KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Programavimo kalbų teorija (P175B124)
Laboratorinių darbų ataskaita

Atliko:

       IFF-7/14 gr. studentas

       Eigijus Kiudys

       2020 m. balandžio 8 d.

Priėmė:

       lekt. Evaldas Guogis

       lekt. Fyleris Tautvydas

KAUNAS 2020

TURINYS

1. Python (L1)

1.1. Darbo užduotis

A large company wishes to monitor the cost of phone calls made by its personnel. To achieve this the PABX logs, for each call, the number called (a string of up to 15 digits) and the duration in minutes. Write a program to process this data and produce a report specifying each call and its cost, based on standard Telecom charges.

International (IDD) numbers start with two zeroes (00) followed by a country code (1–3 digits) followed by a subscriber's number (4–10 digits). National (STD) calls start with one zero (0) followed by an area code (1–5 digits) followed by the subscriber's number (4–7 digits). The price of a call is determined by its destination and its duration. Local calls start with any digit other than 0 and are free.

Input

Input will be in two parts. The first part will be a table of IDD and STD codes, localities and prices as follows:

Code △ Locality name$price in cents per minute

where △ represents a space. Locality names are 25 characters or less. This section is terminated by a line containing 6 zeroes (000000).

The second part contains the log and will consist of a series of lines, one for each call, containing the number dialled and the duration. The file will be terminated a line containing a single #. The numbers will not necessarily be tabulated, although there will be at least one space between them. Telephone numbers will not be ambiguous.

Output

Output will consist of the called number, the country or area called, the subscriber's number, the duration, the cost per minute and the total cost of the call, as shown below. Local calls are costed at zero. If the number has an invalid code, list the area as 'Unknown' and the cost as −1.00.

Note: The first line of the Sample Output below in not a part of the output, but only to show the exact tabulation format it must follow.

Sample Input

```
088925 Broadwood$81
03 Arrowtown$38
0061 Australia$140
000000
031526        22
0061853279   3
0889256287213    122
779760 1
002832769 5
#
```

Sample Output

```
1               17              51   56   62     69
031526          Arrowtown      1526   22  0.38    8.36
0061853279      Australia    853279    3  1.40    4.20
0889256287213   Broadwood   6287213  122  0.81   98.82
779760          Local        779760    1  0.00    0.00
002832769       Unknown                5         -1.00
```

1.2. Programos tekstas

```python
class TelephoneInfo:
    def __init__(self, code, name, price):
        self.code = code
        self.name = name
        self.price = price

    def priceSeconds(self):
        return round(self.price * 0.1, 2)

class TelephoneCalls:
    def __init__(self, number, time):
        self.number = number
        self.time = time

class Data (TelephoneInfo, TelephoneCalls):
    def __init__(self, number, code, name, price, time):
        self.number = number
        self.code = code
        self.name = name
        self.price = price
        self.time = time

    def calcuPrice(self):
        return float(self.price) * 0.1 * float(self.time)

    def __str__(self):
        if self.code != -1:
            return "{0:15} {1:16} {2:8} {3:6} {4:6} {5:.2f}\n".format(self.number,
self.name, self.code, self.time, str(self.priceSeconds()), self.calcuPrice())
        else:
            return "{0:15} {1:16} {2:8} {3:6} {4:6} {5:.2f}\n".format(self.number,
self.name, "", self.time, "", float(self.price))


class Main:

    def __init__(self, read, write):
        self.readFromFile = read
        self.writeToFile = write

    def dataRead(self):
        data = []
        file = open(self.readFromFile, "r")
        for dataFromFile in file:
            data.append(dataFromFile)
        file.close()
        return data

    def splitLine(self,dataFromFile):
        split = []
        listForInfo = []
        listForCalls = []
        info = False
        for i in dataFromFile:

            index = 0
            if len(split) == 2 and info:
                split = []
            elif not info:
                split = []

            for string in i:
```

```python
                if string == " ":
                    split.append(i[0:index])
                    split.append(i[index + 1: len(i)])
                    break
                index += 1
            if index == len(i):
                split.append(i.strip())

            split[0] = split[0].strip()
            if len(split) > 1 and not bool(info):
                split[1] = split[1].strip()
                data = split[1].split('$')
                telephoneInfo = TelephoneInfo(split[0], data[0], float(data[1]) * 0.1)
                listForInfo.append(telephoneInfo)
            elif len(split) == 1 and split[0] == "000000" and not bool(info):
                split = []
                info = True
            elif len(split) > 1 and bool(info):
                split[1] = split[1].strip()
                calls = TelephoneCalls(split[0], split[1])
                listForCalls.append(calls)
        return listForInfo, listForCalls

    def calculatePrice(self, listOfCall, listOfInfo):
        calculatedData = []
        for calls in listOfCall:
            state = False
            for info in listOfInfo:
                if calls.number[0:len(info.code)] == info.code:
                    data = Data(calls.number,
calls.number[len(info.code):len(calls.number)], info.name, info.price,
                                calls.time)
                    calculatedData.append(data)
                    state = True
                    break
                elif calls.number[0] != "0":
                    tempInfo = TelephoneInfo(calls.number, "Local", 0)
                    data = Data(calls.number, tempInfo.code, tempInfo.name,
tempInfo.price, calls.time)
                    calculatedData.append(data)
                    state = True
                    break
            if not bool(state):
                data = Data(calls.number, -1, "Unknown", -1, calls.time)
                calculatedData.append(data)
        return calculatedData

    def saveData(self, calculatedData):
        fSave = open(self.writeToFile, "w+")
        for data in calculatedData:
            fSave.write(str(data))
        fSave.close()

    def run(self):
        fileData = self.dataRead()
        infoList, callsList = self.splitLine(fileData)
        dataList = self.calculatePrice(callsList, infoList)
        self.saveData(dataList)

main = Main("test.txt", "data.txt")
main.run()
```

1.3. Pradiniai duomenys ir rezultatai

Pradiniai duomenys	Rezultatai
``` 088925 Broadwood$81 03 Arrowtown$38 0061 Australia$140 000000 031526 22 0061853279 3 0889256287213 122 779760 1 002832769 5 # ```	``` 031526 Arrowtown 1526 22 0.38 8.36 0061853279 Australia 853279 3 1.4 4.20 0889256287213 Broadwood 6287213 122 0.81 98.82 779760 Local 779760 1 0.0 0.00 002832769 Unknown 5 -1.00 ```

2. Scalatron botas

2.1. Darbo užduotis

Sukurti Scalatron botą.

Reikalavimai:
1. Panaudoti bent kelis master boto išleidžiamus botų padėjėjų tipus (pvz.: minos, raketos į priešus, "kamikadzės", rinkikai, masalas ir pan.)
2. Panaudoti bet kurį vieną iš kelio radimo algoritmų (DFS, BFS, A*, Greedy, Dijkstra).

Realizuotos minos, gyvūnų gaudytojai/rinkikai ir „Kamimadzė" akytvus botai, modifikuotos agresyvios ir apsauginės raketos.

Master botas, gyvūnų gaudytojai/rinkikai ir „Kamikadzė", kelio radimui naudoja Djikstros algoritmą.

2.2. Programos tekstas

```scala
import scala.util.control.Breaks._
import scala.math.sqrt
import java.util
import scala.collection.mutable.ListBuffer


object ControlFunction
{
  def forMaster(bot: Bot) {
    val (directionValue, nearestEnemyMaster, nearestEnemySlave) = analyzeViewAsMaster(bot)

    val dontFireAggressiveMissileUntil = bot.inputAsIntOrElse("dontFireAggressiveMissileUntil", -1)
    val dontFireDefensiveMissileUntil = bot.inputAsIntOrElse("dontFireDefensiveMissileUntil", -1)
    val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)
    val dontPlantMineUntil = bot.inputAsIntOrElse("dontPlantMineUntil", -1)
    val dontReleaseKamekadzeUntil = bot.inputAsIntOrElse("dontReleaseKamekadzeUntil", -1)
    val dontGatherFood = bot.inputAsIntOrElse("dontGatherFood", -1)

    val direction = XY.fromDirection45(directionValue)

    bot.move(direction) //give straight direction
    bot.set("lastDirection" -> directionValue)

    // food gathering bot, this bot gathers 1500 energy with givven and try return back to master
    if(dontGatherFood < bot.time && bot.energy > 500){
      bot.view.offsetToNearest('P') match {
        case Some(delta: XY) =>
          bot.set("rx" -> delta.x, "ry" -> delta.y)
            val unitDelta = XY.fromDirection45((lastDirection + 4) % 8)
            bot.spawn(unitDelta, "energy" -> 400, "mood" -> "Gather")
            bot.set("dontGatherFood" -> (bot.time + delta.stepCount + 1))
        case None =>
      }
      bot.view.offsetToNearest('B') match {
        case Some(delta: XY) =>
          bot.set("rx" -> delta.x, "ry" -> delta.y)
            val unitDelta = XY.fromDirection45((lastDirection + 4) % 8)
            bot.spawn(unitDelta, "energy" -> 500, "mood" -> "Gather")
            bot.set("dontGatherFood" -> (bot.time + delta.stepCount + 1))
        case None =>
      }

