|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | P0 Longest substring without repeating | <chare> | 1 | <pext> | <starting>P0 Longest substring without repeating</category> object Solution1 {  fun returnMax(str: String): Int {  val charList = str.toCharArray().toList()  for(i in charList.size downTo 1) {  val windowedList = charList.windowed(i)  windowedList.forEach { each ->  if(each.groupBy{it}.size == each.size)  return each.size  }  }  return 1  } } fun main() {  println(Solution1.returnMax("dddd")) //1  println(Solution1.returnMax("abcdabcab")) //4  println(Solution1.returnMax("abcabcdeab")) //5  println(Solution1.returnMax("samplestring")) //11 } </ending> | </end> |
| <hitle> | P1 1-100 print two consecutive numbers | <chare> | 1 | <pext> | <starting>P1 1-100 print two consecutive numbers</heading> \*\*Mobile problem\*\* - To be done LIVE online  write a logic to print a simple series from 1-100 condition is the number should be alternate print two consecutive numbers and skip next two numbers  \*\*Print series 2\*\*  \*\*1,2,3,4,5,6……100\*\*  \*\*[Alternate numbers in group of two]\*\*  \*\*12, 56, 910 …\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun main(){  val numbers = (1..100).toList()  val middleResult = numbers.chunked(2)  middleResult.map{ it->  if(middleResult.indexOf(it) % 2 == 0){  print(it.joinToString("") + ", ")   }   } } </ending> | </end> |
| <hitle> | P2 vending machine getChange() | <chare> | 1 | <pext> | <starting>P2 vending machine getChange()</heading> A vending machine has the following denominations: 1c, 5c, 10c, 25c, 50c, and $1. Your task is to write a program that will be used in a vending machine to return change. Assume that the vending machine will always want to return the least number of coins or notes. Devise a function getChange(M, P) where M is how much money was inserted into the machine and P the price of the item selected, that returns an array of intergers representing the number of each denomination to return.  Example: getChange(5, 0.99) // should return [1,0,0,0,0,4] getChange(3.14, 1.99) //should return [0,1,1,0,0,1] getChange(3, 0.01) //should return [4,0,2,1,1,2] getChange(4, 3.14) //should return [1,0,1,1,1,0] getChange(0.45, 0.34) // should return [1,0,1,0,0,0] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun getChange(M:Float, P:Float): IntArray {  val changes = intArrayOf(0, 0, 0, 0, 0, 0)  val coinUnits = floatArrayOf(0.01f, 0.05f, 0.1f, 0.25f, 0.5f, 1f)  var rest = M - P;  for(i in 5 downTo 0) {  while(rest > coinUnits[i]) {  changes[i]++  rest -= coinUnits[i]  }   }  return changes } </ending> | </end> |
| <hitle> | P3 Construction Materials infix | <chare> | 1 | <pext> | <starting>P3 Construction Materials infix</heading> Consider the following code used to check whether two materials are compatible to use in a construction project: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* enum class Category {  HEAVY, LIGHT, FILLER }  data class Material (val category: Category)  // Extension infix function. infix fun Material.compatibleWith(other: Material) : Boolean {  return this.category == Category.LIGHT || other.category == Category.FILLER; }  fun main() {  val mat1 = Material(Category.HEAVY)  val mat2 = Material(Category.LIGHT)  // Use mat1 and mat2 here. } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* What is correct about the compatibleWith function? 1-True. compatibleWith can be called as mat1 compatibleWith mat2 2-False. The compatibleWith function can only be called from other methods of the Material class. 3-False. Placing the compatibleWith function and the Material class in two different files will throw compile-time errors, even when using imports. 4-True. The implicit variable this inside the compatibleWith function refers to a Material instance. </ending> | </end> |
| <hitle> | P4 Toy Drone | <chare> | 1 | <pext> | <starting>P4 Toy Drone</heading> The following Kotlin code is used in a toy drone to adjust its movement: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun adjust(vertical: Int?, horizontal: Int?) : Unit {  if (vertical == null) {  println("Didn't get vertical information")  } else {  // Controls the drone with amplified vertical movements  move(vertical \* 2, horizontal!!)  }  log\_movement(vertical ?: -1, horizontal ?: -1) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Select the correct answers about the adjust function. 1-True. If horizontal is null and vertical is not null, a NullPointerException will be thrown. 2-True. When executing the log\_movement call with vertical having a null value, -1 will be sent as the first argument. 3-False. adjust(-1, -1) will produce the same movements as adjust(null, null). 4-False. The code will throw an exception if both vertical and horizontal are null. 5-False. The vertical == null condition must be replaced with vertical.equals(null) to be able to check for nulls. </ending> | </end> |
| <hitle> | P5 registration roster getUniqueStudentIds() | <chare> | 1 | <pext> | <starting>P5 registration roster getUniqueStudentIds()</heading> You work in a university registrar's office. You have been tasked with removing duplicates from the registration roster for final exams.  Given an array of integers a containing the unique student ID numbers of students, write a function that removes any duplicate numbers and returns a new array in which the ordering of the ID numbers remains the same as it was in the original array. Note that all student IDs are 5-digit numbers beginning with the number 5.  Return -1 as a single element in the array in the case of any invalid input. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun getUniqueStudentIds(a: Array<Int>): Array<Int> {  return a.distinct().toTypedArray()   } fun main() {  val input = arrayOf<Int>(1,10,1,4,6,15,15,15)  println(getUniqueStudentIds(input).toList().toString()) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P6 administrative assistant | <chare> | 1 | <pext> | <starting>P7 administrative assistant</heading> You are an administrative assistant at a university. You have been tasked with determining which students are allowed to sit their final exams based on their classroom attendance in three subjects, physics, chemistry and math.  In total, 50 classes were taught in each subject. The criteria for being allowed to sit the exams are as follows: 1. A student must have attended at least 50% of classes for each subject. 2. A student must have attended 70% of classes overall.  Return "true" if a student is allowed to sit the exams, otherwise, return "false" in boolean format.  Return "false" if the attendance records show signs of having been tampered with, e.g. if attendance is below 0 or more than 50 for any subject.  For example, 1. If the number of classes attended are, physics=40, chemistry=40, and math=30, the student will be allowed to sit the final exams. Return "true". 2. If the number of classes attended are, physics=20, chemistry=40, and math=50, the student will not be allowed to sit the final exams. Return "false". 3. If the number of classes attended are, physics=30, chemistry=30, and math=30, the student will not be allowed to sit the final exams. Return "false". \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun isAllowedToSit( physics: Int, chemistry: Int, math: Int ): Boolean {  if(physics/50f < 0.5f) return false  if(chemistry/50f < 0.5f) return false  if(math/50f < 0.5f) return false  if((physics+chemistry+math)/150f < 0.7f) return false  return true  } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P7 Capitalize sentence | <chare> | 1 | <pext> | <starting>P7 Capitalize sentence</heading> Given a string implement a function which capitalizes first letter of every word in that string. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin idiomatic solution object Solution1 {  fun capitalizeSentence(str: String): String {  return str  .split(" ")  .joinToString(" ") { string ->  string.replaceFirstChar { it.uppercase() }  }  } }  // Iterative solution object Solution2 {  fun capitalizeSentence(str: String): String {  val words = mutableListOf<String>()   str.split(" ").forEach {  words.add(it[0].uppercase() + it.substring(1))  }   return words.joinToString(" ")  } } fun main() {  println(Solution1.capitalizeSentence("flower")) // Flower  println(Solution2.capitalizeSentence("this is a house")) // This Is A House } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Flower This Is A House </ending> | </end> |
| <hitle> | P8 Get odd numbers | <chare> | 1 | <pext> | <starting>P8 Get odd numbers</heading> Given a list of integers return a list that contains only odd integers (only integers which are not a multiple of 2). \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin idiomatic solution object Solution1 {  fun filterOdd(list: List<Int>): List<Int> {  return list.filter { it % 2 == 1 }  } }  // Recursive solution object Solution2 {  fun filterOdd(list: List<Int>): List<Int> {   if (list.isEmpty()) {  return list  }   return if (list.first() % 2 == 1) {  mutableListOf(list.first()) + filterOdd(list.drop(1))  } else {  filterOdd(list.drop(1))  }  } }  // Recursive solution with helper function object Solution3 {  fun filterOdd(list: List<Int>): List<Int> {  val result = mutableListOf<Int>()   fun helper(list: List<Int>) {  if (list.isEmpty()) {  return  }   if (list.first() % 2 == 1) {  result.add(list.first())  }   helper(list.drop(1))  }   helper(list)   return result  } } fun main() {  println(Solution1.filterOdd(listOf(1, 2, 3))) // 1, 3  println(Solution2.filterOdd(listOf(4, 6, 8, 7, 9))) // 7, 9 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* [1, 3] [7, 9] </ending> | </end> |
| <hitle> | P9 Contains range | <chare> | 1 | <pext> | <starting>P9 Contains range</heading> Given two ranges implement a function which checks if range1 contains range2. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun containsRange(range1: IntRange, range2: IntRange) =  range2.first >= range1.first && range2.last <= range1.last } fun main() {  println(Solution1.containsRange(5..7, 5..7)) // true  println(Solution1.containsRange(1..12, 5..7)) // true  println(Solution1.containsRange(5..8, 5..9)) // false } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P10 Add up to | <chare> | 1 | <pext> | <starting>P10 Add up to</heading> Given positive integer n implement a function which calculates sum of all numbers from 1 up to (and including) number n. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin idiomatic solution object Solution1 {  fun addUpTo(n: Int): Int {  return (1..n).sum()  } }  // Kotlin idiomatic solution object Solution2 {  fun addUpTo(n: Int): Int {  return (0..n).fold(0) { accumulated, current -> accumulated + current }  } }  // Recursive solution object Solution3 {  fun addUpTo(n: Int): Int {  if (n == 1) {  return 1  }   return n + addUpTo(n - 1)  } }  // Time Complexity: O(1) // Mathematical formula object Solution4 {  fun addUpTo(n: Int): Int {  return n \* (n + 1) / 2  } }  // Time Complexity: O(n) // Iterative solution object Solution5 {  fun addUpTo(n: Int): Int {  var total = 0   (0..n).forEach { total += it }   return total  } }  // Iterative solution object Solution6 {  fun addUpTo(n: Int): Int {  var total = 0  repeat(n + 1) { total += it }  return total  } } fun main() {  println(Solution6.addUpTo(1)) // 1  println(Solution6.addUpTo(2)) // 3  println(Solution6.addUpTo(3)) // 6 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P11 Linear search | <chare> | 1 | <pext> | <starting>P11 Linear search</heading> Given list of strings and a string return index of th value in the list or -1 if value was not found. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity: O(n) object Solution1 {  fun getIndex(list: List<String>, str: String): Int {  list.forEachIndexed { index, element ->  if (element == str) {  return index  }  }  return -1  } }  fun main() {  println(Solution1.getIndex(listOf("A", "B", "C"), "A")) // 0  println(Solution1.getIndex(listOf("A", "B", "C"), "B")) // 1  println(Solution1.getIndex(listOf("A", "B", "C"), "D")) // -1 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P12 Count down | <chare> | 1 | <pext> | <starting>P12 Count down</heading> Given positive integer n implement a function which returns a list containing all steps from n to 0. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin idiomatic solution object Solution1 {  fun countDown(n: Int): List<Int> {  // Create a range and convert it to a list  return (n downTo 0).toList()  } }  // Recursive solution object Solution2 {  fun countDown(n: Int): List<Int> {  if (n == 0) {  return listOf(0)  }   return mutableListOf(n).also { it.addAll(countDown(n - 1)) }  } }  // Recursive solution with helper function object Solution3 {  fun countDown(n: Int): List<Int> {  // We want to keep return type unchanged while implementing recursive solution, so we will  // use helper method defied inside countDown function.  fun helper(n: Int): MutableList<Int> {  if (n == 0) {  return mutableListOf(0)  }   return mutableListOf(n).also { it.addAll(countDown(n - 1)) }  }   return helper(n).toList()  } }  // Kotlin idiomatic solution private object Solution4 {  private fun countDown(n: Int): List<Int> {  return List(n + 1) { n - it }  } }  fun main() {  println(Solution1.countDown(1)) // [1, 0]  println(Solution1.countDown(4)) // [4, 3, 2, 1, 0] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P13 Count up and down | <chare> | 1 | <pext> | <starting>P13 Count up and down</heading> Given positive integer n implement a function which returns a list containing all steps up from 0 to n and down to 0. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Calculate element value from element position object Solution1 {  fun countUpAndDown(n: Int): List<Int> {  val size = (n \* 2) + 1   return List(size) {  when {  it <= n -> it  else -> (n \* 2) - it  }  }  } }  // Returns a new list combining one collection of numbers from 0 to n-1 and another with numbers from n to 0 object Solution2 {  fun countUpAndDown(n: Int): List<Int> {  return (0 until n) + (n downTo 0)  } }  fun main() {  println(Solution1.countUpAndDown(1))// [0, 1, 0]  println(Solution1.countUpAndDown(2))// [0, 1, 2, 1, 0] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P14 Format train route | <chare> | 1 | <pext> | <starting>P14 Format train route</heading> Given list of strings representing train station names implement a function which will return passenger friendly String: 1. String always starts with Train is calling at 2. If list contains single train station name, then return Train is calling at stationName 3. If list contains multiple train station names, then return comma separated station names, but the last two names are separated by and word. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun formatTrainRoute(stations: List<String>): String {  val prefix = "Train is calling at "   if (stations.size == 1) {  return prefix + stations[0]  }   val lastStation = stations.takeLast(1)  .joinToString { it }   val followingStations = stations.dropLast(1)  .joinToString { it }   return "$prefix$followingStations and $lastStation"  } }  object Solution2 {  fun formatTrainRoute(stations: List<String>): String {  var res = ""  stations.forEach {  res += when (it) {  stations.first() -> it  stations.last() -> " and $it"  else -> ", $it"  }  }  return "Train is calling at $res"  } }  fun main() {  println(Solution1.formatTrainRoute(listOf("Luton")))// "Train is calling at Luton"  println(Solution1.formatTrainRoute(listOf("Luton", "Harpenden")))// "Train is calling at Luton and Harpenden"  println(Solution1.formatTrainRoute(listOf("Luton", "Harpenden", "London")))// "Train is calling at Luton, Harpenden and London" } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P15 Generate all pairs | <chare> | 1 | <pext> | <starting>P15 Generate all pairs</heading> Given positive Int n implement a function which returns a list of containing pairs with all combinations of numbers from 0 to n. Use Kotlin Pair class. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun getAllPairs(n: Int): List<Pair<Int, Int>> {  val result = mutableListOf<Pair<Int, Int>>()   (0..n).forEach { i ->  (0..n).forEach { j ->  result.add(i to j)  }  }   return result  } }  object Solution2 {  fun getAllPairs(n: Int): List<Pair<Int, Int>> {  return (0..n).map { i ->  (0..n).map { j ->  i to j  }  }.flatten()  } }  fun main() {  println(Solution1.getAllPairs(0))// [Pair(0, 0)]  println(Solution1.getAllPairs(1))// [Pair(0, 0), Pair(0, 1), Pair(1, 0), Pair(1, 1)]  println(Solution2.getAllPairs(2))// [Pair(0, 0), Pair(0, 1), Pair(0, 2), Pair(1, 0), Pair(1, 1), Pair(1, 2), Pair(2, 0), Pair(2, 1), Pair(2, 2)] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P16 Count unique values | <chare> | 1 | <pext> | <starting>P16 Count unique values</heading> Given a sorted list of integers implement a function which counts the unique values. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun countUniqueValues(list: List<Int>): Int {  val map = mutableMapOf<Int, Int>()   list.forEach {  var value = map.getOrDefault(it, 0)  value++  map[it] = value  }   return map.count()  } }  object Solution2 {  fun countUniqueValues(list: List<Int>): Int {  return list.toSet().size  } }  object Solution3 {  fun countUniqueValues(list: List<Int>): Int {  return list.distinct().size  } }  object Solution4 {  fun countUniqueValues(list: List<Int>): Int {  return list.groupBy { it }.size  } }  fun main() {  println(Solution1.countUniqueValues(listOf())) // 0  println(Solution1.countUniqueValues(listOf(1, 1, 6))) // 2  println(Solution1.countUniqueValues(listOf(-2, -1, -1, 0, 1))) // 4 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P17 Print numbers | <chare> | 1 | <pext> | <starting>P17 Print numbers</heading> Given positive integer n implement a function which returns a list representing all numbers from n to 1. If n value is zero then empty list should be returned. