example:
//     scala> nth(2, List(1, 1, 2, 3, 5, 8))
//     res0: Int = 2

object P03 {
  // Trivial with builtins.
  def nthBuiltin[A](n: Int, ls: List[A]): A =
    if (n >= 0) ls(n)
    else throw new NoSuchElementException

  // Not that much harder without.
  def nthRecursive[A](n: Int, ls: List[A]): A = (n, ls) match {
    case (0, h :: _   ) => h
    case (n, _ :: tail) => nthRecursive(n - 1, tail)
    case (_, Nil      ) => throw new NoSuchElementException
  }
}
// P04 (*) Find the number of elements of a list.
//     Example:
//     scala> length(List(1, 1, 2, 3, 5, 8))
//     res0: Int = 6

object P04 {
  // Builtins.
  def lengthBuiltin[A](ls: List[A]): Int = ls.length

  // Simple recursive solution.
  def lengthRecursive[A](ls: List[A]): Int = ls match {
    case Nil       => 0
    case _ :: tail => 1 + lengthRecursive(tail)
  }

  // Tail recursive solution.  Theoretically more efficient; with tail-call
  // elimination in the compiler, this would run in constant space.
  // Unfortunately, the JVM doesn't do tail-call elimination in the general
  // case.  Scala *will* do it if the method is either final or is a local
  // function.  In this case, `lengthR` is a local function, so it should
  // be properly optimized.
  // For more information, see
  // http://blog.richdougherty.com/2009/04/tail-calls-tailrec-and-trampolines.html
  def lengthTailRecursive[A](ls: List[A]): Int = {
    def lengthR(result: Int, curList: List[A]): Int = curList match {
      case Nil       => result
      case _ :: tail => lengthR(result + 1, tail)
    }
    lengthR(0, ls)
  }

  // More pure functional solution, with folds.
  def lengthFunctional[A](ls: List[A]): Int = ls.foldLeft(0) { (c, _) => c + 1 }
}
// P05 (*) Reverse a list.
//     Example:
//     scala> reverse(List(1, 1, 2, 3, 5, 8))
//     res0: List[Int] = List(8, 5, 3, 2, 1, 1)

object P05 {
  // Builtin.
  def reverseBuiltin[A](ls: List[A]): List[A] = ls.reverse

  // Simple recursive.  O(n^2)
  def reverseRecursive[A](ls: List[A]): List[A] = ls match {
    case Nil       => Nil
    case h :: tail => reverseRecursive(tail) ::: List(h)
  }

  // Tail recursive.
  def reverseTailRecursive[A](ls: List[A]): List[A] = {
    def reverseR(result: List[A], curList: List[A]): List[A] = curList match {
      case Nil       => result
      case h :: tail => reverseR(h :: result, tail)
    }
    reverseR(Nil, ls)
  }

  // Pure functional
  def reverseFunctional[A](ls: List[A]): List[A] =
    ls.foldLeft(List[A]()) { (r, h) => h :: r }
}
// P06 (*) Find out whether a list is a palindrome.
//     Example:
//     scala> isPalindrome(List(1, 2, 3, 2, 1))
//     res0: Boolean = true

object P06 {
  // In theory, we could be slightly more efficient than this.  This approach
  // traverses the list twice: once to reverse it, and once to check equality.
  // Technically, we only need to check the first half of the list for equality
  // with the first half of the reversed list.  The code to do that more
  // efficiently than this implementation is much more complicated, so we'll
  // leave things with this clear and concise implementation.
  def isPalindrome[A](ls: List[A]): Boolean = ls == ls.reverse
```

```
}
// P07 (**) Flatten a nested list structure.
//     Example:
//     scala> flatten(List(List(1, 1), 2, List(3, List(5, 8))))
//     res0: List[Any] = List(1, 1, 2, 3, 5, 8)

object P07 {
  def flatten(ls: List[Any]): List[Any] = ls flatMap {
    case ms: List[_] => flatten(ms)
    case e => List(e)
  }
}

// P08 (**) Eliminate consecutive duplicates of list elements.
//     If a list contains repeated elements they should be replaced with a
//     single copy of the element.  The order of the elements should not be
//     changed.
//
//     Example:
//     scala> compress(List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e))
//     res0: List[Symbol] = List('a, 'b, 'c, 'a, 'd, 'e)

object P08 {
  // Standard recursive.
  def compressRecursive[A](ls: List[A]): List[A] = ls match {
    case Nil       => Nil
    case h :: tail => h :: compressRecursive(tail.dropWhile(_ == h))
  }

  // Tail recursive.
  def compressTailRecursive[A](ls: List[A]): List[A] = {
    def compressR(result: List[A], curList: List[A]): List[A] = curList match {
      case h :: tail => compressR(h :: result, tail.dropWhile(_ == h))
      case Nil       => result.reverse
    }
    compressR(Nil, ls)
  }

  // Functional.
  def compressFunctional[A](ls: List[A]): List[A] =
    ls.foldRight(List[A]()) { (h, r) =>
      if (r.isEmpty || r.head != h) h :: r
      else r
    }
}
// P09 (**) Pack consecutive duplicates of list elements into sublists.
//     If a list contains repeated elements they should be placed in separate
//     sublists.
//
//     Example:
//     scala> pack(List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e))
//     res0: List[List[Symbol]] = List(List('a, 'a, 'a, 'a), List('b), List('c, 'c), List('a, 'a),
List('d), List('e, 'e, 'e, 'e))

object P09 {
  def pack[A](ls: List[A]): List[List[A]] = {
    if (ls.isEmpty) List(List())
    else {
      val (packed, next) = ls span { _ == ls.head }
      if (next == Nil) List(packed)
      else packed :: pack(next)
    }
  }
}
// P10 (*) Run-length encoding of a list.
//     Use the result of problem P09 to implement the so-called run-length
//     encoding data compression method.  Consecutive duplicates of elements are
//     encoded as tuples (N, E) where N is the number of duplicates of the
//     element E.
//
//     Example:
//     scala> encode(List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e))
//     res0: List[(Int, Symbol)] = List((4,'a), (1,'b), (2,'c), (2,'a), (1,'d), (4,'e))

object P10 {
  import P09.pack
  def encode[A](ls: List[A]): List[(Int, A)] =
    pack(ls) map { e => (e.length, e.head) }
}
// P11 (*) Modified run-length encoding.
//     Modify the result of problem P10 in such a way that if an element has no
//     duplicates it is simply copied into the result list.  Only elements with
//     duplicates are transferred as (N, E) terms.
//
//     Example:
//     scala> encodeModified(List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e))
//     res0: List[Any] = List((4,'a), 'b, (2,'c), (2,'a), 'd, (4,'e))

object P11 {
  import P10.encode
  def encodeModified[A](ls: List[A]): List[Any] =
    encode(ls) map { t => if (t._1 == 1) t._2 else t }

  // Just for fun, here's a more typesafe version.
  def encodeModified2[A](ls: List[A]): List[Either[A, (Int, A)]] =
    encode(ls) map { t => if (t._1 == 1) Left(t._2) else Right(t) }
}
// P12 (**) Decode a run-length encoded list.
//     Given a run-length code list generated as specified in problem P10,
//     construct its uncompressed version.
//
//     Example:
//     scala> decode(List((4, 'a), (1, 'b), (2, 'c), (2, 'a), (1, 'd), (4, 'e)))
//     res0: List[Symbol] = List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e)

object P12 {
  def decode[A](ls: List[(Int, A)]): List[A] =
    ls flatMap { e => List.make(e._1, e._2) }
}
// P13 (**) Run-length encoding of a list (direct solution).
//     Implement the so-called run-length encoding data compression method
//     directly.  I.e. don't use other methods you've written (like P09's
//     pack); do all the work directly.
//
//     Example:
//     scala> encodeDirect(List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e))
//     res0: List[(Int, Symbol)] = List((4,'a), (1,'b), (2,'c), (2,'a), (1,'d), (4,'e))

object P13 {
  // This is basically a modification of P09.
  def encodeDirect[A](ls: List[A]): List[(Int, A)] =
    if (ls.isEmpty) Nil
    else {
      val (packed, next) = ls span { _ == ls.head }
      (packed.length, packed.head) :: encodeDirect(next)
    }
}
// P14 (*) Duplicate the elements of a list.
//     Example:
//     scala> duplicate(List('a, 'b, 'c, 'c, 'd))
//     res0: List[Symbol] = List('a, 'a, 'b, 'b, 'c, 'c, 'c, 'c, 'd, 'd)

object P14 {
  def duplicate[A](ls: List[A]): List[A] = ls flatMap { e => List(e, e) }
}
// P15 (**) Duplicate the elements of a list a given number of times.
//     Example:
//     scala> duplicateN(3, List('a, 'b, 'c, 'c, 'd))
//     res0: List[Symbol] = List('a, 'a, 'a, 'b, 'b, 'b, 'c, 'c, 'c, 'c, 'c, 'c, 'd, 'd, 'd)

object P15 {
  def duplicateN[A](n: Int, ls: List[A]): List[A] =
    ls flatMap { List.make(n, _) }
}
// P16 (**) Drop every Nth element from a list.
//     Example:
//     scala> drop(3, List('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k))
//     res0: List[Symbol] = List('a, 'b, 'd, 'e, 'g, 'h, 'j, 'k)

object P16 {
  // Simple recursion.
  def dropRecursive[A](n: Int, ls: List[A]): List[A] = {
    def dropR(c: Int, curList: List[A]): List[A] = (c, curList) match {
      case (_, Nil)       => Nil
      case (1, _ :: tail) => dropR(n, tail)
      case (_, h :: tail) => h :: dropR(c - 1, tail)
    }
    dropR(n, ls)
  }
  // Tail recursive.
  def dropTailRecursive[A](n: Int, ls: List[A]): List[A] = {
    def dropR(c: Int, curList: List[A], result: List[A]): List[A] = (c, curList) match {
      case (_, Nil)       => result.reverse
      case (1, _ :: tail) => dropR(n, tail, result)
      case (_, h :: tail) => dropR(c - 1, tail, h :: result)
    }
    dropR(n, ls, Nil)
  }

  // Functional.
  def dropFunctional[A](n: Int, ls: List[A]): List[A] =
    ls.zipWithIndex filter { v => (v._2 + 1) % n != 0 } map { _._1 }
}
// P17 (*) Split a list into two parts.
//     The length of the first part is given.  Use a Tuple for your result.
//
//     Example:
//     scala> split(3, List('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k))
//     res0: (List[Symbol], List[Symbol]) = (List('a, 'b, 'c),List('d, 'e, 'f, 'g, 'h, 'i, 'j, 'k))

object P17 {
  // Builtin.
  def splitBuiltin[A](n: Int, ls: List[A]): (List[A], List[A]) = ls.splitAt(n)

  // Simple recursion.
  def splitRecursive[A](n: Int, ls: List[A]): (List[A], List[A]) = (n, ls) match {
    case (_, Nil)       => (Nil, Nil)
    case (0, list)      => (Nil, list)
    case (n, h :: tail) => {
      val (pre, post) = splitRecursive(n - 1, tail)
      (h :: pre, post)
    }
  }

  // Tail recursive.
  def splitTailRecursive[A](n: Int, ls: List[A]): (List[A], List[A]) = {
    def splitR(curN: Int, curL: List[A], pre: List[A]): (List[A], List[A]) =
      (curN, curL) match {
        case (_, Nil)       => (pre.reverse, Nil)
        case (0, list)      => (pre.reverse, list)
        case (n, h :: tail) => splitR(n - 1, tail, h :: pre)
      }
    splitR(n, ls, Nil)
  }

  // Functional (barely not "builtin").
  def splitFunctional[A](n: Int, ls: List[A]): (List[A], List[A]) =
    (ls.take(n), ls.drop(n))
}
// P18 (**) Extract a slice from a list.
//     Given two indices, I and K, the slice is the list containing the elements
//     from and including the Ith element up to but not including the Kth
//     element of the original list.  Start counting the elements with 0.
//
//     Example:
//     scala> slice(3, 7, List('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k))
//     res0: List[Symbol] = List('d, 'e, 'f, 'g)

object P18 {
  // Builtin.
  def sliceBuiltin[A](start: Int, end: Int, ls: List[A]): List[A] =
    ls.slice(start, end)

  // Simple recursive.
  def sliceRecursive[A](start: Int, end: Int, ls: List[A]): List[A] =
    (start, end, ls) match {
      case (_, _, Nil)                 => Nil
      case (_, e, _)         if e <= 0 => Nil
      case (s, e, h :: tail) if s <= 0 => h :: sliceRecursive(0, e - 1, tail)
      case (s, e, h :: tail)           => sliceRecursive(s - 1, e - 1, tail)
    }

  // Tail recursive, using pattern matching.
  def sliceTailRecursive[A](start: Int, end: Int, ls: List[A]): List[A] = {
    def sliceR(count: Int, curList: List[A], result: List[A]): List[A] =
      (count, curList) match {
        case (_, Nil)                     => result.reverse
        case (c, h :: tail) if end <= c   => result.reverse
        case (c, h :: tail) if start <= c => sliceR(c + 1, tail, h :: result)
        case (c, _ :: tail)               => sliceR(c + 1, tail, result)
      }
    sliceR(0, ls, Nil)
  }

  // Since several of the patterns are similar, we can condense the tail recursive
  // solution a little.
  def sliceTailRecursive2[A](start: Int, end: Int, ls: List[A]): List[A] = {
    def sliceR(count: Int, curList: List[A], result: List[A]): List[A] = {
      if (curList.isEmpty || count >= end) result.reverse
      else sliceR(count + 1, curList.tail,
                  if (count >= start) curList.head :: result
                  else result)
    }
    sliceR(0, ls, Nil)
  }

  // Functional.
  def sliceFunctional[A](s: Int, e: Int, ls: List[A]): List[A] =
    ls drop s take (e - (s max 0))
}
// P19 (**) Rotate a list N places to the left.
//     Examples:
//     scala> rotate(3, List('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k))
//     res0: List[Symbol] = List('d, 'e, 'f, 'g, 'h, 'i, 'j, 'k, 'a, 'b, 'c)
//
//     scala> rotate(-2, List('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k))
//     res1: List[Symbol] = List('j, 'k, 'a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i)

object P19 {
  def rotate[A](n: Int, ls: List[A]): List[A] = {
    val nBounded = if (ls.isEmpty) 0 else n % ls.length
    if (nBounded < 0) rotate(nBounded + ls.length, ls)
    else (ls drop nBounded) ::: (ls take nBounded)
  }
}
// P20 (*) Remove the Kth element from a list.
//     Return the list and the removed element in a Tuple.  Elements are
//     numbered from 0.
//
//     Example:
//     scala> removeAt(1, List('a, 'b, 'c, 'd))
//     res0: (List[Symbol], Symbol) = (List('a, 'c, 'd),'b)

object P20 {
  def removeAt[A](n: Int, ls: List[A]): (List[A], A) = ls.splitAt(n) match {
    case (Nil, _) if n < 0 => throw new NoSuchElementException
    case (pre, e :: post)  => (pre ::: post, e)
    case (pre, Nil)        => throw new NoSuchElementException
  }

  // Alternate, with fewer builtins.
  def removeAt2[A](n: Int, ls: List[A]): (List[A], A) =
    if (n < 0) throw new NoSuchElementException
    else (n, ls) match {
      case (_, Nil) => throw new NoSuchElementException
      case (0, h :: tail) => (tail, h)
      case (_, h :: tail) => {
        val (t, e) = removeAt(n - 1, ls.tail)
        (ls.head :: t, e)
      }
    }
}

// P21 (*) Insert an element at a given position into a list.
//     Example:
//     scala> insertAt('new, 1, List('a, 'b, 'c, 'd))
//     res0: List[Symbol] = List('a, 'new, 'b, 'c, 'd)

object P21 {
  def insertAt[A](e: A, n: Int, ls: List[A]): List[A] = ls.splitAt(n) match {
    case (pre, post) => pre ::: e :: post
  }
}
// P22 (*) Create a list containing all integers within a given range.
//     Example:
//     scala> range(4, 9)
//     res0: List[Int] = List(4, 5, 6, 7, 8, 9)

object P22 {
```

```scala
  // Builtin.
  def rangeBuiltin(start: Int, end: Int): List[Int] = List.range(start, end + 1)

  // Recursive.
  def rangeRecursive(start: Int, end: Int): List[Int] =
    if (end < start) Nil
    else start :: rangeRecursive(start + 1, end)

  // Tail recursive.
  def rangeTailRecursive(start: Int, end: Int): List[Int] = {
    def rangeR(end: Int, result: List[Int]): List[Int] = {
      if (end < start) result
      else rangeR(end - 1, end :: result)
    }
    rangeR(end, Nil)
  }

  // The classic functional approach would be to use `unfoldr`, which Scala
  // doesn't have.  So we'll write one and then use it.
  def unfoldRight[A, B](s: B)(f: B => Option[(A, B)]): List[A] =
    f(s) match {
      case None         => Nil
      case Some((r, n)) => r :: unfoldRight(n)(f)
    }
  def rangeFunctional(start: Int, end: Int): List[Int] =
    unfoldRight(start) { n =>
      if (n > end) None
      else Some((n, n + 1))
    }
}
// P23 (**) Extract a given number of randomly selected elements from a list.
//     Example:
//     scala> randomSelect(3, List('a, 'b, 'c, 'd, 'f, 'g, 'h))
//     res0: List[Symbol] = List('e, 'd, 'a)
//
//     Hint: Use the answer to P20.

object P23 {
  import P20.removeAt

  def randomSelect1[A](n: Int, ls: List[A]): List[A] =
    if (n <= 0) Nil
    else {
      val (rest, e) = removeAt((new util.Random).nextInt(ls.length), ls)
      e :: randomSelect1(n - 1, rest)
    }

  // It can be expensive to create a new Random instance every time, so let's
  // only do it once.
  def randomSelect[A](n: Int, ls: List[A]): List[A] = {
    def randomSelectR(n: Int, ls: List[A], r: util.Random): List[A] =
      if (n <= 0) Nil
      else {
        val (rest, e) = removeAt(r.nextInt(ls.length), ls)
        e :: randomSelectR(n - 1, rest, r)
      }
    randomSelectR(n, ls, new util.Random)
  }
}