    }

    // mine planting
```

```scala
if(dontPlantMineUntil < bot.time && bot.energy > 600){

    val unitDelta = XY.fromDirection45((lastDirection + 4) % 8)
    bot.spawn(unitDelta, "energy" -> 400, "mood" -> "Mine")
    bot.set("dontPlantMineUntil" -> (bot.time + 20))

}

// kamekadze if master sees other master kamikaze go to straight slave bot or him self
if(dontReleaseKamekadzeUntil < bot.time && bot.energy > 200) { // fire defensive
missile?
    bot.view.offsetToNearest('m') match {
      case Some(delta: XY) =>
        bot.set("rx" -> delta.x, "ry" -> delta.y)
          val unitDelta = XY.fromDirection45((lastDirection + 4) % 8)
          bot.spawn(unitDelta, "energy" -> 200, "mood" -> "Kamikadze")
          bot.set("dontReleaseKamekadzeUntil" -> (bot.time + delta.stepCount + 1))
       case None =>
    }
}


if(dontFireAggressiveMissileUntil < bot.time && bot.energy > 100) { // fire attack
missile?
    nearestEnemyMaster match {
      case None =>          // no-on nearby
      case Some(relPos) =>    // a master is nearby
          val unitDelta = relPos.signum
          val remainder = relPos - unitDelta // we place slave nearer target, so subtract that
from overall delta
          bot.spawn(unitDelta, "mood" -> "Aggressive", "target" -> remainder)
          bot.set("dontFireAggressiveMissileUntil" -> (bot.time + relPos.stepCount + 1))
    }
}
else
if(dontFireDefensiveMissileUntil < bot.time && bot.energy > 100) { // fire defensive
missile?
    nearestEnemySlave match {
      case None =>          // no-on nearby
      case Some(relPos) =>    // an enemy slave is nearby
         if(relPos.stepCount < 8) {
            // this one's getting too close!
            val unitDelta = relPos.signum
            val remainder = relPos - unitDelta // we place slave nearer target, so subtract
that from overall delta
            bot.spawn(unitDelta, "mood" -> "Defensive", "target" -> remainder)
            bot.set("dontFireDefensiveMissileUntil" -> (bot.time + relPos.stepCount + 1))
         }
    }
// vieta kazkokiai atakai
}
}
```

```scala
def forSlave(bot: MiniBot) {
  bot.inputOrElse("mood", "Lurking") match {
    case "Aggressive" => reactAsAggressiveMissile(bot)
    case "Defensive" => reactAsDefensiveMissile(bot)
    case "Mine" => reactAsMine(bot)
    case "Kamikadze" => reactAsKamekaze(bot)
    case "Gather" => reactAsHarvest(bot)

    case s: String => bot.log("unknown mood: " + s)
  }
}

// mine plant logic
def reactAsMine(bot: MiniBot) {
  bot.view.offsetToNearest('m') match {
    case Some(delta: XY) =>
     bot.set("rx" -> delta.x, "ry" -> delta.y)
     if (delta.length <= 3) {
       // yes -- blow it up!
       bot.explode(4)
     }
    case None =>
  }
  bot.view.offsetToNearest('s') match {
    case Some(delta: XY) =>
     bot.set("rx" -> delta.x, "ry" -> delta.y)
     if (delta.length <= 3) {
       // yes -- blow it up!
       bot.explode(4)
     }
    case None =>
  }
  bot.view.offsetToNearest('b') match {
    case Some(delta: XY) =>
     bot.set("rx" -> delta.x, "ry" -> delta.y)
     if (delta.length < 2) {
       // yes -- blow it up!
       bot.explode(4)
     }
    case None =>
  }

}

// kamikadze exploads near master or slave if it cant find any of them then kamekazde self
distructs
def reactAsKamekaze(bot: MiniBot) {
    val direction45 = analyzeViewAsBot(bot, 1000)
    val direction = XY.fromDirection45(direction45)
    if(direction != XY(0,0))
    {
       bot.move(direction)
       bot.view.offsetToNearest('m') match {
          case Some(delta: XY) =>
```

```scala
                    bot.set("rx" -> delta.x, "ry" -> delta.y)
                    if (delta.length <= 4) {
                       // yes -- blow it up!
                       bot.explode(4)
                    }
                case None =>
              }
            }
            else
            {
              bot.explode(4)
            }
        }


        // start react gathering bot move by found location and obsticles
        def reactAsHarvest(bot: MiniBot) {

                val       (directionValue,    nearestEnemyMaster,    nearestEnemySlave)    =
analyzeViewAsMaster(bot)
                val direction = XY.fromDirection45(directionValue)
                bot.move(direction)
                bot.set("lastDirection" -> direction.toDirection45)

        }



        def reactAsAggressiveMissile(bot: MiniBot) {
          bot.view.offsetToNearest('m') match {
            case Some(delta: XY) =>
              // another master is visible at the given relative position (i.e. position delta)

              // close enough to blow it up?
              if(delta.length <= 2) {
                // yes -- blow it up!
                bot.explode(4)

              } else {
                // no -- move closer!
                bot.move(delta.signum)
                bot.set("rx" -> delta.x, "ry" -> delta.y)
              }
            case None =>
              // no target visible -- follow our targeting strategy
              val target = bot.inputAsXYOrElse("target", XY.Zero)

              // did we arrive at the target?
              if(target.isNonZero) {
                // no -- keep going
                val unitDelta = target.signum // e.g. CellPos(-8,6) => CellPos(-1,1)
                bot.move(unitDelta)

                // compute the remaining delta and encode it into a new 'target' property
```

```scala
          val remainder = target - unitDelta // e.g. = CellPos(-7,5)
          bot.set("target" -> remainder)
        } else
    }
}




def reactAsDefensiveMissile(bot: MiniBot) {
    bot.view.offsetToNearest('s') match {
      case Some(delta: XY) =>
        // another slave is visible at the given relative position (i.e. position delta)
        // move closer!
        bot.move(delta.signum)
        bot.set("rx" -> delta.x, "ry" -> delta.y)

      case None =>
        // no target visible -- follow our targeting strategy
        val target = bot.inputAsXYOrElse("target", XY.Zero)

        // did we arrive at the target?
        if(target.isNonZero) {
          // no -- keep going
          val unitDelta = target.signum // e.g. CellPos(-8,6) => CellPos(-1,1)
          bot.move(unitDelta)

          // compute the remaining delta and encode it into a new 'target' property
          val remainder = target - unitDelta // e.g. = CellPos(-7,5)
          bot.set("target" -> remainder)
        }
    }
}




def analyzeViewAsMaster(bot: Bot) = {
    var view = bot.view
    val directionValue = Array.ofDim[Double](8)
    var nearestEnemyMaster: Option[XY] = None
    var nearestEnemySlave: Option[XY] = None

    val cells = view.cells
    val cellCount = cells.length
    val cellWeights = Array.ofDim[Double](cellCount)
    val indexRel = view.indexFromRelPos(XY(0,0))

    //view.aStarPathfind(cellWeights, bot)
        //bot.log(cells.contains('P').toString)
        //bot.log(cells)

    // this for creates heatmap for helping a* algorith find road to position with weights
        for(i <- 0 until cellCount) {
          val cellRelPos = view.relPosFromIndex(i)
```