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun printNumber(n: Int): List<Int> {  return (n downTo 1).toList()  } }  object Solution2 {  fun printNumber(n: Int): List<Int> =  when (n) {  0 -> emptyList()  else -> listOf(n) + printNumber(n - 1)  } }  object Solution3 {  fun printNumber(n: Int): List<Int> {  val list = mutableListOf<Int>()   if (n == 0) {  return emptyList()  } else {  list.add(n)  }   list.addAll(printNumber(n - 1))  return list  } }  fun main() {  println(Solution1.printNumber(0)) // []  println(Solution1.printNumber(2)) // [2, 1]  println(Solution1.printNumber(5)) // [5, 4, 3, 2, 1] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P18 Print numbers with steps | <chare> | 1 | <pext> | <starting>Print numbers with steps</heading> Given positive integer n and integer step implement a function which returns a list representing all numbers from n to 1 taking step into consideration. If n value is zero then empty list should be returned. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun printNumber(n: Int, step: Int = 1): List<Int> =  (n downTo 1 step step).toList() }  object Solution2 {  fun printNumber(n: Int, step: Int = 1): List<Int> {  fun printNumberRec(n: Int): List<Int> =  when {  n <= 0 -> emptyList()  else -> listOf(n) + printNumberRec(n - step)  }  return printNumberRec(n)  } }  object Solution3 {  fun printNumber(n: Int, step: Int = 1): List<Int> {  val list = mutableListOf<Int>()   if (n <= 0) {  return emptyList()  } else {  list.add(n)  }   list.addAll(printNumber(n - step, step))  return list  } }  fun main() {  println(Solution1.printNumber(0, 1)) // []  println(Solution1.printNumber(6, 1)) // [6, 5, 4, 3, 2, 1]  println(Solution1.printNumber(6, 2)) // [6, 4, 2] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P19 Power | <chare> | 1 | <pext> | <starting>P19 Power</heading> Given positive integer base and positive integer exponent implement a function which calculates power of base of exponent. This function should mimic functionality of Math.pow. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun power(base: Int, exponent: Int): Int {  if (exponent == 1) {  return base  }   return base \* power(base, exponent - 1)  } }  fun main() {  println(Solution1.power(2, 1)) // 1  println(Solution1.power(2, 2)) // 4  println(Solution1.power(3, 3)) // 27 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P20 Factorial | <chare> | 1 | <pext> | <starting>P20 Factorial</heading> Given positive integer n implement a function, which returns a factorial of that integer. Factorial (of positive integer) is number that is created by multiplying all numbers from 1 to n eg. factorial of 3 (3!) is equal to 6 (3 \* 2 \* 1) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // iterative solution object Solution1 {  fun factorial(n: Int): Int {  var total = 1   (1..n).forEach {  total \*= it  }   return total  } }  // another iterative solution private object Solution2 {  private fun factorial(n: Int): Int =  when (n) {  0 -> 1  else -> (n downTo 1).reduce { acc, it -> acc \* it }  } }  // recursive solution private object Solution3 {  private fun factorial(n: Int): Int =  when (n) {  0, 1 -> 1  else -> n \* factorial(n - 1)  } }  // Tail-recursive solution private object Solution4 {  private fun factorial(n: Int): Int {  fun fact(n: Int, acc: Int = 1): Int =  when (n) {  0, 1 -> acc  else -> fact(n - 1, acc \* n)  }  return fact(n)  } }  fun main() {  println(Solution1.factorial(1)) // 1  println(Solution1.factorial(2)) // 2  println(Solution1.factorial(3)) // 6 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P21 Product | <chare> | 1 | <pext> | <starting>P21 Product</heading> Given a list of integers implement a function which returns the product of all numbers in the list. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin idiomatic solution object Solution1 {  fun product(list: List<Int>): Int {  return list.reduce { accumulated, current -> accumulated \* current }  } }  // Recursive solution private object Solution2 {  private fun product(list: List<Int>): Int {  if (list.size == 1) {  return list.first()  }   return list.first() \* product(list.drop(1))  } }  // Tail-recursive solution private object Solution3 {  private fun product(list: List<Int>): Int {  fun prod(acc: Int, list: List<Int>): Int {  if (list.isEmpty()) {  return acc  }  return prod(acc \* list.first(), list.drop(1))  }  return prod(1, list)  } } fun main() {  println(Solution1.product(listOf(1, 2))) // 2  println(Solution1.product(listOf(1, 2, 4)))// 8  println(Solution1.product(listOf(2, 4, 10)))// 80 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P22 Capitalize First | <chare> | 1 | <pext> | <starting>P22 Capitalize First</heading> Given list of strings implement a recursive function which will capitalize first letter of each string in the list. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin idiomatic solution object Solution1 {  fun capitalizeFirst(list: List<String>): List<String> {  return list.map { it.replaceFirstChar { string -> string.uppercaseChar() } }  } }  // Recursive solution private object Solution2 {  private fun capitalizeFirst(list: List<String>): List<String> {  return if (list.isEmpty()) emptyList()  else listOf(list.first().replaceFirstChar { string -> string.uppercaseChar() }) + capitalizeFirst(list.drop(1))  } }  // Recursive solution private object Solution3 {  private fun capitalizeFirst(list: List<String>): List<String> {  if (list.size == 1) {  return list.map { it.replaceFirstChar { string -> string.uppercaseChar() } }  }   return list.take(1)  .map { it.replaceFirstChar { string -> string.uppercaseChar() } } + capitalizeFirst(list.drop(1))  } } fun main() {  println(Solution1.capitalizeFirst(listOf("igor")))// listOf("Igor")  println(Solution1.capitalizeFirst(listOf("igor", "wojda")))// listOf("Igor", "Wojda") } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P23 Decapitalize const | <chare> | 1 | <pext> | <starting>P23 Decapitalize const</heading> Given a string representing constant name (upper case words separated by underscore eg FOO\_BAR) implement a function which converts it into variable name (eg. fooBar): 1. convert first word to lowercase 2. convert next words to lowercase, but first character is still uppercase 3. remove all underscore characters \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin idiomatic solution object Solution1 {  fun decapitalizeConst(str: String): String {  val subsStringsList = str  .split("\_")  .map { it.lowercase() }  .map { it.replaceFirstChar { string -> string.uppercaseChar() } }   return subsStringsList  .joinToString("")  .replaceFirstChar {  it.lowercaseChar()  }  } }  // Another Approach private object Solution2 {  private fun decapitalizeConst(str: String): String? {  val words = str.split("\_").filter { it.isNotEmpty() }   if (words.size == 1) return words.first().lowercase()   return words.mapIndexed { index, word ->  if (index == 0) {  word.lowercase()  } else {  word.first().uppercase() + word.drop(1).lowercase()  }  }.joinToString(separator = "")  } } fun main() {  println(Solution1.decapitalizeConst("FOO"))// foo  println(Solution1.decapitalizeConst("FOO\_BAR"))// fooBar  println(Solution1.decapitalizeConst("\_\_FOO\_BAR\_BAZ"))// fooBarBaz } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P24 Longest word | <chare> | 1 | <pext> | <starting>P24 Longest word</heading> Given a string implement a function which returns longest word in that string. If there are two or more words that have the same length, return the first longest word from the string. Ignore punctuation. Input string can't be empty or blank sering. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun longestWord(str: String): String {  return str.map { if (it.isLetter()) it else ' ' }  .joinToString(separator = "")  .split(" ")  .filterNot { it.isBlank() }  .maxBy { it.length }  } } fun main() {  println(Solution1.longestWord("big flower"))// flower  println(Solution1.longestWord("this is a house"))// house } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P25 Is palindrome | <chare> | 1 | <pext> | <starting>P25 Is palindrome</heading> Given a string, return true if the string is a palindrome or false if it is not. Palindromes are strings that form the same word if it is reversed. Include spaces and punctuation in determining if the string is a palindrome. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // string reverse object Solution1 {  fun isPalindrome(str: String): Boolean {  return str == str.reversed()  } }  // iterative, double pointer solution private object Solution2 {  private fun isPalindrome(str: String): Boolean {  var leftIndex = 0  var rightIndex = str.lastIndex   while (leftIndex <= rightIndex) {  val leftValue = str[leftIndex]  val rightValue = str[rightIndex]   if (leftValue != rightValue) {  return false  }   leftIndex++  rightIndex--  }   return true  } }  // iterative, double pointer solution private object Solution3 {  private fun isPalindrome(str: String): Boolean {  str.forEachIndexed { index, char ->  val rightIndex = str.lastIndex - index   if (char != str[rightIndex])  return false   if (index > rightIndex)  return true  }   return true  } }  // recursive solution private object Solution4 {  private fun isPalindrome(str: String): Boolean {  return if (str.isEmpty() || str.length == 1) {  true  } else {  if (str.first() == str.last()) {  isPalindrome(str.substring(1 until str.lastIndex))  } else {  false  }  }  } } fun main() {  println(Solution1.isPalindrome("abba")) // true  println(Solution1.isPalindrome("abcdefg")) // false } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P26 Is tolerant palindrome | <chare> | 1 | <pext> | <starting>P26 Is tolerant palindrome</heading> Given a string, return true if the string is a palindrome or false if it is not. Palindromes are strings that form the same word when reversed (including spaces and punctuation). In this exercise one bad character is allowed. The string will be (tolerant) palindrome if after removing one character string is a palindrome. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // iterative solution object Solution1 {  fun isTolerantPalindrome(str: String): Boolean {  var characterRemoved = false  var leftIndex = 0  var rightIndex = str.lastIndex  while (leftIndex <= rightIndex) {  if (str[leftIndex] != str[rightIndex]) {  if (characterRemoved) {  return false  }  characterRemoved = true  when {  str[leftIndex + 1] == str[rightIndex] -> leftIndex++  str[leftIndex] == str[rightIndex - 1] -> rightIndex--  else -> return false  }  }  leftIndex++  rightIndex--  }  return true  } } // recursive solution private object Solution2 {  private fun isTolerantPalindrome(str: String, characterRemoved: Boolean = false): Boolean {  return if (str.isEmpty() || str.length == 1) {  true  } else {  if (str.first() == str.last()) {  isTolerantPalindrome(  str.substring(1 until str.lastIndex),  characterRemoved  )  } else {  if (characterRemoved) {  false  } else {  if (str.length == 2) {  return true  }   val removeLeftResult = isTolerantPalindrome(  str.substring(2 until str.lastIndex),  true  )  val removeRightResult = isTolerantPalindrome(  str.substring(1 until str.lastIndex - 1),  true  )   return removeLeftResult || removeRightResult  }  }  }  } }  // recursive solution 2 private object Solution3 {  private fun isTolerantPalindrome(str: String, characterRemoved: Boolean = false): Boolean {  val revStr = str.reversed()  if (revStr == str) return true  if (characterRemoved) return false   // Remove a single non matching character and re-compare  val removeIndex = str.commonPrefixWith(revStr).length  if (removeIndex + 1 > str.length) return false // reached end of string  val reducedStr = str.removeRange(removeIndex, removeIndex + 1)  return if (isTolerantPalindrome(reducedStr, true)) {  true  } else {  val reducedRevStr = revStr.removeRange(removeIndex, removeIndex + 1)  isTolerantPalindrome(reducedRevStr, true)  }  } } fun main() {  println(Solution1.isTolerantPalindrome("abba"))// == true  println(Solution1.isTolerantPalindrome("abb#a"))// == true  println(Solution1.isTolerantPalindrome("abcdefg"))// == false } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P27 Is permutation palindrome | <chare> | 1 | <pext> | <starting>P27 Is permutation palindrome</heading> Given a string, return true if the input string is an a permutation of palindrome or false if it is not. Permutation means that all letters of the string can be used to create a palindrome. Palindromes are strings that form the same word if it is reversed. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // frequency map solution private object Solution1 {  private fun isPermutationPalindrome(str: String, characterRemoved: Boolean = false): Boolean {  val charactersFrequencyMap = str.groupingBy { it }.eachCount()   val filtered = charactersFrequencyMap.filter { it.value % 2 == 1 }   return filtered.size <= 1  } } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* permutationPalindrome("gikig") == true permutationPalindrome("ookvk") == true permutationPalindrome("sows") == false permutationPalindrome("tami") == false </ending> | </end> |
| <hitle> | P28 Is anagram | <chare> | 1 | <pext> | <starting>P28 Is anagram</heading> Given two strings, implement a function to determine if the second string is an anagram of the first. An anagram is a word, phrase, or name formed by rearranging the letters of another, such as cinema, formed from iceman. Only consider characters, not spaces or punctuation. Consider capital letters to be the same as lower case. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun isAnagram(str1: String, str2: String): Boolean {  val a1 = str1.uppercase().filter { it.isLetter() }.groupBy { it }  val a2 = str2.uppercase().filter { it.isLetter() }.groupBy { it }  return a1 == a2  } }  private object Solution2 {  private fun isAnagram(str1: String, str2: String): Boolean {  return getCharFrequency(str1) == getCharFrequency(str2)  }   private fun getCharFrequency(str: String): Map<Char, List<Char>> {  return str.lowercase()  .filter { it.isLetterOrDigit() }  .groupBy { it }  } }  private object Solution3 {  private fun isAnagram(str1: String, str2: String): Boolean {  return getCharFrequency(str1) == getCharFrequency(str2)  }   private fun getCharFrequency(str: String): Map<Char, Int> {  return str.lowercase()  .filter { it.isLetterOrDigit() }  .groupingBy { it }  .eachCount()  } } fun main() {  println(Solution1.isAnagram("rail safety", "fairy tales"))// --> True  println(Solution1.isAnagram("RAIL! SAFETY!", "fairy tales"))// --> True  println(Solution1.isAnagram("Hi there", "Bye there"))// --> False } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P29 Max occurrent char | <chare> | 1 | <pext> | <starting>P29 Max occurrent char</heading> Given a string implement a function which returns the character that is most commonly used in that string. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin idiomatic solution object Solution1 {  fun maxOccurrentChar(str: String): Char? {  if (str.isBlank()) return null  return str.toCharArray()  .groupBy { it }  .maxBy { it.value.size }  .key  } }  // Kotlin idiomatic solution private object Solution2 {  private fun maxOccurrentChar(str: String): Char? {  if (str.isBlank()) return null   return str.toList()  .groupingBy { it }  .eachCount()  .maxBy { it.value }  .key  } }  private object Solution3 {  private fun maxOccurrentChar(str: String): Char? {  if (str.isBlank()) return null   val map = mutableMapOf<Char, Int>()   str.forEach {  map[it] = (map[it] ?: 0) + 1  }   return map.maxBy { it.value }.key  } }  // Recursive naive approach // Time complexity: O(n^2) private object Solution4 {  private fun maxOccurrentChar(str: String): Char? {  if (str.length == 1) {  return str.first()  }   str.forEachIndexed { index, c ->  str.substring(index + 1).forEach {  if (c == it) {  return it  }  }  }   return null  } } fun main() {  println(Solution1.maxOccurrentChar("abcccccccd")) // "c"  println(Solution1.maxOccurrentChar("apple 1231111")) // "1" } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P30 Reverse string | <chare> | 1 | <pext> | <starting>P30 Reverse string</heading> Given a string implement a function which returns a reversed copy of that string. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin idiomatic way object Solution1 {  fun reverse(str: String): String {  return str.reversed()  } }  // Iterative approach private object Solution2 {  private fun reverse(str: String): String {  var reversed = ""  str.forEach {  reversed = it + reversed  }  return reversed  } }  // Recursive approach private object Solution3 {  private fun reverse(str: String): String {  if (str.isEmpty()) {  return str  }   return reverse(str.drop(1)) + str.first()  } }  // Kotlin fold private object Solution4 {  private fun reverse(str: String): String {  return str.foldRight("") { char, reversed -> reversed + char }  } }  // Time complexity: O(n) // Space complexity: O(1) // Reverse in place private object Solution5 {  private fun reverse(str: String): String {  val chars = str.toMutableList()   var leftIndex = 0  var rightIndex = chars.lastIndex   while (leftIndex <= rightIndex) {  val temp = chars[leftIndex]  chars[leftIndex] = chars[rightIndex]  chars[rightIndex] = temp   leftIndex++  rightIndex--  }   return chars.joinToString(transform = { it.toString() }, separator = "")  } } fun main() {  println(Solution1.reverse("apple"))// == 'leppa'  println(Solution1.reverse("hello"))// == 'olleh'  println(Solution1.reverse("Greetings!"))// == '!sgniteerG' } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P31 Find the vowels | <chare> | 1 | <pext> | <starting>P31 Find the vowels</heading> Given string implement a function, which returns the number of vowels used in a string. Vowels are the characters a , e i, o, u and y. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin collection processing object Solution1 {  fun vowels(str: String): Int {  val vowels = listOf('a', 'e', 'i', 'o', 'u', 'y')  return str.count { it.lowercaseChar() in vowels }  } }  // Iterative private object Solution2 {  private fun vowels(str: String): Int {  val vowels = listOf('a', 'e', 'i', 'o', 'u', 'y')  var counter = 0   str.forEach {  if (vowels.contains(it.lowercaseChar())) {  counter++  }  }   return counter  } }  // Regex private object Solution3 {  private fun vowels(str: String): Int {  return Regex("[aeiouy]").findAll(str.lowercase()).count()  } } fun main() {  println(Solution1.vowels("Hi There!")) // 3  println(Solution1.vowels("Why do you ask?")) // 6  println(Solution1.vowels("Why?")) // 1 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P32 Reverse Int | <chare> | 1 | <pext> | <starting>P32 Reverse Int</heading> Given an integer n, return an integer that is the reverse ordering of digits. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* import kotlin.math.sign  object Solution1 {  fun reverseInt(i: Int): Int {  var reverse = i.toString().removePrefix("-").reversed()  return reverse.toInt() \* i.sign  } } fun main() {  println(Solution1.reverseInt(15)) // 51  println(Solution1.reverseInt(981)) // 189  println(Solution1.reverseInt(500)) // 5  println(Solution1.reverseInt(-15)) // -51  println(Solution1.reverseInt(-90)) // -9 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P33 Pyramid generator | <chare> | 1 | <pext> | <starting>P33 Pyramid generator</heading> Given a positive integer n implement a function which returns a list representing the generatePyramid shape with n levels. The function should console log a generatePyramid shape with N levels using the # character. Make sure the generatePyramid has spaces on both the left and right hand sides. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // iterative solution object Solution1 {  fun generatePyramid(n: Int): List<String> {  val list = mutableListOf<String>()  val numColumns = (n \* 2) - 1   (0 until n).forEach { row ->  val numHashes = (row \* 2) + 1  val numSpaces = numColumns - numHashes  var sideString = ""  repeat(numSpaces / 2) { sideString += " " }  var hashString = ""  repeat(numHashes) { hashString += "#" }  list.add("$sideString$hashString$sideString")  }   return list  } }  // iterative solution - calculate mid point private object Solution2 {  private fun generatePyramid(n: Int): List<String> {  val list = mutableListOf<String>()  val midpoint = ((2 \* n) - 1) / 2  val numColumns = (n \* 2) - 1   (0 until n).forEach { row ->  var rowStr = ""  (0 until numColumns).forEach { column ->  rowStr += if (midpoint - row <= column && midpoint + row >= column) {  "#"  } else {  " "  }  }  list.add(rowStr)  }   return list  } }  // simplified iterative solution private object Solution3 {  private fun generatePyramid(n: Int): MutableList<String> {  val list = mutableListOf<String>()  val maxRowLen = n \* 2 - 1   for (i in 1..n) {  val rowLen = i \* 2 - 1   val sideString = " ".repeat((maxRowLen - rowLen) / 2)  val hashString = "#".repeat(rowLen)   list.add("$sideString$hashString$sideString")  }  return list  } } fun main() {  println(Solution1.generatePyramid(1))  println(Solution1.generatePyramid(2))  println(Solution1.generatePyramid(3)) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* pyramid(1)  '#'  pyramid(2)  ' # '  '###'  pyramid(3)  ' # '  ' ### '  '#####' </ending> | </end> |