// P24 (*) Lotto: Draw N different random numbers from the set 1..M.
//     Example:
//     scala> lotto(6, 49)
//     res0: List[Int] = List(23, 1, 17, 33, 21, 37)

object P24 {
  import P23.randomSelect
  def lotto(count: Int, max: Int): List[Int] =
    randomSelect(count, List.range(1, max + 1))
}
// P25 (*) Generate a random permutation of the elements of a list.
//     Hint: Use the solution of problem P23.
//
//     Example:
//     scala> randomPermute(List('a, 'b, 'c, 'd, 'e, 'f))
//     res0: List[Symbol] = List('b, 'a, 'd, 'c, 'e, 'f)

object P25 {
  // This algorithm is O(n^2), but it makes up for that in simplicity of
  // implementation.
  import P23.randomSelect
  def randomPermute1[A](ls: List[A]): List[A] = randomSelect(ls.length, ls)

  // The canonical way to shuffle imperatively is Fisher-Yates.  It requires a
  // mutable array.  This is O(n).
  def randomPermute[A](ls: List[A]): List[A] = {
    val rand = new util.Random
    val a = ls.toArray
    for (i <- a.length - 1 to 1 by -1) {
      val i1 = rand.nextInt(i + 1)
      val t = a(i)
      a.update(i, a(i1))
      a.update(i1, t)
    }
    a.toList
  }

  // Efficient purely functional algorithms for shuffling are a lot harder.  One
  // is described in http://okmij.org/ftp/Haskell/perfect-shuffle.txt using
  // Haskell. Implementing it in Scala is left as an exercise for the reader.
}
// P26 (**) Generate the combinations of K distinct objects chosen from the N
//          elements of a list.
//     In how many ways can a committee of 3 be chosen from a group of 12
//     people?  We all know that there are C(12,3) = 220 possibilities (C(N,K)
//     denotes the well-known binomial coefficient).  For pure mathematicians,
//     this result may be great.  But we want to really generate all the possibilities.
//
//     Example:
//     scala> combinations(3, List('a, 'b, 'c, 'd, 'e, 'f))
//     res0: List[List[Symbol]] = List(List('a, 'b, 'c), List('a, 'b, 'd), List('a, 'b, 'e), ...

object P26 {
  // flatMapSublists is like list.flatMap, but instead of passing each element
  // to the function, it passes successive sublists of L.
  def flatMapSublists[A,B](ls: List[A])(f: (List[A]) => List[B]): List[B] =
    ls match {
      case Nil => Nil
      case sublist@(_ :: tail) => f(sublist) ::: flatMapSublists(tail)(f)
    }

  def combinations[A](n: Int, ls: List[A]): List[List[A]] =
    if (n == 0) List(Nil)
    else flatMapSublists(ls) { sl =>
      combinations(n - 1, sl.tail) map {sl.head :: _}
    }
}
// P27 (**) Group the elements of a set into disjoint subsets.
//     a) In how many ways can a group of 9 people work in 3 disjoint subgroups
//        of 2, 3 and 4 persons?  Write a function that generates all the
//        possibilities.
//
//        Example:
//        scala> group3(List("Aldo", "Beat", "Carla", "David", "Evi", "Flip", "Gary", "Hugo",
"Ida"))
//        res0: List[List[List[String]]] = List(List(List(Aldo, Beat), List(Carla, David, Evi),
List(Flip, Gary, Hugo, Ida)), ...
//
//     b) Generalize the above predicate in a way that we can specify a list
//        of group sizes and the predicate will return a list of groups.
//
//        Example:
//        scala> group(List(2, 2, 5), List("Aldo", "Beat", "Carla", "David", "Evi", "Flip", "Gary",
"Hugo", "Ida"))
//        res0: List[List[List[String]]] = List(List(List(Aldo, Beat), List(Carla, David), List(Evi,
Flip, Gary, Hugo, Ida)), ...
//
//     Note that we do not want permutations of the group members;
//     i.e. ((Aldo, Beat), ...) is the same solution as ((Beat, Aldo), ...).
//     However, we make a difference between ((Aldo, Beat), (Carla, David), ...)
//     and ((Carla, David), (Aldo, Beat), ...).
//
//     You may find more about this combinatorial problem in a good book on
//     discrete mathematics under the term "multinomial coefficients".

object P27 {
  import P26.combinations

  def group3[A](ls: List[A]): List[List[List[A]]] =
    for {
      a <- combinations(2, ls)
      noA = ls -- a
      b <- combinations(3, noA)
    } yield List(a, b, noA -- b)

  def group[A](ns: List[Int], ls: List[A]): List[List[List[A]]] = ns match {
    case Nil     => List(Nil)
    case n :: ns => combinations(n, ls) flatMap { c =>
      group(ns, ls -- c) map {c :: _}
    }
  }
}
// P28 (**) Sorting a list of lists according to length of sublists.
//     a) We suppose that a list contains elements that are lists themselves.
//        The objective is to sort the elements of the list according to their
//        length.  E.g. short lists first, longer lists later, or vice versa.
//
//     Example:
//     scala> lsort(List(List('a, 'b, 'c), List('d, 'e), List('f, 'g, 'h), List('d, 'e), List('i,
'j, 'k, 'l), List('m, 'n), List('o)))
//     res0: List[List[Symbol]] = List(List('o), List('d, 'e), List('d, 'e), List('m, 'n), List('a,
'b, 'c), List('f, 'g, 'h), List('i, 'j, 'k, 'l))
//
//     b) Again, we suppose that a list contains elements that are lists
//        themselves.  But this time the objective is to sort the elements
//        according to their length frequency; i.e. in the default, sorting is
//        done ascendingly, lists with rare lengths are placed, others with a
//        more frequent length come later.
//
//     Example:
//     scala> lsortFreq(List(List('a, 'b, 'c), List('d, 'e), List('f, 'g, 'h), List('d, 'e),
List('i, 'j, 'k, 'l), List('m, 'n), List('o)))
//     res1: List[List[Symbol]] = List(List('i, 'j, 'k, 'l), List('o), List('a, 'b, 'c), List('f,
'g, 'h), List('d, 'e), List('d, 'e), List('m, 'n))
//
//     Note that in the above example, the first two lists in the result have
//     length 4 and 1 and both lengths appear just once.  The third and fourth
//     lists have length 3 and there are two list of this length.  Finally, the
//     last three lists have length 2.  This is the most frequent length.

object P28 {
  import P10.encode

  def lsort[A](ls: List[List[A]]): List[List[A]] =
    ls sort { _.length < _.length }

  def lsortFreq[A](ls: List[List[A]]): List[List[A]] = {
    val freqs = Map(encode(ls map { _.length } sort { _ < _ }) map { _.swap }:_*)
    ls sort { (e1, e2) => freqs(e1.length) < freqs(e2.length) }
  }
}
// P31 (**) Determine whether a given integer number is prime.
//     scala> 7.isPrime
//     res0: Boolean = true

// A fairly naive implementation for primality testing is simply: a number is
// prime if it it not divisible by any prime number less than or equal to its
// square root.
// Here, we use a Stream to create a lazy infinite list of prime numbers.  The
// mutual recursion between `primes` and `isPrime` works because of the limit
// on `isPrime` to the square root of the number being tested.

class S99Int(val start: Int) {
  def isPrime: Boolean =
    (start > 1) && (primes takeWhile { _ <= Math.sqrt(start) } forall { start % _ != 0 })
}

object S99Int {
  val primes = Stream.cons(2, Stream.from(3, 2) filter { _.isPrime })
}

// Readers interested in more sophisticated (and more efficient) primality tests
// are invited to read http://primes.utm.edu/prove/index.html .  Implementation
// in Scala is left as an exercise for the reader.

// Similarly, a more efficient, functional, lazy, infinite prime list can be found
// at http://article.gmane.org/gmane.comp.lang.haskell.cafe/19470 .  (Haskell
// implementation.)
// P32 (**) Determine the greatest common divisor of two positive integer
//          numbers.
```

```scala
//     Use Euclid's algorithm.
//
//     scala> gcd(36, 63)
//     res0: Int = 9

object S99Int {
  def gcd(m: Int, n: Int): Int = if (n == 0) m else gcd(n, m % n)
}
// P33 (*) Determine whether two positive integer numbers are coprime.
//     Two numbers are coprime if their greatest common divisor equals 1.
//
//     scala> 35.isCoprimeTo(64)
//     res0: Boolean = true

class S99Int(val start: Int) {
  def isCoprimeTo(n: Int): Boolean = gcd(start, n) == 1
}
// P34 (**) Calculate Euler's totient function phi(m).
//     Euler's so-called totient function phi(m) is defined as the number of
//     positive integers r (1 <= r < m) that are coprime to m.  As a special
//     case, phi(1) is defined to be 1.
//
//     scala> 10.totient
//     res0: Int = 4

class S99Int(val start: Int) {
  def totient: Int = (1 to start) filter { start.isCoprimeTo(_) } length
}
// P35 (**) Determine the prime factors of a given positive integer.
//     Construct a flat list containing the prime factors in ascending order.
//
//     scala> 315.primeFactors
//     res0: List[Int] = List(3, 3, 5, 7)

class S99Int(val start: Int) {
  def primeFactors: List[Int] = {
    def primeFactorsR(n: Int, ps: Stream[Int]): List[Int] =
      if (n.isPrime) List(n)
      else if (n % ps.head == 0) ps.head :: primeFactorsR(n / ps.head, ps)
      else primeFactorsR(n, ps.tail)
    primeFactorsR(start, primes)
  }
}
// P36 (**) Determine the prime factors of a given positive integer (2).
//     Construct a list containing the prime factors and their multiplicity.
//
//     scala> 315.primeFactorMultiplicity
//     res0: List[(Int, Int)] = List((3,2), (5,1), (7,1))
//
//     Alternately, use a Map for the result.
//     scala> 315.primeFactorMultiplicity
//     res0: Map[Int,Int] = Map(3 -> 2, 5 -> 1, 7 -> 1)

/*
// One approach is to reuse the solution from P10.
class S99Int(val start: Int) {
  import P10.encode
  def primeFactorMultiplicity: List[(Int,Int)] =
    encode(start.primeFactors) map { _.swap }
}
*/

// But we can do it directly.
class S99Int(val start: Int) {
  def primeFactorMultiplicity: Map[Int,Int] = {
    def factorCount(n: Int, p: Int): (Int,Int) =
      if (n % p != 0) (0, n)
      else factorCount(n / p, p) match { case (c, d) => (c + 1, d) }
    def factorsR(n: Int, ps: Stream[Int]): Map[Int, Int] =
      if (n == 1) Map()
      else if (n.isPrime) Map(n -> 1)
      else {
        val nps = ps.dropWhile(n % _ != 0)
        val (count, dividend) = factorCount(n, nps.head)
        Map(nps.head -> count) ++ factorsR(dividend, nps.tail)
      }
    factorsR(start, primes)
  }

  // This also lets us change primeFactors.
  def primeFactors: List[Int] =
    start.primeFactorMultiplicity flatMap { v => List.make(v._2, v._1) } toList
}
// P37 (**) Calculate Euler's totient function phi(m) (improved).
//     See problem P34 for the definition of Euler's totient function.  If the
//     list of the prime factors of a number m is known in the form of problem
//     P36 then the function phi(m>) can be efficiently calculated as follows:
//     Let [[p_1, m_1], [p_2, m_2], [p_3, m_3], ...] be the list of prime
//     factors (and their multiplicities) of a given number m.  Then phi(m) can
//     be calculated with the following formula:
//
//     phi(m) = (p_1-1)*p_1^(m_1-1) * (p_2-1)*p_2^(m_2-1) * (p_3-1)*p_3^(m_3-1) * ...
//
//     Note that a^b stands for the bth power of a.

class S99Int(val start: Int) {
  def totient: Int = start.primeFactorMultiplicity.foldLeft(1) { (r, f) =>
    f match { case (p, m) => r * (p - 1) * Math.pow(p, m - 1).toInt }
  }
}
// P38 (*) Compare the two methods of calculating Euler's totient function.
//     Use the solutions of problems P34 and P37 to compare the algorithms.  Try
//     to calculate phi(10090) as an example.

// Here's an object that will test the relative execution times of the two
// approaches.
// On a 2.4 GHz Athlon 64 X2, here's what happens the first time `test` is called:
//   Preload primes: 20 ms.
//   P34 (10090): 65 ms.
//   P37 (10090): 3 ms.
//
// The JVM tends to profile its execution, though.  Here's a several-iteration run.
//   scala> import P38._
//   import P38._
//
//   scala> test(10090)
//   Preload primes: 9 ms.
//   P34 (10090): 53 ms.
//   P37 (10090): 4 ms.
//
//   scala> test(10090)
//   Preload primes: 2 ms.
//   P34 (10090): 28 ms.
//   P37 (10090): 1 ms.
//
//   scala> test(10090)
//   Preload primes: 2 ms.
//   P34 (10090): 17 ms.
//   P37 (10090): 1 ms.
//
//   scala> test(10090)
//   Preload primes: 2 ms.
//   P34 (10090): 3 ms.
//   P37 (10090): 0 ms.
//
//   scala> test(10090)
//   Preload primes: 4 ms.
//   P34 (10090): 2 ms.
//   P37 (10090): 0 ms.

object P38 {
  import arithmetic.S99Int._

  def time[A](label: String)(block: => A): A = {
    val now = System.currentTimeMillis()
    val ret = block
    println(label + ": " + (System.currentTimeMillis() - now) + " ms.")
    ret
  }

  def test(n: Int) {
    time("Preload primes") {
      primes takeWhile { _ <= Math.sqrt(n) } force
    }
    time("P34 (" + n + ")") {
      n.totientP34
    }
    time("P37 (" + n + ")") {
      n.totient
    }
  }
}
// P39 (*) A list of prime numbers.
//     Given a range of integers by its lower and upper limit, construct a list
//     of all prime numbers in that range.
//
//     scala> listPrimesinRange(7 to 31)
//     res0: List[Int] = List(7, 11, 13, 17, 19, 23, 29, 31)

object S99Int {
  def listPrimesinRange(r: Range): List[Int] =
    primes dropWhile { _ < r.first } takeWhile { _ <= r.last } toList
}
// P40 (**) Goldbach's conjecture.
//     Goldbach's conjecture says that every positive even number greater than 2
//     is the sum of two prime numbers.  E.g. 28 = 5 + 23.  It is one of the
//     most famous facts in number theory that has not been proved to be correct
//     in the general case.  It has been numerically confirmed up to very large
//     numbers (much larger than Scala's Int can represent).  Write a function
//     to find the two prime numbers that sum up to a given even integer.
//
//     scala> 28.goldbach
//     res0: (Int, Int) = (5,23)

class S99Int(val start: Int) {
  def goldbach: (Int,Int) =
    primes takeWhile { _ < start } find { p => (start - p).isPrime } match {
      case None     => throw new IllegalArgumentException
      case Some(p1) => (p1, start - p1)
    }
}
// P41 (**) A list of Goldbach compositions.
//     Given a range of integers by its lower and upper limit, print a list of
//     all even numbers and their Goldbach composition.
//
//     scala> printGoldbachList(9 to 20)
//     10 = 3 + 7
//     12 = 5 + 7
//     14 = 3 + 11
//     16 = 3 + 13
//     18 = 5 + 13
//     20 = 3 + 17
//
//     In most cases, if an even number is written as the sum of two prime
//     numbers, one of them is very small.  Very rarely, the primes are both
//     bigger than, say, 50.  Try to find out how many such cases there are in
//     the range 2..3000.
//
//     Example (minimum value of 50 for the primes):
//     scala> printGoldbachListLimited(1 to 2000, 50)
//     992 = 73 + 919
//     1382 = 61 + 1321
//     1856 = 67 + 1789
//     1928 = 61 + 1867

object S99Int {
  def printGoldbachList(r: Range) {
    printGoldbachListLimited(r, 0)
  }

  def printGoldbachListLimited(r: Range, limit: Int) {
    (r filter { n => n > 2 && n % 2 == 0 } map { n => (n, n.goldbach) }
     filter { _._2._1 >= limit } foreach {
       _ match { case (n, (p1, p2)) => println(n + " = " + p1 + " + " + p2) }
     })
  }
}
// P46 (**) Truth tables for logical expressions.
//     Define functions and, or, nand, nor, xor, impl, and equ (for logical
//     equivalence) which return true or false according to the result of their
//     respective operations; e.g. and(A, B) is true if and only if both A and B
//     are true.
//
//     scala> and(true, true)
//     res0: Boolean = true
//
//     scala> xor(true. true)
//     res1: Boolean = false
//
//     A logical expression in two variables can then be written as a function of
//     two variables, e.g: (a: Boolean, b: Boolean) => and(or(a, b), nand(a, b))
//
//     Now, write a function called table2 which prints the truth table of a
//     given logical expression in two variables.
//
//     scala> table2((a: Boolean, b: Boolean) => and(a, or(a, b)))
//     A     B     result
//     true  true  true
//     true  false true
//     false true  false
//     false false false

// The trick here is not using builtins.  We'll define `not`, `and`, and `or`
// directly (using pattern matching), and the other functions in terms of those
// three.

object S99Logic {
  def not(a: Boolean) = a match {
    case true  => false
    case false => true
  }
  def and(a: Boolean, b: Boolean): Boolean = (a, b) match {
    case (true, true) => true
    case _            => false
  }
  def or(a: Boolean, b: Boolean): Boolean = (a, b) match {
    case (true, _) => true
    case (_, true) => true
    case _         => false
  }
  def equ(a: Boolean, b: Boolean): Boolean = or(and(a, b), and(not(a), not(b)))
  def xor(a: Boolean, b: Boolean): Boolean = not(equ(a, b))
  def nor(a: Boolean, b: Boolean): Boolean = not(or(a, b))
  def nand(a: Boolean, b: Boolean): Boolean = not(and(a, b))
  def impl(a: Boolean, b: Boolean): Boolean = or(not(a), b)

  def table2(f: (Boolean, Boolean) => Boolean) {
    println("A     B     result")
    for {a <- List(true, false);
         b <- List(true, false)} {
     printf("%-5s %-5s %-5s\n", a, b, f(a, b))
    }
  }
}
// P47 (*) Truth tables for logical expressions (2).
//     Continue problem P46 by redefining and, or, etc as operators.  (i.e. make
//     them methods of a new class with an implicit conversion from Boolean.)
//     not will have to be left as a object method.
//
//     scala> table2((a: Boolean, b: Boolean) => a and (a or not(b)))
//     A     B     result
//     true  true  true
//     true  false true
//     false true  false
//     false false false

// For simplicity, we remove `and`, `or`, `equ`, `xor`, `nor`, `nand`, and
// `impl` from the S99Logic object before putting them into a new class.