```
if(cellRelPos.isNonZero && !view.outOfBoundsRel(cellRelPos)) {
 cells(i) match {
 case 'm' => // another master: not dangerous, but an obstacle
    nearestEnemyMaster = Some(cellRelPos)
    for (x <- -4 to 4) {
     for (y <- -4 to 4) {
      val pos = cellRelPos + XY(x, y)
      if (pos.isNonZero && !view.outOfBoundsRel(pos)) {
       cellWeights(view.indexFromRelPos(pos)) += 10000
      }
     }
    }

 case 's' => // another slave: potentially dangerous?
    nearestEnemySlave = Some(cellRelPos)
    for (j <- 0 until cellCount) {
       val pos = view.relPosFromIndex(j)
       if(!bot.view.outOfBoundsRel(pos))
       {
          val stepDistance = cellRelPos.stepsTo(pos)
          if (pos.isNonZero && stepDistance != 0) {
             val stepDistance = cellRelPos.stepsTo(pos)
             cellWeights(j) += 1000 / stepDistance
          }
       }
    }
 case 'P' =>
    val pos = view.relPosFromIndex(i)

    if(!bot.view.outOfBoundsRel(pos))
      {
         val stepDistance = cellRelPos.stepsTo(pos)
         if (stepDistance == 1) cellWeights(i) += 100
         else if (stepDistance == 2) cellWeights(i) += 300
         else cellWeights(i) += 500
      }

 case 'B' =>
    val pos = view.relPosFromIndex(i)
    val stepDistance = cellRelPos.stepsTo(pos)
    if (stepDistance == 1) cellWeights(i) += 50
    else if (stepDistance == 2) cellWeights(i) += 200
    else cellWeights(i) += 520

 case 'b' =>
    for (x <- -2 to 2) {
       for (y <- -2 to 2) {
          val pos = view.relPosFromIndex(i) + XY(x,y)
          if(pos.isNonZero && !view.outOfBoundsRel(pos))
          {
             val index = view.indexFromRelPos(pos)
             cellWeights(index) += 2500000
          }
       }
```

```
                }

            /*case 'p' => // bad plant: bad, but only if I step on it
               cellWeights(i) += 100000*/

            case 'W' => // wall: harmless, just don't walk into it
               for (x <- -1 to 1) {
                for (y <- -1 to 1) {
                 val pos = cellRelPos + XY(x, y)
                 if (pos.isNonZero && !view.outOfBoundsRel(pos)) {
                  cellWeights(view.indexFromRelPos(pos)) += 1500000
                 }
                }
               }

            case '?' =>
               cellWeights(i) += 1500000

            case '_' =>
               cellWeights(i) += 2

            case _ => cellWeights(i) += 1

           }
          }
         }


         var direction45 = 0
         val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)
         val lastCount = bot.inputAsIntOrElse("lastCount", 0).toInt
         val previuosStepCount = bot.inputAsIntOrElse("PreviousStepCount", 0).toInt
         if ((cells.contains('P') || cells.contains('B')) || (lastCount < 1 && lastCount > 2)) {


            // finding closest target by weights around target and distance between target and
 master
            var (closestFood) = findClosestThings(cellCount, cells, view, cellWeights, bot)

            if(!closestFood.isZero)
            {

               var (path, path_index, found) = aStarPathfind(cells, XY.Zero, view, closestFood,
 bot, cellWeights)

               if(found && path.size > 0)
               {
                  if(path(path.size - 1) != XY(0,0))
                  {
                     if(previuosStepCount == path.size)
                     {
                        bot.set("lastCount" -> 1.toString)
                     }
```

```
                        bot.set("PreviousStepCount" -> path.size)
                        direction45 = path(path.size - 1).toDirection45
                        bot.log(direction45.toString)
                        directionValue(direction45)
                    }



                }
                else
                {
                    // if path not found it starts refrence bot algorithn
                    bot.set("lastCount" -> 1.toString)
                }
            }
            else{
                // after failed search of closest target it try again by adding bigger weight to
previuos target
                var (temp_closestFood) = findClosestThings(cellCount, cells, view, cellWeights,
bot)

                if(!bot.view.outOfBoundsRel(temp_closestFood) && !temp_closestFood.isZero)
                {
                    var index_tmp = view.indexFromRelPos(temp_closestFood)
                    cellWeights(index_tmp) += cellWeights(index_tmp) * 2
                    var (closestFood) = findClosestThings(cellCount, cells, view, cellWeights, bot)
                    var (path, path_index, found) = aStarPathfind(cells, XY.Zero, view,
closestFood, bot, cellWeights)

                    if(found && path.size > 0)
                    {
                        if(path(path.size - 1) != XY(0,0))
                        {


                                if(previuosStepCount == path.size)
                                {
                                    bot.set("lastCount" -> 1.toString)
                                }
                                bot.set("PreviousStepCount" -> path.size)
                                direction45 = path(path.size - 1).toDirection45
                                bot.log(direction45.toString)
                                directionValue(direction45)

                        }

                    }
                    else
                    {
                        // if path not found it starts refrence bot algorithn
                        bot.set("lastCount" -> 1.toString)
                    }
                }
```

```scala
                }

            }
            //refrence algorithm is used by bot when there are any targets in screnn or when playres
last step is the same
            else if((!cells.contains('P') && !cells.contains('B')) || (lastCount > 0 && lastCount < 3))
            {

                for(i <- 0 until cellCount) {
                    val cellRelPos = view.relPosFromIndex(i)
                    if(cellRelPos.isNonZero && !view.outOfBoundsRel(cellRelPos)) {
                        val stepDistance = cellRelPos.stepCount
                        val value: Double = cells(i) match {
                            case 'M' =>
                                1500000
                            case 'm' => // another master: not dangerous, but an obstacle
                                nearestEnemyMaster = Some(cellRelPos)
                                if(stepDistance < 2) -1000 else 0

                            case 's' => // another slave: potentially dangerous?
                                nearestEnemySlave = Some(cellRelPos)
                                -100 / stepDistance

                            case 'S' => // out own slave
                                0.0

                            case 'B' => // good beast: valuable, but runs away
                                if(stepDistance == 1) 600
                                else if(stepDistance == 2) 300
                                else (150 - stepDistance * 15).max(10)

                            case 'P' => // good plant: less valuable, but does not run
                                if(stepDistance == 1) 500
                                else if(stepDistance == 2) 300
                                else (150 - stepDistance * 10).max(10)

                            case 'b' => // bad beast: dangerous, but only if very close
                                if(stepDistance < 4) -400 / stepDistance else -50 / stepDistance

                            case 'p' => // bad plant: bad, but only if I step on it
                                if(stepDistance < 2) -1000 else 0

                            case 'W' => // wall: harmless, just don't walk into it
                                if(stepDistance < 3) -1000 else 0

                            case '?' =>
                                -1000

                            case '_' => 10

                            case _ => 0.0
                        }
                        direction45 = cellRelPos.toDirection45
                        directionValue(direction45) += value
```

16

```scala
                val bestDirection45 = directionValue.zipWithIndex.maxBy(_._1)._2
                direction45 = bestDirection45
            }
          }
        bot.set("lastCount" -> (lastCount + 1).toString)


      }



      (direction45, nearestEnemyMaster, nearestEnemySlave)
    }



// closest thing search
def findClosestThings(cellCount: Int, cells: String, view: View, weights: Array[Double],
bot: Bot): (XY) = {
    var distances = ListBuffer[Double]() // list of targets distances
    var indexes = ListBuffer[Int]() // list of target indexes
    var weightsForFood = ListBuffer[Double]() // target weights
    for (i <- 0 until cellCount) {
      val cellRelPos = view.relPosFromIndex(i)
        if(cellRelPos.isNonZero && !view.outOfBoundsRel(cellRelPos)) {
          cells(i) match {

            case 'P' =>
              indexes = indexes :+ i
              var testWeight = 0.0
              var cnt = 0
              for (x <- -4 to 4) {
                for (y <- -4 to 4) {
                  val pos = cellRelPos + XY(x, y)
                  if (pos.isNonZero && !view.outOfBoundsRel(pos)) {
                    testWeight += weights(view.indexFromRelPos(pos))
                    cnt += 1
                  }
                }
              }
              var avgWeight = testWeight / cnt
              val stepDistance = cellRelPos.stepCount
              distances = distances :+ stepDistance.toDouble
              // change distance by weights
                if(avgWeight > 1500000)
                {
                  distances(distances.size-1) += stepDistance / 2
                }
                if(avgWeight < 1500000 && avgWeight > 1000000 && bot.energy >
2500)

                {
                  distances(distances.size-1) -= stepDistance / 3
                }
                if(avgWeight < 1000000)
                {
                  distances(distances.size-1) -= stepDistance / 2
                }
```

```scala
            case 'B' =>
              indexes = indexes :+ i
              var testWeight = 0.0
              var cnt = 0
              for (x <- -4 to 4) {
                for (y <- -4 to 4) {
                  val pos = cellRelPos + XY(x, y)
                  if (pos.isNonZero && !view.outOfBoundsRel(pos)) {
                    testWeight += weights(view.indexFromRelPos(pos))
                    cnt += 1
                  }
                }
              }
              var avgWeight = testWeight / cnt
              val stepDistance = cellRelPos.stepCount
              distances = distances :+ stepDistance.toDouble
              // by weights make shorter or longer distance
                if(avgWeight > 1500000)
                {
                  distances(distances.size-1) += stepDistance / 2
                }
                if(avgWeight < 1500000 && avgWeight > 1000000 && bot.energy >
2500)

