KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Programavimo kalbų teorija (P175B124)

Laboratorinių darbų ataskaita

Atliko:

IFF-1/4 gr. studentas

Mildaras Karvelis

2023 m. gegužės 10 d.

Priėmė:

Doc. Sajavičius Svajūnas

KAUNAS 2023

TURINYS

1. C++ arba Ruby(L1)

1.1. Darbo užduotis

Guess the weights of the girls

Some school children discovered that by getting on a weighing machine in couples, and then exchanging places—one at a time—they could get the correct weight of a whole party on the payment of but one cent. They found that in couples they weighed (in pounds): 129, 125, 124, 123, 122, 121, 120, 118, 116 and 114. What was the weight of each one of the fve little girls if taken separately?

It proves that they must have been clever scholars or they never would have been able to work out the correct answer to such an interesting puzzle question, which is liable to confuse older heads than theirs.

Given a list of 10 integers, representing the weighs of each couple formed from a group of 5 people, determine the weights of each person.

Input

Input starts with a positive integer T, that denotes the number of test cases. Each test case is described by 10 integers $W_1, W_2, \ldots, W_{10}$ in a single line.

$T \leq 3000$; $100 \leq W_1 \leq W_2 \leq \ldots \leq W_{10} \leq 400$

Output

For each test case, print the case number, followed by the 5 weights asked, separated by spaces. Print these numbers in ascending order.

Sample Input

```
3
114 116 118 120 121 122 123 124 125 129
110 111 114 115 118 118 119 122 123 126
180 190 190 196 196 206 216 216 226 232
```

Sample Output

```
Case 1: 56 58 60 64 65
Case 2: 53 57 58 61 65
Case 3: 90 90 100 106 126
```

1.2. Programos tekstas

```
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>

using namespace std;
using namespace std::chrono;


class AllWeight{
    public:
```

```
        int getWeight(int i)
        {
            return weights[i];
        }

        void set(int i, int data)
        {
            weights[i] = data;
        }

        void sort()
        {
            int temp;

            for(int i=0; i<(10-1); i++)
            {
                for(int j=0; j<(10-i-1); j++)
                {
                    if(weights[j]>weights[j+1])
                    {
                        temp = weights[j];
                        weights[j] = weights[j+1];
                        weights[j+1] = temp;
                    }
                }
            }
        }


    private:

        int weights[10];
};

class FinalWeight{
    public:

        int getWeight(int i)
        {
            return weights[i];
        }

        void set(int i, int data)
        {
            weights[i] = data;
        }

        void sort()
        {
            int temp;

            for(int i=0; i<(5-1); i++)
```

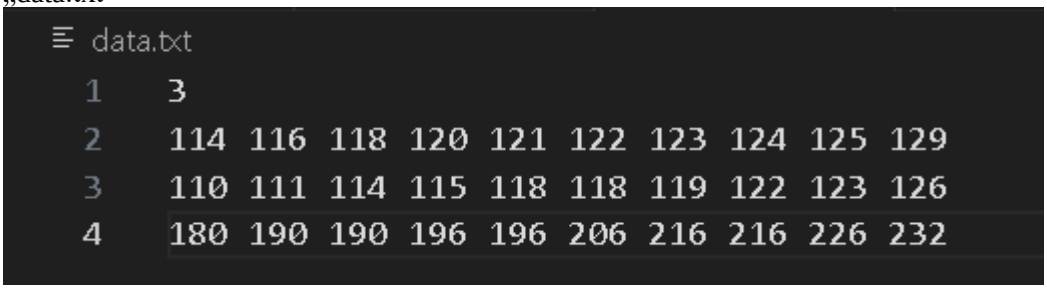
```
            {
                for(int j=0; j<(5-i-1); j++)
                {
                    if(weights[j]>weights[j+1])
                    {
                        temp = weights[j];
                        weights[j] = weights[j+1];
                        weights[j+1] = temp;
                    }
                }
            }
        }

        void printData()
        {
            for(int i = 0; i < 5; i++)
            {
                cout << weights[i] << " ";
            }
        }

    private:
        int weights[5];

};

int main()
{
    int n;
    ifstream fin;
    fstream fout;

    fin.open("data.txt");
    auto chronoStart = chrono::system_clock::now();
    fin >> n;
    AllWeight foo[n];

    int tempValue;
    for(int i = 0; i < n; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            fin >> tempValue;
            foo[i].set(j, tempValue);
        }
    }



    int bendras_sv[n] {0, 0, 0};

    for(int i = 0; i < n; i++)
    {
```

```
        for(int j = 0; j < 10; j++)
        {
            bendras_sv[i]+=foo[i].getWeight(j);
        }
        bendras_sv[i] = bendras_sv[i]/4; //rastas tikras svoris
    }

    for(int i = 0; i < n; i++)
    {
        foo[i].sort();
    }



    int aa[n]; //sunkiausia pora
    int bb[n]; //lengviausia pora
    int ab[n]; //antra pagal sunkuma pora
    int ba[n]; //antra pagal lengvuma pora

    for (int i = 0; i < n; i++)//Masyvas yra isrikiuotas todel galime naudoti tikslius skaicius
    {
        aa[i] = foo[i].getWeight(9);
        bb[i] = foo[i].getWeight(0);
        ab[i] = foo[i].getWeight(8);
        ba[i] = foo[i].getWeight(1);
    }

    FinalWeight svoriai[n];

    for(int i = 0; i < n; i++)
    {
        svoriai[i].set(0, bendras_sv[i] - aa[i] - bb[i]);
        svoriai[i].set(1, ba[i] - svoriai[i].getWeight(0));
        svoriai[i].set(2, bb[i] - svoriai[i].getWeight(1));
        svoriai[i].set(3, ab[i] - svoriai[i].getWeight(0));
        svoriai[i].set(4, aa[i] - svoriai[i].getWeight(3));
    }

    for(int i = 0; i < n; i++)
    {
        svoriai[i].sort();
    }

    fout.open("results.txt");

    for(int i = 0; i < n; i++)
    {
        fout << "Case " << i + 1 << ": ";
        for(int j = 0; j < 5; j++)
        {
            fout << svoriai[i].getWeight(j) << " ";
        }
        fout << endl;
```