class S99Logic(a: Boolean) {
  import S99Logic._

  def and(b: Boolean): Boolean = (a, b) match {
    case (true, true) => true
    case _            => false
  }
  def or(b: Boolean): Boolean = (a, b) match {
    case (true, _) => true
    case (_, true) => true
    case _         => false
  }
  def equ(b: Boolean): Boolean = (a and b) or (not(a) and not(b))
  def xor(b: Boolean): Boolean = not(a equ b)
  def nor(b: Boolean): Boolean = not(a or b)
  def nand(b: Boolean): Boolean = not(a and b)
  def impl(b: Boolean): Boolean = not(a) or b
}

object S99Logic {
  implicit def boolean2S99Logic(a: Boolean): S99Logic = new S99Logic(a)
```

```
}
// P49 (**) Gray code.
//     An n-bit Gray code is a sequence of n-bit strings constructed according
//     to certain rules. For example,
//     n = 1: C(1) = ("0", "1").
//     n = 2: C(2) = ("00", "01", "11", "10").
//     n = 3: C(3) = ("000", "001", "011", "010", "110", "111", "101", "100").
//
//     Find out the construction rules and write a function to generate Gray
//     codes.
//
//     scala> gray(3)
//     res0 List[String] = List(000, 001, 011, 010, 110, 111, 101, 100)
//
//     See if you can use memoization to make the function more
//     efficient.

object P49 {
  def gray(n: Int): List[String] =
    if (n == 0) List("")
    else {
      val lower = gray(n - 1)
      (lower map { "0" + _ }) ::: (lower.reverse map { "1" + _ })
    }

  import scala.collection.mutable
  private val strings = mutable.Map(0 -> List(""))
  def grayMemoized(n: Int): List[String] = {
    if (!strings.contains(n)) {
      strings + (n -> ((grayMemoized(n - 1) map { "0" + _ }) :::
                       (grayMemoized(n - 1).reverse map { "1" + _ })))
    }
    strings(n)
  }
}
// P50 (***) Huffman code.
//     First of all, consult a good book on discrete mathematics or algorithms
//     for a detailed description of Huffman codes!
//
//     We suppose a set of symbols with their frequencies, given as a list of
//     (S, F) Tuples.  E.g. (("a", 45), ("b", 13), ("c", 12), ("d", 16),
//     ("e", 9), ("f", 5)).  Our objective is to construct a list of (S, C)
//     Tuples, where C is the Huffman code word for the symbol S.
//
//     scala> huffman(List(("a", 45), ("b", 13), ("c", 12), ("d", 16), ("e", 9), ("f", 5)))
//     res0: List[String, String] = List((a,0), (b,101), (c,100), (d,111), (e,1101), (f,1100))

// We'll do this functionally, with the two-queue algorithm.  (Scala's priority
// queues are mutable.)
object P50 {
  private abstract sealed class Tree[A] {
    val freq: Int
    def toCode: List[(A, String)] = toCodePrefixed("")
    def toCodePrefixed(prefix: String): List[(A, String)]
  }
  private final case class InternalNode[A](left: Tree[A], right: Tree[A]) extends Tree[A] {
    val freq: Int = left.freq + right.freq
    def toCodePrefixed(prefix: String): List[(A, String)] =
      left.toCodePrefixed(prefix + "0") ::: right.toCodePrefixed(prefix + "1")
  }
  private final case class LeafNode[A](element: A, freq: Int) extends Tree[A] {
    def toCodePrefixed(prefix: String): List[(A, String)] = List((element, prefix))
  }

  def huffman[A](ls: List[(A, Int)]): List[(A, String)] = {
    import collection.immutable.Queue
    def dequeueSmallest(q1: Queue[Tree[A]], q2: Queue[Tree[A]]) = {
      // This ordering chooses q1 in case of ties, which helps minimize tree
      // depth.
      if (q2.isEmpty) (q1.front, q1.dequeue._2, q2)
      else if (q1.isEmpty || q2.front.freq < q1.front.freq) (q2.front, q1, q2.dequeue._2)
      else (q1.front, q1.dequeue._2, q2)
    }
    def huffmanR(q1: Queue[Tree[A]], q2: Queue[Tree[A]]): List[(A, String)] = {
      if (q1.length + q2.length == 1) (if (q1.isEmpty) q2.front else q1.front).toCode
      else {
        val (v1, q3, q4) = dequeueSmallest(q1, q2)
        val (v2, q5, q6) = dequeueSmallest(q3, q4)
        huffmanR(q5, q6.enqueue(InternalNode(v1, v2)))
      }
    }
    huffmanR(Queue.Empty.enqueue(ls sort { _._2 < _._2 } map { e => LeafNode(e._1, e._2) }),
             Queue.Empty)
  }
}
// P01 (*) Find the last element of a list.
//     Example:
//     scala> last(List(1, 1, 2, 3, 5, 8))
//     res0: Int = 8

// The start of the definition of last should be
//     def last[A](l: List[A]): A = ...
// The `[A]` allows us to handle lists of any type.

object P01 {
  // There are several ways to solve this problem.  If we use builtins, it's very
  // easy.
  def lastBuiltin[A](ls: List[A]): A = ls.last

  // The standard functional approach is to recurse down the list until we hit
  // the end.  Scala's pattern matching makes this easy.
  def lastRecursive[A](ls: List[A]): A = ls match {
    case h :: Nil  => h
    case _ :: tail => lastRecursive(tail)
    case _         => throw new NoSuchElementException
  }
}
// P02 (*) Find the last but one element of a list.
//     Example:
//     scala> penultimate(List(1, 1, 2, 3, 5, 8))
//     res0: Int = 5

object P02 {
  // Again, with builtins this is easy.
  def penultimateBuiltin[A](ls: List[A]): A =
    if (ls.isEmpty) throw new NoSuchElementException
    else ls.init.last

  // But pattern matching also makes it easy.
  def penultimateRecursive[A](ls: List[A]): A = ls match {
    case h :: _ :: Nil => h
    case _ :: tail     => penultimateRecursive(tail)
    case _             => throw new NoSuchElementException
  }


  // Just for fun, let's look at making a generic lastNth function.

  // An obvious modification of the builtin solution works.
  def lastNthBuiltin[A](n: Int, ls: List[A]): A = {
    if (n <= 0) throw new IllegalArgumentException
    if (ls.length < n) throw new NoSuchElementException
    ls.takeRight(n).head
  }

  // Here's one approach to a non-builtin solution.
  def lastNthRecursive[A](n: Int, ls: List[A]): A = {
    def lastNthR(count: Int, resultList: List[A], curList: List[A]): A =
      curList match {
        case Nil if count > 0 => throw new NoSuchElementException
        case Nil              => resultList.head
        case _ :: tail        =>
          lastNthR(count - 1,
                   if (count > 0) resultList else resultList.tail,
                   tail)
      }
    if (n <= 0) throw new IllegalArgumentException
    else lastNthR(n, ls, ls)
  }
}
// P03 (*) Find the Kth element of a list.
//     By convention, the first element in the list is element 0.
//
//     Example:
//     scala> nth(2, List(1, 1, 2, 3, 5, 8))
//     res0: Int = 2

object P03 {
  // Trivial with builtins.
  def nthBuiltin[A](n: Int, ls: List[A]): A =
    if (n >= 0) ls(n)
    else throw new NoSuchElementException

  // Not that much harder without.
  def nthRecursive[A](n: Int, ls: List[A]): A = (n, ls) match {
    case (0, h :: _   ) => h
    case (n, _ :: tail) => nthRecursive(n - 1, tail)
    case (_, Nil      ) => throw new NoSuchElementException
  }
}
// P04 (*) Find the number of elements of a list.
//     Example:
//     scala> length(List(1, 1, 2, 3, 5, 8))
//     res0: Int = 6

object P04 {
  // Builtins.
  def lengthBuiltin[A](ls: List[A]): Int = ls.length

  // Simple recursive solution.
  def lengthRecursive[A](ls: List[A]): Int = ls match {
    case Nil       => 0
    case _ :: tail => 1 + lengthRecursive(tail)
  }

  // Tail recursive solution.  Theoretically more efficient; with tail-call
  // elimination in the compiler, this would run in constant space.
  // Unfortunately, the JVM doesn't do tail-call elimination in the general
  // case.  Scala *will* do it if the method is either final or is a local
  // function.  In this case, `lengthR` is a local function, so it should
  // be properly optimized.
  // For more information, see
  // http://blog.richdougherty.com/2009/04/tail-calls-tailrec-and-trampolines.html
  def lengthTailRecursive[A](ls: List[A]): Int = {
    def lengthR(result: Int, curList: List[A]): Int = curList match {
      case Nil       => result
      case _ :: tail => lengthR(result + 1, tail)
    }
    lengthR(0, ls)
  }

  // More pure functional solution, with folds.
  def lengthFunctional[A](ls: List[A]): Int = ls.foldLeft(0) { (c, _) => c + 1 }
}
// P05 (*) Reverse a list.
//     Example:
//     scala> reverse(List(1, 1, 2, 3, 5, 8))
//     res0: List[Int] = List(8, 5, 3, 2, 1, 1)

object P05 {
  // Builtin.
  def reverseBuiltin[A](ls: List[A]): List[A] = ls.reverse

  // Simple recursive.  O(n^2)
  def reverseRecursive[A](ls: List[A]): List[A] = ls match {
    case Nil       => Nil
    case h :: tail => reverseRecursive(tail) ::: List(h)
  }

  // Tail recursive.
  def reverseTailRecursive[A](ls: List[A]): List[A] = {
    def reverseR(result: List[A], curList: List[A]): List[A] = curList match {
      case Nil       => result
      case h :: tail => reverseR(h :: result, tail)
    }
    reverseR(Nil, ls)
  }

  // Pure functional
  def reverseFunctional[A](ls: List[A]): List[A] =
    ls.foldLeft(List[A]()) { (r, h) => h :: r }
}
// P06 (*) Find out whether a list is a palindrome.
//     Example:
//     scala> isPalindrome(List(1, 2, 3, 2, 1))
//     res0: Boolean = true

object P06 {
  // In theory, we could be slightly more efficient than this.  This approach
  // traverses the list twice: once to reverse it, and once to check equality.
  // Technically, we only need to check the first half of the list for equality
  // with the first half of the reversed list.  The code to do that more
  // efficiently than this implementation is much more complicated, so we'll
  // leave things with this clear and concise implementation.
  def isPalindrome[A](ls: List[A]): Boolean = ls == ls.reverse
}
// P07 (**) Flatten a nested list structure.
//     Example:
//     scala> flatten(List(List(1, 1), 2, List(3, List(5, 8))))
//     res0: List[Any] = List(1, 1, 2, 3, 5, 8)

object P07 {
  def flatten(ls: List[Any]): List[Any] = ls flatMap {
    case ms: List[_] => flatten(ms)
    case e => List(e)
  }
}

// P08 (**) Eliminate consecutive duplicates of list elements.
//     If a list contains repeated elements they should be replaced with a
//     single copy of the element.  The order of the elements should not be
//     changed.
//
//     Example:
//     scala> compress(List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e))
//     res0: List[Symbol] = List('a, 'b, 'c, 'a, 'd, 'e)

object P08 {
  // Standard recursive.
  def compressRecursive[A](ls: List[A]): List[A] = ls match {
    case Nil       => Nil
    case h :: tail => h :: compressRecursive(tail.dropWhile(_ == h))
  }

  // Tail recursive.
  def compressTailRecursive[A](ls: List[A]): List[A] = {
    def compressR(result: List[A], curList: List[A]): List[A] = curList match {
      case h :: tail => compressR(h :: result, tail.dropWhile(_ == h))
      case Nil       => result.reverse
    }
    compressR(Nil, ls)
  }

  // Functional.
  def compressFunctional[A](ls: List[A]): List[A] =
    ls.foldRight(List[A]()) { (h, r) =>
      if (r.isEmpty || r.head != h) h :: r
      else r
    }
}
// P09 (**) Pack consecutive duplicates of list elements into sublists.
//     If a list contains repeated elements they should be placed in separate
//     sublists.
//
//     Example:
//     scala> pack(List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e))
//     res0: List[List[Symbol]] = List(List('a, 'a, 'a, 'a), List('b), List('c, 'c), List('a, 'a),
List('d), List('e, 'e, 'e, 'e))

object P09 {
  def pack[A](ls: List[A]): List[List[A]] = {
    if (ls.isEmpty) List(List())
    else {
      val (packed, next) = ls span { _ == ls.head }
      if (next == Nil) List(packed)
      else packed :: pack(next)
    }
  }
}
// P10 (*) Run-length encoding of a list.
//     Use the result of problem P09 to implement the so-called run-length
//     encoding data compression method.  Consecutive duplicates of elements are
//     encoded as tuples (N, E) where N is the number of duplicates of the
//     element E.
//
//     Example:
//     scala> encode(List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e))
//     res0: List[(Int, Symbol)] = List((4,'a), (1,'b), (2,'c), (2,'a), (1,'d), (4,'e))

object P10 {
  import P09.pack
  def encode[A](ls: List[A]): List[(Int, A)] =
    pack(ls) map { e => (e.length, e.head) }
}
// P11 (*) Modified run-length encoding.
//     Modify the result of problem P10 in such a way that if an element has no
//     duplicates it is simply copied into the result list.  Only elements with
//     duplicates are transferred as (N, E) terms.
//
//     Example:
//     scala> encodeModified(List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e))
//     res0: List[Any] = List((4,'a), 'b, (2,'c), (2,'a), 'd, (4,'e))

object P11 {
  import P10.encode
  def encodeModified[A](ls: List[A]): List[Any] =
    encode(ls) map { t => if (t._1 == 1) t._2 else t }

  // Just for fun, here's a more typesafe version.
  def encodeModified2[A](ls: List[A]): List[Either[A, (Int, A)]] =
    encode(ls) map { t => if (t._1 == 1) Left(t._2) else Right(t) }
}
// P12 (**) Decode a run-length encoded list.
//     Given a run-length code list generated as specified in problem P10,
//     construct its uncompressed version.
//
//     Example:
//     scala> decode(List((4, 'a), (1, 'b), (2, 'c), (2, 'a), (1, 'd), (4, 'e)))
//     res0: List[Symbol] = List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e)

object P12 {
  def decode[A](ls: List[(Int, A)]): List[A] =
    ls flatMap { e => List.make(e._1, e._2) }
}
// P13 (**) Run-length encoding of a list (direct solution).
//     Implement the so-called run-length encoding data compression method
//     directly.  I.e. don't use other methods you've written (like P09's
//     pack); do all the work directly.
//
//     Example:
//     scala> encodeDirect(List('a, 'a, 'a, 'a, 'b, 'c, 'c, 'a, 'a, 'd, 'e, 'e, 'e, 'e))
//     res0: List[(Int, Symbol)] = List((4,'a), (1,'b), (2,'c), (2,'a), (1,'d), (4,'e))

object P13 {
  // This is basically a modification of P09.
  def encodeDirect[A](ls: List[A]): List[(Int, A)] =
    if (ls.isEmpty) Nil
    else {
      val (packed, next) = ls span { _ == ls.head }
      (packed.length, packed.head) :: encodeDirect(next)
    }
}
// P14 (*) Duplicate the elements of a list.
//     Example:
//     scala> duplicate(List('a, 'b, 'c, 'c, 'd))
//     res0: List[Symbol] = List('a, 'a, 'b, 'b, 'c, 'c, 'c, 'c, 'd, 'd)

object P14 {
  def duplicate[A](ls: List[A]): List[A] = ls flatMap { e => List(e, e) }
}
// P15 (**) Duplicate the elements of a list a given number of times.
//     Example:
//     scala> duplicateN(3, List('a, 'b, 'c, 'c, 'd))
//     res0: List[Symbol] = List('a, 'a, 'a, 'b, 'b, 'b, 'c, 'c, 'c, 'c, 'c, 'c, 'd, 'd, 'd)

object P15 {
  def duplicateN[A](n: Int, ls: List[A]): List[A] =
    ls flatMap { List.make(n, _) }
}
// P16 (**) Drop every Nth element from a list.
//     Example:
//     scala> drop(3, List('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k))
//     res0: List[Symbol] = List('a, 'b, 'd, 'e, 'g, 'h, 'j, 'k)

object P16 {
  // Simple recursion.
  def dropRecursive[A](n: Int, ls: List[A]): List[A] = {
    def dropR(c: Int, curList: List[A]): List[A] = (c, curList) match {
      case (_, Nil)       => Nil
      case (1, _ :: tail) => dropR(n, tail)
      case (_, h :: tail) => h :: dropR(c - 1, tail)
    }
    dropR(n, ls)
  }

  // Tail recursive.
  def dropTailRecursive[A](n: Int, ls: List[A]): List[A] = {
    def dropR(c: Int, curList: List[A], result: List[A]): List[A] = (c, curList) match {
      case (_, Nil)       => result.reverse
      case (1, _ :: tail) => dropR(n, tail, result)
      case (_, h :: tail) => dropR(c - 1, tail, h :: result)
    }
    dropR(n, ls, Nil)
  }

  // Functional.
  def dropFunctional[A](n: Int, ls: List[A]): List[A] =
    ls.zipWithIndex filter { v => (v._2 + 1) % n != 0 } map { _._1 }
}
// P17 (*) Split a list into two parts.
//     The length of the first part is given.  Use a Tuple for your result.
//
//     Example:
//     scala> split(3, List('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k))
//     res0: (List[Symbol], List[Symbol]) = (List('a, 'b, 'c),List('d, 'e, 'f, 'g, 'h, 'i, 'j, 'k))

object P17 {
  // Builtin.
  def splitBuiltin[A](n: Int, ls: List[A]): (List[A], List[A]) = ls.splitAt(n)

  // Simple recursion.
  def splitRecursive[A](n: Int, ls: List[A]): (List[A], List[A]) = (n, ls) match {
    case (_, Nil)       => (Nil, Nil)
    case (0, list)      => (Nil, list)
    case (n, h :: tail) => {
      val (pre, post) = splitRecursive(n - 1, tail)
      (h :: pre, post)
    }
  }

  // Tail recursive.
  def splitTailRecursive[A](n: Int, ls: List[A]): (List[A], List[A]) = {
    def splitR(curN: Int, curL: List[A], pre: List[A]): (List[A], List[A]) =
      (curN, curL) match {
        case (_, Nil)       => (pre.reverse, Nil)
        case (0, list)      => (pre.reverse, list)
        case (n, h :: tail) => splitR(n - 1, tail, h :: pre)
      }
    splitR(n, ls, Nil)
  }

  // Functional (barely not "builtin").
  def splitFunctional[A](n: Int, ls: List[A]): (List[A], List[A]) =
    (ls.take(n), ls.drop(n))
}
// P18 (**) Extract a slice from a list.
//     Given two indices, I and K, the slice is the list containing the elements
//     from and including the Ith element up to but not including the Kth
//     element of the original list.  Start counting the elements with 0.
//
//     Example:
//     scala> slice(3, 7, List('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k))
//     res0: List[Symbol] = List('d, 'e, 'f, 'g)

object P18 {
  // Builtin.
  def sliceBuiltin[A](start: Int, end: Int, ls: List[A]): List[A] =
    ls.slice(start, end)