| <hitle> | P34 Steps generator | <chare> | 1 | <pext> | <starting>P34 Steps generator</heading> Given a positive integer n implement a function which returns a list representing the generated steps with n levels using the # character. Make sure the step has appropriate number of spaces on the right hand side! \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin repeat function private object Solution1 {  private fun generateSteps(n: Int) = List(n) { "#".repeat(it + 1) + " ".repeat(n - it - 1) } }  // iterative solution private object Solution2 {  private fun generateSteps(n: Int): MutableList<String> {  val list = mutableListOf<String>()   (1..n).forEach { row ->  var item = ""   (1..n).forEach { column ->  val char = if (column <= row) '#' else ' '  item += char  }   list.add(item)  }   return list  } } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P35 FizzBuzz | <chare> | 1 | <pext> | <starting>sqlite</heading> Given positive integer n implement a function which returns a list numbers from 1 to n. However for multiples of three list should contain word Fizz instead of the number and for the multiples of five list should contain word Buzz. For numbers which are multiples of both three and five list should contain FizzBuzz word. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun fizzBuzz(n: Int): List<String> {  val list = mutableListOf<String>()   (1..n).forEach {  val item = when {  it % 3 == 0 && it % 5 == 0 -> "FizzBuzz" // also (it % 15 == 0) because 3x5=15  it % 3 == 0 -> "Fizz"  it % 5 == 0 -> "Buzz"  else -> it.toString()  }   list.add(item)  }   return list  } }  fun main() {  println(Solution1.fizzBuzz(5)) // [1, 2, "Fizz", 4, "Buzz"]  println(Solution1.fizzBuzz(16)) // [1, 2, "Fizz", 4, "Buzz", "Fizz", 7, 8, "Fizz", "Buzz", 11, "Fizz", 13, 14, "FizzBuzz", 16] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P36 Caesar Cipher | <chare> | 1 | <pext> | <starting>P36 Caesar Cipher</heading> Given a string and integer shift implement a function which shifts each character in that string by shift amount ( cesar cipher) eg. if shift is 1 then a becomes b, c becomesd, z became a etc. Allowed input is string containing only lowercase characters from alphabet abcdefghijklmnopqrstuvwxyz. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun encodeCaesarCipher(str: String, shift: Int): String {  val aCode = 'a'.code  val zCode = 'z'.code  val realShift = shift % (zCode - aCode + 1)   return str.map {  var code = it.code  code += realShift   if (code > zCode) {  code = aCode + (code % zCode) - 1  }   code.toChar()  }.joinToString(separator = "")  } }  private object Solution2 {  private fun encodeCaesarCipher(str: String, shift: Int): String {  val alphabet = "abcdefghijklmnopqrstuvwxyz"   var encoded = ""   str.forEach {  val indexInAlphabet = alphabet.indexOf(it)  val newIndex = (indexInAlphabet + shift) % alphabet.length  encoded += alphabet[newIndex]  }   return encoded  } } fun main() {  println(Solution1.encodeCaesarCipher("abc", 1)) // bcd  println(Solution1.encodeCaesarCipher("abc", 3)) // def  println(Solution1.encodeCaesarCipher("xyz", 1)) // yza } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P37 Has repeated char | <chare> | 1 | <pext> | <starting>P37 Has repeated char</heading> Given a string implement a function which returns true if string contains at least one repeated character, otherwise it returns false. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity: O(n) object Solution1 {  fun hasRepeatedChar(str: String): Boolean {  val frequency = str.groupingBy { it }.eachCount()  return frequency.any { it.value > 1 }  } }  // Time complexity: O(n) private object Solution2 {  private fun hasRepeatedChar(str: String): Boolean {  val frequency = str.groupingBy { it }.eachCount()   frequency.forEach {  if (it.value > 1) {  return true  }  }   return false  } } fun main() {  println(Solution1.hasRepeatedChar("abc")) // false"  println(Solution1.hasRepeatedChar("aabc")) // true  println(Solution1.hasRepeatedChar("aabcc")) // true } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P38 Spiral matrix generator | <chare> | 1 | <pext> | <starting>P38 Spiral matrix generator</heading> Given integer n implement a function which generates a spiral numbers matrix (size n x n). Numbers in matrix are placed according to spiral - starting top left corner and then go right, down, left, up, right... \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun generateSpiralMatrix(n: Int): List<MutableList<Int?>> {  val list = MutableList(n) {  MutableList<Int?>(n) { null }  }   var direction = MatrixDirection.RIGHT  var rowIndex = 0  var columnIndex = 0  val maxNumber = n \* n  val lastIndex = n - 1   (1..maxNumber).forEach {  list[rowIndex][columnIndex] = it   when (direction) {  MatrixDirection.RIGHT -> {  val rightValue = if (columnIndex == lastIndex) null else list[rowIndex][columnIndex + 1]   if (rightValue == null && columnIndex < lastIndex) {  columnIndex++  } else {  direction = MatrixDirection.DOWN  rowIndex++  }  }  MatrixDirection.DOWN -> {  val downValue = if (rowIndex == lastIndex) null else list[rowIndex + 1][columnIndex]   if (downValue == null && rowIndex < lastIndex) {  rowIndex++  } else {  columnIndex--  direction = MatrixDirection.LEFT  }  }  MatrixDirection.LEFT -> {  val leftValue = if (columnIndex == 0) null else list[rowIndex][columnIndex - 1]   if (leftValue == null && columnIndex > 0) {  columnIndex--  } else {  rowIndex--  direction = MatrixDirection.UP  }  }  MatrixDirection.UP -> {  val upValue = if (rowIndex == 0) null else list[rowIndex - 1][columnIndex]   if (upValue == null && rowIndex > 0) { // rowIndex == 0  rowIndex--  } else {  columnIndex++  direction = MatrixDirection.RIGHT  }  }  }  }   return list  }   private enum class MatrixDirection { UP, DOWN, LEFT, RIGHT } } fun main() {  println(Solution1.generateSpiralMatrix(2))  println(Solution1.generateSpiralMatrix(3))  println(Solution1.generateSpiralMatrix(4)) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Example 1 generateSpiralMatrix(2)  [[1, 2],  [4, 3]] Example 2 generateSpiralMatrix(3)  [[1, 2, 3],  [8, 9, 4],  [7, 6, 5]] Example 3 generateSpiralMatrix(4)  [[1, 2, 3, 4],  [12, 13, 14, 5],  [11, 16, 15, 6],  [10, 9, 8, 7]] </ending> | </end> |
| <hitle> | P39 Find rectangle | <chare> | 1 | <pext> | <starting>Find rectangle</heading> Image is represent as a simple 2D array where every pixel is a 1 or a 0. The image you get is known to have a single rectangle of 0s on a background of 1s. Write a function that takes in the image and returns list containing coordinates of top-left and bottom-right pixels represented as list of internees: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity: O(n\*m) object Solution1 {  fun findRectangle(image: List<List<Int>>): List<Int>? {   // This helper function may be used with challange variation where multiple rectangles exists in the image  fun getRectangleFromTopLeftCorner(rowIndex: Int, columnIndex: Int): List<Int> {  // find right x coordinate  var rightColumnIndex = columnIndex   while (image.getOrNull(rowIndex)?.getOrNull(rightColumnIndex + 1) == 0) {  rightColumnIndex++  }   // find bottom y coordinate  var bottomRowIndex = rowIndex   while (image.getOrNull(bottomRowIndex + 1)?.getOrNull(columnIndex) == 0) {  bottomRowIndex++  }   return listOf(rowIndex, columnIndex, bottomRowIndex, rightColumnIndex)  }   image.forEachIndexed { rowIndex, rowItem ->  rowItem.forEachIndexed { columnIndex, pixelValue ->  if (pixelValue == 0) {  return getRectangleFromTopLeftCorner(rowIndex, columnIndex)  }  }  }   return null  } }  // Time complexity: O(n\*m) private object Solution2 {  private fun findRectangle(image: List<List<Int>>): List<Int>? {  var top: Pair<Int, Int>? = null  var bottom: Pair<Int, Int>? = null   image.forEachIndexed { rowIndex, rowItem ->  rowItem.forEachIndexed { columnIndex, pixelValue ->  if (pixelValue == 0) {  if (top == null) {  top = rowIndex to columnIndex  } else {  bottom = rowIndex to columnIndex  }  }  }  }   // handle edge case where rectangle has 1 pixel  if (bottom == null) {  bottom = top  }   return if (top == null || bottom == null) {  null  } else {  // Smart cast does not work with inlined lambdas so we have to use non-null assertion operator  // More: https://youtrack.jetbrains.com/issue/KT-7186  listOf(top!!.first, top!!.second, bottom!!.first, bottom!!.second)  }  } } fun main() {  val image = listOf(  listOf(1, 1, 0, 0, 0, 1),  listOf(1, 1, 0, 0, 0, 1),  listOf(1, 1, 1, 1, 1, 1),  listOf(1, 1, 1, 1, 1, 1)  )  println(Solution1.findRectangle(image)) // [0, 2, 1, 4] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P40 Queue | <chare> | 1 | <pext> | <starting>P40 Queue</heading> Implement a queue data structure. Adding to the queue should store an element until it is removed. First element added to a queue will be the first that is removed (FIFO). The queue should be a class with methods:  add method - adds element to a queue (enqueue the element) remove method - removes the element from a queue (enqueue the element) peek method - returns last element (the one that should be returned) without removing it from the queue isEmpty method - return true if queue is empty, otherwise return false \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* /\* Linked List based implementation Time complexity: Insertion: O(1) Removal: O(1) Searching: O(n) Access: O(n) \*/ object Solution1 {  class Queue<E> {  var size = 0  private set   private var first: Node<E>? = null  private var last: Node<E>? = null    fun    fun add(element: E) {  val node = Node(element)   if (first == null) {  first = node  } else {  last?.next = node  }   size++  last = node  }   fun remove(): E? {  if (size == 0) return null   val node = first  first = node?.next  size--  return node?.data  }   fun peek() = first?.data   fun isEmpty() = first == null  }   private data class Node<E>(val data: E, var next: Node<E>? = null) }  /\* List based implementation It's important to notice that adding element to the beginning of the array or removing element from the beginning is expensive operation (all subsequent items have to be re-indexed): Option A - add to the end, remove from the beginning: Insert at the end O(1) Remove at the end O(1) Option B - add to the beginning, remove from the end: Bad idea to add elements at the beginning: Insert at the beginning O(n) Remove at the beginning O(n) - we have to re-index all the other elements in the list If we would add new element to the beginning of the list (expensive), we would have to remove them from the end of the list (cheap). If we would add new element to the end of the array (cheap) we would have to remove ti from the beginning (expensive). Because of that the list based implementation can't be efficient. We could use linked list based implementation instead. Time complexity (add at the beginning and remove from the end): Insertion: O(n) Removal: O(1) Searching: O(n) Access: O(n) Solution time complexity (add at the end and remove from the beginning): Insertion: O(1) Removal: O(n) Searching: O(n) Access: O(n) \*/ private object Solution2 {  private class Queue<E> {  private val list = mutableListOf<E>()   fun add(element: E) {  list.add(element)  }   fun remove() = if (list.isEmpty()) null else list.removeAt(0)   fun peek() = list.firstOrNull()   fun isEmpty() = list.isEmpty()   val size get() = list.size  } }  // Two Stack based implementation private object Solution3 {  private class Queue<E> {  val primaryStack = Stack<E>()  val temporaryStack = Stack<E>()   val size get() = primaryStack.size   fun add(element: E) {  primaryStack.add(element)  val a = 2  }   fun remove(): E? {  moveElementsToTemporaryStack()  val result = temporaryStack.remove()  moveElementsToPrimaryStack()  return result  }   fun peek(): E? {  moveElementsToTemporaryStack()  val result = temporaryStack.peek()  moveElementsToPrimaryStack()   return result  }   private fun moveElementsToTemporaryStack() {  while (primaryStack.peek() != null) {  primaryStack.remove()?.let { temporaryStack.add(it) }  }  }   private fun moveElementsToPrimaryStack() {  while (temporaryStack.peek() != null) {  temporaryStack.remove()?.let { primaryStack.add(it) }  }  }   fun isEmpty() = primaryStack.isEmpty()  }   private class Stack<E> {  private val list = mutableListOf<E>()   val size get() = list.size   fun add(element: E) {  list.add(element)  }   fun remove() = if (list.isEmpty()) null else list.removeAt(list.lastIndex)   fun peek() = list.lastOrNull()   fun isEmpty() = list.isEmpty()  } } fun main() {  val q = Solution1.Queue<Char>()  q.isEmpty() // true  q.add('A')  q.isEmpty() // false  q.add('B')  q.add('C')  q.remove() // A  q.peek() // A  q.peek() // A  q.remove() // B  q.remove() // C  q.remove() // null  println(q.toString()) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P41 Combine two queues | <chare> | 1 | <pext> | <starting>P41 Combine two queues</heading> Given two queues implement a function which combines the contents of each queue into third queue. The third queue should contain the content of the two queues. The function should handle queues of different lengths without inserting null into the new one. Take into consideration the fact that both queues may be of different length. IntQueue can only expose add, remove, and peek methods to external clients. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  class Queue<E> {  private val list = mutableListOf<E>()   fun add(element: E) {  list.add(element)  }   fun remove() = if (list.isEmpty()) null else list.removeAt(0)   fun peek() = list.firstOrNull()  }   fun combine(q1: Queue<\*>, q2: Queue<\*>): Queue<\*> {  val result = Queue<Any>()   while (true) {  q1.remove()?.let { result.add(it) }  q2.remove()?.let { result.add(it) }   if (q1.peek() == null && q2.peek() == null) {  break  }  }   return result  } fun main() {  val queueOne = Queue<Int>()  queueOne.add(1)  queueOne.add(2)   val queueTwo = Queue<String>()  queueTwo.add("Hi")  queueTwo.add("There")   val queueThree = combine(queueOne, queueTwo)  queueThree.remove() // 1  queueThree.remove() // 'Hi'  queueThree.remove() // 2  queueThree.remove() // 'There' } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P42 Stack | <chare> | 1 | <pext> | <starting>P42 Stack</heading> Implement a stack data structure. Adding to the stack should store an element until it is removed. First element added to a stack will be the last that is removed (FILO). The stack should be a class with members: 1. add method - adds element to the stack 2. remove method - removes the "top" element from the stack 3. peek method - returns "top" element (the one that should be returned)  4. without removing (removeping) it from the stack 5. isEmpty() - returns true if there are elements on the stack, otherwise return false size - numbers of items in the stack \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* /\* LinkedList based implementation Time complexity: Insertion: O(1) Removal: O(1) Searching: O(n) Access: O(n) \*/ object Solution1 {  class Stack<E> {  private var first: Node<E>? = null   var size = 0  private set   fun add(element: E) {  val node = Node(element)   if (first == null) {  first = node  } else {  node.next = first  first = node  }   size++  }   fun remove(): E? {  if (size == 0) return null   val node = first  first = first?.next  size--  return node?.data  }   fun peek(): E? = first?.data   fun isEmpty() = first == null  }   private data class Node<E>(val data: E, var next: Node<E>? = null) }  /\* List based implementation It's important to notice that we should add new element to the end of the list, because adding to the beginning is expensive (all subsequent items have to be re-indexed): Option A - add to the end, remove from the beginning: Insert at the end O(1) Remove at the end O(1) Option B - add to the beginning, remove from the end: Bad idea to add elements at the beginning: Insert at the beginning O(n) Remove at the beginning O(n) - we have to re-index all the other elements in the list Solution time complexity: Insertion: O(1) Removal: O(1) Searching: O(n) Access: O(n) \*/ private object Solution2 {  private class Stack<E> {  private val list = mutableListOf<E>()   val size get() = list.size   fun add(element: E) {  list.add(element)  }   fun remove() = if (list.isEmpty()) null else list.removeAt(list.lastIndex)   fun peek() = list.lastOrNull()   fun isEmpty() = list.isEmpty()  } } fun main() {  val s = Solution1.Stack<Int>()  s.add(1)  s.add(2)  s.remove() // 2  s.remove() // 1  s.remove() // null } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P43 Bubble sort | <chare> | 1 | <pext> | <starting>P43 Bubble sort</heading> Sort list of numbers from lowest number to greatest number using bubble sort. Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity (Best): Ω(n) - all data besides one element are sorted // Time complexity (Average): Θ(n^2) // Time complexity (Worst): O(n^2) // Space complexity: O(1) object Solution1 {  fun bubbleSort(list: List<Int>): List<Number> {  val sorted = list.toMutableList()   (0..sorted.lastIndex).forEach { i ->  var swapped = false  (0 until (sorted.size - i - 1)).forEach { j ->  val element = sorted[j]  val nextElement = sorted[j + 1]   if (element > nextElement) {  sorted[j] = nextElement  sorted[j + 1] = element  swapped = true  }  }   if (!swapped) {  return sorted  }  }   return sorted  } } fun main() {  println(Solution1.bubbleSort(listOf(5, 1, 4, 2, 8))) //[1, 2, 4, 5, 8] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P44 Selection sort | <chare> | 1 | <pext> | <starting>P44 Selection sort</heading> Sort list of numbers from lowest number to greatest number using selection sort. Algorithm: Sort an list by repeatedly finding the minimum element (ascending order) from unsorted part and putting it at the beginning of the list. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity (Best): Ω(n^2) // Time complexity (Average): Θ(n^2) // Time complexity (Worst): O(n^2) // Space complexity: O(1) object Solution1 {  fun selectionSort(list: List<Int>): List<Number> {  val sorted = list.toMutableList()   (0..sorted.lastIndex).forEach { i ->  var indexOfMin = i   ((i + 1)..sorted.lastIndex).forEach { j ->  if (sorted[j] < sorted[indexOfMin]) {  indexOfMin = j  }  }   if (indexOfMin != i) {  val temp = sorted[i]  sorted[i] = sorted[indexOfMin]  sorted[indexOfMin] = temp  }  }   return sorted  } }  fun main() {  println(Solution1.selectionSort(listOf(5, 1, 4, 2, 8))) //[1, 2, 4, 5, 8] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P45 Square equals | <chare> | 1 | <pext> | <starting>P45 Square equals</heading> Given two lists of integers implement a function which returns true if every value in the first list has it's corresponding value squared in the second list. The frequency of values must be the same eg. if there are two values in first list, two exactly two squared values must exist in second list. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  // Time complexity: O(n)  // Create two frequency map for each list and compare them  fun squareEquals(list: List<Int>, squared: List<Int>): Boolean {  if (list.size != squared.size) {  return false  }   val frequencyCounter1 = list.map { it \* it }.groupBy { it }   val frequencyCounter2 = squared.groupBy { it }   return frequencyCounter1 == frequencyCounter2  } }  private object Solution2 {  // Time complexity: O(n^2)  // Loop through first list and look for value square in the second list. If square if found  // remove it from second list to make sure that item frequency match.  private fun squareEquals(list: List<Int>, squared: List<Int>): Boolean {  val mutableList2 = squared.toMutableList()   if (list.size != squared.size) {  return false  }   list.forEach {  // under the hood 'indexOf' iterates through entire list (it's like nested loop)  val index = mutableList2.indexOf(it \* it)   if (index == -1) {  return false  }   mutableList2.removeAt(index)  }   return true  } } fun main() {  println(Solution1.squareEquals(listOf(2), listOf(4))) // true  println(Solution1.squareEquals(listOf(1, 2, 3), listOf(9, 1, 4))) // true  println(Solution1.squareEquals(listOf(1, 2, 3), listOf(9, 1, 7))) // false (does not have square of 3)  println(Solution1.squareEquals(listOf(1, 2, 3), listOf(9, 1))) // false (does not have square of 2)  println(Solution1.squareEquals(listOf(1, 2, 1), listOf(4, 1, 4))) // false (frequency must be the same) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P46 Sum zero | <chare> | 1 | <pext> | <starting>P46 Sum zero</heading> Given sorted list of integers implement a function which finds the first pair where the sum equals to 0. Return an pair that includes both values that sum to zero or null if a pair does not exist. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  // Optimal solution using double pointer.  // Time complexity: O(n)  fun sumZero(list: List<Int>): Pair<Int, Int>? {  if (list.isEmpty()) {  return null  }   var pointer1 = 0  var pointer2 = list.lastIndex   // declare loop  while (pointer1 != pointer2) {  val element1 = list[pointer1]  val element2 = list[pointer2]  val sum = element1 + element2   when {  sum == 0 -> return Pair(element1, element2)  sum > 0 -> pointer2--  sum < 0 -> pointer1++  }  }   return null  } }  // Naive solution using nested loop // Time complexity: O(n^2) private object Solution2 {  fun sumZero(list: List<Int>): Pair<Int, Int>? {  list.forEachIndexed { index, element1 ->  (index..list.lastIndex).forEach { index2 ->  if (element1 + list[index2] == 0) {  return element1 to list[index2]  }  }  }   return null  } } fun main() {  println(Solution1.sumZero(listOf(1, 2))) // null  println(Solution1.sumZero(listOf(-3, -2, 0, 1, 2))) // Pair(-2, 2) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P47 Digit frequency | <chare> | 1 | <pext> | <starting>Digit frequency</heading> Given two integers implement a function which determines if they have the same frequency of digits. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity: O(n) // Generate digit frequency map for each integer and compare them object Solution1 {  fun equalDigitFrequency(i1: Int, i2: Int): Boolean {  val i1Str = i1.toString()  val i2Str = i2.toString()   if (i1Str.length != i2Str.length) {  return false  }   val frequencyCounter1 = i1Str.groupingBy { it }.eachCount()  val frequencyCounter2 = i2Str.groupingBy { it }.eachCount()  return frequencyCounter1 == frequencyCounter2  } }  // Time complexity: O(n^2) // Loop through each character of first integer and look for this character in another integer. If character if found // remove it from second integer to make sure that character frequency match. private object Solution2 {  private fun equalDigitFrequency(i1: Int, i2: Int): Boolean {  val i1Str = i1.toString().toList()  val i2Str = i2.toString().toMutableList()   if (i1Str.size != i2Str.size) {  return false  }   i1Str.forEach {  // under the hood 'indexOf' iterates through entire list (it's like nested loop)  val index = i2Str.indexOf(it)   if (index == -1) {  return false  }   i2Str.removeAt(index)  }   return true  } } fun main() {  println(Solution1.equalDigitFrequency(123, 321)) // true  println(Solution1.equalDigitFrequency(123, 312)) // true  println(Solution1.equalDigitFrequency(4557, 745)) // false } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P48 Get duplicated arguments | <chare> | 1 | <pext> | <starting>P48 Get duplicated arguments</heading> Given variable number of arguments (list of strings) checks whether there are any duplicates among the arguments and return list of all unique duplicates. If no arguments are passes return empty list. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity: O(n) // Determine the frequency of each argument and then filter arguments with frequency > 1 object Solution1 {  fun getDuplicatedArguments(vararg strings: String): List<String>? = strings  .groupingBy { it }  .eachCount()  .filter { it.value != 1 }  .map { it.key } }  // Time complexity: O(n) // Optimal solution using double pointer. private object Solution2 {  fun getDuplicatedArguments(vararg strings: String): List<String>? {  var pointer1 = 0  var pointer2 = 1  val result = mutableSetOf<String>()  val sortedList = strings.sorted()   for (it in 0 until sortedList.lastIndex) {  if (sortedList[pointer1] == sortedList[pointer2]) {  result.add(sortedList[pointer1])  }   pointer1++  pointer2++  }   return result.toList()  } }  // Time complexity: O(n^2) // Loop through all arguments and compare given argument with all other arguments. private object Solution3 {  fun getDuplicatedArguments(vararg strings: String): List<String>? {  val result = mutableSetOf<String>()   strings.forEachIndexed { index, element1 ->  (index + 1..strings.lastIndex).forEach { index2 ->  if (element1 == strings[index2]) {  result.add(element1)  }  }  }   return result.toList()  } } fun main() {  println(Solution1.getDuplicatedArguments("a", "b", "c")) //empty list  println(Solution1.getDuplicatedArguments("a", "b", "c", "a")) // [a] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P49 Find the pair with average | <chare> | 1 | <pext> | <starting>P49 Find the pair with average</heading> Given sorted list of integers and target average implement a function which determine if there is a pair of values in the list where their average equals target average. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity: O(n) // Optimal solution using double pointer. object Solution1 {  fun hasAverage(list: List<Int>, average: Double): Boolean {  if (list.size < 2) {  return false  }   var pointer1 = 0  var pointer2 = list.lastIndex   while (pointer1 != pointer2) {  val pairAvg: Double = (list[pointer1].toDouble() + list[pointer2].toDouble()) / 2   when {  pairAvg == average -> return true  pairAvg < average -> pointer1++  pairAvg > average -> pointer2--  }  }   return false  } } fun main() {  println(Solution1.hasAverage(listOf(), 1.0)) // false  println(Solution1.hasAverage(listOf(3, 4, 7, 9), 6.5)) // true  println(Solution1.hasAverage(listOf(3, 4, 7, 9), 3.0)) // false  println(Solution1.hasAverage(listOf(3, 5, 9, 7, 11, 14), 8.0)) // true } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P50 Binary search tree | <chare> | 1 | <pext> | <starting>P50 Binary search tree</heading> Implement binary search tree (BST). \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  class BinarySearchTree<E : Comparable<E>> {  var root: BinaryNode<E>? = null  private set   fun add(element: E) {  val newNode = BinaryNode(element)   if (root == null) {  root = newNode  return  }   var current: BinaryNode<E> = root ?: return   while (true) {  when {  current.data == element -> {  return  }  element < current.data -> {  if (current.left == null) {  current.left = newNode  return  }   current.left?.let { current = it }  }  element > current.data -> {  if (current.right == null) {  current.right = newNode  return  }   current.right?.let { current = it }  }  }  }  }   fun contains(element: E): Boolean {  var current = root   while (true) {  if (current == null) {  break  } else if (current.data == element) {  return true  } else if (element < current.data) {  current = current.left  } else if (element > current.data) {  current = current.right  }  }   return false  }   fun isEmpty() = root == null  }   data class BinaryNode<E : Comparable<E>>(  val data: E,  var left: BinaryNode<E>? = null,  var right: BinaryNode<E>? = null  ) } fun main() {  val bst = Solution1.BinarySearchTree<Int>()  bst.add(6)  bst.add(3)  bst.add(1)  bst.add(8) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P51 Tree level width | <chare> | 1 | <pext> | <starting>P51 Tree level width</heading> Given the root node of a tree, return an list where each element is the width of the tree at each level. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun levelWidth(tree: Node): List<Int> {  val result = mutableListOf<Int>()   val nodes = mutableListOf(tree)  var levelCurrentNode = 0  var levelTotalNodes = 1  var nextLevelTotalNodes = 0   while (nodes.isNotEmpty()) {  val node = nodes.removeAt(0)  levelCurrentNode++  nodes.addAll(node.children)  nextLevelTotalNodes += node.children.size   if (levelCurrentNode == levelTotalNodes) {  result.add(levelTotalNodes)  levelCurrentNode = 0  levelTotalNodes = nextLevelTotalNodes  nextLevelTotalNodes = 0  }  }   return result  }   private class Node(val data: String, val children: MutableList<Node> = mutableListOf()) {  fun add(data: String) {  add(Node(data))  }   fun add(node: Node) {  children.add(node)  }   fun remove(data: String) {  children.removeAll { it.data == data }  }  } }  private object Solution2 {  // Counters array solution  private fun levelWidth(tree: Node): List<Int> {  val rowSeparator = null  val counters = mutableListOf<Int>()  val nodes = mutableListOf<Node?>()  nodes.add(tree)  nodes.add(rowSeparator)  counters.add(0)   while (nodes.size >= 2) { // 2 means node and separator  val node = nodes.removeAt(0)   if (node == rowSeparator) { // end of row  nodes.add(rowSeparator)  counters.add(0)  } else {  nodes.addAll(node.children)  counters[counters.lastIndex]++  }  }   return counters  }   private class Node(val data: String, val children: MutableList<Node> = mutableListOf()) {  fun add(data: String) {  add(Node(data))  }   fun add(node: Node) {  children.add(node)  }   fun remove(data: String) {  children.removeAll { it.data == data }  }  } } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P52 Binary Search Tree (insert) | <chare> | 1 | <pext> | <starting>P52 Binary Search Tree (insert)</heading> Having binary node (data, left, right) we need to implement two methods binary search tree methods:  insert - accepts an argument data, then create an insert a new node at the appropriate location in the tree. contains - accepts a data argument and return the true if node with given value already exists in the tree, otherwise returns false Requirements that are always true for any given node in Binary Search Tree:  parent node value is always greater then value of all nodes in the left subtree and less than value of all nodes in the right subtree left node value is also a valid BST right node value is also a valid BST \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  data class Node<E : Comparable<E>>(  var data: E,  var left: Node<E>? = null,  var right: Node<E>? = null  ) {  fun insert(e: E) {  if (e < data) { // left node  if (left == null) {  left = Node(e)  } else {  left?.insert(e)  }  } else if (e > data) { // right node  if (right == null) {  right = Node(e)  } else {  right?.insert(e)  }  }  }   fun contains(e: E): Boolean = when {  e == data -> true  e < data -> left?.contains(e) ?: false  e > data -> right?.contains(e) ?: false  else -> false  }  } } fun main() {  val tree = Solution1.Node(1)  tree.insert(4)  tree.insert(2)  println(tree.contains(4)) // true  println(tree.contains(99)) // false } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P53 Binary Tree (validate) | <chare> | 1 | <pext> | <starting>P53 Binary Tree (validate)</heading> Given a node, validate the binary search tree, ensuring that every node's left hand child is less than the parent node's value, and that every node's right hand child is greater than the parent Requirements that are always true for any given node in Binary Search Tree: parent node value is always greater then value of all nodes in the left subtree and less than value of all nodes in the right subtree left node value is also a valid BST right node value is also a valid BST challenge | solution \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P54 Binary Tree (validate) | <chare> | 1 | <pext> | <starting>P54 Binary Tree (validate)</heading> Given a node, validate the binary search tree, ensuring that every node's left hand child is less than the parent node's value, and that every node's right hand child is greater than the parent Requirements that are always true for any given node in Binary Search Tree: parent node value is always greater then value of all nodes in the left subtree and less than value of all nodes in the right subtree left node value is also a valid BST right node value is also a valid BST \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun isValidSearchBinaryTree(node: Node<Int>, min: Int? = null, max: Int? = null): Boolean {  if (min != null && node.data < min) {  return false  }  if (max != null && node.data > max) {  return false  }   return node.left?.let { isValidSearchBinaryTree(it, min, node.data) } ?: true &&  node.right?.let { isValidSearchBinaryTree(it, node.data, max) } ?: true  }   data class Node<E : Comparable<E>>(  var data: E,  var left: Node<E>? = null,  var right: Node<E>? = null  ) {  fun insert(e: E) {  if (e < data) { // left node  if (left == null) {  left = Node(e)  } else {  left?.insert(e)  }  } else if (e > data) { // right node  if (right == null) {  right = Node(e)  } else {  right?.insert(e)  }  }  }  } } fun main() {  val tree = Solution1.Node(1)  tree.insert(4)  tree.insert(2)  println(Solution1.isValidSearchBinaryTree(tree)) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P55 Any callback | <chare> | 1 | <pext> | <starting>P55 Any callback</heading> Given list of integers and callback implement a recursive function which returns true if simple value in the list passed to callback returns true, otherwise return false. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun anyCallback(list: List<Int>, callback: (Int) -> Boolean): Boolean {  if (list.size == 1) {  return callback(list.first())  }   return callback(list.first()) || anyCallback(list.drop(1), callback) } fun main() {  val callback1:((Int) -> Boolean) = { it > 3 }  println(anyCallback(listOf(1, 2, 3, 4), callback1))//shouldBeEqualTo true  val callback2:((Int) -> Boolean) = { it > 3 }  println(anyCallback(listOf(1, 2, 3), callback2))// shouldBeEqualTo false } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P56 Binary search | <chare> | 1 | <pext> | <starting>P56 Binary search</heading> Given list of sorted elements and a element return index of that element in the list or -1 if element was not found. Use binary search algorithm. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity (Best): Ω(1) // Time complexity (Average): Θ(log(n)) // Time complexity(Worst): O(log(n)) object Solution1 {  fun binarySearch(list: List<Char>, element: Char): Int {  var left = 0  var right = list.size - 1   while (left <= right) {  val middle = (right + left) / 2   // Check if x is present at mid  if (list[middle] == element)  return middle   if (list[middle] < element)  left = middle + 1 // x greater than middle, so ignore left half  else  right = middle - 1 // x greater than middle, so , ignore right half  }   // if we reach here, then element was  // not present  return -1  } } fun main() {  println(Solution1.binarySearch(listOf('A', 'B', 'C'), 'A')) // 0  println(Solution1.binarySearch(listOf('A', 'B', 'C'), 'B')) // 1  println(Solution1.binarySearch(listOf('A', 'B', 'C'), 'H')) // -1 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P57 Surrounded letter | <chare> | 1 | <pext> | <starting>P57 Surrounded letter</heading> Given a string containing letters and + characters implement a function which determines if each letter in the string is surrounded by + character. There may be more than one + character between letters (+a++b+) and letters may be surrounded by the same + character (+a+b+). \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Regex solution object Solution1 {  fun surroundedLetter(str: String): Boolean {  val pattern = Regex("(?=(\\+[a-z]\\+))")  .findAll(str)  .count()   val letters = str.count { it.isLetter() }   return pattern == letters  } }  // Iterative solution object Solution2 {  fun surroundedLetter(str: String): Boolean {  if (str.length < 3) {  return false  }   if (str.first().isLetter() || str.last().isLetter()) {  return false  }   var previousCharacter = str.first()   str.drop(1).forEach { currentCharacter ->  if (currentCharacter.isLetter() && previousCharacter.isLetter()) {  return false  }   previousCharacter = currentCharacter  }   return true  } } fun main() {  println(Solution1.surroundedLetter("+a+")) // true  println(Solution1.surroundedLetter("+ab+")) // false  println(Solution1.surroundedLetter("+a+b+")) // true  println(Solution1.surroundedLetter("+a++b++")) // true } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P58 Insertion sort | <chare> | 1 | <pext> | <starting>P58 Insertion sort</heading> Sort list of numbers from lowest number to greatest number using insertion sort. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity (Best): Ω(n) - all data besides one element are sorted // Time complexity (Average): Θ(n^2) // Time complexity (Worst): O(n^2) // Space complexity: O(1) object Solution1 {  fun insertionSort(list: MutableList<Int>): List<Int> {  (1..list.lastIndex).forEach { i ->  var j = i - 1   while (j >= 0 && list[j] > list[i]) {  j--  }   list.removeAt(i).also {  list.add(j + 1, it)  }  }   return list  } }  fun main() {  println(Solution1.insertionSort(mutableListOf(4,5,2,6,4,56))) //[2, 4, 4, 5, 6, 56] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P59 Quick sort | <chare> | 1 | <pext> | <starting>P59 Quick sort</heading> Sort list of numbers from lowest number to greatest number using quick sort. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity (Best): Ω(n log(n)) // Time complexity (Average): Θ(l log(n)) // Time complexity (Worst): O(n^2) // Space complexity: O(log(n)) object Solution1 {  fun quickSort(list: MutableList<Int>, left: Int = 0, right: Int = list.lastIndex): List<Number> {  // Rearrange elements and returns correct index of the pivot  // All elements smaller than element will be on the left side of the array (smaller indexes)  // All elements larger than element will be on the left side of the array (lager indexes)  fun pivot(list: MutableList<Int>, start: Int = 0, end: Int = list.lastIndex): Int {  val pivot = list[start] // We decide that pivot is our first element (it can be any element)  var swapIndex = start // first index that we can swap (number of element that are less than pivot)   (start + 1..end).forEach {  if (pivot > list[it]) {  swapIndex++  list.swap(it, swapIndex)  }  }   list.swap(start, swapIndex)   return swapIndex  }   if (left < right) {  val pivotIndex = pivot(list, left, right)   // Sort left part  quickSort(list, left, pivotIndex - 1)   // Sort right part  quickSort(list, pivotIndex + 1, right)  }   return list  }   private fun <T> MutableList<T>.swap(index1: Int, index2: Int) {  val tmp = this[index1]  this[index1] = this[index2]  this[index2] = tmp  } }  // Time complexity (Best): Ω(n^2) // Time complexity (Average): Θ(n^2) // Time complexity (Worst): O(n^2) // Space complexity: O(n) private object Solution2 {  private fun quickSort(list: MutableList<Int>): List<Number> {  if (list.isEmpty()) {  return list  }   val pivot = list.first() // We selected first element as pivot (it can be any element)  var pivotIndex = 0 // first index that we can swap (number of element that are less than pivot)   (0..list.lastIndex).forEach {  if (pivot > list[it]) {  list.swap(it, pivotIndex + 1)  pivotIndex++  }  }   // Move element to the correct index  // All elements smaller than element will be on the left side of the array (smaller indexes)  // All elements larger than element will be on the left side of the array (lager indexes)  list.swap(0, pivotIndex)   // Create left sub-list  val left = list.subList(0, pivotIndex)   // Create right sub-list  val right = list.subList(pivotIndex + 1, list.size)   return quickSort(left) + listOf(pivot) + quickSort(right)  }   private fun <T> MutableList<T>.swap(index1: Int, index2: Int) {  val tmp = this[index1]  this[index1] = this[index2]  this[index2] = tmp  } }  fun main() {  println(Solution1.quickSort(mutableListOf(9,5,2,6,4,8))) //[2, 4, 5, 6, 8, 9] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P60 Radix sort | <chare> | 1 | <pext> | <starting>P60 Radix sort</heading> Sort list of numbers from lowest number to greatest number using radix sort. Algorithm needs to support only positive integers. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity (Best): Ω(n^2) // Time complexity (Average): Θ(n^2) // Time complexity (Worst): O(n^2) // Space complexity: O(1) object Solution1 {  fun radixSort(list: List<Int>): List<Number> {  // create temp list  val tempList = list.toMutableList()   // got through the list as many times as there are digits in the longest number  val maxDigits = maxDigits(list)   (0 until maxDigits).forEach { digitRightIndex ->  // crate 10 buckets  val buckets = mutableMapOf<Char, MutableList<Int>>()  ('0'..'9').forEach { buckets[it] = mutableListOf() }   // add numbers to the buckets  tempList.forEach {  val digit = it.getDigitAt(digitRightIndex)  buckets[digit]?.add(it)  }   // clean temp list (all numbers are in buckets by now)  tempList.clear()   // take numbers from buckets and add it to the temp list  ('0'..'9').forEach { digit ->  buckets[digit]?.let { tempList.addAll(it) }  }  }   return tempList  }   private fun Int.getDigitAt(index: Int): Char {  val str = this.toString()  val rightIndex = str.lastIndex - index  return str.getOrElse(rightIndex) { '0' }  }   // This is mathematical alternative to this.toString().length witch has better performance  private val Int.digitCount: Int  get() = when (this) {  0 -> 1  else -> Math.log10(Math.abs(this.toDouble())).toInt() + 1  }   private fun maxDigits(list: List<Int>) = list.map { it.digitCount }.maxOrNull() ?: 0 }  fun main() {  println(Solution1.radixSort(listOf(170, 45, 75, 90, 2, 802, 2, 66))) //[2, 2, 45, 66, 75, 90, 170, 802] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P61 Doubly linked list | <chare> | 1 | <pext> | <starting>P61 Doubly linked list</heading> Implement doubly linked list data structure. This is quite big challenge, so we will split it into multiple multiple methods and properties that we will implement one my one.  We also want to handle various edge cases, because we are assuming that developer using our class many do certain mistakes. That's why each step has one or more tests associated with it. Tests are commented out default, so before staring each step we will have to enable all tests related to given step by uncommenting it. To quickly uncomment the test select all lines containing test method and press Cmd + / keys (Comment with line Comment action). \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  private class DoublyLinkedList<E> : Iterable<Node<E>> {  var head: Node<E>? = null   var first: Node<E>? = null  get() = head   val last: Node<E>?  get() {  var node = head   while (node?.next != null) {  node = node.next  }   return node  }   val size: Int  get() {  var node = head  var count = 0   while (node != null) {  count++  node = node.next  }   return count  }   fun clear() {  head = null  }   fun insertFirst(e: E) {  insertAt(e, 0)  }   fun removeFirst() {  head = head?.next  head?.prev = null  }   override fun iterator() = object : Iterator<Node<E>> {  var node = head  override fun hasNext() = node != null   override fun next(): Node<E> {  val currentNode = node  node = node?.next  return currentNode!!  }  }   fun removeLast() {  if (head?.next == null) {  head = null  } else {  last?.prev?.next = null  }  }   fun insertLast(data: E) {  var node = head   while (node?.next != null) {  node = node.next  }   val newNode = Node(data, node)   if (node == null) {  head = newNode  } else {  node.next = newNode  }  }   fun insertAt(data: E, index: Int) {  val newNode = Node(data)   if (head == null || index == 0) {  newNode.next = head  head?.prev = newNode  head = newNode  return  }   var count = 0  var node = head  var prevNode: Node<E>? = null   while (node != null) {  if (count == index) {  break  }   count++  prevNode = node  node = node.next  }   val nextNode = prevNode?.next  prevNode?.next = newNode  newNode.prev = prevNode  newNode.next = nextNode  nextNode?.prev = newNode  }   fun getAt(index: Int): Node<E>? {  var node = head  var count = 0   while (node != null) {  if (count == index) {  return node  }   node = node.next  count++  }   return null  }   fun setAt(data: E, index: Int) {  getAt(index)?.data = data  }   fun removeAt(index: Int) {  val node = getAt(index) ?: return  val prevNode = node.prev  val nextNode = node.next   if (prevNode == null && nextNode == null) {  head = null  } else if (prevNode != null && nextNode != null) {  prevNode.next = nextNode  nextNode.prev = prevNode  } else if (prevNode == null && nextNode != null) {  head = nextNode  nextNode.prev = null  node.next = null  } else if (prevNode != null && nextNode == null) {  prevNode.next = null  node.prev = null  }  }   operator fun plus(linkedList: DoublyLinkedList<E>): DoublyLinkedList<E> {  val result = DoublyLinkedList<E>()  forEach { result.insertLast(it.data) }  linkedList.forEach { result.insertLast(it.data) }  return result  }  }   private data class Node<T>(  var data: T,  var prev: Node<T>? = null,  var next: Node<T>? = null  ) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P62 Singly linked list, Linked list midpoint, Circular check, Linked list take n-th element from last | <chare> | 1 | <pext> | <starting>P62 Singly linked list, Linked list midpoint, Circular check, Linked list take n-th element from last</heading> Implement singly linked list data structure. This is quite big challenge, so we will split it into multiple multiple methods and properties that we will implement one my one.  We also want to handle various edge cases, because we are assuming that developer using our class many do certain mistakes. That's why each step has one or more tests associated with it. Tests are commented out default, so before staring each step we will have to enable all tests related to given step by uncommenting it. To quickly uncomment the test select all lines containing test method and press Cmd + / keys (Comment with line Comment action). \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  fun fromLast(list: SinglyLinkedList<Char>, endIndex: Int): Node<Char>? {  var slow = list.first  var fast = list.first   repeat(endIndex) { fast = fast?.next }   while (fast?.next != null) {  slow = slow?.next  fast = fast?.next  }   return slow  }   fun circularCheck(list: SinglyLinkedList<Char>): Boolean {  var slowPointer = list.first  var fastPointer = list.first   while (fastPointer != null) {  list.forEach {  slowPointer = slowPointer?.next  fastPointer = fastPointer?.next?.next   if (fastPointer != null && slowPointer == fastPointer) {  return true  }  }  }   return false  }  fun midpoint(list: SinglyLinkedList<Char>): Node<Char>? {  var slowPointer = list.first  var fastPointer = list.first   list.forEach {  if (fastPointer?.next?.next == null) {  return slowPointer  }  slowPointer = slowPointer?.next  fastPointer = fastPointer?.next?.next  }   return null  }  class SinglyLinkedList<E> : Iterable<Node<E>> {  var head: Node<E>? = null   val first: Node<E>?  get() = getAt(0)   val last: Node<E>?  get() = getAt(size - 1)   val size: Int  get() {  var count = 0  var node = head   while (node != null) {  count++  node = node.next  }   return count  }   fun insertFirst(data: E) {  insertAt(data, 0)  }   fun insertLast(data: E) {  insertAt(data, size)  }   fun insertAt(data: E, index: Int) {  if (index == 0) {  head = Node(data, head)  } else {  val prevNode = getAt(index - 1) ?: last  val node = prevNode?.next  prevNode?.next = Node(data, node)  }  }   fun removeFirst() {  removeAt(0)  }   fun removeLast() {  removeAt(size - 1)  }   fun removeAt(index: Int) {  if (index == 0) {  head = head?.next  } else {  val prevNode = getAt(index - 1)  val nextNode = prevNode?.next?.next  prevNode?.next = nextNode  }  }   fun setAt(data: E, index: Int) {  getAt(index)?.data = data  }   fun getAt(index: Int): Node<E>? {  if (head == null) {  return null  }   var node = head  var count = 0   while (node != null) {  if (count == index) {  return node  }   count++  node = node.next  }   return null  }   fun clear() {  head = null  }   override fun toString() = head.toString()   override fun iterator() = object : Iterator<Node<E>> {  var node = head  override fun hasNext() = node != null   override fun next(): Node<E> {  val currentNode = node  node = node?.next  return currentNode!!  }  }   operator fun plus(linkedList: SinglyLinkedList<E>): SinglyLinkedList<E> {  val result = SinglyLinkedList<E>()  forEach { result.insertLast(it.data) }  linkedList.forEach { result.insertLast(it.data) }  return result  }  }   data class Node<T>(  var data: T,  var next: Node<T>? = null  )  fun main() {  var l = SinglyLinkedList<Char> ()  val a = Node ('a')  val b = Node ('b')  val c = Node ('c')  l.head = a  a.next = b  b.next = c  c.next = b  println(circularCheck(l)) // true    l = SinglyLinkedList<Char> ()  l.insertLast('a')  l.insertLast('b')  l.insertLast('c')  l.insertLast('d')  println(midpoint(l)) // 'b'    l = SinglyLinkedList<Char>()  l.insertLast('a')  l.insertLast('b')  l.insertLast('c')  l.insertLast('d')  l.insertLast('e')  println(fromLast(l, 0)?.data) // shouldBeEqualTo 'e'  println(fromLast(l, 3)?.data) // shouldBeEqualTo 'b' } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P63 Max binary heap | <chare> | 1 | <pext> | <starting>P63 Max binary heap</heading> Implement Max binary heap. In the Max Binary Heap each node may have from 0 to 2 children. Parent nodes are always larger than child nodes, in other words all children are always smaller than a parents. Heap always have to be filled from "left side" meaning that we can add children to new level only if current level is full (each parent from previous level have two children). \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  class MaxBinaryHeap<E : Comparable<E>> {  val items = mutableListOf<E>()   fun add(element: E) {  items.add(element)  bubbleUpLastLeaf()  }   fun removeMax(): E? {   if (items.isEmpty()) return null   if (items.size == 1) {  return items.removeAt(0)  }   items.swap(0, items.lastIndex)   val max = items.removeAt(items.lastIndex)   if (items.size > 1) {  bubbleDownRootElement()  }   return max  }   private fun bubbleDownRootElement() {  var elementIndex = 0  var leftChildIndex = getLeftChildIndex(elementIndex)  var rightChildIndex = getRightChildIndex(elementIndex)   // Only two elements  if (items.getOrNull(rightChildIndex) == null) {  if (items[elementIndex] < items[leftChildIndex]) {  items.swap(elementIndex, leftChildIndex)  }  return  }   while ((items[elementIndex] < items[leftChildIndex]) ||  items[elementIndex] < items[rightChildIndex]  ) {   elementIndex = if (rightChildIndex > items.lastIndex ||  items[leftChildIndex] > items[rightChildIndex]  ) {  items.swap(leftChildIndex, elementIndex)  leftChildIndex  } else {  items.swap(rightChildIndex, elementIndex)  rightChildIndex  }   leftChildIndex = getLeftChildIndex(elementIndex)  rightChildIndex = getRightChildIndex(elementIndex)  }  }   private fun bubbleUpLastLeaf() {  var elementIndex = items.lastIndex  var parentIndex = getParentIndex(elementIndex)   while (items[elementIndex] > items[parentIndex]) {  items.swap(elementIndex, parentIndex)   elementIndex = parentIndex  parentIndex = getParentIndex(elementIndex)  }  }   private fun getParentIndex(index: Int) = (index - 1) / 2   private fun getLeftChildIndex(index: Int) = (index \* 2) + 1   private fun getRightChildIndex(index: Int) = (index \* 2) + 2   fun isEmpty() = items.isEmpty()   private fun <T> MutableList<T>.swap(index1: Int, index2: Int) {  val tmp = this[index1]  this[index1] = this[index2]  this[index2] = tmp  }  } } fun main() {  val heap = Solution1.MaxBinaryHeap<Int>()  heap.add(9)  heap.add(7)  heap.add(6)  heap.add(3)  heap.add(2)  heap.add(4) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // ----------Heap------------ // // 9 // / \ // 7 6 // / \ / // 3 2 4 // // -------------------------- </ending> | </end> |