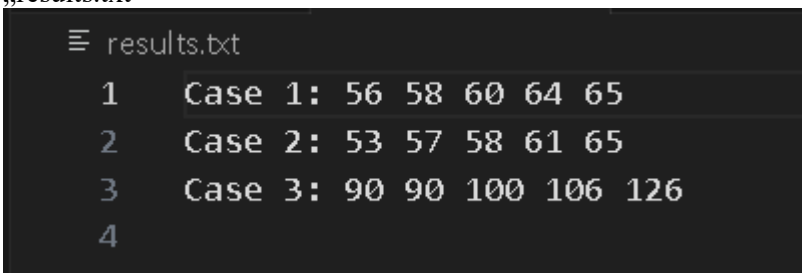
```
    }



     //Ends the clock
    auto chronoEnd = chrono::system_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(chronoEnd -
chronoStart);
    //Writes time taken
    cout << "Program Finished" << endl;
    cout << "Time Elapsed in microseconds: " << elapsed.count() << endl;


    fin.close(); fout.close();
    return 0;
}
```

1.3. Pradiniai duomenys ir rezultatai

„data.txt“



„results.txt“



2. Scala (L2)

2.1. Boto kodas

```
import scalatron.botwar.botPlugin._

object ControlFunction
{
   import util.Random
   val rnd = new Random()

   def forMaster(bot: Bot) {
      //Analyzes grid
      val (enemiesInVision, myOwnSlavesInVision, directionValue, nearestEnemyMaster, nearestEnemySlave,
nearestBeast) = analyzeViewAsMaster(bot.view)

      //Inputs
      val dontFireAggressiveMissileUntil = bot.inputAsIntOrElse("dontFireAggressiveMissileUntil", -1)
      val dontFireDefensiveMissileUntil = bot.inputAsIntOrElse("dontFireDefensiveMissileUntil", -1)
```

```
      val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)

      // Determine movement direction
      directionValue(lastDirection) += 10 // try to break ties by favoring the last direction
      val bestDirection45 = directionValue.zipWithIndex.maxBy(_._1)._2
      val direction = XY.fromDirection45(bestDirection45)
      bot.move(direction)
      bot.set("lastDirection" -> bestDirection45)

      // fire attack missile?
      if(dontFireAggressiveMissileUntil < bot.time && bot.energy > 100) {
         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
      // fire defensive missile?
      if(dontFireDefensiveMissileUntil < bot.time && bot.energy > 100) {
         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))
               }
         }
      }

      // Make a slave to defend against beasts?
      if(enemiesInVision >= 3 && bot.energy >= 100 && myOwnSlavesInVision < 5) {
         nearestBeast match {
            case None =>          // no-on nearby
            case Some(relPos) =>  // a beast is nearby
               if(relPos.stepCount < 5) {
                  // A breast is close, make a slave
                  val chance1 = rnd.nextInt(3)
                  var chance2 = rnd.nextInt(3)-1
                  val chance3 = rnd.nextInt(3)-1
                  if(chance2 == 0 && chance3 == 0){
                     chance2 = 1
                  }
                  chance1 match {
                     case 0 =>
                        bot.spawn(XY(chance2,chance3), "mood" -> "Mine")
                     case 1 =>
                        bot.spawn(XY(chance2,chance3), "mood" -> "Bomb")
                     case 2 =>
                        bot.spawn(XY(chance2,chance3), "mood" -> "Bait")
                     case default =>
                        throw new IllegalStateException("Error: " + default)
```

```
                }
                bot.say("Made a new slave")
            }
        }
    }

    // Logs
    bot.log("Owned slaves on screen - " + myOwnSlavesInVision.toString)
    bot.log("Number of enemies on screen - " + enemiesInVision.toString)
}

def forSlave(bot: MiniBot) {
    bot.inputOrElse("mood", "Lurking") match {
        case "Bomb" => reactAsBomb(bot)
        case "Mine" => reactAsMine(bot)
        case "Bait" => reactAsBait(bot)
        case "Aggressive" => reactAsAggressiveMissile(bot)
        case "Defensive" => reactAsDefensiveMissile(bot)
        case s: String => bot.log("Error: " + s)
    }
}

def reactAsBait(bot: MiniBot) {
    val dx = rnd.nextInt(3)-1
    val dy = rnd.nextInt(3)-1
    bot.move(XY(dx,dy))
}

def reactAsBomb(bot: MiniBot) {
    bot.explode(4)
}

def reactAsMine(bot: MiniBot) {
    bot.view.offsetToNearest('b') match {
        case Some(delta: XY) =>
            if(delta.length <= 2) {
                bot.explode(4)
            }
    }
}

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
               val remainder = target - unitDelta // e.g. = CellPos(-7,5)
               bot.set("target" -> remainder)
            } else {
               // yes -- but we did not detonate yet, and are not pursuing anything?!? => switch purpose
               bot.set("mood" -> "Lurking", "target" -> "")
               bot.say("Lurking")
            }
      }
   }


   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)
            } else {
               // yes -- but we did not annihilate yet, and are not pursuing anything?!? => switch purpose
               bot.set("mood" -> "Lurking", "target" -> "")
               bot.say("Lurking")
            }
      }
   }


   /** Analyze the view, building a map of attractiveness for the 45-degree directions and
     * recording other relevant data, such as the nearest elements of various kinds.
     */
  def analyzeViewAsMaster(view: View) = {
      //Enemies in vision
      var enemiesInVision = 0;
      var myOwnSlavesInVision = 0;