                {
                  distances(distances.size-1) -= stepDistance / 3
                }
                if(avgWeight < 1000000)
                {
                  distances(distances.size-1) -= stepDistance / 2
                }

            case _ =>


          }
        }

    }

    if(distances.nonEmpty)
    {
      // searching minimum distance index
      val temp_index = distances.indexOf(distances.min)
      // geting from indexed real position index, then from this index get position where is
target
      (view.relPosFromIndex(indexes(temp_index)))
    }
    else
    {
      (XY(0,1))
    }
  }
```

```
// a* path finding algorithm
def aStarPathfind(cells: String, startingPoint: XY, view: View, destination: XY, bot: Bot,
weights: Array[Double]) = {
        var open_list = ListBuffer[XY]() // list where coordinates are added after it selected
        var open_list_f = ListBuffer[Double]() // list where coordinates weight is added after
selection
        // open lists are constantly changing because this is temporary lis for value saving
        var closed_list = ListBuffer[Boolean]() // boolean list for checking if coordinates is used
        var parent = ListBuffer[Int]() // previous index list for ex. parent(child index) = parent
index
        var parent_coordinates = ListBuffer[XY]() // previous coordinates list for ex. parent(child
index) = parent coordinates
        var g = ListBuffer[Float]() // g weight list for finding path. g is distance fom center to
other move pair
        var f = ListBuffer[Float]() // g + h weight list for finding path
        var h = ListBuffer[Float]() // h weight list for finding path. h is distance between last and
other position pair
        var foundDest = false // boolean for returning if destination is found
        var loopingPos = startingPoint; // current position
        var path = ListBuffer[XY]() // founded path coordinates
        var path_index = ListBuffer[Int]() // first coordinates index of path list

        // init of lists
        for(i <- 0 until cells.length)
        {
           closed_list = closed_list :+ false
           parent = parent :+ -1
           parent_coordinates = parent_coordinates :+ XY(-1,-1)
           g = g :+ Float.MaxValue
           f = f :+ Float.MaxValue
           h = h :+ Float.MaxValue
        }

        //bot.log(cells.size.toString)

        // setuping first element
        var index = view.indexFromRelPos(startingPoint)
        f.update(index, (0.0).toFloat)
        g.update(index, (0.0).toFloat)
        h.update(index, (0.0).toFloat)
        parent.update(index, index)
        parent_coordinates.update(index, startingPoint)
        open_list = open_list :+ startingPoint
        open_list_f = open_list_f :+ 0.0

        var count = 0

        // breakable ussage for breaking while when end is found
        breakable{
           while(!open_list.isEmpty)
           {
              // get element from temporary list and the delete it
              loopingPos = open_list(0)
```

```
            var parentIndex = view.indexFromRelPos(loopingPos)
            open_list.remove(0)
            open_list_f.remove(0)

            // set that position is visited
            closed_list.update(index, false)

            // checking neighbours around selected element for finding next element
            for(x <- -1 to 1)
            {
               for(y <- -1 to 1)
               {
                  // prevent form adding zero coordinates
                  if((x != 0 && y != 0) ||(x == 0 && y != 0) || (x != 0 && y == 0))
                  {

                     var pos = loopingPos + XY(x, y)

                     index = view.indexFromRelPos(pos)

                     if (!view.outOfBoundsRel(pos))
                     {
                        // destination found
                        if(pos == destination)
                        {
                           // add last coordinates and index
                           parent_coordinates.update(index, pos)
                           parent.update(index, parentIndex)
                           // trace path
                           var    (temp_path,    temp_path_index)    =    tracePath(parent,
parent_coordinates, pos, startingPoint, view, bot)
                           path = temp_path
                           path_index = temp_path_index
                           foundDest = true
                           break
                        }
                        else if(closed_list(index) == false && isUnBlocked(cells, index)) // check
if element is not blocked and coordinates ins not used
                        {
                           // calculate vaerage weight around new element. this helps decide new
coordinates with geat map
                           var average_weight = 0.0
                           var count = 0
                           for(x_tmp <- -1 to 1)
                           {
                              for(y_tmp <- -1 to 1)
                              {
                                 var temp_pos = pos + XY(x_tmp, y_tmp)
                                 if(!bot.view.outOfBoundsRel(temp_pos))
                                 {
                                 var temp_index = view.indexFromRelPos(pos)
                                 average_weight = average_weight + weights(temp_index)
                                 average_weight += weights(index)
                                 count += 1
```

```scala
                                    }
                                }
                            }
                            average_weight = average_weight / count
                            // calculating weight for coordinates
                            var gNew = 0.0
                            if((x == 0 && y != 0) || (x != 0 && y == 0))
                            {
                               gNew = g(parentIndex) + 1.0;
                            }
                            else
                            {
                               gNew = g(parentIndex) + 1.414;
                            }
                            gNew = gNew
                            var hNew = calculateHValue(pos, destination)
                            hNew = hNew
                            var fNew = gNew + hNew + average_weight;

                            // check if use this coordinates or more efficent coordinates is already
selected
                            if (f(index) == Float.MaxValue || f(index) > fNew)
                            {
                               // add new coordinates for next coordinates
                               open_list = open_list :+ pos
                               open_list_f = open_list_f :+ fNew
                               //bot.log(index.toString)
                               // update all values fom init to real values
                               f.update(index, fNew.toFloat)
                               g.update(index, gNew.toFloat)
                               h.update(index, hNew.toFloat)
                               parent.update(index, parentIndex)
                               parent_coordinates.update(index, loopingPos)


                            }

                        }


                    }
                }
            }

        }

    }
    (path, path_index, foundDest)

}

def isUnBlocked(colums: String, index_check: Int): (Boolean) =
```

```scala
    {
       // Returns true if the cell is not blocked else false
       if (colums(index_check) != 'w' && colums(index_check) != '?' && colums(index_check)
!= 'b')
       {
          (true)
       }
       else
       {
          (false)
       }
    }

    def calculateHValue(pos: XY, dest: XY): (Double)=
    {
    // Return using the distance formula
    (sqrt((pos.x-dest.x)*(pos.x-dest.x) + (pos.y-dest.y)*(pos.y-dest.y)))
    }

    def tracePath(parent: ListBuffer[Int], parent_coordinates: ListBuffer[XY], last: XY, dest:
XY, view: View, bot: Bot) = {

       // go from back to beggining and return reversed list as path
       var Path = ListBuffer[XY]()
       var index = view.indexFromRelPos(last)
       var indexGo = ListBuffer[Int]()

       while(parent(index) != view.indexFromRelPos(dest))
       {

          Path = parent_coordinates(index) +=: Path
          indexGo = index +=: indexGo
          index = parent(index)
       }

       Path = parent_coordinates(index) +=: Path
       indexGo = index +=: indexGo
       (Path, indexGo)

    }


    // almost the same as master bot changed target search
    def analyzeViewAsBot(bot: Bot, masterVal: Int) = {
       var view = bot.view

       val directionValue = Array.ofDim[Double](8)
       var nearestEnemyMaster: XY = XY(0,0)
       var nearestEnemySlave: XY = XY(0,0)
       var nearestEnemy: XY = XY(0,0)
       var nearestEnemyIndex: Int = 0
       var nearestDistance: Int = 0
       val cells = view.cells
       val cellCount = cells.length
```

```scala
            val cellWeights = Array.ofDim[Double](cellCount)
          val indexRel = view.indexFromRelPos(XY(0,0))

              // this for creates heatmap for helping a* algorith find road to position with weight.
When heatmap ia updating closest enemy is updating too
              for(i <- 0 until cellCount) {
               val cellRelPos = view.relPosFromIndex(i)
               if(cellRelPos.isNonZero && !view.outOfBoundsRel(cellRelPos)) {
                cells(i) match {
                case 'M' =>
                     cellWeights(i) += 1500000
                case 'm' => // another master: not dangerous, but an obstacle
                   nearestEnemyMaster = cellRelPos
                   if(!bot.view.outOfBoundsRel(cellRelPos))
                   {
                      for (x <- -4 to 4) {
                       for (y <- -4 to 4) {
                        val pos = cellRelPos + XY(x, y)
                        val stepDistance = cellRelPos.stepsTo(pos)
                        if (pos.isNonZero && !view.outOfBoundsRel(pos) && stepDistance != 0)
{
                          cellWeights(view.indexFromRelPos(pos)) += 10000
                        }
                       }
                      }