  // Simple recursive.
  def sliceRecursive[A](start: Int, end: Int, ls: List[A]): List[A] =
    (start, end, ls) match {
      case (_, _, Nil)                 => Nil
      case (_, e, _)         if e <= 0 => Nil
      case (s, e, h :: tail) if s <= 0 => h :: sliceRecursive(0, e - 1, tail)
      case (s, e, h :: tail)           => sliceRecursive(s - 1, e - 1, tail)
    }

  // Tail recursive, using pattern matching.
  def sliceTailRecursive[A](start: Int, end: Int, ls: List[A]): List[A] = {
    def sliceR(count: Int, curList: List[A], result: List[A]): List[A] =
      (count, curList) match {
        case (_, Nil)                     => result.reverse
        case (c, h :: tail) if end <= c   => result.reverse
        case (c, h :: tail) if start <= c => sliceR(c + 1, tail, h :: result)
        case (c, _ :: tail)               => sliceR(c + 1, tail, result)
      }
    sliceR(0, ls, Nil)
  }

  // Since several of the patterns are similar, we can condense the tail recursive
  // solution a little.
  def sliceTailRecursive2[A](start: Int, end: Int, ls: List[A]): List[A] = {
    def sliceR(count: Int, curList: List[A], result: List[A]): List[A] = {
      if (curList.isEmpty || count >= end) result.reverse
      else sliceR(count + 1, curList.tail,
                  if (count >= start) curList.head :: result
                  else result)
    }
    sliceR(0, ls, Nil)
  }

  // Functional.
  def sliceFunctional[A](s: Int, e: Int, ls: List[A]): List[A] =
    ls drop s take (e - (s max 0))
}
// P19 (**) Rotate a list N places to the left.
//     Examples:
//     scala> rotate(3, List('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k))
//     res0: List[Symbol] = List('d, 'e, 'f, 'g, 'h, 'i, 'j, 'k, 'a, 'b, 'c)
//
//     scala> rotate(-2, List('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j, 'k))
//     res1: List[Symbol] = List('j, 'k, 'a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i)

object P19 {
  def rotate[A](n: Int, ls: List[A]): List[A] = {
    val nBounded = if (ls.isEmpty) 0 else n % ls.length
    if (nBounded < 0) rotate(nBounded + ls.length, ls)
    else (ls drop nBounded) ::: (ls take nBounded)
  }
}
// P20 (*) Remove the Kth element from a list.
//     Return the list and the removed element in a Tuple.  Elements are
//     numbered from 0.
//
//     Example:
//     scala> removeAt(1, List('a, 'b, 'c, 'd))
//     res0: (List[Symbol], Symbol) = (List('a, 'c, 'd),'b)

object P20 {
  def removeAt[A](n: Int, ls: List[A]): (List[A], A) = ls.splitAt(n) match {
    case (Nil, _) if n < 0 => throw new NoSuchElementException
    case (pre, e :: post)  => (pre ::: post, e)
    case (pre, Nil)        => throw new NoSuchElementException
  }

  // Alternate, with fewer builtins.
  def removeAt2[A](n: Int, ls: List[A]): (List[A], A) =
    if (n < 0) throw new NoSuchElementException
    else (n, ls) match {
      case (_, Nil) => throw new NoSuchElementException
      case (0, h :: tail) => (tail, h)
      case (_, h :: tail) => {
        val (t, e) = removeAt(n - 1, ls.tail)
        (ls.head :: t, e)
      }
    }
}

// P21 (*) Insert an element at a given position into a list.
//     Example:
//     scala> insertAt('new, 1, List('a, 'b, 'c, 'd))
//     res0: List[Symbol] = List('a, 'new, 'b, 'c, 'd)

object P21 {
  def insertAt[A](e: A, n: Int, ls: List[A]): List[A] = ls.splitAt(n) match {
    case (pre, post) => pre ::: e :: post
  }
}
// P22 (*) Create a list containing all integers within a given range.
```

```scala
//     Example:
//     scala> range(4, 9)
//     res0: List[Int] = List(4, 5, 6, 7, 8, 9)

object P22 {
  // Builtin.
  def rangeBuiltin(start: Int, end: Int): List[Int] = List.range(start, end + 1)

  // Recursive.
  def rangeRecursive(start: Int, end: Int): List[Int] =
    if (end < start) Nil
    else start :: rangeRecursive(start + 1, end)

  // Tail recursive.
  def rangeTailRecursive(start: Int, end: Int): List[Int] = {
    def rangeR(end: Int, result: List[Int]): List[Int] = {
      if (end < start) result
      else rangeR(end - 1, end :: result)
    }
    rangeR(end, Nil)
  }

  // The classic functional approach would be to use `unfoldr`, which Scala
  // doesn't have.  So we'll write one and then use it.
  def unfoldRight[A, B](s: B)(f: B => Option[(A, B)]): List[A] =
    f(s) match {
      case None         => Nil
      case Some((r, n)) => r :: unfoldRight(n)(f)
    }
  def rangeFunctional(start: Int, end: Int): List[Int] =
    unfoldRight(start) { n =>
      if (n > end) None
      else Some((n, n + 1))
    }
}
// P23 (**) Extract a given number of randomly selected elements from a list.
//     Example:
//     scala> randomSelect(3, List('a, 'b, 'c, 'd, 'f, 'g, 'h))
//     res0: List[Symbol] = List('e, 'd, 'a)
//
//     Hint: Use the answer to P20.

object P23 {
  import P20.removeAt

  def randomSelect1[A](n: Int, ls: List[A]): List[A] =
    if (n <= 0) Nil
    else {
      val (rest, e) = removeAt((new util.Random).nextInt(ls.length), ls)
      e :: randomSelect1(n - 1, rest)
    }

  // It can be expensive to create a new Random instance every time, so let's
  // only do it once.
  def randomSelect[A](n: Int, ls: List[A]): List[A] = {
    def randomSelectR(n: Int, ls: List[A], r: util.Random): List[A] =
      if (n <= 0) Nil
      else {
        val (rest, e) = removeAt(r.nextInt(ls.length), ls)
        e :: randomSelectR(n - 1, rest, r)
      }
    randomSelectR(n, ls, new util.Random)
  }
}

// P24 (*) Lotto: Draw N different random numbers from the set 1..M.
//     Example:
//     scala> lotto(6, 49)
//     res0: List[Int] = List(23, 1, 17, 33, 21, 37)

object P24 {
  import P23.randomSelect
  def lotto(count: Int, max: Int): List[Int] =
    randomSelect(count, List.range(1, max + 1))
}
// P25 (*) Generate a random permutation of the elements of a list.
//     Hint: Use the solution of problem P23.
//
//     Example:
//     scala> randomPermute(List('a, 'b, 'c, 'd, 'e, 'f))
//     res0: List[Symbol] = List('b, 'a, 'd, 'c, 'e, 'f)

object P25 {
  // This algorithm is O(n^2), but it makes up for that in simplicity of
  // implementation.
  import P23.randomSelect
  def randomPermute1[A](ls: List[A]): List[A] = randomSelect(ls.length, ls)

  // The canonical way to shuffle imperatively is Fisher-Yates.  It requires a
  // mutable array.  This is O(n).
  def randomPermute[A](ls: List[A]): List[A] = {
    val rand = new util.Random
    val a = ls.toArray
    for (i <- a.length - 1 to 1 by -1) {
      val i1 = rand.nextInt(i + 1)
      val t = a(i)
      a.update(i, a(i1))
      a.update(i1, t)
    }
    a.toList
  }

  // Efficient purely functional algorithms for shuffling are a lot harder.  One
  // is described in http://okmij.org/ftp/Haskell/perfect-shuffle.txt using
  // Haskell. Implementing it in Scala is left as an exercise for the reader.
}
// P26 (**) Generate the combinations of K distinct objects chosen from the N
//          elements of a list.
//     In how many ways can a committee of 3 be chosen from a group of 12
//     people?  We all know that there are C(12,3) = 220 possibilities (C(N,K)
//     denotes the well-known binomial coefficient).  For pure mathematicians,
//     this result may be great.  But we want to really generate all the possibilities.
//
//     Example:
//     scala> combinations(3, List('a, 'b, 'c, 'd, 'e, 'f))
//     res0: List[List[Symbol]] = List(List('a, 'b, 'c), List('a, 'b, 'd), List('a, 'b, 'e), ...

object P26 {
  // flatMapSublists is like list.flatMap, but instead of passing each element
  // to the function, it passes successive sublists of L.
  def flatMapSublists[A,B](ls: List[A])(f: (List[A]) => List[B]): List[B] =
    ls match {
      case Nil => Nil
      case sublist@(_ :: tail) => f(sublist) ::: flatMapSublists(tail)(f)
    }

  def combinations[A](n: Int, ls: List[A]): List[List[A]] =
    if (n == 0) List(Nil)
    else flatMapSublists(ls) { sl =>
      combinations(n - 1, sl.tail) map {sl.head :: _}
    }
}
// P27 (**) Group the elements of a set into disjoint subsets.
//     a) In how many ways can a group of 9 people work in 3 disjoint subgroups
//        of 2, 3 and 4 persons?  Write a function that generates all the
//        possibilities.
//
//        Example:
//        scala> group3(List("Aldo", "Beat", "Carla", "David", "Evi", "Flip", "Gary", "Hugo",
"Ida"))
//        res0: List[List[List[String]]] = List(List(List(Aldo, Beat), List(Carla, David, Evi),
List(Flip, Gary, Hugo, Ida)), ...
//
//     b) Generalize the above predicate in a way that we can specify a list
//        of group sizes and the predicate will return a list of groups.
//
//        Example:
//        scala> group(List(2, 2, 5), List("Aldo", "Beat", "Carla", "David", "Evi", "Flip", "Gary",
"Hugo", "Ida"))
//        res0: List[List[List[String]]] = List(List(List(Aldo, Beat), List(Carla, David), List(Evi,
Flip, Gary, Hugo, Ida)), ...
//
//     Note that we do not want permutations of the group members;
//     i.e. ((Aldo, Beat), ...) is the same solution as ((Beat, Aldo), ...).
//     However, we make a difference between ((Aldo, Beat), (Carla, David), ...)
//     and ((Carla, David), (Aldo, Beat), ...).
//
//     You may find more about this combinatorial problem in a good book on
//     discrete mathematics under the term "multinomial coefficients".
object P27 {
  import P26.combinations

  def group3[A](ls: List[A]): List[List[List[A]]] =
    for {
      a <- combinations(2, ls)
      noA = ls -- a
      b <- combinations(3, noA)
    } yield List(a, b, noA -- b)

  def group[A](ns: List[Int], ls: List[A]): List[List[List[A]]] = ns match {
    case Nil     => List(Nil)
    case n :: ns => combinations(n, ls) flatMap { c =>
      group(ns, ls -- c) map {c :: _}
    }
  }
}
// P28 (**) Sorting a list of lists according to length of sublists.
//     a) We suppose that a list contains elements that are lists themselves.
//        The objective is to sort the elements of the list according to their
//        length.  E.g. short lists first, longer lists later, or vice versa.
//
//     Example:
//     scala> lsort(List(List('a, 'b, 'c), List('d, 'e), List('f, 'g, 'h), List('d, 'e), List('i,
'j, 'k, 'l), List('m, 'n), List('o)))
//     res0: List[List[Symbol]] = List(List('o), List('d, 'e), List('d, 'e), List('m, 'n), List('a,
'b, 'c), List('f, 'g, 'h), List('i, 'j, 'k, 'l))
//
//     b) Again, we suppose that a list contains elements that are lists
//        themselves.  But this time the objective is to sort the elements
//        according to their length frequency; i.e. in the default, sorting is
//        done ascendingly, lists with rare lengths are placed, others with a
//        more frequent length come later.
//
//     Example:
//     scala> lsortFreq(List(List('a, 'b, 'c), List('d, 'e), List('f, 'g, 'h), List('d, 'e),
List('i, 'j, 'k, 'l), List('m, 'n), List('o)))
//     res1: List[List[Symbol]] = List(List('i, 'j, 'k, 'l), List('o), List('a, 'b, 'c), List('f,
'g, 'h), List('d, 'e), List('d, 'e), List('m, 'n))
//
//     Note that in the above example, the first two lists in the result have
//     length 4 and 1 and both lengths appear just once.  The third and fourth
//     lists have length 3 and there are two list of this length.  Finally, the
//     last three lists have length 2.  This is the most frequent length.

object P28 {
  import P10.encode

  def lsort[A](ls: List[List[A]]): List[List[A]] =
    ls sort { _.length < _.length }

  def lsortFreq[A](ls: List[List[A]]): List[List[A]] = {
    val freqs = Map(encode(ls map { _.length } sort { _ < _ }) map { _.swap }:_*)
    ls sort { (e1, e2) => freqs(e1.length) < freqs(e2.length) }
  }
}
// P31 (**) Determine whether a given integer number is prime.
//     scala> 7.isPrime
//     res0: Boolean = true

// A fairly naive implementation for primality testing is simply: a number is
// prime if it it not divisible by any prime number less than or equal to its
// square root.
// Here, we use a Stream to create a lazy infinite list of prime numbers.  The
// mutual recursion between `primes` and `isPrime` works because of the limit
// on `isPrime` to the square root of the number being tested.

class S99Int(val start: Int) {
  def isPrime: Boolean =
    (start > 1) && (primes takeWhile { _ <= Math.sqrt(start) } forall { start % _ != 0 })
}

object S99Int {
  val primes = Stream.cons(2, Stream.from(3, 2) filter { _.isPrime })
}

// Readers interested in more sophisticated (and more efficient) primality tests
// are invited to read http://primes.utm.edu/prove/index.html .  Implementation
// in Scala is left as an exercise for the reader.
// Similarly, a more efficient, functional, lazy, infinite prime list can be found
// at http://article.gmane.org/gmane.comp.lang.haskell.cafe/19470 .  (Haskell
// implementation.)
// P32 (**) Determine the greatest common divisor of two positive integer
//          numbers.
//     Use Euclid's algorithm.
//
//     scala> gcd(36, 63)
//     res0: Int = 9

object S99Int {
  def gcd(m: Int, n: Int): Int = if (n == 0) m else gcd(n, m % n)
}
// P33 (*) Determine whether two positive integer numbers are coprime.
//     Two numbers are coprime if their greatest common divisor equals 1.
//
//     scala> 35.isCoprimeTo(64)
//     res0: Boolean = true

class S99Int(val start: Int) {
  def isCoprimeTo(n: Int): Boolean = gcd(start, n) == 1
}
// P34 (**) Calculate Euler's totient function phi(m).
//     Euler's so-called totient function phi(m) is defined as the number of
//     positive integers r (1 <= r < m) that are coprime to m.  As a special
//     case, phi(1) is defined to be 1.
//
//     scala> 10.totient
//     res0: Int = 4

class S99Int(val start: Int) {
  def totient: Int = (1 to start) filter { start.isCoprimeTo(_) } length
}
// P35 (**) Determine the prime factors of a given positive integer.
//     Construct a flat list containing the prime factors in ascending order.
//
//     scala> 315.primeFactors
//     res0: List[Int] = List(3, 3, 5, 7)

class S99Int(val start: Int) {
  def primeFactors: List[Int] = {
    def primeFactorsR(n: Int, ps: Stream[Int]): List[Int] =
      if (n.isPrime) List(n)
      else if (n % ps.head == 0) ps.head :: primeFactorsR(n / ps.head, ps)
      else primeFactorsR(n, ps.tail)
    primeFactorsR(start, primes)
  }
}
// P36 (**) Determine the prime factors of a given positive integer (2).
//     Construct a list containing the prime factors and their multiplicity.
//
//     scala> 315.primeFactorMultiplicity
//     res0: List[(Int, Int)] = List((3,2), (5,1), (7,1))
//
//     Alternately, use a Map for the result.
//     scala> 315.primeFactorMultiplicity
//     res0: Map[Int,Int] = Map(3 -> 2, 5 -> 1, 7 -> 1)

/*
// One approach is to reuse the solution from P10.
class S99Int(val start: Int) {
  import P10.encode
  def primeFactorMultiplicity: List[(Int,Int)] =
    encode(start.primeFactors) map { _.swap }
}
*/

// But we can do it directly.
class S99Int(val start: Int) {
  def primeFactorMultiplicity: Map[Int,Int] = {
    def factorCount(n: Int, p: Int): (Int,Int) =
      if (n % p != 0) (0, n)
      else factorCount(n / p, p) match { case (c, d) => (c + 1, d) }
    def factorsR(n: Int, ps: Stream[Int]): Map[Int, Int] =
      if (n == 1) Map()
      else if (n.isPrime) Map(n -> 1)
      else {
        val nps = ps.dropWhile(n % _ != 0)
        val (count, dividend) = factorCount(n, nps.head)
        Map(nps.head -> count) ++ factorsR(dividend, nps.tail)
      }
    factorsR(start, primes)
  }

  // This also lets us change primeFactors.
  def primeFactors: List[Int] =
    start.primeFactorMultiplicity flatMap { v => List.make(v._2, v._1) } toList
}
// P37 (**) Calculate Euler's totient function phi(m) (improved).
//     See problem P34 for the definition of Euler's totient function.  If the
//     list of the prime factors of a number m is known in the form of problem
//     P36 then the function phi(m>) can be efficiently calculated as follows:
//     Let [[p_1, m_1], [p_2, m_2], [p_3, m_3], ...] be the list of prime
//     factors (and their multiplicities) of a given number m.  Then phi(m) can
//     be calculated with the following formula:
//
//     phi(m) = (p_1-1)*p_1^(m_1-1) * (p_2-1)*p_2^(m_2-1) * (p_3-1)*p_3^(m_3-1) * ...
//
//     Note that a^b stands for the bth power of a.

class S99Int(val start: Int) {
  def totient: Int = start.primeFactorMultiplicity.foldLeft(1) { (r, f) =>
    f match { case (p, m) => r * (p - 1) * Math.pow(p, m - 1).toInt }
  }
}
// P38 (*) Compare the two methods of calculating Euler's totient function.
//     Use the solutions of problems P34 and P37 to compare the algorithms.  Try
//     to calculate phi(10090) as an example.