      //Directional values
      val directionValue = Array.ofDim[Double](8)
      //Nearest dangers
```

```
var nearestEnemyMaster: Option[XY] = None
var nearestEnemySlave: Option[XY] = None
var nearestBeast: Option[XY] = None

//Take all cells
val cells = view.cells
val cellCount = cells.length

for(i <- 0 until cellCount) {
   val currentCellRelPos = view.relPosFromIndex(i)
   if(currentCellRelPos.isNonZero) {
      val stepDistance = currentCellRelPos.stepCount
      val value: Double = cells(i) match {
         case 'm' => // another master: not dangerous, but an obstacle
            nearestEnemyMaster = Some(currentCellRelPos)
            enemiesInVision = 1
            if(stepDistance < 7) -2500 else 0
         case 's' => // another slave: potentially dangerous?
            nearestEnemySlave = Some(currentCellRelPos)
            enemiesInVision += 1
            -1000 / stepDistance
         case 'S' => // out own slave
            myOwnSlavesInVision += 1
            10.0
         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
            nearestBeast = Some(currentCellRelPos)
            enemiesInVision += 1
            if(stepDistance < 5) -600 / stepDistance else -200 / stepDistance

         case 'p' => // bad plant: bad, but only if I step on it
            if(stepDistance < 3) -1000 else 0

         case 'W' => // wall: harmless, just don't walk into it
            if(stepDistance < 4) -1000 else 0

         case _ => 0.0
      }
      val direction45 = currentCellRelPos.toDirection45
      directionValue(direction45) += value
   }
}
//BFS path finding algorithm
import java.util.ArrayDeque
if (cells.contains('B') || cells.contains('P')) {
   //Make an array to store distance
   val distanceArray = Array.ofDim[Double](cellCount)
   for (i <- 0 until cellCount) {
      cells(i) match {
         case 'P' =>
```

```
                //Saves the distance to a plant
                val cellRelPos = view.relPosFromIndex(i)
                if (cellRelPos.isNonZero) {
                  val stepDistance = cellRelPos.stepCount
                  distanceArray(i) += stepDistance
                }
                //Saves the distance to a beast
              case 'B' =>
                val cellRelPos = view.relPosFromIndex(i)
                if (cellRelPos.isNonZero) {
                  val stepDistance = cellRelPos.stepCount
                  distanceArray(i) += stepDistance
                }
              case default =>
            }
          }
          //Find the destination
          val destination = view.relPosFromIndex(distanceArray.indexOf(distanceArray.filter(_ != 0).min))
          //Directions
          val dirVector = Vector(XY.Up, XY.RightUp, XY.Right, XY.DownRight, XY.Down, XY.LeftDown,
XY.Left, XY.UpLeft)
          //an empty queue to store a list of positions that represent the path to reach the destination
          val aq = new ArrayDeque[List[XY]]()
          //a list of positions as the shortest path
          var path: Option[List[XY]] = None
          //current position of the agent
          var visited = Set[XY](XY.Zero)
          //adds the starting position
          aq.add(List(XY.Zero))
          // looping the queue while its not empty and the path variable is still 'None'
          while (!aq.isEmpty && path.isEmpty) {
            val steps@(point :: _) = aq.pop()
            //Check all posible directions
            for (dir <- dirVector) {
              //Gets the new location
              val dirTarget = point + dir
              val cell = view.cellAtRelPos(dirTarget)
              //If nothing is blocking the way
              if (cell != 'W' && cell != 'm' && cell != 's' && cell != 'b' && cell != 'p' && !visited(dirTarget))
              {
                //Add it to the queue or the path if the destination is reaced
                if (dirTarget == destination)
                {
                  path = Some(dirTarget :: steps)
                }
                else
                {
                  visited += dirTarget
                  aq.add(dirTarget :: steps)
                }
              }
            }
          }
          if(path.get.size < 5){
            directionValue(path.get(path.get.size - 2).toDirection45) += 10000
          }
          else if(path.get.size < 8){
            directionValue(path.get(path.get.size - 2).toDirection45) += 5000
          }
```

```
          else{
            directionValue(path.get(path.get.size - 2).toDirection45) += 2500
          }
        }
        (enemiesInVision, myOwnSlavesInVision, directionValue, nearestEnemyMaster, nearestEnemySlave,
nearestBeast)
    }
}


// -------------------------------------------------------------------------------------------------
// 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
        }
    }
}


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


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