| <hitle> | P64 List chunking | <chare> | 1 | <pext> | <starting>P64 List chunking</heading> Given a list and chunk size, divide the list into multiple sub lists where each sub-list is of length of chunk size. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  object Solution1 {  fun chunk(list: List<Int>, size: Int): List<List<Int>> {  val chunks = mutableListOf<MutableList<Int>>()   list.forEach {  if (chunks.size == 0 || chunks.last().size == size) {  chunks.add(mutableListOf(it))  } else {  chunks.last().add(it)  }  }   return chunks  } }  private object Solution2 {  private fun chunk(list: List<Int>, size: Int): List<List<Int>> {  val chunks = mutableListOf<List<Int>>()   for (i in 0..list.lastIndex step size) {  var rightIndex = if (i + size < list.size) i + size else list.size  chunks.add(list.subList(i, rightIndex))  }   return chunks  } }  private object Solution3 {  private fun chunk(list: List<Int>, size: Int): List<List<Int>> {  var index = 0  val chunks = mutableListOf<List<Int>>()   while (index <= list.lastIndex) {  var rightIndex = if (index + size < list.size) index + size else list.size  chunks.add(list.subList(index, rightIndex))   index += size  }   return chunks  } } fun main() {  println(Solution1.chunk(listOf(1, 2, 3, 4), 2))// --> [[ 1, 2], [3, 4]]  println(Solution1.chunk(listOf(1, 2, 3, 4, 5), 2))// --> [[ 1, 2], [3, 4], [5]]  println(Solution1.chunk(listOf(1, 2, 3, 4, 5, 6, 7, 8), 3))// --> [[ 1, 2, 3], [4, 5, 6], [7, 8]]  println(Solution1.chunk(listOf(1, 2, 3, 4, 5), 4))// --> [[ 1, 2, 3, 4], [5]]  println(Solution1.chunk(listOf(1, 2, 3, 4, 5), 10))// --> [[ 1, 2, 3, 4, 5]] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P65 Is substring | <chare> | 1 | <pext> | <starting>P65 Is substring</heading> Given two strings implement a function which determines whether the characters in the second string is a substring of the characters in the first string (check if second string exists as continuous/unbroken chain of characters the first string). \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Kotlin Idiomatic Approach object Solution1 {  fun isSubstring(str: String, subStr: String): Boolean {  return str.contains(subStr) && str.isNotEmpty() && subStr.isNotEmpty()  } }  // Time complexity: O(n\*m) // Space complexity: O(1) // // Optimal solution using double pointer. private object Solution2 {  private fun isSubstring(str: String, subStr: String): Boolean {  if (str.isEmpty() || subStr.isEmpty()) return false  if (str.length < subStr.length) return false   var pointer1 = 0  var pointer2 = 0   while (pointer1 <= str.lastIndex) {  if (str[pointer1] == subStr[pointer2]) {  pointer1++  pointer2++  if (pointer2 == subStr.length) {  return true  }  } else {  pointer1 = pointer1 - pointer2 + 1  pointer2 = 0  }  }   return false  } }  // Time complexity: O(n\*m) // Space complexity: ??, but more than O(1) // Number of iterations (n) is bounded by the length of the first string // and String.drop requires copying the entire remaining string (on it's own it has O(m) complexity) // First of 5 chars, needs 5 iterations at most and 15 character copied (5+4+3+2+1=15). Second is copied less often. // // Recursive solution private object Solution3 {  private fun isSubstring(str: String, subStr: String): Boolean {  fun isExactMatch(str: String, subStr: String): Boolean {  if (subStr.length > str.length) {  return false  }   return when {  str.isEmpty() && subStr.isEmpty() -> true  str.isNotEmpty() && subStr.isEmpty() -> true  else -> str[0] == subStr[0] && isExactMatch(str.drop(1), subStr.drop(1))  }  }   if (subStr.length > str.length) {  return false  }  if (str.isEmpty() || subStr.isEmpty()) {  return false  }   return isExactMatch(str, subStr) || isSubstring(str.drop(1), subStr)  } }  // Time complexity: O(n\*m) // Space complexity: O(1) // This recursive solution is faster than solution with String.drop because it uses double pointer // // Recursive solution private fun isSubstring(str: String, subStr: String): Boolean {  if (str.isEmpty() || subStr.isEmpty()) {  return false  }   fun helper(first: String, second: String, firstPointer1: Int = 0, secondPointer2: Int = 0): Boolean {  if (firstPointer1 > first.lastIndex) {  return false  }   return if (first[firstPointer1] == second[secondPointer2]) {  val localPointer1 = firstPointer1 + 1  val localPointer2 = secondPointer2 + 1   when {  localPointer1 <= first.lastIndex && localPointer2 <= second.lastIndex -> {  helper(first, second, localPointer1, localPointer2)  }  localPointer2 <= second.lastIndex && localPointer1 > first.lastIndex -> false  else -> true  }  } else {  val p1 = firstPointer1 - secondPointer2 + 1   if (p1 > first.lastIndex) {  return false  } else {  helper(first, second, p1, 0)  }  }  }   return helper(str, subStr) }  fun main() {  println(Solution1.isSubstring("go home", "ome")) // true  println(Solution1.isSubstring("go home", "me")) // true  println(Solution1.isSubstring("go home", "abc")) // false } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P66 Merge sort | <chare> | 1 | <pext> | <starting>P66 Merge sort</heading> Sort list of numbers from lowest number to greatest number using merge sort. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Time complexity (Best): Ω(n log(n)) // Time complexity (Average): Θ(n log(n)) // Time complexity (Worst): O(n log(n)) // Space complexity: O(n) object Solution1 {  fun mergeSort(list: List<Int>): List<Int> {  if (list.size <= 1) return list   // Take two sorted lists and merge them together into one sorted list  fun merge(left: MutableList<Int>, right: MutableList<Int>): MutableList<Int> {  val result = mutableListOf<Int>()   while (left.isNotEmpty() && right.isNotEmpty()) {  if (left.first() < right.first()) {  result.add(left.removeAt(0))  } else {  result.add(right.removeAt(0))  }  }   result.addAll(left)  result.addAll(right)   return result  }   val mid = list.size / 2  val left = mergeSort(list.subList(0, mid)).toMutableList()  val right = mergeSort(list.subList(mid, list.size)).toMutableList()  return merge(left, right)  } }  fun main() {  println(Solution1.mergeSort(listOf(5, 1, 4, 2))) //[1, 2, 4, 5] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  [5, 1, 4, 2] Split  [5, 1] [4, 2] Split  [5] [1] [4] [2] All lists are sorted  [1, 5] [2,4] Merge  [1, 2, 4, 5] Merge </ending> | </end> |
| <hitle> | P67 Recursive cache fibonacci | <chare> | 1 | <pext> | <starting>P67 Recursive cache fibonacci</heading> Print out the n-th entry in the fibonacciSequence series. The fibonacci series is an ordering of numbers where each number is the sum of the preceding two. For example, the sequence [0, 1, 1, 2, 3, 5, 8, 13, 21, 34] forms the first ten entries of the fibonacci sequence. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  data class MethodCache(val n: Int, val result: Int)   fun fibonacciSequenceRecursiveCached(  n: Int,  methodCache: MutableList<MethodCache> = mutableListOf()  ): Int {  var result = methodCache.firstOrNull { it.n == n }?.result   if (result == null) {  if (n < 2) {  methodCache.add(MethodCache(n, n))  return n  }   result = fibonacciSequenceRecursiveCached(n - 1, methodCache) +  fibonacciSequenceRecursiveCached(n - 2, methodCache)  methodCache.add(MethodCache(n, result))  }   return result  } }  fun main() {  for(i in 1..10) {  println(Solution1.fibonacciSequenceRecursiveCached(i))  } } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P68 Max sub-list sum | <chare> | 1 | <pext> | <starting>P68 Max sub-list sum</heading> Given list of integers and integer n implement a function which calculates the maximum sum of n consecutive elements in the list (sum of n digits next to another that have the largest sum).  You can use helper max function to deal with Kotlin nullability. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* import kotlin.math.max  // Time Complexity: O(n) // Space Complexity: O(1) // Use "sliding window" - store sum in single variable and with each iteration add (current item) // and remove (first item before current sub-list) object Solution1 {  fun maxSubListSum(list: List<Int>, numElements: Int): Int? {  if (list.size < numElements) {  return null  }   var maxSum = list.take(numElements).sum()  var tempSum = maxSum   (numElements..list.lastIndex).forEach { index ->  tempSum = tempSum - list[index - numElements] + list[index]  maxSum = max(maxSum, tempSum)  }   return maxSum  } }  private object Solution2 {  private fun maxSubListSum(list: List<Int>, numElements: Int): Int? {  if (list.size < numElements) {  return null  }   return list.foldIndexed(0 to 0) { i, (sum, max), next ->  (sum + next - (list.getOrNull(i - numElements) ?: 0)).let {  it to if (it > max) it else max  }  }.second  } }  // Time Complexity: O(n\*m) // Loop through the list and at each index loop again to calculate sum of sublist (from index to index + n) private object Solution3 {  private fun maxSubListSum(list: List<Int>, numElements: Int): Int? {  if (list.size < numElements) {  return null  }   var maxSum: Int? = null   for (i in 0..list.size - numElements) {  var tempSum: Int? = null   for (j in i until (i + numElements)) {  if (tempSum == null) {  tempSum = list[j]  } else {  tempSum += list[j]  }  }   maxSum = max(maxSum, tempSum)  }   return maxSum  }   private fun max(i1: Int?, i2: Int?): Int? {  return when {  i1 != null && i2 != null -> max(i1, i2)  i1 != null && i2 == null -> i1  i1 == null && i2 != null -> i2  else -> null  }  } }  private object Solution4 {  private fun maxSubListSum(list: List<Int>, numElements: Int): Int? {  if (list.isEmpty()) return null   return (0..list.size - numElements)  .map { i -> list.subList(i, i + numElements).sum() }  .maxOrNull()  } } fun main() {  println(Solution1.maxSubListSum(listOf<Int>(), 3)) // null  println(Solution1.maxSubListSum(listOf(4, 2, 7), 2)) // 9  println(Solution1.maxSubListSum(listOf(4, 2, 7, 5, 8, 9, 5, 1), 3)) // 22 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P69 Min sub-list length | <chare> | 1 | <pext> | <starting>P69 Min sub-list length</heading> Given list of positive integers and positive integer n implement a function which finds minimum length of sub-list where sum all elements is equal or greater than n  You can use helper min function to deal with Kotlin nullability. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* import kotlin.math.max  // Time complexity: O(n) // Space complexity O(n) // Use sliding window object Solution1 {  fun minSubListLength(list: List<Int>, sum: Int): Int {  var total = 0  var start = 0  var end = 0  var minLen: Int? = null   while (start < list.size) {  // if current window doesn't add up to the given numElements then move the window to right  if (total < sum && end < list.size) {  total += list[end]  end++  }  // if current window adds up to at least the numElements given then we can shrink the window  else if (total >= sum) {  minLen = min(minLen, end - start)  total -= list[start]  start++  }  // current total less than required total but we reach the end, need this or else we'll be in an infinite loop  else {  break  }  }   return minLen ?: 0  }   private fun min(i1: Int?, i2: Int?): Int? {  return when {  i1 != null && i2 != null -> Math.min(i1, i2)  i1 != null && i2 == null -> i1  i1 == null && i2 != null -> i2  else -> null  }  } }  // Time complexity: O(n^2) // Loop through all the elements and then loop through all sublists private object Solution2 {  fun minSubListLength(list: List<Int>, sum: Int): Int {  var minListLength: Int? = null   repeat(list.size) { index ->  var subListSum = 0  var numItems = 0  val subList = list.subList(index, list.size)   for (item in subList) {  subListSum += item  numItems++   if (subListSum >= sum) {  minListLength = min(minListLength, numItems)  break  }  }  }   return minListLength ?: 0  }   private fun min(i1: Int?, i2: Int?): Int? {  return when {  i1 != null && i2 != null -> Math.min(i1, i2)  i1 != null && i2 == null -> i1  i1 == null && i2 != null -> i2  else -> null  }  } } fun main() {  println(Solution1.minSubListLength(listOf(1, 3, 4), 5)) // 2 (smallest sub-list [3, 4])  println(Solution1.minSubListLength(listOf(1, 2, 11, 5, 9, 4, 6), 22)) // 3 (smallest sub-list [11, 5, 9])  println(Solution1.minSubListLength(listOf(1, 20, 11, 5, 9, 4, 6), 200)) // 0 (non of the integers sums up to 200) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P70 List subtract | <chare> | 1 | <pext> | <starting>P70 List subtract</heading> Given two lists implement a function which will determine witch elements from first list have to be subtracted (removed) to obtain second list (what elements needs to be removed from first list so it it would be equal to second list). There may be more than one element with the same value. For simplicity we assume that input is always correct (it is always possible to remove values from first list to form second list). \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun getSubtraction(list1: List<String>, list2: List<String>): List<String> {  val frequencyMap1 = list1.groupingBy { it }.eachCount()  val frequencyMap2 = list2.groupingBy { it }.eachCount()  val itemList1 = frequencyMap1.flatMap { entry -> List(entry.value) { entry.key } }.toMutableList()  val itemList2 = frequencyMap2.flatMap { entry -> List(entry.value) { entry.key } }  itemList2.forEach { itemList1.remove(it) }   return itemList1  } }  fun main() {  println(Solution1.getSubtraction(listOf("A", "B", "C"), listOf("A"))) // "B", "C"  println(Solution1.getSubtraction(listOf("A", "B", "C"), listOf("A", "B"))) // "C"  println(Solution1.getSubtraction(listOf("A", "B", "C", "A"), listOf("A", "B"))) // "C", "A"  println(Solution1.getSubtraction(listOf("A", "B", "C"), listOf("A", "B", "C"))) // nothing } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P71 Coins | <chare> | 1 | <pext> | <starting>P71 Coins</heading> Given an amount of money and an array of coins write a method that computes the number of ways to make the amount of money with coins of the available coins. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* object Solution1 {  fun getCoins(amount: Int, coins: List<Int>): Int {  // array of zeros from 0..amount  val waysOfDoingNCents = IntArray(amount + 1)   waysOfDoingNCents[0] = 1   for (coin in coins) {  for (higherAmount in coin..amount) {  val higherAmountRemainder = higherAmount - coin  waysOfDoingNCents[higherAmount] += waysOfDoingNCents[higherAmountRemainder]  }  }   return waysOfDoingNCents[amount]  } }   fun main() {  println(Solution1.getCoins(4, listOf(1, 2, 3))) // 4 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P72 char to ascii toInt() | <chare> | 1 | <pext> | <starting>P72 char to ascii toInt()</heading>  val c = 'a'  val ascii = c.toInt()  println("The ASCII value of $c is: $ascii") \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* The ASCII value of a is: 97 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P73 Roots of Quadratic | <chare> | 1 | <pext> | <starting>P73 Roots of Quadratic</heading> fun main(args: Array<String>) {   val a = 2.3  val b = 4  val c = 5.6  val root1: Double  val root2: Double  val output: String   val determinant = b \* b - 4.0 \* a \* c   // condition for real and different roots  if (determinant > 0) {  root1 = (-b + Math.sqrt(determinant)) / (2 \* a)  root2 = (-b - Math.sqrt(determinant)) / (2 \* a)   output = "root1 = %.2f and root2 = %.2f".format(root1, root2)  }  // Condition for real and equal roots  else if (determinant == 0.0) {  root2 = -b / (2 \* a)  root1 = root2   output = "root1 = root2 = %.2f;".format(root1)  }  // If roots are not real  else {  val realPart = -b / (2 \* a)  val imaginaryPart = Math.sqrt(-determinant) / (2 \* a)   output = "root1 = %.2f+%.2fi and root2 = %.2f-%.2fi".format(realPart, imaginaryPart, realPart, imaginaryPart)  }   println(output) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* root1 = -0.87+1.30i and root2 = -0.87-1.30i \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P74 Check a Leap Year | <chare> | 1 | <pext> | <starting>P74 Check a Leap Year</heading> fun main(args: Array<String>) {   val year = 1900  var leap = false   if (year % 4 == 0) {  if (year % 100 == 0) {  // year is divisible by 400, hence the year is a leap year  leap = year % 400 == 0  } else  leap = true  } else  leap = false   println(if (leap) "$year is a leap year." else "$year is not a leap year.") } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1900 is not a leap year. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P75 GCD ( greatest common divisor) | <chare> | 1 | <pext> | <starting>P75 GCD ( greatest common divisor)</heading> fun main(args: Array<String>) {  val n1 = 81  val n2 = 153  var gcd = 1   var i = 1  while (i <= n1 && i <= n2) {  // Checks if i is factor of both integers  if (n1 % i == 0 && n2 % i == 0)  gcd = i  ++i  }   println("G.C.D of $n1 and $n2 is $gcd") } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* G.C.D of 81 and 153 is 9 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P76 LCM (Least Common Multiple) | <chare> | 1 | <pext> | <starting>P76 LCM (Least Common Multiple)</heading> fun main(args: Array<String>) {  val n1 = 72  val n2 = 120  var lcm: Int   // maximum number between n1 and n2 is stored in lcm  lcm = if (n1 > n2) n1 else n2   // Always true  while (true) {  if (lcm % n1 == 0 && lcm % n2 == 0) {  println("The LCM of $n1 and $n2 is $lcm.")  break  }  ++lcm  } } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* The LCM of 72 and 120 is 360. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P77 Prime Check | <chare> | 1 | <pext> | <starting>P77 Prime Check</heading> fun main(args: Array<String>) {  val num = 29  var flag = false  for (i in 2..num / 2) {  // condition for nonprime number  if (num % i == 0) {  flag = true  break  }  }   if (!flag)  println("$num is a prime number.")  else  println("$num is not a prime number.") } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 29 is a prime number. