KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

Programavimo kalbų teorija (P175B124)

Laboratorinių darbų ataskaita

Atliko:

IFF-1/2 gr. studentas

Lukas Borinskij

2023 m. balandžio 27 d.

Priėmė:

Doc. Aštrys Kirvaitis

Lekt. Tautvydas Fyleris

KAUNAS 2023

TURINYS

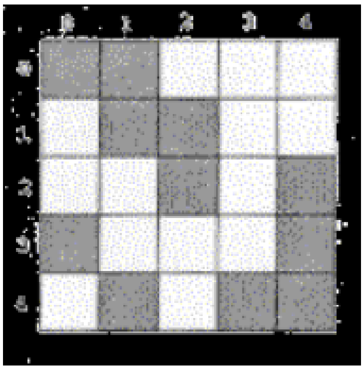
1. C++ arba Ruby (L1)

1.1. *Darbo užduotis Couting Cells in a Blob*

871 Counting Cells in a Blob

Consider a two-dimensional grid of cells, each of which may be empty or filled. Filled cells form blobs. The filled cells that are connected form the same bigger blob. Two cells are said to be connected if they are adjacent to each other horizontally, vertically, or diagonally. There may be several blobs on the grid. Your job is to find the largest blob (in terms of number of cells) on the grid.

Write a program that determines the size of the largest blob for a given set of blobs.



The figure illustrates a grid with 3 blobs (the largest contains 5 cells).

Input

The input begins with a single positive integer on a line by itself indicating the number of the cases following, each of them as described below. This line is followed by a blank line, and there is also a blank line between two consecutive inputs.

The grid is given as a set of strings, each composed of `0`'s and `1`'s. The '`1`' indicates that the cell is filled and '`0`' indicates an empty cell. The strings should be converted into the grid format.

The largest grid that sould be considered is a 25×25 grid.

Output

For each test case, the output must follow the description below. The outputs of two consecutive cases will be separated by a blank line.

The output is the size of the largest blob found on the grid.