                   }
                   nearestEnemy = nearestEnemyMaster
                   nearestEnemyIndex = i
                   nearestDistance = cellRelPos.stepCount

                case 's' => // another slave: potentially dangerous?
                   nearestEnemySlave = cellRelPos
                   for (j <- 0 until cellCount) {
                      val pos = view.relPosFromIndex(j)
                      if(!bot.view.outOfBoundsRel(pos))
                      {
                         val stepDistance = cellRelPos.stepsTo(pos)
                         if (pos.isNonZero && !view.outOfBoundsRel(pos) && stepDistance != 0)
{
                            val stepDistance = cellRelPos.stepsTo(pos)
                            cellWeights(j) += 1000 / stepDistance
                         }
                      }
                   }
                   if(nearestDistance > cellRelPos.stepCount)
                   {
                      nearestEnemy = nearestEnemySlave
                      nearestEnemyIndex = i
                   }
                case 'P' =>
                   val pos = view.relPosFromIndex(i)
                   val stepDistance = cellRelPos.stepsTo(pos)
                      if(pos.isNonZero && !view.outOfBoundsRel(pos) && stepDistance != 0)
```

```scala
                {
                  if (stepDistance == 1) cellWeights(i) += 100
                  else if (stepDistance == 2) cellWeights(i) += 300
                  else cellWeights(i) += 500
                }

          case 'B' =>
            val pos = view.relPosFromIndex(i)
            val stepDistance = cellRelPos.stepsTo(pos)
            if (stepDistance == 1) cellWeights(i) += 50
            else if (stepDistance == 2) cellWeights(i) += 200
            else cellWeights(i) += 520

          case 'b' =>
            for (x <- -2 to 2) {
              for (y <- -2 to 2) {
                val pos = view.relPosFromIndex(i) + XY(x,y)
                val stepDistance = cellRelPos.stepsTo(pos)
                if(pos.isNonZero && !view.outOfBoundsRel(pos) && stepDistance != 0)
                {
                  val index = view.indexFromRelPos(pos)
                  cellWeights(index) += 2500000
                }
              }
            }


          /*case 'p' => // bad plant: bad, but only if I step on it
            cellWeights(i) += 100000*/

          case 'W' => // wall: harmless, just don't walk into it
            for (x <- -1 to 1) {
              for (y <- -1 to 1) {
                val pos = cellRelPos + XY(x, y)
                val stepDistance = cellRelPos.stepsTo(pos)
                if(pos.isNonZero && !view.outOfBoundsRel(pos) && stepDistance != 0)
                {
                  cellWeights(view.indexFromRelPos(pos)) += 1500000
                }
              }
            }

          case '?' =>
            cellWeights(i) += 1500000

          case '_' =>
              cellWeights(i) += 2

        case _  => cellWeights(i) += 1

      }
    }
}
```

```
var direction45 = 0
val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)
val lastCount = bot.inputAsIntOrElse("lastCount", 0).toInt
val previuosStepCount = bot.inputAsIntOrElse("PreviousStepCount", 0).toInt
if ((cells.contains('m') || cells.contains('s')) || (lastCount < 1 && lastCount > 2)) {

    // check if nearest enemy is not at master position
    if(!nearestEnemy.isZero)
    {
        var (path, path_index, found) = aStarPathfind(cells, XY.Zero, view, nearestEnemy,
bot, cellWeights)

        if(found && path.size > 0)
        {
            if(path(path.size - 1) != XY(0,0))
            {
                if(previuosStepCount == path.size)
                {
                    bot.set("lastCount" -> 1.toString)
                }
                bot.set("PreviousStepCount" -> path.size)
                direction45 = path(path.size - 1).toDirection45
                bot.log(direction45.toString)
                directionValue(direction45)
            }


        }
        else
        {
            bot.set("lastCount" -> 1.toString)
        }
    }
    else{

        // check if enemy is still in view
        if(!bot.view.outOfBoundsRel(nearestEnemy) && !nearestEnemy.isZero)
        {
            // update weights after enemy is at master position
            var index_tmp = view.indexFromRelPos(nearestEnemy)
            cellWeights(index_tmp) += cellWeights(index_tmp) * 2
            // search for path
            var (path,  path_index,  found)  =  aStarPathfind(cells,  XY.Zero,  view,
nearestEnemy, bot, cellWeights)

            if(found && path.size > 0)
            {
                if(path(path.size - 1) != XY(0,0))
                {

                    if(previuosStepCount == path.size)
                    {
                        bot.set("lastCount" -> 1.toString)
```

```scala
                            }
                            bot.set("PreviousStepCount" -> path.size)
                            direction45 = path(path.size - 1).toDirection45
                            bot.log(direction45.toString)
                            directionValue(direction45)
                        }

                    }
                    else
                    {
                        bot.set("lastCount" -> 1.toString)
                    }
                }


            }

        }
        else if((!cells.contains('m') && !cells.contains('s')) || (lastCount > 0 && lastCount < 3))
// use refrence algorithm if enemy is not in view or path by a* is not found
        {

            for(i <- 0 until cellCount) {
                val cellRelPos = view.relPosFromIndex(i)
                if(cellRelPos.isNonZero && !view.outOfBoundsRel(cellRelPos)) {
                    val stepDistance = cellRelPos.stepCount
                    val value: Double = cells(i) match {
                        case 'm' =>
                            700

                        case 's' => // another slave: potentially dangerous?
                            700 / stepDistance

                        case 'S' => // out own slave
                            0.0

                        case 'B' => // good beast: valuable, but runs away
                            if(stepDistance == 1) -600
                            else if(stepDistance == 2) -300
                            else -(150 - stepDistance * 15).max(10)

                        case 'P' => // good plant: less valuable, but does not run
                            if(stepDistance == 1) -500
                            else if(stepDistance == 2) -00
                            else -(150 - stepDistance * 10).max(10)

                        case 'b' => // bad beast: dangerous, but only if very close
                            if(stepDistance < 4) -400 / stepDistance else -50 / stepDistance

                        case 'p' => // bad plant: bad, but only if I step on it
                            if(stepDistance < 2) -1000 else 0

                        case 'W' => // wall: harmless, just don't walk into it
                            if(stepDistance < 3) -1000 else 0
```

```scala
              case '?' =>
                -1000

              case '_' => 10

              case _ => 0.0
            }
            direction45 = cellRelPos.toDirection45
            directionValue(direction45) += value
            val bestDirection45 = directionValue.zipWithIndex.maxBy(_._1)._2
            direction45 = bestDirection45
          }
        }

    }


    (direction45)
  }

  // almoust the same as master bot target changes if harvest bot have 1500 energy adn is
vissable by bot
  def analyzeViewAsHarvest(bot: Bot) = {
  var view = bot.view
  // cia suranda vieta kur eiti
  val directionValue = Array.ofDim[Double](8)
  var nearestEnemyMaster: Option[XY] = None
  var nearestEnemySlave: Option[XY] = None
  var nearestMaster: XY = XY(0,0)
  val cells = view.cells
  val cellCount = cells.length
  val cellWeights = Array.ofDim[Double](cellCount)
  val indexRel = view.indexFromRelPos(XY(0,0))

    // heat map
    for(i <- 0 until cellCount) {
        val cellRelPos = view.relPosFromIndex(i)
        if(cellRelPos.isNonZero && !view.outOfBoundsRel(cellRelPos)) {
         cells(i) match {
         case 'M' =>
           nearestMaster = cellRelPos

         case 'm' =>
           nearestEnemyMaster = Some(cellRelPos)
           for (x <- -4 to 4) {
            for (y <- -4 to 4) {
             val pos = cellRelPos + XY(x, y)
             if (pos.isNonZero && !view.outOfBoundsRel(pos)) {
              cellWeights(view.indexFromRelPos(pos)) += 10000
             }
            }
           }
```