// Here's an object that will test the relative execution times of the two
// approaches.
// On a 2.4 GHz Athlon 64 X2, here's what happens the first time `test` is called:
//   Preload primes: 20 ms.
//   P34 (10090): 65 ms.
//   P37 (10090): 3 ms.
//
// The JVM tends to profile its execution, though.  Here's a several-iteration run.
//   scala> import P38._
//   import P38._
//
//   scala> test(10090)
//   Preload primes: 9 ms.
//   P34 (10090): 53 ms.
//   P37 (10090): 4 ms.
//
//   scala> test(10090)
//   Preload primes: 2 ms.
//   P34 (10090): 28 ms.
//   P37 (10090): 1 ms.
//
//   scala> test(10090)
//   Preload primes: 2 ms.
//   P34 (10090): 17 ms.
//   P37 (10090): 1 ms.
//
//   scala> test(10090)
//   Preload primes: 2 ms.
//   P34 (10090): 3 ms.
//   P37 (10090): 0 ms.
//
//   scala> test(10090)
//   Preload primes: 4 ms.
//   P34 (10090): 2 ms.
//   P37 (10090): 0 ms.

object P38 {
  import arithmetic.S99Int._

  def time[A](label: String)(block: => A): A = {
    val now = System.currentTimeMillis()
    val ret = block
    println(label + ": " + (System.currentTimeMillis() - now) + " ms.")
    ret
  }

  def test(n: Int) {
    time("Preload primes") {
      primes takeWhile { _ <= Math.sqrt(n) } force
    }
    time("P34 (" + n + ")") {
      n.totientP34
    }
    time("P37 (" + n + ")") {
      n.totient
    }
  }
}
// P39 (*) A list of prime numbers.
//     Given a range of integers by its lower and upper limit, construct a list
//     of all prime numbers in that range.
//
//     scala> listPrimesinRange(7 to 31)
//     res0: List[Int] = List(7, 11, 13, 17, 19, 23, 29, 31)

object S99Int {
  def listPrimesinRange(r: Range): List[Int] =
    primes dropWhile { _ < r.first } takeWhile { _ <= r.last } toList
}
// P40 (**) Goldbach's conjecture.
//     Goldbach's conjecture says that every positive even number greater than 2
//     is the sum of two prime numbers.  E.g. 28 = 5 + 23.  It is one of the
//     most famous facts in number theory that has not been proved to be correct
//     in the general case.  It has been numerically confirmed up to very large
//     numbers (much larger than Scala's Int can represent).  Write a function
//     to find the two prime numbers that sum up to a given even integer.
//
//     scala> 28.goldbach
//     res0: (Int, Int) = (5,23)

class S99Int(val start: Int) {
  def goldbach: (Int,Int) =
    primes takeWhile { _ < start } find { p => (start - p).isPrime } match {
      case None     => throw new IllegalArgumentException
      case Some(p1) => (p1, start - p1)
    }
}
// P41 (**) A list of Goldbach compositions.
//     Given a range of integers by its lower and upper limit, print a list of
//     all even numbers and their Goldbach composition.
//
//     scala> printGoldbachList(9 to 20)
//     10 = 3 + 7
//     12 = 5 + 7
//     14 = 3 + 11
//     16 = 3 + 13
//     18 = 5 + 13
//     20 = 3 + 17
//
//     In most cases, if an even number is written as the sum of two prime
//     numbers, one of them is very small.  Very rarely, the primes are both
//     bigger than, say, 50.  Try to find out how many such cases there are in
//     the range 2..3000.
//
//     Example (minimum value of 50 for the primes):
//     scala> printGoldbachListLimited(1 to 2000, 50)
//     992 = 73 + 919
//     1382 = 61 + 1321
//     1856 = 67 + 1789
//     1928 = 61 + 1867

object S99Int {
  def printGoldbachList(r: Range) {
    printGoldbachListLimited(r, 0)
  }

  def printGoldbachListLimited(r: Range, limit: Int) {
    (r filter { n => n > 2 && n % 2 == 0 } map { n => (n, n.goldbach) }
     filter { _._2._1 >= limit } foreach {
       _ match { case (n, (p1, p2)) => println(n + " = " + p1 + " + " + p2) }
     })
  }
}
// P46 (**) Truth tables for logical expressions.
//     Define functions and, or, nand, nor, xor, impl, and equ (for logical
//     equivalence) which return true or false according to the result of their
//     respective operations; e.g. and(A, B) is true if and only if both A and B
//     are true.
//
//     scala> and(true, true)
//     res0: Boolean = true
```

```
//
//     scala> xor(true. true)
//     res1: Boolean = false
//
//     A logical expression in two variables can then be written as a function of
//     two variables, e.g: (a: Boolean, b: Boolean) => and(or(a, b), nand(a, b))
//
//     Now, write a function called table2 which prints the truth table of a
//     given logical expression in two variables.
//
//     scala> table2((a: Boolean, b: Boolean) => and(a, or(a, b)))
//     A     B     result
//     true  true  true
//     true  false true
//     false true  false
//     false false false

// The trick here is not using builtins.  We'll define `not`, `and`, and `or`
// directly (using pattern matching), and the other functions in terms of those
// three.

object S99Logic {
  def not(a: Boolean) = a match {
    case true  => false
    case false => true
  }
  def and(a: Boolean, b: Boolean): Boolean = (a, b) match {
    case (true, true) => true
    case _            => false
  }
  def or(a: Boolean, b: Boolean): Boolean = (a, b) match {
    case (true, _) => true
    case (_, true) => true
    case _         => false
  }
  def equ(a: Boolean, b: Boolean): Boolean = or(and(a, b), and(not(a), not(b)))
  def xor(a: Boolean, b: Boolean): Boolean = not(equ(a, b))
  def nor(a: Boolean, b: Boolean): Boolean = not(or(a, b))
  def nand(a: Boolean, b: Boolean): Boolean = not(and(a, b))
  def impl(a: Boolean, b: Boolean): Boolean = or(not(a), b)

  def table2(f: (Boolean, Boolean) => Boolean) {
    println("A     B     result")
    for {a <- List(true, false);
         b <- List(true, false)} {
     printf("%-5s %-5s %-5s\n", a, b, f(a, b))
    }
  }
}
// P47 (*) Truth tables for logical expressions (2).
//     Continue problem P46 by redefining and, or, etc as operators.  (i.e. make
//     them methods of a new class with an implicit conversion from Boolean.)
//     not will have to be left as a object method.
//
//     scala> table2((a: Boolean, b: Boolean) => a and (a or not(b)))
//     A     B     result
//     true  true  true
//     true  false true
//     false true  false
//     false false false

// For simplicity, we remove `and`, `or`, `equ`, `xor`, `nor`, `nand`, and
// `impl` from the S99Logic object before putting them into a new class.

class S99Logic(a: Boolean) {
  import S99Logic._

  def and(b: Boolean): Boolean = (a, b) match {
    case (true, true) => true
    case _            => false
  }
  def or(b: Boolean): Boolean = (a, b) match {
    case (true, _) => true
    case (_, true) => true
    case _         => false
  }
  def equ(b: Boolean): Boolean = (a and b) or (not(a) and not(b))
  def xor(b: Boolean): Boolean = not(a equ b)
  def nor(b: Boolean): Boolean = not(a or b)
  def nand(b: Boolean): Boolean = not(a and b)
  def impl(b: Boolean): Boolean = not(a) or b
}

object S99Logic {
  implicit def boolean2S99Logic(a: Boolean): S99Logic = new S99Logic(a)
}
// P49 (**) Gray code.
//     An n-bit Gray code is a sequence of n-bit strings constructed according
//     to certain rules. For example,
//     n = 1: C(1) = ("0", "1").
//     n = 2: C(2) = ("00", "01", "11", "10").
//     n = 3: C(3) = ("000", "001", "011", "010", "110", "111", "101", "100").
//
//     Find out the construction rules and write a function to generate Gray
//     codes.
//
//     scala> gray(3)
//     res0 List[String] = List(000, 001, 011, 010, 110, 111, 101, 100)
//
//     See if you can use memoization to make the function more
//     efficient.

object P49 {
  def gray(n: Int): List[String] =
    if (n == 0) List("")
    else {
      val lower = gray(n - 1)
      (lower map { "0" + _ }) ::: (lower.reverse map { "1" + _ })
    }

  import scala.collection.mutable
  private val strings = mutable.Map(0 -> List(""))
  def grayMemoized(n: Int): List[String] = {
    if (!strings.contains(n)) {
      strings + (n -> ((grayMemoized(n - 1) map { "0" + _ }) :::
                       (grayMemoized(n - 1).reverse map { "1" + _ })))
    }
    strings(n)
  }
}
// P50 (***) Huffman code.
//     First of all, consult a good book on discrete mathematics or algorithms
//     for a detailed description of Huffman codes!
//
//     We suppose a set of symbols with their frequencies, given as a list of
//     (S, F) Tuples.  E.g. (("a", 45), ("b", 13), ("c", 12), ("d", 16),
//     ("e", 9), ("f", 5)).  Our objective is to construct a list of (S, C)
//     Tuples, where C is the Huffman code word for the symbol S.
//
//     scala> huffman(List(("a", 45), ("b", 13), ("c", 12), ("d", 16), ("e", 9), ("f", 5)))
//     res0: List[String, String] = List((a,0), (b,101), (c,100), (d,111), (e,1101), (f,1100))

// We'll do this functionally, with the two-queue algorithm.  (Scala's priority
// queues are mutable.)
object P50 {
  private abstract sealed class Tree[A] {
    val freq: Int
    def toCode: List[(A, String)] = toCodePrefixed("")
    def toCodePrefixed(prefix: String): List[(A, String)]
  }
  private final case class InternalNode[A](left: Tree[A], right: Tree[A]) extends Tree[A] {
    val freq: Int = left.freq + right.freq
    def toCodePrefixed(prefix: String): List[(A, String)] =
      left.toCodePrefixed(prefix + "0") ::: right.toCodePrefixed(prefix + "1")
  }
  private final case class LeafNode[A](element: A, freq: Int) extends Tree[A] {
    def toCodePrefixed(prefix: String): List[(A, String)] = List((element, prefix))
  }

  def huffman[A](ls: List[(A, Int)]): List[(A, String)] = {
    import collection.immutable.Queue
    def dequeueSmallest(q1: Queue[Tree[A]], q2: Queue[Tree[A]]) = {
      // This ordering chooses q1 in case of ties, which helps minimize tree
      // depth.
      if (q2.isEmpty) (q1.front, q1.dequeue._2, q2)
      else if (q1.isEmpty || q2.front.freq < q1.front.freq) (q2.front, q1, q2.dequeue._2)
      else (q1.front, q1.dequeue._2, q2)
    }
    def huffmanR(q1: Queue[Tree[A]], q2: Queue[Tree[A]]): List[(A, String)] = {
      if (q1.length + q2.length == 1) (if (q1.isEmpty) q2.front else q1.front).toCode
      else {
        val (v1, q3, q4) = dequeueSmallest(q1, q2)
        val (v2, q5, q6) = dequeueSmallest(q3, q4)
        huffmanR(q5, q6.enqueue(InternalNode(v1, v2)))
      }
    }
    huffmanR(Queue.Empty.enqueue(ls sort { _._2 < _._2 } map { e => LeafNode(e._1, e._2) }),
             Queue.Empty)
  }
}
// P55 (**) Construct completely balanced binary trees.
//     In a completely balanced binary tree, the following property holds for
//     every node: The number of nodes in its left subtree and the number of
//     nodes in its right subtree are almost equal, which means their difference
//     is not greater than one.
//
//     Define an object named Tree.  Write a function Tree.cBalanced to
//     construct completely balanced binary trees for a given number of nodes.
//     The function should generate all solutions.  The function should take as
//     parameters the number of nodes and a single value to put in all of them.
//
//     scala> Tree.cBalanced(4, "x")
//     res0: List(Node[String]) = List(T(x T(x . .) T(x . T(x . .))), T(x T(x . .) T(x T(x . .) .)),
...

object Tree {
  def cBalanced[T](nodes: Int, value: T): List[Tree[T]] = nodes match {
    case n if n < 1 => List(End)
    case n if n % 2 == 1 => {
      val subtrees = cBalanced(n / 2, value)
      subtrees.flatMap(l => subtrees.map(r => Node(value, l, r)))
    }
    case n if n % 2 == 0 => {
      val lesserSubtrees = cBalanced((n - 1) / 2, value)
      val greaterSubtrees = cBalanced((n - 1) / 2 + 1, value)
      lesserSubtrees.flatMap(l => greaterSubtrees.flatMap(g => List(Node(value, l, g), Node(value,
g, l))))
    }
  }
}
// P56 (**) Symmetric binary trees.
//     Let us call a binary tree symmetric if you can draw a vertical line
//     through the root node and then the right subtree is the mirror image of
//     the left subtree.  Add an isSymmetric method to the Tree class to check
//     whether a given binary tree is symmetric.  Hint: Write an isMirrorOf
//     method first to check whether one tree is the mirror image of another.
//     We are only interested in the structure, not in the contents of the
//     nodes.
//
//     scala> Node('a', Node('b'), Node('c')).isSymmetric
//     res0: Boolean = true

sealed abstract class Tree[+T] {
  def isMirrorOf[V](tree: Tree[V]): Boolean
  def isSymmetric: Boolean
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def isMirrorOf[V](tree: Tree[V]): Boolean = tree match {
    case t: Node[V] => left.isMirrorOf(t.right) && right.isMirrorOf(t.left)
    case _          => false
  }
  def isSymmetric: Boolean = left.isMirrorOf(right)
}

case object End extends Tree[Nothing] {
  def isMirrorOf[V](tree: Tree[V]): Boolean = tree == End
  def isSymmetric: Boolean = true
}
// P57 (**) Binary search trees (dictionaries).
//     Write a function to add an element to a binary search tree.
//
//     scala> End.addValue(2)
//     res0: Node[Int] = T(2 . .)
//
//     scala> res0.addValue(3)
//     res1: Node[Int] = T(2 . T(3 . .))
//
//     scala> res1.addValue(0)
//     res2: Node[Int] = T(2 T(0 . .) T(3 . .))
```

```
//
//     Hint: The abstract definition of addValue in Tree should be
//     `def addValue[U >: T <% Ordered[U]](x: U): Tree[U]`.  The `>: T` is
//     because addValue's parameters need to be _contravariant_ in T.
//     (Conceptually, we're adding nodes above existing nodes.  In order for the
//     subnodes to be of type T or any subtype, the upper nodes must be of type
//     T or any supertype.)  The `<% Ordered[U]` allows us to use the < operator
//     on the values in the tree.
//
//     Use that function to construct a binary tree from a list of integers.
//
//     scala> Tree.fromList(List(3, 2, 5, 7, 1))
//     res3: Node[Int] = T(3 T(2 T(1 . .) .) T(5 . T(7 . .)))
//
//     Finally, use that function to test your solution to P56.
//
//     scala> Tree.fromList(List(5, 3, 18, 1, 4, 12, 21)).isSymmetric
//     res4: Boolean = true
//
//     scala> Tree.fromList(List(3, 2, 5, 7, 4)).isSymmetric
//     res5: Boolean = false

sealed abstract class Tree[+T] {
  def addValue[U >: T <% Ordered[U]](x: U): Tree[U]
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def addValue[U >: T <% Ordered[U]](x: U) =
    if (x < value) Node(value, left.addValue(x), right)
    else Node(value, left, right.addValue(x))
}

case object End extends Tree[Nothing] {
  // We still need the view bound here because it's just syntatic sugar for
  // `def addValue[U](x: U)(implicit f: (U) => Ordered[U])` and if we left it
  // out, the compiler would see the different parameter list and think we
  // were overloading `addValue` instead of overriding it.
  def addValue[U <% Ordered[U]](x: U) = Node(x)
}

object Tree {
  def fromList[T <% Ordered[T]](l: List[T]): Tree[T] =
    l.foldLeft(End: Tree[T])((r, e) => r.addValue(e))
}
// P58 (**) Generate-and-test paradigm.
//     Apply the generate-and-test paradigm to construct all symmetric,
//     completely balanced binary trees with a given number of nodes.
//
//     scala> Tree.symmetricBalancedTrees(5, "x")
//     res0: List[Node[String]] = List(T(x T(x . T(x . .)) T(x T(x . .) .)), T(x T(x T(x . .) .) T(x
. T(x . .))))

object Tree {
  def symmetricBalancedTrees[T](nodes: Int, value: T): List[Tree[T]] =
    cBalanced(nodes, value).filter(_.isSymmetric)
}
// P59 (**) Construct height-balanced binary trees.
//     In a height-balanced binary tree, the following property holds for every
//     node: The height of its left subtree and the height of its right subtree
//     are almost equal, which means their difference is not greater than one.
//
//     Write a method Tree.hbalTrees to construct height-balanced binary trees
//     for a given height with a supplied value for the nodes.  The function
//     should generate all solutions.
//
//     scala> Tree.hbalTrees(3, "x")
//     res0: List[Node[String]] = List(T(x T(x T(x . .) T(x . .)) T(x T(x . .) T(x . .))), T(x T(x
T(x . .) T(x . .)) T(x T(x . .) .)), ...

object Tree {
  def hbalTrees[T](height: Int, value: T): List[Tree[T]] = height match {
    case n if n < 1 => List(End)
    case 1          => List(Node(value))
    case _ => {
      val fullHeight = hbalTrees(height - 1, value)
      val short = hbalTrees(height - 2, value)
      fullHeight.flatMap((l) => fullHeight.map((r) => Node(value, l, r))) :::
      fullHeight.flatMap((f) => short.flatMap((s) => List(Node(value, f, s), Node(value, s, f))))
    }
  }
```

```
}
// P60 (**) Construct height-balanced binary trees with a given number of nodes.
//     Consider a height-balanced binary tree of height H.  What is the maximum
//     number of nodes it can contain?  Clearly, MaxN = 2H - 1.  However, what
//     is the minimum number MinN?  This question is more difficult.  Try to
//     find a recursive statement and turn it into a function minHbalNodes that
//     takes a height and returns MinN.
//
//     scala> minHbalNodes(3)
//     res0: Int = 4
//
//     On the other hand, we might ask: what is the maximum height H a
//     height-balanced binary tree with N nodes can have?  Write a maxHbalHeight
//     function.
//
//     scala> maxHbalHeight(4)
//     res1: Int = 3
//
//     Now, we can attack the main problem: construct all the height-balanced
//     binary trees with a given nuber of nodes.
//
//     scala> Tree.hbalTreesWithNodes(4, "x")
//     res2: List[Node[String]] = List(T(x T(x T(x . .) .) T(x . .)), T(x T(x . T(x . .)) T(x . .)),
...
//
//     Find out how many height-balanced trees exist for N = 15.

