KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

Programavimo kalbų teorija (P175B124)

Laboratorinių darbų ataskaita

Atliko:

IFF-1/1 gr. studentas

Vytenis Kriščiūnas

2023 m. balandžio 25 d.

Priėmė:

Doc. Dr. Svajūnas Sajavičius

KAUNAS 2023

TURINYS

1. C++20 arba Ruby (L1) 3

1.1. Darbo užduotis 3

1.2. Programos tekstas 3

1.3. Pradiniai duomenys ir rezultatai 5

2. Scala (L2) 6

2.1. Darbo užduotis 6

2.2. Programos tekstas 7

3. F# (L3) 16

3.1. Darbo užduotis 16

3.2. Programos tekstas 17

3.3. Pradiniai duomenys ir rezultatai 17

4. Prolog (L4) 17

# C++20 arba Ruby (L1)

## Darbo užduotis

278 Chess

Almost everyone knows the problem of putting eight queens on an 8×8 chessboard such that no Queen can take another Queen. Jan Timman (a famous Dutch chessplayer) wants to know the maximum number of chesspieces of one kind which can be put on an m × n board with a certain size such that no piece can take another. Because it’s rather difficult to find a solution by hand, he asks your help to solve the problem.

He doesn’t need to know the answer for every piece. Pawns seems rather uninteresting and he doesn’t like Bishops anyway. He only wants to know how many Rooks, Knights, Queens or Kings can be placed on one board, such that one piece can’t take any other.

Input

The first line of input contains the number of problems. A problem is stated on one line and consists of one character from the following set ‘r’, ‘k’, ‘Q’, ‘K’, meaning respectively the chesspieces Rook, Knight, Queen or King. The character is followed by the integers m (4 ≤ m ≤ 10) and n (4 ≤ n ≤ 10), meaning the number of rows and the number of columns or the board.

Output

For each problem specification in the input your program should output the maximum number of chesspieces which can be put on a board with the given formats so they are not in position to take any other piece.

Note: The bottom left square is 1, 1.

Sample Input

2

r 6 7

k 8 8

Sample Output

6

32

## Programos tekstas

#include <fstream>

#include <iostream>

#include <format>

#include <chrono>

#include <list>

using namespace std;

class Piece

{

public:

char Name;

int M, N;

Piece(char name, int m, int n) : Name(name), M(m), N(n) {}

};

class Pieces

{

private:

list<Piece> pieces;

int Number;

public:

Pieces(int number = 0) : Number(number){}

void Add(Piece piece)

{

pieces.push\_back(piece);

}

Piece Get(int index)

{

list<Piece>::iterator it = pieces.begin();

advance(it, index);

return \*it;

}

int GetNumber()

{

return Number;

}

};

class InOutUtils

{

public:

static Pieces Read(const string fileName)

{

int number, m, n;

char name;

ifstream file(fileName);

if (!file.is\_open())

{

cerr << "Unable to open a file " << fileName << endl;

return 0;

}

file >> number;

Pieces pieces = Pieces(number);

for (int i = 0; i < number; i++)

{

file >> name >> m >> n;

Piece piece = Piece(name, m, n);

pieces.Add(piece);

}

file.close();

return pieces;

}

static void Write(const string fileName, list<int> totalCounts)

{

ofstream file(fileName);

if (!file.is\_open())

{

cerr << "Unable to open a file " << fileName << endl;

return;

}

for (int count : totalCounts)

{

file << count << endl;

}

file.close();

}

};

class TaskUtils

{

public:

static list<int> Calculate(Pieces pieces)

{

list<int> totalCounts;

for (int i = 0; i < pieces.GetNumber(); i++)

{

Piece piece = pieces.Get(i);

if (piece.Name == 'r' || piece.Name == 'Q')

{

totalCounts.push\_back(piece.M);

}

else if (piece.Name == 'K')

{

int m = piece.M / 2 + piece.M % 2;

int n = piece.N / 2 + piece.N % 2;

totalCounts.push\_back(m \* n);

}

else

{

int m1 = piece.M / 2 + piece.M % 2;

int m2 = piece.M / 2;

int n1 = piece.N / 2 + piece.N % 2;

int n2 = piece.N / 2;

int rezult = m1 \* n1 + m2 \* n2;

totalCounts.push\_back(rezult);