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P78 Add Two Matrix | <chare> | 1 | <pext> | <starting>P78 Add Two Matrix</heading> fun main(args: Array<String>) {  val rows = 2  val columns = 3  val firstMatrix = arrayOf(intArrayOf(2, 3, 4), intArrayOf(5, 2, 3))  val secondMatrix = arrayOf(intArrayOf(-4, 5, 3), intArrayOf(5, 6, 3))   // Adding Two matrices  val sum = Array(rows) { IntArray(columns) }  for (i in 0..rows - 1) {  for (j in 0..columns - 1) {  sum[i][j] = firstMatrix[i][j] + secondMatrix[i][j]  }  }   // Displaying the result  println("Sum of two matrices is: ")  for (row in sum) {  for (column in row) {  print("$column ")  }  println()  } } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Sum of two matrices is: -2 8 7  10 8 6  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P79 Multiply Two Matrices | <chare> | 1 | <pext> | <starting>P79 Multiply Two Matrices</heading> fun main(args: Array<String>) {  val r1 = 2  val c1 = 3  val r2 = 3  val c2 = 2  val firstMatrix = arrayOf(intArrayOf(3, -2, 5), intArrayOf(3, 0, 4))  val secondMatrix = arrayOf(intArrayOf(2, 3), intArrayOf(-9, 0), intArrayOf(0, 4))   // Mutliplying Two matrices  val product = Array(r1) { IntArray(c2) }  for (i in 0..r1 - 1) {  for (j in 0..c2 - 1) {  for (k in 0..c1 - 1) {  product[i][j] += firstMatrix[i][k] \* secondMatrix[k][j]  }  }  }   // Displaying the result  println("Product of two matrices is: ")  for (row in product) {  for (column in row) {  print("$column ")  }  println()  } } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Product of two matrices is: 24 29  6 25  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P80 Add Two Complex Numbers | <chare> | 1 | <pext> | <starting>P80 Add Two Complex Numbers</heading> class Complex(internal var real: Double, internal var imag: Double)  fun main(args: Array<String>) {  val n1 = Complex(2.3, 4.5)  val n2 = Complex(3.4, 5.0)  val temp: Complex   temp = add(n1, n2)   System.out.printf("Sum = %.1f + %.1fi", temp.real, temp.imag) }  fun add(n1: Complex, n2: Complex): Complex {  val temp = Complex(0.0, 0.0)   temp.real = n1.real + n2.real  temp.imag = n1.imag + n2.imag   return temp } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Sum = 5.7 + 9.5i \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P81 minus time | <chare> | 1 | <pext> | <starting>P81 minus time</heading> class Time(internal var hours: Int, internal var minutes: Int, internal var seconds: Int)  fun main(args: Array<String>) {  val start = Time(12, 34, 55)  val stop = Time(8, 12, 15)  val diff: Time   diff = difference(start, stop)   print("TIME DIFFERENCE: ${start.hours}:${start.minutes}:${start.seconds} - ")  print("${stop.hours}:${stop.minutes}:${stop.seconds} ")  print("= ${diff.hours}:${diff.minutes}:${diff.seconds}") }  fun difference(start: Time, stop: Time): Time {  val diff = Time(0, 0, 0)   if (stop.seconds > start.seconds) {  --start.minutes  start.seconds += 60  }   diff.seconds = start.seconds - stop.seconds  if (stop.minutes > start.minutes) {  --start.hours  start.minutes += 60  }   diff.minutes = start.minutes - stop.minutes  diff.hours = start.hours - stop.hours   return diff } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TIME DIFFERENCE: 12:34:55 - 8:12:15 = 4:22:40 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P82 Remove All Whitespaces regex | <chare> | 1 | <pext> | <starting>P82 Remove All Whitespaces</heading> fun main(args: Array<String>) {  var sentence = "T his is b ett er."  println("Original sentence: $sentence")   sentence = sentence.replace("\\s".toRegex(), "")  println("After replacement: $sentence") } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Original sentence: T his is b ett er. After replacement: Thisisbetter. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P83 Convert String to Date LocalDate.parse() | <chare> | 1 | <pext> | <starting>P83 Convert String to Date</heading> import java.time.LocalDate import java.time.format.DateTimeFormatter  fun main(args: Array<String>) {  // Format y-M-d or yyyy-MM-d  val string = "2017-07-25"  val date = LocalDate.parse(string, DateTimeFormatter.ISO\_DATE)   println(date) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2017-07-25 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P84 println("%.4f".format(num)) | <chare> | 1 | <pext> | <starting>P84 println("%.4f".format(num))</heading> fun main(args: Array<String>) {  val num = 1.34567  println("%.4f".format(num)) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1.3457 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P85 arraycopy() | <chare> | 1 | <pext> | <starting>P85 arraycopy()</heading> import java.util.Arrays fun main(args: Array<String>) {  val array1 = intArrayOf(1, 2, 3)  val array2 = intArrayOf(4, 5, 6)  val aLen = array1.size  val bLen = array2.size  val result = IntArray(aLen + bLen)  System.arraycopy(array1, 0, result, 0, aLen)  System.arraycopy(array2, 0, result, aLen, bLen)  println(Arrays.toString(result)) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* [1, 2, 3, 4, 5, 6] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P86 Current Date LocalDateTime.now() | <chare> | 1 | <pext> | <starting>P86 Current Date LocalDateTime.now()</heading> import java.time.LocalDateTime fun main(args: Array<String>) {  val current = LocalDateTime.now()  println("Current Date and Time is: $current") } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Current Date and Time is: 2017-08-02T11:25:44.973 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P87 Add Two Dates | <chare> | 1 | <pext> | <starting>P87 Add Two Dates</heading> import java.util.Calendar fun main(args: Array<String>) {  val c1 = Calendar.getInstance()  val c2 = Calendar.getInstance()  val cTotal = c1.clone() as Calendar  cTotal.add(Calendar.YEAR, c2.get(Calendar.YEAR))  cTotal.add(Calendar.MONTH, c2.get(Calendar.MONTH) + 1) // Zero-based months  cTotal.add(Calendar.DATE, c2.get(Calendar.DATE))  cTotal.add(Calendar.HOUR\_OF\_DAY, c2.get(Calendar.HOUR\_OF\_DAY))  cTotal.add(Calendar.MINUTE, c2.get(Calendar.MINUTE))  cTotal.add(Calendar.SECOND, c2.get(Calendar.SECOND))  cTotal.add(Calendar.MILLISECOND, c2.get(Calendar.MILLISECOND))  println("${c1.time} + ${c2.time} = ${cTotal.time}") } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Tue Aug 08 10:20:56 NPT 2017 + Tue Aug 08 10:20:56 NPT 2017 = Mon Apr 16 20:41:53 NPT 4035 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P88 Join Two Lists addAll() | <chare> | 1 | <pext> | <starting>P88 Join Two Lists addAll()</heading> import java.util.ArrayList fun main(args: Array<String>) {  val list1 = ArrayList<String>()  list1.add("a")  val list2 = ArrayList<String>()  list2.add("b")  val joined = ArrayList<String>()  joined.addAll(list1)  joined.addAll(list2)  println("list1: $list1")  println("list2: $list2")  println("joined: $joined") } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* list1: [a] list2: [b] joined: [a, b] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P89 Convert Array to Set HashSet(Arrays.asList(\*array)) | <chare> | 1 | <pext> | <starting>P89 Convert Array to Set</heading> import java.util.\*  fun main(args: Array<String>) {   val array = arrayOf("a", "b", "c")  val set = HashSet(Arrays.asList(\*array))   println("Set: $set")  } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Set: [a, b, c] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P90 Byte to Hex format("%02X", b) | <chare> | 1 | <pext> | <starting>P90 Byte to Hex format("%02X", b)</heading> fun main(args: Array<String>) {   val bytes = byteArrayOf(10, 2, 15, 11)   for (b in bytes) {  val st = String.format("%02X", b)  print(st)  } } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 0A020F0B \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P91 Create String from file | <chare> | 1 | <pext> | <starting>P91 Create String from file</heading> import java.nio.charset.Charset import java.nio.file.Files import java.nio.file.Paths fun main(args: Array<String>) {   val path = System.getProperty("user.dir") + "\\src\\test.txt"  val encoding = Charset.defaultCharset();   val lines = Files.readAllLines(Paths.get(path), encoding)  println(lines)  } </ending> | </end> |
| <hitle> | P92 Append Text to file | <chare> | 1 | <pext> | <starting>P92 Append Text to file</heading> import java.io.IOException import java.nio.file.Files import java.nio.file.Paths import java.nio.file.StandardOpenOption fun main(args: Array<String>) {  val path = System.getProperty("user.dir") + "\\src\\test.txt"  val text = "Added text"  try {  Files.write(Paths.get(path), text.toByteArray(), StandardOpenOption.APPEND)  } catch (e: IOException) {  }  } </ending> | </end> |
| <hitle> | P93 InputStream | <chare> | 1 | <pext> | <starting>P93 InputStream</heading> import java.io.\*  fun main(args: Array<String>) {   val stream = ByteArrayInputStream("Hello there!".toByteArray())  val sb = StringBuilder()  var line: String?   val br = BufferedReader(InputStreamReader(stream))  line = br.readLine()   while (line != null) {  sb.append(line)  line = br.readLine()  }  br.close()   println(sb)  } </ending> | </end> |
| <hitle> | P94 OutputStream | <chare> | 1 | <pext> | <starting>P94 OutputStream</heading> import java.io.\* fun main(args: Array<String>) {  val stream = ByteArrayOutputStream()  val line = "Hello there!"  stream.write(line.toByteArray())  val finalString = String(stream.toByteArray())  println(finalString) } </ending> | </end> |
| <hitle> | P95 Lookup enum | <chare> | 1 | <pext> | <starting>P95 Lookup enum</heading> enum class TextStyle {  BOLD, ITALICS, UNDERLINE, STRIKETHROUGH } fun main(args: Array<String>) {  val style = "Bold"  val textStyle = TextStyle.valueOf(style.toUpperCase())  println(textStyle) } </ending> | </end> |
| <hitle> | P96 Sort a Map Values | <chare> | 1 | <pext> | <starting>P96 Sort a Map Values</heading> fun main(args: Array<String>) {   var capitals = hashMapOf<String, String>()  capitals.put("Nepal", "Kathmandu")  capitals.put("India", "New Delhi")  capitals.put("United States", "Washington")  capitals.put("England", "London")  capitals.put("Australia", "Canberra")   val result = capitals.toList().sortedBy { (\_, value) -> value}.toMap()   for (entry in result) {  print("Key: " + entry.key)  println(" Value: " + entry.value)  } } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Key: Australia Value: Canberra Key: Nepal Value: Kathmandu Key: England Value: London Key: India Value: New Delhi Key: United States Value: Washington \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P97 string numeric parseDouble() | <chare> | 1 | <pext> | <starting>P97 string numeric parseDouble()</heading> import java.lang.Double.parseDouble fun main(args: Array<String>) {  val string = "12345s15"  var numeric = true  try {  val num = parseDouble(string)  } catch (e: NumberFormatException) {  numeric = false  }  if (numeric)  println("$string is a number")  else  println("$string is not a number") } </ending> | </end> |
| <hitle> | P98 Build a pile of Cubes pow(1/3) | <chare> | 1 | <pext> | <starting>P98 Build a pile of Cubes pow(1/3)</heading> import kotlin.math.pow fun findNb(m: Long): Long {  val cubrt = m.toDouble().pow(1/3.toDouble()).toInt()  var sum: Long = 0  for(i in 1..cubrt) {  sum += i.toDouble().pow(3).toLong()  if(sum > m ) return -1  else if (sum == m) return i.toLong()  }  return -1 } fun main() {  println(findNb(1071225)) // 45 } </ending> | </end> |
| <hitle> | P99 Count the Digit k\*k | <chare> | 1 | <pext> | <starting>P99 Count the Digit k\*k</heading> n = 10, d = 1  the k\*k are 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 We are using the digit 1 in: 1, 16, 81, 100. The total count is then 4. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun nbDig(n:Int, d:Int):Int {  val dchar = d.toString().toCharArray().first()  val result = (0..n).map{ (it\*it).toString().toCharArray().toList() }.flatMap{it}.filter { it == dchar }.count()  return result } fun main() {  println(nbDig(10,1)) //4  println(nbDig(25,1)) //11  println(nbDig(5750,0)) //4700 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P100 Tribonacci Sequence | <chare> | 1 | <pext> | <starting>P100 Tribonacci Sequence</heading> fun tribonacci(signature: DoubleArray, n: Int): DoubleArray {  if(signature.size < 3) return doubleArrayOf()  val res:MutableList<Double> = mutableListOf()  res.addAll(signature.toList())  var start = 4  while(start <= n) {  res.add(res[start-4]+res[start-3]+res[start-2])  start++  }  return res.toDoubleArray() } fun main() {  println(tribonacci(doubleArrayOf(1.0,1.0,1.0),10)) //[1.0,1.0,1.0,3.0,5.0,9.0,17.0,31.0,57.0,105.0] } </ending> | </end> |
| <hitle> | P101 Divisibility thirteen | <chare> | 1 | <pext> | <starting>P101 Divisibility thirteen</heading> val seq = listOf(1, 10, 9, 12, 3, 4);  tailrec fun thirt(n:Long):Long {  val v = n.toString().toList()  .reversed()  .mapIndexed { index, c -> c.toString().toLong() \* seq[index % seq.size] }  .sum()  return if(v < 100) v else thirt(v) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* import kotlin.math.\* fun Long.length() = when(this) {  0L -> 1L  else -> log10(abs(toDouble())).toLong() + 1 } fun thirt(n:Long):Long {  val pattern = listOf(1,10,9,12,3,4)  var remainders = mutableListOf(1,10,9,12,3,4)  val digitCount = n.length()  while( remainders.size < digitCount )  {  remainders.addAll(pattern)  }  val multipliers = remainders.take(digitCount.toInt())  var sum = n  var prevsum = 0L  var resn = n  while (sum != prevsum) {  prevsum = sum  sum = 0  val digitList = resn.toString().map{it.toString().toLong()}.reversed()  for(i in 0..digitList.lastIndex) {  sum += digitList[i] \* multipliers[i]  }  resn = sum  }  return resn } fun main() {  println(thirt(1234567)) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P102 extract file name | <chare> | 1 | <pext> | <starting>P102 extract file name</heading> fun extractFileName(dirtyFileName: String): String {  val splits = dirtyFileName.split(".").map {it.split("\_").filter{!it.all { char -> char.isDigit()}}}  return splits[0].joinToString("\_") + "." + splits[1].joinToString("") } fun extractFileName(self:String)=self.substringAfter("\_").substringBeforeLast(".") fun main() {  println(extractFileName("1231231223123131\_FILE\_NAME.EXTENSION.OTHEREXTENSION")) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Inputs: 1231231223123131\_FILE\_NAME.EXTENSION.OTHEREXTENSION 1\_This\_is\_an\_otherExample.mpg.OTHEREXTENSIONadasdassdassds34 1231231223123131\_myFile.tar.gz2 Outputs FILE\_NAME.EXTENSION This\_is\_an\_otherExample.mpg myFile.tar </ending> | </end> |
| <hitle> | P103 Multiples of 3 or 5 | <chare> | 1 | <pext> | <starting>P103 Multiples of 3 or 5</heading> fun solution(number: Int) = (1 until number).filter { it % 3 == 0 || it % 5 == 0 }.sum() </ending> | </end> |
| <hitle> | P104 | <chare> | 1 | <pext> | <starting>sqlite</heading> fun main() {  var fomular = "Fe(NO3)2K4[ON(SO3)2]2"  var periodic = mutableListOf("H","He","Li","Be","B","C","N","O","F","Ne","Na","Mg","Al","Si","P","S","Cl","Ar",  "K","Ca","Sc","Ti","V","Cr","Mn","Fe","Co","Ni","Cu","Zn","Ga","Ge","As","Se","Br",  "Kr","Rb","Sr","Y","Zr","Nb","Mo","Tc","Ru","Rh","Pd","Ag","Cd","In","Sn","Sb","Te",  "I","Xe","Cs","Ba","La","Ce","Pr","Nd","Pm","Sm","Eu","Gd","Tb","Dy","Ho","Er","Tm",  "Yb","Lu","Hf","Ta","W","Re","Os","Ir","Pt","Au","Hg","Tl","Pb","Bi","Po","At","Rn",  "Fr","Ra","Ac","Th","Pa","U","Np","Pu","Am","Cm","Bk","Cf","Es","Fm","Md","No","Lr",  "Rf","Db","Sg","Bh","Hs","Mt","Ds","Rg","Uub","Uuq",)  periodic.sortByDescending {it.length}  var terms = periodic.map {Pair(it, "\*"+periodic.indexOf(it)+"\*")}  terms.forEach {  fomular = fomular.replace(it.first, it.second)  }  println(fomular)  var periStart = false  var braketStart1 = false  var braketStart2 = false  var braketStart3 = false  val fomChars = fomular.toCharArray()  var curNums = ""  var curItem = mutableMapOf("periodic" to 0, "bracket1" to false, "bracket2" to false, "bracket3" to false)  var opts = mutableMapOf(0 to curItem)  var index = 0  for(i in 0..fomChars.lastIndex) {  if(fomChars[i] == '\*') {  if(periStart == true) {  periStart = false   println(curNums)  curItem["periodic"] = curNums.toInt()  curItem["bracket1"] = braketStart1  curItem["bracket2"] = braketStart2  curItem["bracket3"] = braketStart3  opts.put(index to curItem)  index++  }   else {  periStart = true  }   curNums = ""  }  else if(fomChars[i] == '{') {  braketStart3 = true  }  else if(fomChars[i] == '}') {  braketStart3 = false  }  else if(fomChars[i] == '[') {  braketStart2 = true  }  else if(fomChars[i] == ']') {  braketStart2 = false  }  else if(fomChars[i] == '(') {  braketStart1 = true  }  else if(fomChars[i] == ')') {  braketStart1 = false  }  else if(fomChars[i].isDigit()){  curNums += fomChars[i]  }  }  println(opts) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P105 Josephus Survivor | <chare> | 1 | <pext> | <starting>P105 Josephus Survivor</heading> josephus\_survivor(7,3) => means 7 people in a circle; one every 3 is eliminated until one remains [1,2,3,4,5,6,7] - initial sequence [1,2,4,5,6,7] => 3 is counted out [1,2,4,5,7] => 6 is counted out [1,4,5,7] => 2 is counted out [1,4,5] => 7 is counted out [1,4] => 5 is counted out [4] => 1 counted out, 4 is the last element - the survivor! \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun josephusSurvivor(n: Int, k: Int): Int {  if(k<1) return 1  if(k==1) return n  val list = (1..n).toMutableList()  var curpos = -1  while(list.size>1) {  while(curpos <= list.lastIndex)  {  curpos = curpos + k  if(curpos <= list.lastIndex) list[curpos] = -1  }  if(curpos > list.lastIndex) curpos = curpos - list.lastIndex - k - 1  list.removeIf{it == -1}  }  return list[0] } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P106 Persistent Bugger | <chare> | 1 | <pext> | <starting>P106 Persistent Bugger</heading> 39 --> 3 (because 3\*9 = 27, 2\*7 = 14, 1\*4 = 4 and 4 has only one digit) 999 --> 4 (because 9\*9\*9 = 729, 7\*2\*9 = 126, 1\*2\*6 = 12, and finally 1\*2 = 2) 4 --> 0 (because 4 is already a one-digit number) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun persistence1(num: Int) : Int {  var str = num.toString()  var depth = 0;  while(str.length > 1) {  var multi = 1;  str.forEach {  multi = multi \* it.toString().toInt()  }  str = multi.toString()  depth++  }  return depth } fun persistence(num: Int) = generateSequence(num) {  it.toString().map(Character::getNumericValue).reduce { mult, element -> mult \* element }  }.takeWhile { it > 9 }.count() fun assertEquals(n: Int, k:Int): Boolean {  return n == k } fun main() {  println(assertEquals(3, persistence(39)))  println(assertEquals(0, persistence(4)))  println(assertEquals(2, persistence(25)))  println(assertEquals(4, persistence(999))) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P107 RGB To Hex Conversion | <chare> | 1 | <pext> | <starting>sqlite</heading> rgb(255, 255, 255) // returns FFFFFF rgb(255, 255, 300) // returns FFFFFF rgb(0, 0, 0) // returns 000000 rgb(148, 0, 211) // returns 9400D3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun rgb(r: Int, g: Int, b: Int) = String.format("%02X%02X%02X", r.coerceIn(0..255), g.coerceIn(0..255), b.coerceIn(0..255)) fun rgb(r: Int, g: Int, b: Int): String {  println(r.toHex() + g.toHex() + b.toHex())  return r.toHex() + g.toHex() + b.toHex() } fun Int.toHex() : String{  if(this < 0) return "00"  if(this > 255) return "FF"  val hex = this.toString(16).uppercase().let {  if(it.length<=1) "0"+it  else it  }  return hex } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P108 Playing with digits | <chare> | 1 | <pext> | <starting>P108 Playing with digits</heading> digPow(89, 1) should return 1 since 8¹ + 9² = 89 = 89 \* 1 digPow(92, 1) should return -1 since there is no k such as 9¹ + 2² equals 92 \* k digPow(695, 2) should return 2 since 6² + 9³ + 5⁴= 1390 = 695 \* 2 digPow(46288, 3) should return 51 since 4³ + 6⁴+ 2⁵ + 8⁶ + 8⁷ = 2360688 = 46288 \* 51 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* import kotlin.math.\* fun digPow(n: Int, p: Int): Int {  val diglist = n.toString().toCharArray().map {it.toString().toFloat()}  var specsum = 0  var power = p  diglist.forEach {  specsum += it.pow(power).toInt()  power++  }  if(specsum % n == 0) return specsum/n  else return -1 } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* import kotlin.math.pow fun digPow(n: Int, p: Int) = n.toString().mapIndexed { i, c -> c.toString().toDouble().pow(p + i).toInt() }.sum().let { if (it % n == 0) it / n else -1 } </ending> | </end> |