sealed abstract class Tree[+T] {
  def nodeCount: Int
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def nodeCount: Int = left.nodeCount + right.nodeCount + 1
}

case object End extends Tree[Nothing] {
  def nodeCount: Int = 0
}

object Tree {
  def minHbalNodes(height: Int): Int = height match {
    case n if n < 1 => 0
    case 1          => 1
    case n          => minHbalNodes(n - 1) + minHbalNodes(n - 2) + 1
  }
  def maxHbalNodes(height: Int): Int = 2 * height - 1
  def minHbalHeight(nodes: Int): Int =
    if (nodes == 0) 0
    else minHbalHeight(nodes / 2) + 1
  def maxHbalHeight(nodes: Int): Int =
    Stream.from(1).takeWhile(minHbalNodes(_) <= nodes).last
  def hbalTreesWithNodes[T](nodes: Int, value: T): List[Tree[T]] =
    (minHbalHeight(nodes) to maxHbalHeight(nodes)).flatMap(hbalTrees(_, value)).filter(_.nodeCount
== nodes).toList
}
// 61A (*) Collect the leaves of a binary tree in a list.
//     A leaf is a node with no successors.  Write a method leafList to
//     collect them in a list.
//
//     scala> Node('a', Node('b'), Node('c', Node('d'), Node('e'))).leafList
//     res0: List[Char] = List(b, d, e)

// Note that leafCount from P61 is no longer an abstract method.

sealed abstract class Tree[+T] {
  def leafCount: Int = leafList.length
  def leafList: List[T]
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def leafList: List[T] = (left, right) match {
    case (End, End) => List(value)
    case _          => left.leafList ::: right.leafList
  }
}

case object End extends Tree[Nothing] {
  def leafList = Nil
}
// P61 (*) Count the leaves of a binary tree.
//     A leaf is a node with no successors.  Write a method leafCount to count
//     them.
//
//     scala> Node('x', Node('x'), End).leafCount
//     res0: Int = 1

sealed abstract class Tree[+T] {
  def leafCount: Int
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def leafCount: Int = (left, right) match {
    case (End, End) => 1
    case _          => left.leafCount + right.leafCount
  }
}

case object End extends Tree[Nothing] {
  def leafCount: Int = 0
}
// P62B (*) Collect the nodes at a given level in a list.
//      A node of a binary tree is at level N if the path from the root to the
//      node has length N-1.  The root node is at level 1.  Write a method
//      atLevel to collect all nodes at a given level in a list.
//
//      scala> Node('a', Node('b'), Node('c', Node('d'), Node('e'))).atLevel(2)
//      res0: List[Char] = List(b, c)
//
//      Using atLevel it is easy to construct a method levelOrder which creates
//      the level-order sequence of the nodes.  However, there are more
//      efficient ways to do that.

sealed abstract class Tree[+T] {
  def atLevel(level: Int): List[T]
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def atLevel(level: Int): List[T] = level match {
    case n if n < 1 => Nil
    case 1          => List(value)
    case n          => left.atLevel(n - 1) ::: right.atLevel(n - 1)
  }
}

case object End extends Tree[Nothing] {
  def atLevel(level: Int) = Nil
}
// P62 (*) Collect the internal nodes of a binary tree in a list.
//     An internal node of a binary tree has either one or two non-empty
//     successors.  Write a method internalList to collect them in a list.
//
//     scala> Node('a', Node('b'), Node('c', Node('d'), Node('e'))).internalList
//     res0: List[Char] = List(a, c)

sealed abstract class Tree[+T] {
  def internalList: List[T]
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def internalList: List[T] = (left, right) match {
    case (End, End) => Nil
    case _          => value :: left.internalList ::: right.internalList
  }
}

case object End extends Tree[Nothing] {
  def internalList = Nil
}
// P63 (**) Construct a complete binary tree.
//     A complete binary tree with height H is defined as follows: The levels
//     1,2,3,...,H-1 contain the maximum number of nodes (i.e 2^(i-1) at the
//     level i, note that we start counting the levels from 1 at the root).  In
//     level H, which may contain less than the maximum possible number of
//     nodes, all the nodes are "left-adjusted".  This means that in a
//     levelorder tree traversal all internal nodes come first, the leaves come
//     second, and empty successors (the Ends which are not really nodes!) come
//     last.
//
//     Particularly, complete binary trees are used as data structures (or
//     addressing schemes) for heaps.
```

```
//
//     We can assign an address number to each node in a complete binary tree by
//     enumerating the nodes in levelorder, starting at the root with number 1.
//     In doing so, we realize that for every node X with address A the
//     following property holds: The address of X's left and right successors
//     are 2*A and 2*A+1, respectively, supposed the successors do exist.  This
//     fact can be used to elegantly construct a complete binary tree structure.
//     Write a method completeBinaryTree that takes as parameters the number of
//     nodes and the value to put in each node.
//
//     scala> Tree.completeBinaryTree(6, "x")
//     res0: Node[String] = T(x T(x T(x . .) T(x . .)) T(x T(x . .) .))

object Tree {
  def completeBinaryTree[T](nodes: Int, value: T): Tree[T] = {
    def generateTree(addr: Int): Tree[T] =
      if (addr > nodes) End
      else Node(value, generateTree(2 * addr), generateTree(2 * addr + 1))
    generateTree(1)
  }
}
// P64 (**) Layout a binary tree (1).
//     As a preparation for drawing a tree, a layout algorithm is required to
//     determine the position of each node in a rectangular grid.  Several
//     layout methods are conceivable, one of them is shown in the illustration
//     below.
//
//     In this layout strategy, the position of a node v is obtained by the
//     following two rules:
//     * x(v) is equal to the position of the node v in the inorder sequence
//     * y(v) is equal to the depth of the node v in the tree
//
//     In order to store the position of the nodes, we add a new class with the
//     additional information.
//
//     case class PositionedNode[+T](override val value: T, override val left: Tree[T], override val
right: Tree[T], x: Int, y: Int) extends Node[T](value, left, right) {
//       override def toString = "T[" + x.toString + "," + y.toString + "](" + value.toString + " "
+ left.toString + " " + right.toString + ")"
//     }
//
//     Write a method layoutBinaryTree that turns a tree of normal Nodes into a
//     tree of PositionedNodes.
//
//     scala> Node('a', Node('b', End, Node('c')), Node('d')).layoutBinaryTree
//     res0: PositionedNode[Char] = T[3,1](a T[1,2](b . T[2,3](c . .)) T[4,2](d . .))

sealed abstract class Tree[+T] {
  def layoutBinaryTree: Tree[T] = layoutBinaryTreeInternal(1, 1)._1
  def layoutBinaryTreeInternal(x: Int, depth: Int): (Tree[T], Int)
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def layoutBinaryTreeInternal(x: Int, depth: Int): (Tree[T], Int) = {
    val (leftTree, myX) = left.layoutBinaryTreeInternal(x, depth + 1)
    val (rightTree, nextX) = right.layoutBinaryTreeInternal(myX + 1, depth + 1)
    (PositionedNode(value, leftTree, rightTree, myX, depth), nextX)
  }
}

case class PositionedNode[+T](override val value: T, override val left: Tree[T], override val right:
Tree[T], x: Int, y: Int) extends Node[T](value, left, right) {
  override def toString = "T[" + x.toString + "," + y.toString + "](" + value.toString + " " +
left.toString + " " + right.toString + ")"
}

case object End extends Tree[Nothing] {
  def layoutBinaryTreeInternal(x: Int, depth: Int) = (End, x)
}
// P65 (**) Layout a binary tree (2).
//     An alternative layout method is depicted in the illustration opposite.
//     Find out the rules and write the corresponding method.  Hint: On a given
//     level, the horizontal distance between neighboring nodes is constant.
//
//     Use the same conventions as in problem P64.
//
//     scala> Node('a', Node('b', End, Node('c')), Node('d')).layoutBinaryTree2
//     res0: PositionedNode[Char] = T[3,1]('a T[1,2]('b . T[2,3]('c . .)) T[5,2]('d . .))

// The layout rules for a node v with parent u and depth d are as follows:
```

```
// * x(v) is x(u) plus or minus 2^(m-d), where m is the maximum depth of the
//   tree.  The leftmost node has x(v) == 1.
// * y(v) == d

sealed abstract class Tree[+T] {
  def treeDepth: Int
  def leftmostNodeDepth: Int
  def layoutBinaryTree2: Tree[T] = {
    val d = treeDepth
    val x0 = (2 to leftmostNodeDepth).map((n) => Math.pow(2, d - n).toInt).reduceLeft(_+_) + 1
    layoutBinaryTree2Internal(x0, 1, d - 2)
  }
  def layoutBinaryTree2Internal(x: Int, depth: Int, exp: Int): Tree[T]
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def treeDepth: Int = (left.treeDepth max right.treeDepth) + 1
  def leftmostNodeDepth: Int = left.leftmostNodeDepth + 1
  def layoutBinaryTree2Internal(x: Int, depth: Int, exp: Int): Tree[T] =
    PositionedNode(
      value,
      left.layoutBinaryTree2Internal(x - Math.pow(2, exp).toInt, depth + 1, exp - 1),
      right.layoutBinaryTree2Internal(x + Math.pow(2, exp).toInt, depth + 1, exp - 1),
      x, depth)
}

case object End extends Tree[Nothing] {
  def treeDepth: Int = 0
  def leftmostNodeDepth: Int = 0
  def layoutBinaryTree2Internal(x: Int, depth: Int, exp: Int) = End
}
// P66 (***) Layout a binary tree (3).
//     Yet another layout strategy is shown in the illustration opposite.  The
//     method yields a very compact layout while maintaining a certain symmetry
//     in every node.  Find out the rules and write the corresponding method.
//     Hint: Consider the horizontal distance between a node and its successor
//     nodes.  How tight can you pack together two subtrees to construct the
//     combined binary tree?
//
//     Use the same conventions as in problem P64 and P65.  Note: This is a
//     difficult problem.  Don't give up too early!
//
//     scala> Node('a', Node('b', End, Node('c')), Node('d')).layoutBinaryTree3
//     res0: PositionedNode[Char] = T[2,1]('a T[1,2]('b . T[2,3]('c . .)) T[3,2]('d . .))

// One way of tacking this problem is to first consider the envelope bounding
// the subtree of a given node, relative to the node's own position.  Each node
// then queries its subtrees for their envelopes and then determines how much
// to shift them to maintain at least one unit of space between adjacent nodes.
// From there, actually calculating the layout is relatively easy.

// We add a `bounds` property to the tree, which is a list of pairs of integers
// giving the left and right offset from a given node for each tree level
// beneath it.  The toplevel node uses `bounds` to determine the x position of
// the root of the tree, and then each node uses its shift calculation to tell
// its subnodes their locations.

sealed abstract class Tree[+T] {
  def bounds: List[(Int,Int)]
  def layoutBinaryTree3: Tree[T] =
    layoutBinaryTree3Internal(bounds.map(_._1).reduceLeft(_ min _) * -1 + 1, 1)
  def layoutBinaryTree3Internal(x: Int, depth: Int): Tree[T]
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def bounds: List[(Int,Int)] = {
    def lowerBounds = (left.bounds, right.bounds) match {
      case (Nil, Nil) => Nil
      case (lb, Nil)  => lb.map((b) => (b._1 - 1, b._2 - 1))
      case (Nil, rb)  => rb.map((b) => (b._1 + 1, b._2 + 1))
      case (lb, rb) => {
        val shift = lb.zip(rb).map((e) => (e._1._2 - e._2._1) / 2 + 1).reduceLeft(_ max _)
        lb.map(Some(_)).zipAll(rb.map(Some(_)), None, None).map(_ match {
          case (Some((a, b)), Some((c, d))) => (a - shift, d + shift)
          case (Some((a, b)), None)         => (a - shift, b - shift)
          case (None, Some((c, d)))         => (c + shift, d + shift)
          case (None, None) => throw new Exception  // Placate the compiler; can't get here.
        })
      }
    }
    (0, 0) :: lowerBounds
  }
  def layoutBinaryTree3Internal(x: Int, depth: Int): Tree[T] = bounds match {
    case _ :: (bl, br) :: _ => PositionedNode(
      value, left.layoutBinaryTree3Internal(x + bl, depth + 1),
      right.layoutBinaryTree3Internal(x + br, depth + 1), x, depth)
    case _ => PositionedNode(value, End, End, x, depth)
  }
}

case object End extends Tree[Nothing] {
  def bounds: List[(Int,Int)] = Nil
  def layoutBinaryTree3Internal(x: Int, depth: Int) = End
}
// P67 (**) A string representation of binary trees.
//     Somebody represents binary trees as strings of the following type (see
//     example opposite):
//
//     a(b(d,e),c(,f(g,)))
//
//     Write a method which generates this string representation, if the tree
//     is given as usual (in Nodes and Ends).  Use that method for the Tree
//     class's and subclass's toString methods.  Then write a method (on the
//     Tree object) which does this inverse; i.e. given the string
//     representation, construct the tree in the usual form.
//
//     For simplicity, suppose the information in the nodes is a single letter
//     and there are no spaces in the string.
//
//     scala> Node('a', Node('b', Node('d'), Node('e')), Node('c', End, Node('f', Node('g'),
End))).toString
//     res0: String = a(b(d,e),c(,f(g,)))
//
//     scala> Tree.fromString("a(b(d,e),c(,f(g,)))")
//     res1: Node[Char] = a(b(d,e),c(,f(g,)))

// TODO: Implement string2Tree with parser combinators.

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  override def toString = (left, right) match {
    case (End, End) => value.toString
    case _ => value.toString + "(" + left + "," + right + ")"
  }
}

case class PositionedNode[+T](override val value: T, override val left: Tree[T], override val right:
Tree[T], x: Int, y: Int) extends Node[T](value, left, right) {
  override def toString = (left, right) match {
    case (End, End) => value + "[" + x + "," + y + "]"
    case _ => value + "[" + x + "," + y + "](" + left + "," + right + ")"
  }
}

case object End extends Tree[Nothing] {
  override def toString = ""
}

object Tree {
  def string2Tree(s: String): Tree[Char] = {
    def extractTreeString(s: String, start: Int, end: Char): (String,Int) = {
      def updateNesting(nesting: Int, pos: Int): Int = s(pos) match {
        case '(' => nesting + 1
        case ')' => nesting - 1
        case _   => nesting
      }
      def findStringEnd(pos: Int, nesting: Int): Int =
        if (s(pos) == end && nesting == 0) pos
        else findStringEnd(pos + 1, updateNesting(nesting, pos))
      val strEnd = findStringEnd(start, 0)
      (s.substring(start, strEnd), strEnd)
    }
    s.length match {
      case 0 => End
      case 1 => Node(s(0))
      case _ => {
        val (leftStr, commaPos) = extractTreeString(s, 2, ',')
        val (rightStr, _) = extractTreeString(s, commaPos + 1, ')')
        Node(s(0), string2Tree(leftStr), string2Tree(rightStr))
      }
    }
  }
}
// P68 (**) Preorder and inorder sequences of binary trees.
//     We consider binary trees with nodes that are identified by single
//     lower-case letters, as in the example of problem P67.
//
//     a) Write methods preorder and inorder that construct the preorder and
//        inorder sequence of a given binary tree, respectively.  The results
//        should be lists, e.g. List('a','b','d','e','c','f','g') for the
//        preorder sequence of the example in problem P67.
//
//     scala> Tree.string2Tree("a(b(d,e),c(,f(g,)))").preorder
//     res0: List[Char] = List(a, b, d, e, c, f, g)
//
//     scala> Tree.string2Tree("a(b(d,e),c(,f(g,)))").inorder
//     res1: List[Char] = List(d, b, e, a, c, g, f)
//
//     b) If both the preorder sequence and the inorder sequence of the nodes of
//        a binary tree are given, then the tree is determined unambiguously.
//        Write a method preInTree that does the job.
//
//     scala> Tree.preInTree(List('a', 'b', 'd', 'e', 'c', 'f', 'g'), List('d', 'b', 'e', 'a', 'c',
'g', 'f'))
//     res2: Node[Char] = a(b(d,e),c(,f(g,)))
//
//     What happens if the same character appears in more than one node?  Try,
//     for instance, Tree.preInTree(List('a', 'b', 'a'), List('b', 'a', 'a')).