Sample Input

```
1

11000
01100
00101
10001
01011
```

Sample Output

```
5
```

1.2. *Programos tekstas*

```
#include <iostream>
#include <algorithm>
#include <fstream>
#include <vector>
```

```cpp
#include <string>
#include <chrono>


using namespace std;
//Demes klase
class Blob
{
private :
	int id;
	int size;
public :
	Blob()
	{

	}
	Blob()
	{
		size = 0;
	}

	void addCell()
	{
		size++;
	}

	int getSize() const
	{
		return size;
	}


};

//tinklelio klase
class Grid

{
private :
	const int N = 25;
	vector<vector<int>> cells; //langeliai
	vector<vector<bool>> visited; //aplankyti

	int numRows, numCols; //eiluciu sk ir stulpeliu sk
	 //Paieška į gylį (Depth-first search)
	int dfs(int row, int col)
	{
		int count = 1;
		visited[row][col] = true;



		// Patikrinkite visas gretimas langelius
		for (int i = row - 1; i <= row + 1; i++) {
			for (int j = col - 1; j <= col + 1; j++) {
				if (i >= 0 && i < numRows && j >= 0 && j < numCols &&
!visited[i][j] && cells[i][j] == 1) {
					count += dfs(i, j);
				}
			}
		}

		return count;
	}
public:
	Grid()
	{
```

```
	}
	Grid(const vector<string>& input)
	{
		numRows = input.size();
		numCols = input[0].length();
		cells.resize(numRows, vector<int>(numCols));
		visited.resize(numRows, vector<bool>(numCols, false));

		// Tinklelio inicijacija
		for (int i = 0; i < numRows; i++) {
			for (int j = 0; j < numCols; j++) {
				cells[i][j] = input[i][j] - '0';
			}
		}
	}
	//Surandame dydziausai deme
	int findLargestBlob()
	{
		int largestBlobSize = 0;

		for (int i = 0; i < numRows; i++) {
			for (int j = 0; j < numCols; j++) {
				if (!visited[i][j] && cells[i][j] == 1) {
					Blob blob;
					int blobSize = dfs(i, j);
					largestBlobSize = max(largestBlobSize, blobSize);
				}
			}
		}

		return largestBlobSize;
	}
};

class InOutUtils {
public:



};
	static vector<string> readDataFromFile(string filename)
	{
		vector<string> data;
		ifstream infile(filename);

		if (infile)
		{
			string line;
			while (getline(infile, line))
			{
				data.push_back(line);
			}
		}
 else
	{
  cerr << "ivyko klaida." << endl;

	}


return data;
	}



	static void writeDataToFile(int data, string filename) {
		ofstream outfile(filename);
```

```
		if (outfile) {
			outfile << data << endl;
		}
		else {
			cerr << "Klaida" << endl;
		}
	}

	static void printDataToConsole(vector<string> data) {
		for (string line : data) {
			cout << line << endl;
		}
	}
;


int main()
{

	auto start = chrono::steady_clock::now();


	chrono::time_point<std::chrono::system_clock> start, end;
	auto start = chrono::steady_clock::now();
	vector<Blob> blobs;
	vector<Grid> grid;

	int numCases;
	cin >> numCases;

	for (int i = 0; i < numCases; i++) {
		if (i > 0) {
			cout << endl; // isvesti atskiriame tuscia eilute
		}

		vector<string> input;
		string line;

		// skaitome inputa

		while (getline(cin, line)) {
			if (line.empty()) {
				break;
			}

			input.push_back(line);
		}

		Grid grid(input);
		int largestBlobSize = grid.findLargestBlob();
		cout << largestBlobSize << endl;

		auto end = chrono::steady_clock::now();
		chrono::duration<double> elapsed_seconds = end - start;
		cout << "Elapsed time: " << elapsed_seconds.count() << "s\n";


	}

	return 0;
}
```