```scala
case 's' =>
  nearestEnemySlave = Some(cellRelPos)
  for (j <- 0 until cellCount) {
    val pos = view.relPosFromIndex(j)
    if(!bot.view.outOfBoundsRel(pos))
    {
      val stepDistance = cellRelPos.stepsTo(pos)
      if (pos.isNonZero && stepDistance != 0) {
        val stepDistance = cellRelPos.stepsTo(pos)
        cellWeights(j) += 1000 / stepDistance
      }
    }
  }
case 'P' =>
  val pos = view.relPosFromIndex(i)
  if(!bot.view.outOfBoundsRel(pos))
  {
    val stepDistance = cellRelPos.stepsTo(pos)
    if (stepDistance == 1) cellWeights(i) += 100
    else if (stepDistance == 2) cellWeights(i) += 300
    else cellWeights(i) += 500
  }

case 'B' =>
  val pos = view.relPosFromIndex(i)
  val stepDistance = cellRelPos.stepsTo(pos)
  if (stepDistance == 1) cellWeights(i) += 50
  else if (stepDistance == 2) cellWeights(i) += 200
  else cellWeights(i) += 520

case 'b' =>
  for (x <- -2 to 2) {
    for (y <- -2 to 2) {
      val pos = view.relPosFromIndex(i) + XY(x,y)
      if(pos.isNonZero && !view.outOfBoundsRel(pos))
      {
        val index = view.indexFromRelPos(pos)
        cellWeights(index) += 2500000
      }
    }
  }


case 'W' => // wall: harmless, just don't walk into it
  for (x <- -1 to 1) {
    for (y <- -1 to 1) {
      val pos = cellRelPos + XY(x, y)
      if (pos.isNonZero && !view.outOfBoundsRel(pos)) {
        cellWeights(view.indexFromRelPos(pos)) += 1500000
      }
    }
  }

case '?' =>
```

```
                      cellWeights(i) += 1500000

                  case '_' =>
                    cellWeights(i) += 2

                  case _ => cellWeights(i) += 1

                }
              }
            }


          var direction45 = 0
          val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)
          val lastCount = bot.inputAsIntOrElse("lastCount", 0).toInt
          val previuosStepCount = bot.inputAsIntOrElse("PreviousStepCount", 0).toInt
          if ((cells.contains('P') || cells.contains('B') || (cells.contains('M') && bot.energy > 1500))
|| (lastCount < 1 && lastCount > 2)) {


              var (closestFood) = findClosestThings(cellCount, cells, view, cellWeights, bot)

              if(cells.contains('M') && bot.energy > 1500)
              {
                closestFood = nearestMaster
              }

              if(!closestFood.isZero)
              {
                  var (path, path_index, found) = aStarPathfind(cells, XY.Zero, view, closestFood,
bot, cellWeights)

                  if(found && path.size > 0)
                    {
                      if(path(path.size - 1) != XY(0,0))
                      {

                          if(previuosStepCount == path.size)
                          {
                              bot.set("lastCount" -> 1.toString)
                          }
                          bot.set("PreviousStepCount" -> path.size)
                          direction45 = path(path.size - 1).toDirection45
                          bot.log(direction45.toString)
                          directionValue(direction45)
                      }


                    }
                    else
                    {
                      bot.set("lastCount" -> 1.toString)
                    }
              }
```

```
            else{

                var (temp_closestFood) = findClosestThings(cellCount, cells, view, cellWeights,
bot)

                if(cells.contains('M') && bot.energy > 1500)
                {
                    var closestFood = nearestMaster
                    var (path, path_index, found) = aStarPathfind(cells, XY.Zero, view,
closestFood, bot, cellWeights)

                        if(found && path.size > 0)
                        {
                            if(path(path.size - 1) != XY(0,0))
                            {

                                    if(previuosStepCount == path.size)
                                    {
                                        bot.set("lastCount" -> 1.toString)
                                    }
                                    bot.set("PreviousStepCount" -> path.size)
                                    direction45 = path(path.size - 1).toDirection45
                                    bot.log(direction45.toString)
                                    directionValue(direction45)
                            }

                        }
                        else
                        {
                            bot.set("lastCount" -> 1.toString)
                        }
                }
                else
                {

                    if(!bot.view.outOfBoundsRel(temp_closestFood)                    &&
!temp_closestFood.isZero)
                    {
                        var index_tmp = view.indexFromRelPos(temp_closestFood)
                        cellWeights(index_tmp) += cellWeights(index_tmp) * 2

                        var (closestFood) = findClosestThings(cellCount, cells, view, cellWeights,
bot)
                        var (path, path_index, found) = aStarPathfind(cells, XY.Zero, view,
closestFood, bot, cellWeights)

                        if(found && path.size > 0)
                        {
                            if(path(path.size - 1) != XY(0,0))
                            {

                                    if(previuosStepCount == path.size)
                                    {
                                        bot.set("lastCount" -> 1.toString)
```

```
                    }
                    bot.set("PreviousStepCount" -> path.size)
                    direction45 = path(path.size - 1).toDirection45
                    bot.log(direction45.toString)
                    directionValue(direction45)
                }

            }
            else
            {
                bot.set("lastCount" -> 1.toString)
            }
        }
    }


}

}
else if((!cells.contains('P') && !cells.contains('B') && !cells.contains('M')) || (lastCount
> 0 && lastCount < 3))
{

    for(i <- 0 until cellCount) {
        val cellRelPos = view.relPosFromIndex(i)
        if(cellRelPos.isNonZero && !view.outOfBoundsRel(cellRelPos)) {
            val stepDistance = cellRelPos.stepCount
            val value: Double = cells(i) match {
                case 'M' =>
                    1500000
                case 'm' => // another master: not dangerous, but an obstacle
                    nearestEnemyMaster = Some(cellRelPos)
                    if(stepDistance < 2) -1000 else 0

                case 's' => // another slave: potentially dangerous?
                    nearestEnemySlave = Some(cellRelPos)
                    -100 / stepDistance

                case 'S' => // out own slave
                    0.0

                case 'B' => // good beast: valuable, but runs away
                    if(stepDistance == 1) 600
                    else if(stepDistance == 2) 300
                    else (150 - stepDistance * 15).max(10)

                case 'P' => // good plant: less valuable, but does not run
                    if(stepDistance == 1) 500
                    else if(stepDistance == 2) 300
                    else (150 - stepDistance * 10).max(10)

                case 'b' => // bad beast: dangerous, but only if very close
                    if(stepDistance < 4) -400 / stepDistance else -50 / stepDistance
```

```scala
                case 'p' => // bad plant: bad, but only if I step on it
                   if(stepDistance < 2) -1000 else 0

                case 'W' => // wall: harmless, just don't walk into it
                   if(stepDistance < 3) -1000 else 0

                case '?' =>
                   -1000

                case '_' => 10

                case _ => 0.0
              }
              direction45 = cellRelPos.toDirection45
              directionValue(direction45) += value
              val bestDirection45 = directionValue.zipWithIndex.maxBy(_._1)._2
              direction45 = bestDirection45
            }
          }
          bot.set("lastCount" -> (lastCount + 1).toString)

        }



        (direction45)
      }



}


// ----------------------------------------------------------------------------------------------------
// Framework
// ----------------------------------------------------------------------------------------------------

class ControlFunctionFactory {
   def create = (input: String) => {
      val (opcode, params) = CommandParser(input)
      opcode match {
         case "React" =>
            val bot = new BotImpl(params)
            if( bot.generation == 0 ) {
               ControlFunction.forMaster(bot)
            } else {
               ControlFunction.forSlave(bot)
            }
            bot.toString
         case _ => "" // OK
      }
   }
```

```scala
    }


    // -----------------------------------------------------------------------------------------


    trait Bot {
       // inputs

       def inputOrElse(key: String, fallback: String): String
       def inputAsIntOrElse(key: String, fallback: Int): Int
       def inputAsXYOrElse(keyPrefix: String, fallback: XY): XY
       def view: View
       def energy: Int
       def time: Int
       def generation: Int

       // outputs
       def move(delta: XY) : Bot
       def say(text: String) : Bot
       def status(text: String) : Bot
       def spawn(offset: XY, params: (String,Any)*) : Bot
       def set(params: (String,Any)*) : Bot
       def log(text: String) : Bot
    }

    trait MiniBot extends Bot {
       // inputs
       def offsetToMaster: XY

       // outputs
       def explode(blastRadius: Int) : Bot
    }


    case class BotImpl(inputParams: Map[String, String]) extends MiniBot {
       // input
       def inputOrElse(key: String, fallback: String) = inputParams.getOrElse(key, fallback)
       def              inputAsIntOrElse(key:              String,           fallback:        Int)            =
    inputParams.get(key).map(_.toInt).getOrElse(fallback)
       def inputAsXYOrElse(key: String, fallback: XY) = inputParams.get(key).map(s =>
    XY(s)).getOrElse(fallback)

       val view = View(inputParams("view"))
       val energy = inputParams("energy").toInt
       val time = inputParams("time").toInt
       val generation = inputParams("generation").toInt
       def offsetToMaster = inputAsXYOrElse("master", XY.Zero)


       // output

       private var stateParams = Map.empty[String,Any]     // holds "Set()" commands
       private var commands = ""                        // holds all other commands
```

```
    private var debugOutput = ""                    // holds all "Log()" output

    /** Appends a new command to the command string; returns 'this' for fluent API. */
    private def append(s: String) : Bot = { commands += (if(commands.isEmpty) s else "|" + s);
this }