sealed abstract class Tree[+T] {
  def preorder: List[T]
  def inorder: List[T]
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def preorder: List[T] = value :: left.preorder ::: right.preorder
  def inorder: List[T] = left.inorder ::: value :: right.inorder
}

case object End extends Tree[Nothing] {
  def preorder = Nil
  def inorder = Nil
}

object Tree {
  def preInTree[T](pre: List[T], in: List[T]): Tree[T] = pre match {
    case Nil       => End
    case v :: preTail => {
      val (leftIn, rightIn) = in.span(_ != v)
      Node(v, preInTree(preTail.take(leftIn.length), leftIn),
           preInTree(preTail.drop(leftIn.length), rightIn))
    }
  }
}
// P69 (**) Dotstring representation of binary trees.
//     We consider again binary trees with nodes that are identified by single
//     lower-case letters, as in the example of problem P67.  Such a tree can be
//     represented by the preorder sequence of its nodes in which dots (.) are
//     inserted where an empty subtree (End) is encountered during the tree
//     traversal.  For example, the tree shown in problem P67 is represented as
//     "abd..e..c.fg...".  First, try to establish a syntax (BNF or syntax
//     diagrams) and then write two methods, toDotstring and fromDotstring,
//     which do the conversion in both directions.
//
//     scala> Tree.string2Tree("a(b(d,e),c(,f(g,)))").toDotstring
//     res0: String = abd..e..c.fg...
//
//     scala> Tree.fromDotstring("abd..e..c.fg...")
//     res1: Node[Char] = a(b(d,e),c(,f(g,)))

sealed abstract class Tree[+T] {
  def toDotstring: String
}

case class Node[+T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T] {
  def toDotstring: String = value.toString + left.toDotstring + right.toDotstring
}

case object End extends Tree[Nothing] {
  def toDotstring: String = "."
}
```

```
object Tree {
  def fromDotstring(ds: String): Tree[Char] = {
    def fromDotstringR(pos: Int): (Tree[Char], Int) = ds(pos) match {
      case '.' => (End, pos + 1)
        case c   => {
          val (lTree, lPos) = fromDotstringR(pos + 1)
          val (rTree, rPos) = fromDotstringR(lPos)
          (Node(c, lTree, rTree), rPos)
        }
    }
    fromDotstringR(0)._1
  }
}
// P70C (*) Count the nodes of a multiway tree.
//      Write a method nodeCount which counts the nodes of a given multiway
//      tree.
//
//      scala> MTree('a', List(MTree('f'))).nodeCount
//      res0: Int = 2

case class MTree[+T](value: T, children: List[MTree[T]]) {
  def nodeCount: Int = children.foldLeft(1)(_ + _.nodeCount)
}
// P70 (**) Tree construction from a node string.
//     We suppose that the nodes of a multiway tree contain single characters.
//     In the depth-first order sequence of its nodes, a special character ^ has
//     been inserted whenever, during the tree traversal, the move is a
//     backtrack to the previous level.
//
//     By this rule, the tree in the figure opposite is represented as:
//     afg^^c^bd^e^^^
//
//     Define the syntax of the string and write a function string2MTree to
//     construct an MTree from a String.  Make the function an implicit
//     conversion from String.  Write the reverse function, and make it the
//     toString method of MTree.
//
//     scala> MTree('a', List(MTree('f', List(MTree('g'))), MTree('c'), MTree('b', List(MTree('d'),
MTree('e'))))).toString
//     res0: String = afg^^c^bd^e^^^

case class MTree[+T](value: T, children: List[MTree[T]]) {
  override def toString = value.toString + children.map(_.toString + "^").mkString("")
}

object MTree {
  implicit def string2MTree(s: String): MTree[Char] = {
    def nextStrBound(pos: Int, nesting: Int): Int =
      if (nesting == 0) pos
      else nextStrBound(pos + 1, if (s(pos) == '^') nesting - 1 else nesting + 1)
    def splitChildStrings(pos: Int): List[String] =
      if (pos >= s.length) Nil
      else {
        val end = nextStrBound(pos + 1, 1)
        s.substring(pos, end - 1) :: splitChildStrings(end)
      }
    MTree(s(0), splitChildStrings(1).map(string2MTree(_)))
  }
}
// P71 (*) Determine the internal path length of a tree.
//     We define the internal path length of a multiway tree as the total sum of
//     the path lengths from the root to all nodes of the tree.  By this
//     definition, the tree in the figure of problem P70 has an internal path
//     length of 9.  Write a method internalPathLength to return that sum.
//
//     scala> "afg^^c^bd^e^^^".internalPathLength
//     res0: Int = 9

case class MTree[+T](value: T, children: List[MTree[T]]) {
  def internalPathLength: Int =
    children.foldLeft(0)((r, c) => r + c.nodeCount + c.internalPathLength)
}
// P72 (*) Construct the postorder sequence of the tree nodes.
//     Write a method postorder which constructs the postorder sequence of the
//     nodes of a multiway tree.  The result should be a List.
//
//     scala> "afg^^c^bd^e^^^".postorder
//     res0: List[Char] = List(g, f, c, d, e, b, a)
```

```
case class MTree[+T](value: T, children: List[MTree[T]]) {
  def postorder: List[T] = children.flatMap(_.postorder) ::: List(value)
}
// P73 (**) Lisp-like tree representation.
//     There is a particular notation for multiway trees in Lisp.  Lisp is a
//     prominent functional programming language.  In Lisp almost everything is
//     a list.
//
//     Our example tree would be represented in Lisp as (a (f g) c (b d e)).
//
//     Note that in the "lispy" notation a node with successors (children)in the
//     tree is always the first element in a list, followed by its children.
//     The "lispy" representation of a multiway tree is a sequence of atoms and
//     parentheses '(' and ')', with the atoms separated by spaces.  We can
//     represent this syntax as a Scala String.  Write a method lispyTree which
//     constructs a "lispy string" from an MTree.
//
//     scala> MTree("a", List(MTree("b", List(MTree("c"))))).lispyTree
//     res0: String = (a (b c))
//
//     As a second, even more interesting, exercise try to write a method that
//     takes a "lispy" list and turns it into a multiway tree.
//
//     [Note: This is certainly one way of looking at Lisp notation, but it's
//      not how the language actually represents that syntax internally.  I can
//      elaborate more on this, if requested.  <PMG>]

case class MTree[+T](value: T, children: List[MTree[T]]) {
  def lispyTree: String =
    if (children == Nil) value.toString
    else "(" + value.toString + " " + children.map(_.lispyTree).mkString(" ") + ")"
}

object MTree {
  def fromLispyString(s: String): MTree[String] = {
    def setNesting(nesting: Int, c: Char): Int = c match {
      case '(' => nesting + 1
      case ')' => nesting - 1
      case _   => nesting
    }
    def nextSpace(pos: Int, nesting: Int): Int =
      if ((s(pos) == ' ' || s(pos) == ')') && nesting == 0) pos
      else nextSpace(pos + 1, setNesting(nesting, s(pos)))
    def nextNonSpace(pos: Int): Int =
      if (s(pos) == ' ') nextNonSpace(pos + 1)
      else pos
    def listSubstrings(pos: Int): List[String] =
      if (pos > s.length || s(pos) == ')') Nil
      else {
        val end = nextSpace(pos, 0)
        s.substring(pos, end) :: (if (s(end) == ')') Nil else listSubstrings(nextNonSpace(end)))
      }
    if (s(0) != '(') MTree(s)
    else {
      val vEnd = nextSpace(1, 0)
      MTree(s.substring(1, vEnd), listSubstrings(nextNonSpace(vEnd)).map(fromLispyString(_)))
    }
  }
}
// P80 (***) Conversions.
//     Write methods to generate the graph-term and adjacency-list forms from a
//     Graph.  Write another method to output the human-friendly form for a
//     graph.  Make it the toString method for Graph.  Write one more function
//     to create a graph of Chars and Ints from a human-friendly string.  Make
//     it implicitly convert from Strings to Graphs.
//
//     scala> Graph.fromString("[b-c, f-c, g-h, d, f-b, k-f, h-g]").toTermForm
//     res0: (List[String], List[(String, String, Unit)]) = (List(d, k, h, c, f, g, b),List((h,g,
()), (k,f,()), (f,b,()), (g,h,()), (f,c,()), (b,c,())))
//
//     scala> Digraph.fromStringLabel("[p>q/9, m>q/7, k, p>m/5]").toAdjacentForm
//     res1: List[(String, List[(String, Int)])] = List((m,List((q,7))), (p,List((m,5), (q,9))),
(k,List()), (q,List()))

abstract class GraphBase[T, U] {
  case class Edge(n1: Node, n2: Node, value: U) {
    override def toString = value match {
      case () => n1.value + edgeSep + n2.value
      case v  => n1.value + edgeSep + n2.value + labelSep + v
    }
  }

  val edgeSep: String
  val labelSep: String = "/"

  override def toString = {
    val (edgeStrs, unlinkedNodes) =
      edges.foldLeft((Nil: List[String], nodes.values.toList))((r, e) => (e.toString :: r._1,
r._2.filter((n) => n != e.n1 && n != e.n2)))
    "[" + (unlinkedNodes.map(_.value.toString) ::: edgeStrs).mkString(", ") + "]"
  }
  def toTermForm: (List[T], List[(T, T, U)]) =
    (nodes.keys.toList, edges.map((e) => (e.n1.value, e.n2.value, e.value)))
  def toAdjacentForm: List[(T, List[(T, U)])] =
    nodes.values.toList.map((n) => (n.value, n.adj.map((e) =>
      (edgeTarget(e, n).get.value, e.value))))
}

class Graph[T, U] extends GraphBase[T, U] {
  val edgeSep: String = "-"
}

class Digraph[T, U] extends GraphBase[T, U] {
  val edgeSep: String = ">"
}

abstract class GraphObjBase {
    val edgeSep: String
    val labelSep: String = "/"

    def fromStringBase[U, V](s: String)(mkGraph: (List[String], List[V]) => GraphClass[String, U],
mkDigraph: (List[String], List[V]) => GraphClass[String, U])(parseEdge: String => V):
GraphClass[String, U] = {
      assert(s(0) == '[')
      assert(s(s.length - 1) == ']')
      val tokens = s.substring(1, s.length - 1).split(", *").toList
      val nodes = tokens.flatMap(_.replaceAll("/.*", "").split("[->]")).removeDuplicates
      val edges = tokens.filter(_.matches(".*[->].*")).map(parseEdge)
      tokens.find(_.matches(".*>.*")) match {
        case None    => mkGraph(nodes, edges)
        case Some(_) => mkDigraph(nodes, edges)
      }
    }
    def fromString(s: String): GraphClass[String, Unit] =
      fromStringBase(s)(Graph.term[String], Digraph.term[String]) { t =>
        val split = t.split("[->]")
        (split(0), split(1))
      }
    def fromStringLabel(s: String): GraphClass[String, Int] =
      fromStringBase(s)(Graph.termLabel[String, Int], Graph.termLabel[String, Int]) { t =>
        val split = t.split("[->]")
        val split2 = split(1).split("/")
        (split(0), split2(0), split2(1).toInt)
      }
}

object Graph extends GraphObjBase {
  val edgeSep: String = "-"
}

object Digraph extends GraphObjBase {
  val edgeSep: String = ">"
}
// P81 (**) Path from one node to another one.
//     Write a method named findPaths to find an acyclic path from one node to
//     another in a graph.  The method should return all paths.
//
//     scala> Digraph.fromStringLabel("[p>q/9, m>q/7, k, p>m/5]").findPaths("p", "q")
//     res0: List[List[String]] = List(List(p, q), List(p, m, q))
//
//     scala> Digraph.fromStringLabel("[p>q/9, m>q/7, k, p>m/5]").findPaths("p", "k")
//     res1: List[List[String]] = List()

abstract class GraphBase[T, U] {
  def findPaths(source: T, dest: T): List[List[T]] = {
    def findPathsR(curNode: Node, curPath: List[T]): List[List[T]] = {
      if (curNode.value == dest) List(curPath)
      else curNode.adj.map(edgeTarget(_, curNode).get).filter(n => !
curPath.contains(n.value)).flatMap(n => findPathsR(n, n.value :: curPath))
    }
    findPathsR(nodes(source), List(source)).map(_.reverse)
  }
}
// P82 (*) Cycle from a given node.
//     Write a method named findCycles to find closed paths (cycles) starting at
//     a given node in a graph.  The method should return all cycles.
//
//     scala> Graph.fromString("[b-c, f-c, g-h, d, f-b, k-f, h-g]").findCycles("f")
//     res0: List[List[String]] = List(List(f, c, b, f), List(f, b, c, f))

abstract class GraphBase[T, U] {
  def findCycles(source: T): List[List[T]] = {
    val n = nodes(source)
    n.adj.map(edgeTarget(_, n).get.value).flatMap(findPaths(_, source)).map(source ::
_).filter(_.lengthCompare(3) > 0)
  }
}
// P83 (**) Construct all spanning trees.
//     Write a method spanningTrees to construct all spanning trees of a given
//     graph.  With this method, find out how many spanning trees there are for
//     the graph depicted to the right.  The data of this example graph can be
//     found below.  When you have a correct solution for the spanningTrees
//     method, use it to define two other useful methods: isTree and
//     isConnected.  Both are five-minute tasks!
//
//     Graph:
//     Graph.term(List('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'),
//                List(('a', 'b'), ('a', 'd'), ('b', 'c'), ('b', 'e'),
//                     ('c', 'e'), ('d', 'e'), ('d', 'f'), ('d', 'g'),
//                     ('e', 'h'), ('f', 'g'), ('g', 'h')))
//
//     scala> Graph.fromString("[a-b, b-c, a-c]").spanningTrees
//     res0: List[Graph[String,Unit]] = List([a-b, b-c], [a-c, b-c], [a-b, a-c])

// Only undirected graphs have spanning trees.

class Graph[T, U] extends GraphBase[T, U] {
  // edgeConnectsToGraph is needed for P84, so it's not an internal function.
  def edgeConnectsToGraph[T,U](e: Edge, nodes: List[Node]): Boolean =
    !(nodes.contains(e.n1) == nodes.contains(e.n2))  // xor
  def spanningTrees = {
    def spanningTreesR(graphEdges: List[Edge], graphNodes: List[Node], treeEdges: List[Edge]):
List[Graph[T,U]] = {
      if (graphNodes == Nil) List(Graph.termLabel(nodes.keys.toList, treeEdges.map(_.toTuple)))
      else if (graphEdges == Nil) Nil
      else graphEdges.filter(edgeConnectsToGraph(_, graphNodes)) flatMap { ge =>
        spanningTreesR(graphEdges.remove(_ == ge),
                       graphNodes.filter(edgeTarget(ge, _) == None),
                       ge :: treeEdges)
                                                                  }
    }
    spanningTreesR(edges, nodes.values.toList.tail, Nil).removeDuplicates
  }
  def isTree: Boolean = spanningTrees.lengthCompare(1) == 0
  def isConnected: Boolean = spanningTrees.lengthCompare(0) > 0
}
// P84 (**) Construct the minimal spanning tree.
//     Write a method minimalSpanningTree to construct the minimal spanning tree
//     of a given labeled graph.  Hint: Use Prim's Algorithm.  A small
//     modification of the solution of P83 does the trick.  The data of the
//     example graph to the right can be found below.
//
//     Graph:
//     Graph.termLabel(
//       List('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'),
//            List(('a', 'b', 5), ('a', 'd', 3), ('b', 'c', 2), ('b', 'e', 4),
//                 ('c', 'e', 6), ('d', 'e', 7), ('d', 'f', 4), ('d', 'g', 3),
//                 ('e', 'h', 5), ('f', 'g', 4), ('g', 'h', 1)))
//
//     scala> Graph.fromStringLabel("[a-b/1, b-c/2, a-c/3]").minimalSpanningTree
//     res0: Graph[String,Int] = [a-b/1, b-c/2]

class Graph[T, U] extends GraphBase[T, U] {
  def minimalSpanningTree(implicit f: (U) => Ordered[U]): Graph[T,U] = {
    def minimalSpanningTreeR(graphEdges: List[Edge], graphNodes: List[Node], treeEdges: List[Edge]):
Graph[T,U] =
      if (graphNodes == Nil) Graph.termLabel(nodes.keys.toList, treeEdges.map(_.toTuple))
      else {
        val nextEdge = graphEdges.filter(edgeConnectsToGraph(_, graphNodes)).reduceLeft((r, e) => if
(r.value < e.value) r else e)
```

```
        minimalSpanningTreeR(graphEdges - nextEdge,
                             graphNodes.filter(edgeTarget(nextEdge, _) == None),
                             nextEdge :: treeEdges)
      }
    minimalSpanningTreeR(edges, nodes.values.toList.tail, Nil)
  }
}
// P85 (**) Graph isomorphism.
//     Two graphs G1(N1,E1) and G2(N2,E2) are isomorphic if there is a bijection
//     f: N1 → N2 such that for any nodes X,Y of N1, X and Y are adjacent if and
//     only if f(X) and f(Y) are adjacent.
//
//     Write a method that determines whether two graphs are isomorphic.
//
//     scala> Graph.fromString("[a-b]").isIsomorphicTo(Graph.fromString("[5-7]"))
//     res0: Boolean = true

// This problem is in NP (it's one of the few for which it is unknown whether
// it's NP-complete or not), so it gets slow very quickly.  Essentially, we
// consider every mapping from the nodes of one graph into the other, which
// is O(n!).  There are some heuristics to prune the search tree
// (isValidMapping), but that can only go so far.
//
// For a more efficient algorithm (O(n^2 log n)), see "Practical Graph
// Isomorphism" by Brendan D. McKay of Vanderbilt University.
// http://cs.anu.edu.au/~bdm/nauty/PGI/

abstract class GraphBase[T, U] {
  def isIsomorphicTo[R,S](o: GraphBase[R,S]): Boolean = {
    // Build a lazy list so we only have to evaluate as much as necessary.
    def listMappings(tNodes: List[Node], oNodes: List[o.Node]) =
      tNodes.projection.flatMap(tn => oNodes.projection.map((tn, _)))
    // Used on partially-filled isomorphisms to weed out some early.
    def isValidMapping(iso: Map[Node,o.Node]): Boolean =
      nodes.values forall {tn =>
        (!iso.contains(tn) ||
         tn.neighbors.filter(iso.contains).forall(tnn => iso(tn).neighbors.contains(iso(tnn))))
      }
    def isValidCompleteMapping(iso: Map[Node,o.Node]): Boolean =
      nodes.values forall {tn =>
        Set(tn.neighbors.map(iso.apply): _*) == Set(iso(tn).neighbors: _*)
      }
    def isIsomorphicToR(tNodes: List[Node], oNodes: List[o.Node], iso: Map[Node,o.Node]): Boolean =
      if (tNodes == Nil) isValidCompleteMapping(iso)
      else listMappings(tNodes, oNodes).filter(p => isValidMapping(iso + p)) exists {p =>
        isIsomorphicToR(tNodes - p._1, oNodes - p._2, iso + p)
      }
    isIsomorphicToR(nodes.values.toList, o.nodes.values.toList, Map())
  }
}
// P86 (**) Node degree and graph coloration.
//     a) Write a method Node.degree that determines the degree of a given node.
//
//     scala> Graph.fromString("[a-b, b-c, a-c, a-d]").nodes("a").degree
//     res0: Int = 3
//
//     b) Write a method that lists all nodes of a graph sorted according to
//        decreasing degree.
//
//     scala> Graph.fromString("[a-b, b-c, a-c, a-d]").nodesByDegree
//     res1: List[Graph[String,Unit]#Node] = List(Node(a), Node(c), Node(b), Node(d))
//
//     c) Use Welsh-Powell's algorithm to paint the nodes of a graph in such a
//        way that adjacent nodes have different colors.  Make a method
//        colorNodes that returns a list of tuples, each of which contains a
//        node and an integer representing its color.
//
//     scala> Graph.fromString("[a-b, b-c, a-c, a-d]").colorNodes
//     res2: List[(Graph[String,Unit]#Node,Int)] = List((Node(a),1), (Node(b),2), (Node(c), 3),
(Node(d), 2))

abstract class GraphBase[T, U] {
  case class Node(value: T) {
    def degree: Int = edges.length
  }

  def nodesByDegree: List[Node] = nodes.values.toList.sort(_.degree > _.degree)
  def colorNodes: List[(Node,Int)] = {
    import collection.immutable.Set
    def applyColor(color: Int, uncolored: List[Node], colored: List[(Node,Int)], adjacentNodes:
```