| <hitle> | P109 Strip Comments | <chare> | 1 | <pext> | <starting>P109 Strip Comments</heading> fun solution(input: String, markers: CharArray): String {  val list = input.split("\n")  val reslist = mutableListOf<String>()  val clearlist = mutableListOf<String>()  list.forEach {  for(i in 0..it.lastIndex) {  if(markers.indexOf(it.get(i))>=0) {  reslist.add(it.substring(0,i))  break  }  else if(i==it.lastIndex) {  reslist.add(it)  }  }  }  reslist.forEach {  for(i in it.lastIndex downTo 0) {  if(it.get(i) != ' ') {  clearlist.add(it.substring(0,i+1))  break;  }  else if(i==0) {  clearlist.add(it)  }  }  }  return clearlist.joinToString("\n") } fun main() {  println(solution("apples, plums % and bananas\npears\noranges !applesauce", charArrayOf('%', '!')))  println(solution("Q @b\nu\ne -e f g", charArrayOf('@', '-'))) } </ending> | </end> |
| <hitle> | P110 Factorial decomposition | <chare> | 1 | <pext> | <starting>P110 Factorial decomposition</heading> fun decomp(m:Int):String {  val primelist = mutableListOf<Int>()  for(i in 2..m) {  var root = i  while(root > 1) {  for(j in 2..root) {  if(root % j == 0) {  root = root / j  primelist.add(j)  break;  }  }  }  }  primelist.sort()  val reslist = primelist.groupingBy{it}.eachCount()  println(reslist)  return reslist.map {  if(it.value <= 1) it.key  else it.key.toString() + "^" + it.value.toString()  }.joinToString(" \* ") } fun main() {  println(decomp(17))// "2^15 \* 3^6 \* 5^3 \* 7^2 \* 11 \* 13 \* 17"));  println(decomp(5))// "2^3 \* 3 \* 5");  println(decomp(22))// "2^19 \* 3^9 \* 5^4 \* 7^3 \* 11^2 \* 13 \* 17 \* 19");  println(decomp(14))// "2^11 \* 3^5 \* 5^2 \* 7^2 \* 11 \* 13");  println(decomp(25))// "2^22 \* 3^10 \* 5^6 \* 7^3 \* 11^2 \* 13 \* 17 \* 19 \* 23"); } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* import java.util.\* fun decomp(m:Int):String {  val map = TreeMap<Int, Int>()  for (i in 2..m) {  val decMap = decompSmall(i)  decMap.forEach { (key, value) -> map[key] = (map[key] ?: 0) + value}  }  return map.map { if (it.value > 1) "${it.key}^${it.value}" else "${it.key}" }.joinToString(" \* ") }  fun decompSmall(m: Int): Map<Int, Int> {  var div = 2  var mm = m  val result = TreeMap<Int, Int>()  while (mm > 1) {  if (mm % div == 0) {  result[div] = (result[div] ?: 0) + 1  mm /= div  } else {  div++  }  }  return result } </ending> | </end> |
| <hitle> | P111 Primes in numbers | <chare> | 1 | <pext> | <starting>P111 Primes in numbers</heading> import java.util.\* fun factors(l: Int): String {  var div = 2  var mm = l  val result = TreeMap<Int, Int>()  while (mm > 1) {  if (mm % div == 0) {  result[div] = (result[div] ?: 0) + 1  mm /= div  } else {  div++  }  }  return result.map { if (it.value > 1) "(${it.key}\*\*${it.value})" else "(${it.key})" }.joinToString("") } fun main() {  println(factors(1024))// "(2\*\*10)")  println(factors(7775460))// "(2\*\*2)(3\*\*3)(5)(7)(11\*\*2)(17)") } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun factors(l: Int): String {  var n=l  var str=""  for(i in 2..n)  {  var c=0  while(n%i==0){c++; n/=i }  if(c>1) str+="("+i+"\*\*"+c+")";  if(c==1) str+="("+i+")";  }  return str } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P112 Square into Squares. Protect trees! | <chare> | 1 | <pext> | <starting>P112 Square into Squares. Protect trees!</heading> The function "decompose" will take a positive integer n and return the decomposition of N = n² as:  [x1 ... xk] or "x1 ... xk" or Just [x1 ... xk] or Some [x1 ... xk] or {x1 ... xk} or "[x1,x2, ... ,xk]" \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* import kotlin.math.sqrt fun decompose(n: Long): String {  val decomposeArray = decomposer(n, n \* n) ?: return "null"  decomposeArray.removeAt(decomposeArray.size - 1)  return decomposeArray.joinToString(" ") } private fun decomposer(n: Long, remain: Long): MutableList<Long>? {  if (remain == 0L) return mutableListOf(n)  for (i in n - 1 downTo 1) {  if (remain - i \* i >= 0) decomposer(i, remain - i \* i)?.apply { add(n); return this }  }  return null } fun decompose1(n: Long): String {  var remain:Long = n\*n - (n-1)\*(n-1)  val loop = mutableListOf<Long>()  loop.add(n-1)  while(loop.size > 0) {  if(remain == 0L) return loop.sorted().joinToString(" ")  else if(remain < 0) {  loop[loop.lastIndex]--  while(loop.size>0 && loop.last() <= 0) {  loop.removeLast()  if(loop.size>0) loop[loop.lastIndex]--  }  if(loop.size <= 0) return "null"  if(loop.last() <= 0) return "null"  remain = n\*n - loop.map{it\*it}.sum()  }   else if(remain > 0) {  while(remain > 0) {  val cur = sqrt(remain.toDouble()).toLong()  remain = remain - cur\*cur  if(loop.size>0) {  if(loop.last() > cur){  loop.add(cur)   }  else {  remain = -1  }  }  else loop.add(cur)  }  }  }  return "null" } fun main() {  println(decompose(2))// "null")  println(decompose(11))// "1 2 4 10")  println(decompose(12))// "1 2 3 7 9")  println(decompose(4))// "null")  println(decompose(625))// 2 5 8 34 624  println(decompose(7100))// 2 3 5 119 7099  println(decompose(12345))// 2 6 157 12344  println(decompose(1234567))// 2 8 32 1571 1234566  println(decompose(7654321))// 6 10 69 3912 7654320 } </ending> | </end> |
| <hitle> | P113 Weight for weight | <chare> | 1 | <pext> | <starting>P113 Weight for weight</heading> "56 65 74 100 99 68 86 180 90" ordered by numbers weights becomes:  "100 180 90 56 65 74 68 86 99" \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fun orderWeight(i: String) = i.split(' ').sortedWith(compareBy({ it.sumBy { it - '0' } }, { it })).joinToString(" ") \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  </ending> | </end> |
| <hitle> | P114 Build a pile of Cubes | <chare> | 1 | <pext> | <starting>P114 Build a pile of Cubes</heading> n^3+(n−1)^3+(n−2)^3+...+1^3=m \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* import kotlin.math.pow fun findNb(m: Long): Long {  val cubrt = m.toDouble().pow(1/3.toDouble()).toInt()  var sum: Long = 0  for(i in 1..cubrt) {  sum += i.toDouble().pow(3).toLong()  if(sum > m ) return -1  else if (sum == m) return i.toLong()  }  return -1 } fun main(args: Array<String>) {  println(findNb(1071225)) //45  println(findNb(56396345062501)) //-1  println(findNb(6132680780625)) //2225  println(findNb(28080884739601)) //-1 } </ending> | </end> |
| <hitle> | P115 LinkedList | <chare> | 1 | <pext> | class LinkedList<T> {  private var head: Node<T>? = null  private var tail: Node<T>? = null  private var size = 0  fun isEmpty() = size == 0  override fun toString(): String {  if(isEmpty()) {  return "Empty List"  } else {  return head.toString()  }  }  fun push(value: T) {  head = Node(value = value, next = head)  if(tail == null) {  tail = head  }  size++  }  fun append(value: T) {  if(isEmpty()) {  push(value)  return  }  tail?.next = Node(value = value)  tail = tail?.next  size ++  }  fun nodeAt(index: Int): Node<T>? {  var currentNode = head  var currentIndex = 0  while(currentNode != null && currentIndex < index) {  currentNode = currentNode.next  currentIndex++  }  return currentNode  }  fun insert(value: T, afterNode: Node<T>): Node<T> {  if(tail == afterNode) {  append(value)  return tail!!  }  val newNode = Node(value = value, next = afterNode.next)  afterNode.next = newNode  size++  return newNode  }  fun removeAt(index: Int): Node<T>? {  var currentNode = head  var currentIndex = 0  while(currentNode != null && currentIndex < index - 1) {  currentNode = currentNode.next  currentIndex++  }  if(currentNode == null) return null  if(currentNode.next == null) return null  val res = currentNode.next  currentNode.next = res?.next  size--  return res  }  }  fun main() {  val node1 = Node(value = 1)  val node2 = Node(value = 2)  val node3 = Node(value = 3)  node1.next = node2  node2.next = node3  println(node1)  val list = LinkedList<Int>()  list.append(100)  list.append(200)  list.append(300)  println(list)  } | </end> |
| <hitle> | P116 Remove Duplicates Sorted ListNode | <chare> | 1 | <pext> | class ListNode(var value: Int) {  var next: ListNode? = null  fun printString() {  print("->$value");  if(next == null) {  println()  }  else {  print(next?.printString())  }  }  }  fun iteratorList(curNode:ListNode?) {  var thisNode = curNode  if(thisNode == null) return  if(thisNode?.next == null) return  if(thisNode?.value == thisNode?.next?.value)  {  thisNode.next = thisNode?.next?.next  }  iteratorList(thisNode?.next)  }  fun deleteDuplicates(head: ListNode?): ListNode? {  // iteratorList(head)  var curNode = head  if(curNode == null) return head  while(curNode?.next != null) {  if(curNode.value == curNode.next?.value) {  curNode.next = curNode.next?.next  }  else {  curNode = curNode.next  }  }  return head  }  fun mergeTwoLists(list1: ListNode?, list2: ListNode?): ListNode? {  var head1 = list1  var head2 = list2  var newList:ListNode? = null  var newHead:ListNode? = null  while(head1 != null || head2 != null) {  var processNode:ListNode? = null  if(head1 == null) {  processNode = head2  head2 = head2?.next  } else if(head2 == null) {  processNode = head1  head1 = head1?.next  }  else {  if(head1.value < head2.value) {  processNode = head1  head1 = head1.next  }  else if(head1.value > head2.value) {  processNode = head2  head2 = head2.next  }  else {  processNode = ListNode(head1.value)  head1 = head1.next  head2 = head2.next  }  }  processNode?.let {  if(newList == null){  newList = ListNode(processNode.value)  if(newHead == null) {  newHead = newList  }  } else {  newList?.next = ListNode(processNode.value)  newList = newList?.next  }  }  }  return newHead  }  fun main() {  val list = mutableListOf<ListNode>()  for(i in 0..5) {a  list.add(ListNode(i))  }  for(i in 0..4) {  list[i].next = list[i+1]  }  list[2].value = 1  list[0].printString()  println()  deleteDuplicates(list[0])  list[0].printString()  println()  val list2 = mutableListOf<ListNode>()  for(i in 3..8) {  list2.add(ListNode(i))  }  for(i in 0..list2.lastIndex-1) {  list2[i].next = list2[i+1]  }  list2[0].printString()  println()  val mergedList = mergeTwoLists(list[0], list2[0])  mergedList?.printString()  println()  } | </end> |
| <hitle> | P116 Search Longest Word | <chare> | 1 | <pext> | fun longestWord(words: List<String>): String {  val sorted = words.sorted()  val len = words.size  val stepList = IntArray(len)  println(stepList.toList())  var res = ""  var step = 0  for(i in 0..len - 1) {  stepList[i] = -1  if(sorted[i].length == 1) {  step++  stepList[i] = step  } else if(i > 0) {  if(sorted[i] == sorted[i -1]) {  stepList[i] == stepList[i - 1]  } else if(sorted[i - 1].length + 1 == sorted[i].length  && sorted[i].substring(0, sorted[i - 1].length) == sorted[i - 1]) {  stepList[i] = step  }  }  if(i > 0) {  if(stepList[i - 1] > 0 && stepList[i - 1] != stepList[i]) {  res = maxString(res, sorted[i - 1])  }  else if(stepList[i] > 0 && i == len - 1) {  res = maxString(res, sorted[i])  }  }  }  println(sorted)  println(stepList.toList())  return res  }  fun maxString(a: String, b: String) = if (a.length < b.length) b else a  fun main() {  println(longestWord(listOf("w","wo","wor","worl","world")))  println(longestWord(listOf("w", "banan", "a","banana", "ban", "wo", "wol", "apply", "bana", "app","appl", "ap","apple","apple", "b", "ba", "wolf")))  } | </end> |
| <hitle> | P117 Rotate Array | <chare> | 1 | <pext> | fun rotate(nums: IntArray, k: Int): Unit {  if(nums.isEmpty() || k == 0) return;    var tempArray = IntArray(nums.size)    for(i in nums.indices) {  tempArray[(i+k) % nums.size] = nums[i]  }    for(i in nums.indices) {  nums[i] = tempArray[i]  }  } | </end> |
| <hitle> | P118 Word Break | <chare> | 1 | <pext> | fun wordBreak(s: String, wordDict: List<String>): Boolean {  if(s.isEmpty()) return false  val len = s.length  val wordSet = HashSet(wordDict)  val data = BooleanArray(len + 1)  data[0] = true  for(hi in 1..len){  for(lo in 0..hi){  if(data[lo] && wordSet.contains(s.substring(lo, hi))){  data[hi] = true  break  }  }  }  return data[len]  } | </end> |
| <hitle> | P119 Check isPalindrome | <chare> | 1 | <pext> | fun isPalindrome(str: String): Boolean {  if(str.length <= 0) return false  var lo = 0  var hi = str.length-1  while(lo <= hi) {  if(str.get(lo) != str.get(hi)) {  return false  }  lo++  hi--  }  return true  } | </end> |
| <hitle> | P120 Check Duplicates in String | <chare> | 1 | <pext> | fun getDuplicates(str: String): MutableList<Char> {  val original = str.lowercase().toMutableList()  val removing = str.lowercase().toMutableList()  val result = mutableListOf<Char>()  while(removing.size > 0) {  val curchar = removing.removeAt(0)  original.remove(curchar)  if(original.remove(curchar)) {  result.add(curchar)  }  }  return result  } | </end> |
| <hitle> | P121 Check Duplicates in String | <chare> | 1 | <pext> | fun wordSubsets1(words1: Array<String>, words2: Array<String>): List<String> {  val res = mutableListOf<String>()  words1.forEach {  res.add(it)  for(i in 0..words2.size - 1) {  if(it.indexOf(words2[i]) < 0) {  res.removeLast()  break  }  }  }  return res  }  fun wordSubsets(words1: Array<String>, words2: Array<String>): List<String> {  val res = mutableListOf<String>()  words1.forEach {  res.add(it)  for(i in 0..words2.size - 1) {  if(it.indexOf(words2[i]) < 0) {  res.removeLast()  break  }  }  }  return res  }  fun main() {  println(wordSubsets(arrayOf("amazon","apple","facebook","google","leetcode"), arrayOf("e","o")))  } | </end> |
| <hitle> | P122 intersect | <chare> | 1 | <pext> | fun intersect(nums1: IntArray, nums2: IntArray): IntArray {  val res = mutableListOf<Int>()  val longArr = if(nums1.size > nums2.size) nums1.toMutableList() else nums2.toMutableList()  val shortArr = if(nums1.size > nums2.size) nums2 else nums1  shortArr.forEach {  if(longArr.remove(it)) res.add(it)  }  return res.toIntArray()  }  fun main() {  println(intersect(intArrayOf(1,2,2,1), intArrayOf(2,2)).toList())  println(intersect(intArrayOf(4,9,5), intArrayOf(9,4,9,8,4)).toList())  } | </end> |
| <hitle> | P123 firstUniqeChar | <chare> | 1 | <pext> | fun firstUniqeChar(s: String): Int {  if(s.length <= 0 ) return -1  if(s.length <= 1) return 0  val chars = s.toCharArray().toMutableList()  var repeatChars = HashSet<Char>()  while(chars.size > 0) {  val res = chars[0]  chars.removeAt(0)  if(chars.size <= 0) {  if(repeatChars.add(res)) {  return s.indexOf(res)  }  else {  return -1  }  }  if(chars.remove(res)) {  repeatChars.add(res)  }  else {  println(repeatChars)  if(repeatChars.add(res)) {  return s.indexOf(res)  }  }  }  return -1  }  fun main() {  // println(firstUniqChar("leetcode"))  // println(firstUniqChar("loveleetcode"))  // println(firstUniqChar("aabb"))  println(firstUniqChar("aadadaad"))  } | </end> |
| <hitle> | P124 containsDuplicate | <chare> | 1 | <pext> | fun containsDuplicate(nums: IntArray): Boolean {  val norepeat = HashSet<Int>()  nums.forEach {  if(!norepeat.add(it)) {  return true  }  }  return false  }  fun main() {  println(containsDuplicate(intArrayOf(1,2,3,1)))  println(containsDuplicate(intArrayOf(1,2,3,4)))  } | </end> |
| <hitle> | P124 binarySearch | <chare> | 1 | <pext> | fun binarySearch(nums: List<Int>, target: Int): Boolean {  if(nums.size <= 0) return false  val sorted = nums.sorted()  println(sorted)  var left = 0  var right = nums.size-1  var middle = left + (right - left) / 2  while(sorted[middle] != target) {  if(right - left == 1) {  if(sorted[left] == target || sorted[right] == target) return true  else return false  }  else {  if(sorted[middle] < target) {  left = middle  middle = left + (right - left) / 2  }  else {  right = middle  middle = left + (right - left) / 2  }  }  println("$left, $middle, $right")  }  return true;  }  --------------------isPerfectSquare---------------  fun isPerfectSquare(num: Int): Boolean {  val list = (1..num).toList()  var left = 0  var right = list.size - 1  while(left <= right) {  var mid = left + (right - left) / 2  if(list[mid] \* list[mid] == num) {  return true  }  if(list[mid] \* list[mid] > num) {  right = mid - 1  }  else {  left = mid + 1  }  }  return false  }  fun mySqrt(x: Int): Int {  if(x <= 0) return 0  val list = (1..x).toList()  var left = 1  var right = list.size  while(left <= right) {  var mid = left + (right - left) / 2  if(mid \* mid == x) {  return mid  }  if(mid \* mid > x) {  right = mid - 1  }  else {  left = mid + 1  }  println("$left, $mid, $right")  }  return left-1  }  ------------peakIndexInMountain-------------------  Foka gatun peakIndexInMountainArray(arr: IntArray): Int {  if(arr.size < 3) return -1  var left = 1  var right = arr.size - 2  while(left <= right) {  var mid = left + (right - left) / 2  if(arr[mid-1] < arr[mid] && arr[mid] > arr[mid+1]) {  return mid  }  else if(arr[mid-1] > arr[mid]) {  right = mid - 1  }  else {  left = mid + 1  }  }  return -1  }  fun main() {  println(peakIndexInMountainArray(intArrayOf(0,1,0)))  println(peakIndexInMountainArray(intArrayOf(0,2,1,0)))  println(peakIndexInMountainArray(intArrayOf(0,1,2,0)))  println(peakIndexInMountainArray(intArrayOf(0,10,5,2)))  } | </end> |