1.3. Pradiniai duomenys ir rezultatai

Pradiniai duomenys	Rezultatai
1 11000 01100 00101 10001 01011	5

2. Scala (L2)

2.1. Programos tekstas

```
package scalatron.botwar.botPlugin.lukbor
import scala.collection.mutable.Queue
object ControlFunction {
 def forMaster(bot: Bot) {
 val (directionValue, nearestEnemyMaster, nearestEnemySlave, nearestBadBeast, nearestGoodFruit)
= analyzeViewAsMaster(bot.view)
 val dontFireAggressiveMissileUntil = bot.inputAsIntOrElse("dontFireAggressiveMissileUntil", -1)
 val dontFireDefensiveMissileUntil = bot.inputAsIntOrElse("dontFireDefensiveMissileUntil", -1)
 val dontFireMineUnitUntil = bot.inputAsIntOrElse("dontFireMineUnitUntil", -1)
 val dontFireKamikazeUnitUntil = bot.inputAsIntOrElse("dontFireKamikazeUnitUntil", -1)
 val lastDirection = bot.inputAsIntOrElse("lastDirection", 0)
 val bfsDirection = BFS(bot, bot.view)
 if(bfsDirection == XY(0,0)) {
 // determine movement direction
 directionValue(lastDirection) += 15 // 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)
 } else{
 bot.move(bfsDirection)
 }
 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))
}
}
}
//////////////////////////////////////////////////////////////////////////////////////////////////////
if (dontFireMineUnitUntil < bot.time && bot.energy > 100) {
nearestBadBeast match {
case Some(relPos) =>
val unitDelta = relPos.signum
val remainder = relPos - unitDelta
bot.spawn(unitDelta, "mood" -> "Mine", "target" -> remainder)
bot.set("dontFireMineUnitUntil" -> (bot.time + relPos.stepCount + 1))
case None =>
}
}
if (dontFireKamikazeUnitUntil < bot.time && bot.energy > 100) {
nearestGoodFruit match {
case Some(relPos) =>
val unitDelta = relPos.signum
val near = relPos - unitDelta
bot.spawn(unitDelta, "mood" -> "Kamikaze", "target" -> near)
bot.set("dontFireKamikazeUnitUntil" -> (bot.time + relPos.stepCount + 1))
case None =>
}
}
//////////////////////////////////////////////////////////////////////////////////////////////////////
}
def forSlave(bot: MiniBot) {
bot.inputOrElse("mood", "Lurking") match {
case "Aggressive" => reactAsAggressiveMissile(bot)
case "Defensive" => reactAsDefensiveMissile(bot)
//////////////////////////////////////////////////////////////
case "Mine" => reactAsMineUnit(bot)
case "Kamikaze" => reactAsKamikazeUnit(bot)
//////////////////////////////////////////////////////////////
case s: String => bot.log("unknown mood: " + s)
}
}
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")
}
}
}
/////////////////////////////////////////////////////////////////////////////////////////////////
def reactAsKamikazeUnit(bot: MiniBot) = {
bot.view.offsetToNearest('b') match {
case Some(delta: XY) =>
if (delta.length <= 1) {
bot.explode(2)
}
bot.move(delta)

bot.set("rx" -> delta.x, "ry" -> delta.y)
case None =>
}
}
def reactAsMineUnit(bot: MiniBot) = {
bot.view.offsetToNearest('s') match {
case Some(delta: XY) =>
if (delta.length <= 3) {
bot.explode(5)
}
case None =>
}
bot.view.offsetToNearest('b') match {
case Some(delta: XY) =>
if (delta.length <= 3) {
```

```
bot.explode(5)
}
case None =>
}
}
/////////////////////////////////////////////////////////////////////////////////////////
def BFS(bot: Bot, view: View): XY = {
var visited = Set[XY]() // already visited
val queue = new Queue[XY]()
var path = Map[XY, XY]() // road
val length = view.size / 2 // trange
queue.enqueue(XY(0,0)) // queue starting location
while (queue.nonEmpty) {
val current = queue.dequeue()
for (x <- -1 to 1; y <- -1 to 1)
{
if (!(x == 0 && y == 0)) {
val newXY = current + XY(x, y) // x + dx, y + dy
if ( newXY.x > -length && newXY.x < length && newXY.y > -length && newXY.y < length) // maze
length
{
val neighbour = view(newXY)
if ( neighbour != 'W' && neighbour != 's' && neighbour != 'b' && neighbour != 'p')
{
if (!visited.contains(newXY)) // if valid cell not found
{
queue.enqueue(newXY) // not visited
visited += newXY // no doubles
path += (newXY -> current) // path
if ((bot.generation != 0 && bot.energy >= 1000 && neighbour == 'M') || neighbour == 'P' ||
neighbour == 'B' )
{
var result = newXY // if true
while (path(result) != XY(0, 0)) // so path length != 0
{
result = path(result) // path
}
return result
}
}
}
}

}
}
}
XY(0, 0)
}
/** 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) = {
val directionValue = Array.ofDim[Double](8)
var nearestEnemyMaster: Option[XY] = None
var nearestEnemySlave: Option[XY] = None
var nearestBadBeast: Option[XY] = None
var nearestGoodFruit: Option[XY] = None
val cells = view.cells
val cellCount = cells.length
```

```
 for (i <- 0 until cellCount) {
 val cellRelPos = view.relPosFromIndex(i)
 if (cellRelPos.isNonZero) {
 val stepDistance = cellRelPos.stepCount
 val value: Double = cells(i) match {
 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
 nearestGoodFruit = Some(cellRelPos)
 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
 nearestBadBeast = Some(cellRelPos)
 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 < 2) -1000 else 0
 case _ => 0.0
 }
 val direction45 = cellRelPos.toDirection45
 directionValue(direction45) += value
 }
 }
 (directionValue, nearestEnemyMaster, nearestEnemySlave, nearestBadBeast, nearestGoodFruit)
 }
}

// -------------------------------------------------------------------------------------------------
// 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)
 }
 }
}
```

3. Haskell

3.1. darbo užduotis

Problems in Computer Science are often classified as belonging to a certain class of problems (e.g., NP, Unsolvable, Recursive). In this problem you will be analyzing a property of an algorithm whose classification is not known for all possible inputs.