```
Set[Node]): List[(Node,Int)] =
      uncolored match {
        case Nil => colored
        case n :: tail => {
          val newAdjacent = adjacentNodes ++ n.neighbors
          applyColor(color, tail.dropWhile(newAdjacent.apply), (n, color) :: colored, newAdjacent)
        }
      }
    def colorNodesR(color: Int, uncolored: List[Node], colored: List[(Node,Int)]): List[(Node,Int)]
=
      if (uncolored == Nil) colored
      else {
        val newColored = applyColor(color, uncolored, colored, Set())
        colorNodesR(color + 1, uncolored -- newColored.map(_._1), newColored)
      }
    colorNodesR(1, nodesByDegree, Nil)
  }
}
// P87 (**) Depth-first order graph traversal.
//     Write a method that generates a depth-first order graph traversal
//     sequence.  The starting point should be specified, and the output should
//     be a list of nodes that are reachable from this starting point (in
//     depth-first order).
//
//     scala> Graph.fromString("[a-b, b-c, e, a-c, a-d]").nodesByDepthFrom("d")
//     res0: List[String] = List(c, b, a, d)

// Node.nodesByDepth is a little inefficient.  With immutable Lists, it ends up
// rebuilding the entire list every time it adds a node.  It would be more
// efficient to build the list backwards (adding new elements to the beginning)
// and then reverse it in nodesByDepthFrom.
//
// Similarly, nodesByDepthR isn't tail recursive.  If a node has more neighbors
// than there are stack frames, this will be a problem.

abstract class GraphBase[T, U] {
  case class Node(value: T) {
    def nodesByDepth(seen: Set[Node]): List[Node] = {
      def nodesByDepthR(neighbors: List[Node], s: Set[Node]): List[Node] =
        neighbors match {
          case Nil => Nil
          case n :: tail if s(n) => nodesByDepthR(tail, s)
          case n :: tail => {
            val subnodes = n.nodesByDepth(s)
            subnodes ::: nodesByDepthR(tail, s ++ subnodes)
          }
        }
      nodesByDepthR(neighbors, seen + this) ::: List(this)
    }
  }

  def nodesByDepthFrom(start: T): List[T] =
    nodes(start).nodesByDepth(Set()).map(_.value)
}
// P88 (**) Connected components.
//     Write a function that splits a graph into its connected components.
//
//     scala> Graph.fromString("[a-b, c]").splitGraph
//     res0: List[Graph[String,Unit]] = List([a-b], [c])

abstract class GraphBase[T, U] {
  case class Node(value: T) {
    // partners are all nodes either adjacent to this node or to which this
    // node is adjacent.
    def partners: List[Node] =
      edges.map(edgePartner(_, this)).remove(_.isEmpty).map(_.get).removeDuplicates
  }

  // If node N is a member of edge E, returns the other member of the edge.
  // This differs from edgeTarget in that if it is given an edge in a directed
  // graph and N is the target of that edge, it will still return the source
  // node for the edge.
  def edgePartner(e: Edge, n: Node): Option[Node] =
    if (e.n1 == n) Some(e.n2)
    else if (e.n2 == n) Some(e.n1)
    else None

  def splitGraph: List[GraphBase[T,U]] = {
    def nodes2Graph(nodes: List[Node]): GraphBase[T,U] = {
      val adjacentForm = nodes.map(n => (n.value, n.adj.map(e =>
        (edgeTarget(e, n).get.value, e.value))))
      this match {
        case _: Graph[_,_] => Graph.adjacentLabel(adjacentForm)
        case _: Digraph[_,_] => Digraph.adjacentLabel(adjacentForm)
      }
    }
    def findConnectedNodes(candidates: List[Node], soFar: List[Node]): List[Node] =
      candidates match {
        case Nil => soFar
        case n :: tail => {
          val newNodes = n.partners -- soFar - n
          findConnectedNodes(tail.union(newNodes), n :: soFar)
        }
      }
    def splitGraphR(unsplit: List[Node]): List[GraphBase[T,U]] = unsplit match {
      case Nil => Nil
      case n :: tail => {
        val connectedNodes = findConnectedNodes(List(n), Nil)
        nodes2Graph(connectedNodes) :: splitGraphR(unsplit -- connectedNodes)
      }
    }
    splitGraphR(nodes.values.toList)
  }
}

// Just to avoid repetition, we can now define Graph.edgeTarget in terms of
// edgePartner.
class Graph[T, U] extends GraphBase[T, U] {
  def edgeTarget(e: Edge, n: Node): Option[Node] = edgePartner(e, n)
}
// P89 (**) Bipartite graphs.
//     Write a function that determines whether a given graph is bipartite.
//
//     scala> Digraph.fromString("[a>b, c>a, d>b]").isBipartite
//     res0: Boolean = true
//
//     scala> Graph.fromString("[a-b, b-c, c-a]").isBipartite
//     res1: Boolean = false
//
//     scala> Graph.fromString("[a-b, b-c, d]").isBipartite
//     res2: Boolean = true
//
//     scala> Graph.fromString("[a-b, b-c, d, e-f, f-g, g-e, h]").isBipartite
//     res3: Boolean = false

abstract class GraphBase[T, U] {
  def isBipartiteInternal = {
    def isBipartiteR(evenToCheck: List[Node], oddToCheck: List[Node], evenSeen: Set[Node], oddSeen:
Set[Node]): Boolean =
      (evenToCheck, oddToCheck) match {
        case (Nil, Nil) => true
        case (e :: eTail, odd) =>
          e.partners.forall(!evenSeen(_)) && isBipartiteR(eTail,
odd.union(e.partners.remove(oddSeen(_))), evenSeen + e, oddSeen ++ e.partners)
        case (Nil, o :: oTail) =>
          o.partners.forall(!oddSeen(_)) && isBipartiteR(o.partners.remove(oddSeen(_)), oTail,
evenSeen ++ o.partners, oddSeen + o)
      }
    isBipartiteR(List(nodes.values.next), Nil, Set(), Set())
  }
  def isBipartite: Boolean = {
    nodes.isEmpty || splitGraph.forall(_.isBipartiteInternal)
  }
}

// P90 (**) Eight queens problem
//     This is a classical problem in computer science.  The objective is to
//     place eight queens on a chessboard so that no two queens are attacking
//     each other; i.e., no two queens are in the same row, the same column, or
//     on the same diagonal.
//
//     Hint: Represent the positions of the queens as a list of numbers 1..N.
//     Example: List(4, 2, 7, 3, 6, 8, 5, 1) means that the queen in the first
//     column is in row 4, the queen in the second column is in row 2, etc.  Use
//     the generate-and-test paradigm.

object P90 {
  def eightQueens = {
    def validDiagonals(qs: List[Int], upper: Int, lower: Int): Boolean = qs match {
      case Nil => true
      case q :: tail => q != upper && q != lower && validDiagonals(tail, upper + 1, lower - 1)
    }
    def valid(qs: List[Int]): Boolean = qs match {
      case Nil => true
      case q :: tail => validDiagonals(tail, q + 1, q - 1)
    }
    def eightQueensR(curQueens: List[Int], remainingCols: Set[Int]): List[List[Int]] =
      if (!valid(curQueens)) Nil
      else if (remainingCols.isEmpty) List(curQueens)
      else remainingCols.toList.flatMap(c => eightQueensR(c :: curQueens, remainingCols - c))
    eightQueensR(Nil, Set() ++ (1 to 8))
  }
}
// P91 (**) Knight's tour.
//     Another famous problem is this one: How can a knight jump on an N×N
//     chessboard in such a way that it visits every square exactly once?
//
//     Hints: Represent the squares by pairs of their coordinates of the form
//     (X, Y), where both X and Y are integers between 1 and N. (Alternately,
//     define a Point class for the same purpose.)  Write a function
//     jump(N, (X, Y), (U, V)) to express the fact that a knight can jump from
//     (X, Y) to (U, V) on a N×N chessboard.  And finally, represent the
//     solution of our problem as a list of knight positions (the knight's
//     tour).

case class Point(x: Int, y: Int) {
  def +(o: Point) = Point(x + o.x, y + o.y)
  def -(o: Point) = Point(x - o.x, y - o.y)
  def +(t: (Int,Int)) = Point(x + t._1, y + t._2)
  def -(t: (Int,Int)) = Point(x - t._1, y - t._2)
  def jumps(n: Int): List[Point] =
    List((2, 1), (1, 2), (-1, 2), (-2, 1)) flatMap {x =>
      List(this + x, this - x)
    } filter {p =>
      (p.x min p.y) >= 1 && (p.x max p.y) <= n
    }
}

object P91 {
  // Utility function.
  // Filters out the positions already seen in the given set, and orders the
  // points with the one with the fewest possibilities first, in line with
  // Warnsdorff's heuristic.
  def jumps(p: Point, n: Int, s: Set[Point]): List[Point] = {
    def filtered(p: Point) = p.jumps(n).remove(s(_))
    filtered(p).sort(filtered(_).length < filtered(_).length)
  }

  // Find one tour.
  // Because we're using the stack for our state here, one of the simplest ways
  // to signal a result is to throw an exception and not worry about properly
  // unwinding the stack.
  class TourFound(val tour: List[Point]) extends Exception
  def knightsTour(n: Int): List[Point] = knightsTour(n, Point(1, 1), (p,s) => s.size == n*n)
  def knightsTourClosed(n: Int): List[Point] =
    knightsTour(n, Point(1, 1), (p,s) => s.size == n*n && p.jumps(n).contains(Point(1, 1)))
  def knightsTour(n: Int, start: Point, done: (Point,Set[Point]) => Boolean): List[Point] =
    try {
      findPath(n, start, Set(start), List(start), done)
      Nil
    } catch {
      case t: TourFound => t.tour
    }
  def findPath(n: Int, p: Point, s: Set[Point], soFar: List[Point], done: (Point,Set[Point]) =>
Boolean): Unit =
    if (done(p, s)) throw new TourFound(soFar)
    else jumps(p, n, s).foreach(q => findPath(n, q, s + q, q :: soFar, done))

  // Find all tours.
  // This is actually easier than finding just one.  The drawback, of course,
  // is that there are so many that this will take a long time and use a large
  // amount of memory building the list.
  def knightsTourComplete(n: Int): List[List[Point]] = knightsTourComplete(n, Point(1, 1))
  def knightsTourCompleteClosed(n: Int): List[List[Point]] =
    knightsTourComplete(n, Point(1, 1)).filter(_.head.jumps(n).contains(Point(1, 1)))
  def knightsTourComplete(n: Int, start: Point): List[List[Point]] =
    findAllPaths(n, start, Set(start), List(start))
  def findAllPaths(n: Int, p: Point, s: Set[Point], soFar: List[Point]): List[List[Point]] =
    if (s.size == n*n) List(soFar)
    else jumps(p, n, s).flatMap(q => findAllPaths(n, q, s + q, q :: soFar))
  // Find all tours, lazily.
  // Instead of implicitly embedding our state into the stack, here we
  // explicitly keep a list containing what were stack frames in the other
  // versions.  Also, rather than mapping across a list in each function call,
  // we just make a frame for each potential destination and drop each one onto
  // our imitation stack.
  case class Frame(n: Int, p: Point, s: Set[Point], soFar: List[Point])
  def knightsTourLazy(n: Int): Stream[List[Point]] = knightsTourLazy(n, Point(1, 1))
  def knightsTourLazyClosed(n: Int): Stream[List[Point]] =
    knightsTourLazy(n, Point(1, 1)).filter(_.head.jumps(n).contains(Point(1, 1)))
  def knightsTourLazy(n: Int, start: Point): Stream[List[Point]] =
    nextTour(List(Frame(n, start, Set(start), List(start))))
  def nextTour(stack: List[Frame]): Stream[List[Point]] = stack match {
    case Nil => Stream.empty
    case Frame(n, p, s, soFar) :: tail =>
      if (s.size == n*n) Stream.cons(soFar, nextTour(tail))
      else nextTour(jumps(p, n, s).map(q => Frame(n, q, s + q, q :: soFar)) ::: tail)
  }
}
// P92 (***) Von Koch's conjecture.
//     Several years ago I met a mathematician who was intrigued by a problem
//     for which he didn't know a solution.  His name was Von Koch, and I don't
//     know whether the problem has been solved since.  [The "I" here refers to
//     the author of the Prolog problems.  <PMG>]
//
//     d e-f      1 5-4  * *1*
//     | |        | |    6 2
//     a-b-c  ->  7-3-6  *4*3*
//     |          |      5
//     g          2      *
//
//     Anyway the puzzle goes like this: Given a tree with N nodes (and hence
//     N-1 edges), find a way to enumerate the nodes from 1 to N and,
//     accordingly, the edges from 1 to N-1 in such a way, that for each edge K
//     the difference of its node numbers is equal to K.  The conjecture is that
//     this is always possible.
//
//     For small trees the problem is easy to solve by hand.  However, for
//     larger trees, and 14 is already very large, it is extremely difficult to
//     find a solution.  And remember, we don't know for sure whether there is
//     always a solution!
//
//     Write a function that calculates a numbering scheme for a given tree.
//     What is the solution for the larger tree pictured below?
//
//     i g d-k   p
//      \| |     |
//       a-c-e-q-b
//      /| |   |
//     h b f   m
// P97 (**) Sudoku.
//     Sudoku puzzles go like this:
//
//     Problem statement                Solution
//
//     .  .  4 | 8  .  . | .  1  7      9  3  4 | 8  2  5 | 6  1  7
//             |         |                      |         |
//     6  7  . | 9  .  . | .  .  .      6  7  2 | 9  1  4 | 8  5  3
//             |         |                      |         |
//     5  .  8 | .  3  . | .  .  4      5  1  8 | 6  3  7 | 9  2  4
//     --------+---------+--------      --------+---------+--------
//     3  .  . | 7  4  . | 1  .  .      3  2  5 | 7  4  8 | 1  6  9
//             |         |                      |         |
//     .  6  9 | .  .  . | 7  8  .      4  6  9 | 1  5  3 | 7  8  2
//             |         |                      |         |
//     .  .  1 | .  6  9 | .  .  5      7  8  1 | 2  6  9 | 4  3  5
//     --------+---------+--------      --------+---------+--------
//     1  .  . | .  8  . | 3  .  6      1  9  7 | 5  8  2 | 3  4  6
//             |         |                      |         |
//     .  .  . | .  .  6 | .  9  1      8  5  3 | 4  7  6 | 2  9  1
//             |         |                      |         |
//     2  4  . | .  .  1 | 5  .  .      2  4  6 | 3  9  1 | 5  7  8
//
//
//     Every spot in the puzzle belongs to a (horizontal) row and a (vertical)
//     column, as well as to one single 3×3 square (which we call "square" for
//     short).  At the beginning, some of the spots carry a single-digit number
//     between 1 and 9.  The problem is to fill the missing spots with digits
//     in such a way that every number between 1 and 9 appears exactly once in
//     each row, in each column, and in each square.

// This is a very simple, functional-style implementation.  It uses mutable
// Arrays because they're the only O(1) access structure that Scala currently
// has.  (An O(1) access, O(1) update, immutable data structure might be added
// to Scala 2.8.)  The updates are all done functionally, which results in a
// lot of copying of data.  Hopefully, this will become more efficient in
// Scala 2.8.
//
// Anyway, the approach is pretty simple.  A sudoku board is represented as an
// array of Eithers.  Left elements represent solved cells, while Right
// elements contain the set of possibilities for their cell.  Solving the
// board consists of trying possibilities for each unsolved cell until either
// a solution is found or a contradiction (in the form of an empty Right set)
// is found.
//
// The code is pretty flexible.  The sudoku board can hold any type of data,
// although string2Board generates the traditional values 1..9.  Likewise,
// the board can be any size, but it will only work properly if the array
// passed in has a length that is a fourth power.

class SudokuBoard[T](aboard: Array[Either[T,Set[T]]]) {
  type Board = Array[Either[T,Set[T]]]
  val width = Math.sqrt(aboard.length)
  val squareWidth = Math.sqrt(width)
  val board: Board = cleanBoard(aboard)

  override def toString = {
    def mkSep =
      List.make(squareWidth, List.make(squareWidth * 3,
'-').mkString("")).mkString("+").substring(1, width * 3 + squareWidth - 2)
    def chunk[U](n: Int, seq: Seq[U]): Seq[Seq[U]] =
      (0 until seq.length by n).toList.zip((n to seq.length by n).toList).map {
        case (start, end) => seq.slice(start, end)
      }
    chunk(width * squareWidth, board.map {
      case Left(n) => n.toString
      case _       => "."
    }).map {
      chunk(width, _).map {
        chunk(squareWidth, _).map {
          _.mkString("  ")
        }.mkString(" | ")
      }.mkString("\n")
    }.mkString("\n" + mkSep + "\n")
  }

  private def removeFromRow(row: Int, n: T, b: Board): Board =
    removeFromIndices((row * width) to ((row + 1) * width - 1), n, b)
  private def removeFromCol(col: Int, n: T, b: Board): Board =
    removeFromIndices(col until width * width by width, n, b)
  private def removeFromSquare(square: Int, n: T, b: Board): Board = {
    val start = (square / squareWidth) * (width * squareWidth) + square % squareWidth * squareWidth
    val indices = Stream.from(start,
width).take(squareWidth).flatMap(Stream.from(_).take(squareWidth))
    removeFromIndices(indices, n, b)
  }
  private def removeFromIndices(is: Seq[Int], n: T, brd: Board): Board =
    is.foldLeft(brd) {(b, i) =>
      if (b(i).isRight) b(i) = Right(b(i).right.get - n)
      b
    }
  private def cleanBoard(brd: Board): Board =
    brd.zipWithIndex.foldLeft(brd) {
      case (b, (Left(n), i)) =>
        removeFromSquare(i / width / squareWidth * squareWidth + i % width / squareWidth, n,
                         removeFromRow(i / width, n,
                                       removeFromCol(i % width, n, b)))
      case (b, _) => b
    }
  def setCell(row: Int, col: Int, value: T): SudokuBoard[T] = {
    val newBoard = board.map(c => c)
    newBoard(row * width + col) = Left(value)
    new SudokuBoard(newBoard)
  }
}

object SudokuBoard {
  implicit def string2Board(s: String): SudokuBoard[Int] = {
    val boardArr: Array[Either[Int,Set[Int]]] = new Array(s.length)
    for (i <- 0 until s.length; c = s(i); n = c - '0') {
      boardArr(i) = if (1 <= n && n <= 9) Left(n)
                    else Right(Set() ++ (1 to 9))
    }
    new SudokuBoard(boardArr)
  }
  final def solve[T](b: SudokuBoard[T]): Option[SudokuBoard[T]] =
    b.board.zipWithIndex.find(_._1.isRight) match {
      case None => Some(b)
      case Some((Right(ns), _)) if ns.isEmpty => None
      case Some((Right(ns), i)) => ns.projection.map {n =>
        solve(b.setCell(i / b.width, i % b.width, n))
      }.find(_.isDefined) match {
        case None => None
        case Some(ans) => ans
      }
      case _ => throw new Exception  // Placate the compiler.
    }
}
```