    /** Renders commands and stateParams into a control function return string. */
    override def toString = {
      var result = commands
      if(!stateParams.isEmpty) {
        if(!result.isEmpty) result += "|"
        result += stateParams.map(e => e._1 + "=" + e._2).mkString("Set(",",",")")
      }
      if(!debugOutput.isEmpty) {
        if(!result.isEmpty) result += "|"
        result += "Log(text=" + debugOutput + ")"
      }
      result
    }

    def log(text: String) = { debugOutput += text + "\n"; this }
    def move(direction: XY) = append("Move(direction=" + direction + ")")
    def say(text: String) = append("Say(text=" + text + ")")
    def status(text: String) = append("Status(text=" + text + ")")
    def explode(blastRadius: Int) = append("Explode(size=" + blastRadius + ")")
    def spawn(offset: XY, params: (String,Any)*) =
      append("Spawn(direction=" + offset +
        (if(params.isEmpty) "" else "," + params.map(e => e._1 + "=" + e._2).mkString(",")) +
        ")")
    def set(params: (String,Any)*) = { stateParams ++= params; this }
    def set(keyPrefix: String, xy: XY) = { stateParams ++= List(keyPrefix+"x" -> xy.x,
keyPrefix+"y" -> xy.y); this }
  }


  // -------------------------------------------------------------------------------------------------


  /** Utility methods for parsing strings containing a single command of the format
    * "Command(key=value,key=value,...)"
    */
  object CommandParser {
    /** "Command(..)" => ("Command", Map( ("key" -> "value"), ("key" -> "value"), ..}) */
    def apply(command: String): (String, Map[String, String]) = {
      /** "key=value" => ("key","value") */
      def splitParameterIntoKeyValue(param: String): (String, String) = {
        val segments = param.split('=')
        (segments(0), if(segments.length>=2) segments(1) else "")
      }

      val segments = command.split('(')
      if( segments.length != 2 )
        throw new IllegalStateException("invalid command: " + command)
      val opcode = segments(0)
```

```scala
      val params = segments(1).dropRight(1).split(',')
      val keyValuePairs = params.map(splitParameterIntoKeyValue).toMap
      (opcode, keyValuePairs)
    }
  }


  // ------------------------------------------------------------------------------------------------


  /** Utility class for managing 2D cell coordinates.
    * The coordinate (0,0) corresponds to the top-left corner of the arena on screen.
    * The direction (1,-1) points right and up.
    */
  case class XY(x: Int, y: Int) {
    override def toString = x + ":" + y

    def isNonZero = x != 0 || y != 0
    def isZero = x == 0 && y == 0
    def isNonNegative = x >= 0 && y >= 0

    def updateX(newX: Int) = XY(newX, y)
    def updateY(newY: Int) = XY(x, newY)

    def addToX(dx: Int) = XY(x + dx, y)
    def addToY(dy: Int) = XY(x, y + dy)

    def +(pos: XY) = XY(x + pos.x, y + pos.y)
    def -(pos: XY) = XY(x - pos.x, y - pos.y)
    def *(factor: Double) = XY((x * factor).intValue, (y * factor).intValue)

    def distanceTo(pos: XY): Double = (this - pos).length // Phythagorean
    def length: Double = math.sqrt(x * x + y * y) // Phythagorean

    def stepsTo(pos: XY): Int = (this - pos).stepCount // steps to reach pos: max delta X or Y
    def stepCount: Int = x.abs.max(y.abs) // steps from (0,0) to get here: max X or Y

    def signum = XY(x.signum, y.signum)

    def negate = XY(-x, -y)
    def negateX = XY(-x, y)
    def negateY = XY(x, -y)

    /** Returns the direction index with 'Right' being index 0, then clockwise in 45 degree steps.
    */
    def toDirection45: Int = {
      val unit = signum
      unit.x match {
        case -1 =>
          unit.y match {
            case -1 =>
              if(x < y * 3) Direction45.Left
              else if(y < x * 3) Direction45.Up
              else Direction45.UpLeft
```

```scala
                          case 0 =>
                             Direction45.Left
                          case 1 =>
                             if(-x > y * 3) Direction45.Left
                             else if(y > -x * 3) Direction45.Down
                             else Direction45.LeftDown
                       }
                 case 0 =>
                    unit.y match {
                       case 1 => Direction45.Down
                       case 0 => throw new IllegalArgumentException("cannot compute direction index
for (0,0)")
                       case -1 => Direction45.Up
                    }
                 case 1 =>
                    unit.y match {
                       case -1 =>
                          if(x > -y * 3) Direction45.Right
                          else if(-y > x * 3) Direction45.Up
                          else Direction45.RightUp
                       case 0 =>
                          Direction45.Right
                       case 1 =>
                          if(x > y * 3) Direction45.Right
                          else if(y > x * 3) Direction45.Down
                          else Direction45.DownRight
                    }
              }
           }

    def rotateCounterClockwise45 = XY.fromDirection45((signum.toDirection45 + 1) % 8)
    def rotateCounterClockwise90 = XY.fromDirection45((signum.toDirection45 + 2) % 8)
    def rotateClockwise45 = XY.fromDirection45((signum.toDirection45 + 7) % 8)
    def rotateClockwise90 = XY.fromDirection45((signum.toDirection45 + 6) % 8)


    def wrap(boardSize: XY) = {
       val fixedX = if(x < 0) boardSize.x + x else if(x >= boardSize.x) x - boardSize.x else x
       val fixedY = if(y < 0) boardSize.y + y else if(y >= boardSize.y) y - boardSize.y else y
       if(fixedX != x || fixedY != y) XY(fixedX, fixedY) else this
    }
}


object XY {
  /** Parse an XY value from XY.toString format, e.g. "2:3". */
  def apply(s: String) : XY = { val a = s.split(':'); XY(a(0).toInt,a(1).toInt) }

  val Zero = XY(0, 0)
  val One = XY(1, 1)

  val Right    = XY( 1,  0)
  val RightUp  = XY( 1, -1)
  val Up       = XY( 0, -1)
```
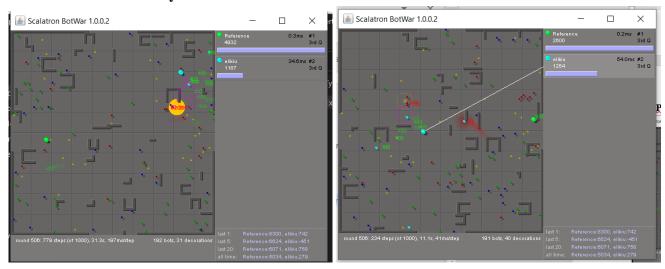
```scala
  val UpLeft    = XY(-1, -1)
  val Left      = XY(-1,  0)
  val LeftDown  = XY(-1,  1)
  val Down      = XY( 0,  1)
  val DownRight = XY( 1,  1)

  def fromDirection45(index: Int): XY = index match {
    case Direction45.Right => Right
    case Direction45.RightUp => RightUp
    case Direction45.Up => Up
    case Direction45.UpLeft => UpLeft
    case Direction45.Left => Left
    case Direction45.LeftDown => LeftDown
    case Direction45.Down => Down
    case Direction45.DownRight => DownRight
  }

  def fromDirection90(index: Int): XY = index match {
    case Direction90.Right => Right
    case Direction90.Up => Up
    case Direction90.Left => Left
    case Direction90.Down => Down
  }

  def apply(array: Array[Int]): XY = XY(array(0), array(1))
}


object Direction45 {
  val Right = 0
  val RightUp = 1
  val Up = 2
  val UpLeft = 3
  val Left = 4
  val LeftDown = 5
  val Down = 6
  val DownRight = 7
}


object Direction90 {
  val Right = 0
  val Up = 1
  val Left = 2
  val Down = 3
}


// -------------------------------------------------------------------------------------------


case class View(cells: String) {
```

37

```scala
    val size = math.sqrt(cells.length).toInt
    val center = XY(size / 2, size / 2)
    val cellCount = cells.length

    def apply(relPos: XY) = cellAtRelPos(relPos)

    def indexFromAbsPos(absPos: XY) = absPos.x + absPos.y * size
    def absPosFromIndex(index: Int) = XY(index % size, index / size)
    def absPosFromRelPos(relPos: XY) = relPos + center
    def cellAtAbsPos(absPos: XY) = cells.charAt(indexFromAbsPos(absPos))

    def indexFromRelPos(relPos: XY) = indexFromAbsPos(absPosFromRelPos(relPos))
    def relPosFromAbsPos(absPos: XY) = absPos - center
    def relPosFromIndex(index: Int) = relPosFromAbsPos(absPosFromIndex(index))
    def cellAtRelPos(relPos: XY) = cells.charAt(indexFromRelPos(relPos))

    def offsetToNearest(c: Char) = {
      val matchingXY = cells.view.zipWithIndex.filter(_._1 == c)
      if( matchingXY.isEmpty )
        None
      else {
        val nearest = matchingXY.map(p => relPosFromIndex(p._2)).minBy(_.length)
        Some(nearest)
      }
    }