Consider the following algorithm:

1. input n
2. print n
3. if $n = 1$ then STOP
4. if n is odd then $n \longleftarrow 3n + 1$
5. else $n \longleftarrow n/2$
6. GOTO 2

Given the input 22, the following sequence of numbers will be printed

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

It is conjectured that the algorithm above will terminate (when a 1 is printed) for any integral input value. Despite the simplicity of the algorithm, it is unknown whether this conjecture is true. It has been verified, however, for all integers n such that $0 < n < 1,000,000$ (and, in fact, for many more numbers than this.)

Given an input n, it is possible to determine the number of numbers printed before and including the 1 is printed. For a given n this is called the *cycle-length* of n. In the example above, the cycle length of 22 is 16.

For any two numbers i and j you are to determine the maximum cycle length over all numbers between and including both i and j.

Input

The input will consist of a series of pairs of integers i and j, one pair of integers per line. All integers will be less than 10,000 and greater than 0.

You should process all pairs of integers and for each pair determine the maximum cycle length over all integers between and including i and j.

You can assume that no operation overflows a 32-bit integer.

Output

For each pair of input integers i and j you should output i, j, and the maximum cycle length for integers between and including i and j. These three numbers should be separated by at least one space with all three numbers on one line and with one line of output for each line of input. The integers i and j must appear in the output in the same order in which they appeared in the input and should be followed by the maximum cycle length (on the same line).

Sample Input

```
1 10
100 200
201 210
900 1000
```

Sample Output

```
1 10 20
100 200 125
201 210 89
900 1000 174
```

3.2. Programos tekstas

```haskell
import Data.Array
import System.IO
import Control.Monad
import Data.Maybe
--paima sveikąjį skaičių n ir grąžina žingsnių skaičių, reikalingą norint pasiekti 1 Collatz sekoje, pradedant nuo n.
collatzSteps :: Int -> Int
collatzSteps 1 = 0
collatzSteps n =  1 +  collatzSteps (collatz n)

--taikome reikalinga taisykle
collatz :: Int -> Int
collatz n | even n     = n `div` 2
          | otherwise = 3 * n + 1


collatzMax :: (Int,Int) -> [(Int, Int)]
collatzMax (a, n) = map (\x -> (collatzSteps x, x)) [a .. n]

getMax :: (Int,Int) -> Int
getMax (a,n) = fst(maximum (collatzMax (a,n))) + 1

getData :: [String] -> Maybe (Int,Int)
getData [x,y] = do
                let a = read x :: Int
                let b = read y :: Int
                return (a,b)
--Spausdiname
printData :: [String] -> [Int] -> String
printData [] [] = ""
printData [x] [y] = (show x) ++ " " ++ (show y)
printData (x:xs) (y:ys) =
        (show x) ++ " " ++ (show y) ++ "\n" ++ printData xs ys

--mainas iskvieciame funkcijas
main = do
        contents <- readFile "data.txt"
        let beleka = lines contents
        let beleka2 = [words n | n <- beleka]
        let beleka3 = [getData n | n <- beleka2]
        let beleka4 = [getMax (fromMaybe (1,1) n) | n <- beleka3]
        let output = printData beleka beleka4
        putStrLn $ filter (/='"') output
```

3.3. *Pradiniai duomenys ir rezultatai*

```
A1 10
100 200
201 210
900 1000
```

Rez →

```
1 10 20
100 200 125
201 210 89
900 1000 174
```