```
    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)
      }
    }
}
```

2.2. Išvada

Naudotas BFS kelio radimo algoritmo botas yra greitesnis už scalatron kūrėjo sukurtą botą.

3. Haskell (Lab2)

3.1. Užduotis

Tri-du is a card game inspired in the popular game of Truco. The game uses a normal deck of 52 cards, with 13 cards of each suit, but suits are ignored. What is used is the value of the cards, considered as integers between 1 to 13.

In the game, each player gets three cards. The rules are simple:

- A Three of a Kind (three cards of the same value) wins over a Pair (two cards of the same value).
- A Three of a Kind formed by cards of a larger value wins over a Three of a Kind formed by cards of a smaller value.
- A Pair formed by cards of a larger value wins over a Pair formed by cards of a smaller value.

Note that the game may not have a winner in many situations; in those cases, the cards are returned to the deck, which is re-shuffled and a new game starts.

A player received already two of the three cards, and knows their values. Your task is to write a program to determine the value of the third card that maximizes the probability of that player winning the game.

Input

The input contains several test cases. In each test case, the input consists of a single line, which contains two integers A ($1 \leq A \leq 13$) and B ($1 \leq B \leq 13$) that indicates the value of the two first received cards.

Output

For each test case in the input, your program must produce a single line, containing exactly one integer, representing the value of the card that maximizes the probability of the player winning the game.

Sample Input

```
10 7
2 2
```

Sample Output

```
10
2
```

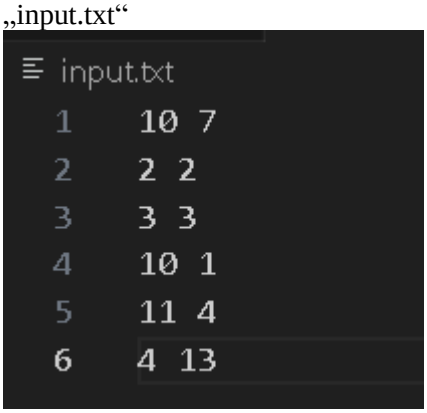
3.2. Programos kodas

```
import System.IO

maxCard :: Int -> Int -> Int
maxCard a b
  | a == b = a
  | otherwise = max a b

main :: IO ()
main = do
   contents <- getContents
   let cases = lines contents
   let results = map (\line -> let [a, b] = map read (words line) in maxCard a b) cases
```

mapM_ print results

3.3. Rezultatai

„input.txt“



Output terminale

```
PS C:\Users\milda\OneDrive\Stalinis kompiuteris\Haskel> Get-Content .\input.txt | .\TriDu
10
2
3
10
11
13
PS C:\Users\milda\OneDrive\Stalinis kompiuteris\Haskel>
```