    def outOfBoundsRel(relPos: XY) = {
      if(math.abs(relPos.x) > center.x || math.abs(relPos.y) > center.y){
        true
      }
      else{
        false
      }
    }

    def outOfBoundsAbs(absPos: XY) = {
      if(absPos.x < 0 || absPos.x > (size-1) || absPos.y < 0 || absPos.y > (size-1)){
        true
      }
      else{
        false
      }
    }
  }
}
```

2.3. Pardiniai duomenys ir rezultatai

3. Haskell (L3)

3.1. Darbo užduotis

713 Adding Reversed Numbers

The Antique Comedians of Malidinesia prefer comedies to tragedies. Unfortunately, most of the ancient plays are tragedies. Therefore the dramatic advisor of ACM has decided to transfigure some tragedies into comedies. Obviously, this work is very hard because the basic sense of the play must be kept intact, although all the things change to their opposites. For example the numbers: if any number appears in the tragedy, it must be converted to its reversed form before being accepted into the comedy play.

Reversed number is a number written in arabic numerals but the order of digits is reversed. The first digit becomes last and vice versa. For example, if the main hero had 1245 strawberries in the tragedy, he has 5421 of them now. Note that all the leading zeros are omitted. That means if the number ends with a zero, the zero is lost by reversing (e.g. 1200 gives 21). Also note that the reversed number never has any trailing zeros.

ACM needs to calculate with reversed numbers. Your task is to add two reversed numbers and output their reversed sum. Of course, the result is not unique because any particular number is a reversed form of several numbers (e.g. 21 could be 12, 120 or 1200 before reversing). Thus we must assume that no zeros were lost by reversing (e.g. assume that the original number was 12).

Input

The input consists of N cases. The first line of the input contains only positive integer N. Then follow the cases. Each case consists of exactly one line with two positive integers separated by space. These are the reversed numbers you are to add. Numbers will be at most 200 characters long.

Output

For each case, print exactly one line containing only one integer — the reversed sum of two reversed numbers. Omit any leading zeros in the output.

Sample Input

```
3
24 1
4358 754
305 794
```

Sample Output

```
34
1998
1
```

3.2. Programos tekstas

```haskell
module Main where
import System.IO
import Data.Data
import Data.List
import Text.Read

-- remove elements from list from 0 to n
removeEle :: Int -> [a] -> [a]
removeEle n xs
  | (n <= 0) || null xs = xs
  | otherwise = removeEle (n - 1) (tail xs)

-- map for checking if elementi is selected type
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ []     = []

mapMaybe f (x:xs) =
  case f x of
    Just y  -> y : mapMaybe f xs
    Nothing -> mapMaybe f xs

-- get file handle with selected name
getFileHandle:: String -> IO Handle
getFileHandle name = openFile name ReadMode

-- append results to selected file recursively
writeLines fileName list listOriginalLength = do
    notEnd ((length list) > 0)
  where notEnd True = do
          file <- openFile fileName AppendMode
          hPutStrLn file ( show (head list))
          hClose file
          writeLines fileName (removeEle 1 list) listOriginalLength

-- read lines from file while file is not at end
getLines :: Handle -> IO [String]
getLines hndl = do
    eof <- hIsEOF hndl
    notEnded eof
  where notEnded False =  do
          let line = hGetLine hndl
          lineConverted <- line
          rest <- getLines hndl
          return (lineConverted:rest)
        notEnded True = return []


-- split lines as words
getArray :: [String] -> IO [[String]]
getArray list = do
    notEnded (length list > 0)
  where notEnded True =  do
          let lineConverted = words (head list)
          let firstVal = if (length lineConverted) > 0 then lineConverted !! 0 else ""
          let secondVal = if (length lineConverted) > 1 then lineConverted !! 1 else ""
          let word = [firstVal, secondVal]
          let removedEl = removeEle 1 list
          let restAnsw = getArray removedEl
          answ <- restAnsw
          return (word:answ)
        notEnded False = return []
```

```haskell
-- reverse element, it works as reversing list
myReverse :: [a] -> [a]
myReverse [] = []
myReverse (x:xs) = (myReverse xs) ++ [x]


-- calculate reverse int sum
calcReverse :: [[Int]] -> IO [Int]
calcReverse list =
  notEnded ((length list) > 0) -- condition to work
  where
    notEnded True = do
      let first = head list -- list of elements
      let forReverseFirst = show (first !! 0)
      let forReverseSecond = show (first !! 1)
      let firstReverse = read (myReverse forReverseFirst) :: Int -- reverse first element
and convert to int
      let secondReverse = read (myReverse forReverseSecond) :: Int -- reverse second
element and convert to int
      let sum = firstReverse + secondReverse -- sum of two reverse elements
      let resReverse = read (myReverse $ show sum) :: Int -- reverse answer
      let datList = removeEle 1 list -- remove one element from list every time until list
doesnt have any elements
      let returnVal = calcReverse datList
      ans <- returnVal
      return (resReverse:ans) -- add las value to list
    notEnded False = do
      return []

main :: IO ()
main = do
  let myint = 1 ::Int
  --putStrLn "hello world"
  file <- getFileHandle "input.txt"
  fileForFirst <- getFileHandle "input.txt" -- reader
  lenghtFileLines <- getLines fileForFirst -- read file for first line
  hClose fileForFirst
  let filterLenghtForFirst = filter (\x -> length x > 0) lenghtFileLines -- filter first
list from emty lines
  let firstElement = words (filterLenghtForFirst !! 0) -- get first element
  let lenghtFile = if (length firstElement == 1) then (read (firstElement !! 0) :: Int)+1
-- adding one because we need to delete one value more
                   else -1 -- checks if first line is length or pair
  linesFromFile <- getLines file -- read lines from file
  let filterLenght = filter (\x -> length x > 0) linesFromFile -- filter all lines for
empty lines
  let reverseList = reverse (filterLenght) -- create reverse list for removing elements by
first element if it is given length
  let lengthRemove = (length reverseList) - lenghtFile -- calculate how many elements need
to be removed
  if lenghtFile > 0 && length reverseList > 0 then do
      let removed = removeEle lengthRemove reverseList -- remove elements from list by
calculated elements
      let reverseBack = reverse removed -- reverse back list for using
      let stringArray = getArray reverseBack
      array <- stringArray
      let filterForUse = map (filter (\x -> length x > 0 )) array -- filtering for empy
values
      let filterForUseSec = filter (\x -> length x > 1 ) filterForUse -- filtering for
empy arrays
      let filterInt = map ( mapMaybe (\x -> readMaybe x :: Maybe Int) ) filterForUseSec --
removing elements that are not numbers
      let filterForUseThird = filter (\x -> length x > 1 ) filterInt -- filtering for
empty arrays
```

```haskell
        let answer = calcReverse filterForUseThird -- use calculation function
        answ <- answer
        file <- openFile "result.txt" WriteMode -- creates file if needed
        hPutStr file "" -- rewrite file with empty value
        hClose file -- close file
        writeLines "result.txt" answ (length answ) -- use function for writing answers to
file
      else if lenghtFile <= 0 && length filterLenght > 0 then do
        let stringArray = getArray filterLenght
        array <- stringArray
        let filterForUse = map (filter (\x -> length x > 0 )) array -- filtering for empy
values
        let filterForUseSec = filter (\x -> length x > 1 ) filterForUse -- filtering for
empy arrays
        let filterInt = map ( mapMaybe (\x -> readMaybe x :: Maybe Int) ) filterForUseSec --
removing elements that are not numbers
        let filterForUseThird = filter (\x -> length x > 1 ) filterInt -- filtering for
empty arrays
        let answer = calcReverse filterForUseThird -- use calculation function
        answ <- answer
        file <- openFile "result.txt" WriteMode -- creates file if needed
        hPutStr file "" -- rewrite file with empty value
        hClose file -- close file
        writeLines "result.txt" answ (length answ) -- use function for writing answers to
file
      else print "Error"

--  something <- calculateAnswer linesFromFile
--  print $ show lenghtFile
-- check if string can be converted to int
```

3.3. Pradiniai duomenys ir rezultatai

Pradiniai duomenys	Rezultatai
3 24 1 4358 754 305 794	34 1998 1

Pradiniai duomenys	Rezultatai
3 24 1 4358 754 305 794 10 5 10 7a 1 2 10 11	34 1998 1

Pradiniai duomenys	Rezultatai
7 24 1 4358 754 305 794 10 5 10 7a 1 2 10 11	34 1998 1 6 3 21

Pradiniai duomenys	Rezultatai
24 1 4358 754 305 794 10 5 10 7a 1 2 10 11	34 1998 1 6 3 21

Pradiniai duomenys	Rezultatai
24 1 4358 754 305 794 10 5 10 7a 1 2 10 11	34 1998 1 6 8 3 21