}

}

return totalCounts;

}

};

int main()

{

typedef chrono::high\_resolution\_clock Time;

typedef chrono::duration<float> duration;

auto start = Time::now();

Pieces pieces = InOutUtils::Read("inputFile.txt");

list<int> totalCounts = TaskUtils::Calculate(pieces);

InOutUtils::Write("Rezults.txt", totalCounts);

auto stop = Time::now();

duration totalTime = chrono::duration\_cast<chrono::microseconds>(stop - start);

cout << "Duration: " << totalTime.count() << " ms." <<endl;

return 0;

}

## Pradiniai duomenys ir rezultatai

inputFile.txt tekstas:

24

K 8 10

K 7 9

K 7 7

K 8 8

K 6 10

K 5 10

Q 8 10

Q 7 9

Q 7 7

Q 8 8

Q 6 10

Q 5 10

k 8 10

k 7 9

k 7 7

k 8 8

k 6 10

k 5 10

r 8 10

r 7 9

r 7 7

r 8 8

r 6 10

r 5 10

Rezults.txt tekstas:

20

20

16

16

15

15

8

7

7

8

6

5

40

32

25

32

30

25

8

7

7

8

6

5

Konsolės rezultatai:

Duration: 0.002962 ms.

# Scala (L2)

## Darbo užduotis

**Aprašymas**

Antroje užduotyje pradedame mokytis funkcinę/objektinę kalbą "Scala". http://www.scala-lang.org/  
Jos kompiliatorių rasite virtualioje mašinoje.

Naudosime programavimo įrankį / žaidimo kūrimo imitatorių "Scalatron", parsisiųsti galite iš: http://scalatron.github.io  
Užduotis: atsiųsti, įsidiegti ir naudojantis Scalatron API Scala kalba parašyti savo "bot'ą".

Scalatron`ą galima pasileisti su komanda:  
java -server -jar Scalatron.jar -x 100 -y 100 -steps 1000 -maxfps 1000

Galima naudotis visa medžiaga ir pateikiamais kodo pavyzdžiais.   
Žaidime pateikiamas "reference" (pavyzdinis/etaloninis) botas, nuo kurio galima pradėti programuoti.  
Rekomenduojama pereiti visas naršyklėje pateikiamas Scalatron pamokas (tutorials).

**Reikalavimai programai/botui**

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).

## Programos tekstas

// Example Bot #1: The Reference Bot

/\*\* This bot builds a 'direction value map' that assigns an attractiveness score to

\* each of the eight available 45-degree directions. Additional behaviors:

\* - aggressive missiles: approach an enemy master, then explode

\* - defensive missiles: approach an enemy slave and annihilate it

\*

\* The master bot uses the following state parameters:

\* - dontFireAggressiveMissileUntil

\* - dontFireDefensiveMissileUntil

\* - lastDirection

\* The mini-bots use the following state parameters:

\* - mood = Aggressive | Defensive | Lurking

\* - target = remaining offset to target location

\*/

object ControlFunction

{

def forMaster(bot: Bot) {

val (directionValue, nearestEnemyMaster, nearestEnemySlave) = analyzeViewAsMaster(bot.view)

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)

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))

}

}

}

val rand = new scala.util.Random

//Spawns resource colector

if (bot.energy > 100 && bot.time % 100 > 24) {

bot.spawn(XY(1, 0), "mood"-> "Hungry")

}

}

def forSlave(bot: MiniBot) {

bot.inputOrElse("mood", "Lurking") match {

case "Hungry" => reactAsColectorBot(bot)

case "Aggressive" => reactAsAggressiveMissile(bot)

case "Defensive" => reactAsDefensiveMissile(bot)

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

}

}

def reactAsColectorBot(bot: MiniBot) {

val BFS = bot.view.BFS match {

case None => bot.offsetToMaster.signum

case Some(delta: XY) => delta.signum

}

bot status("Tasty")

bot.move(BFS)

}

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) = {

val directionValue = Array.ofDim[Double](8)

var nearestEnemyMaster: Option[XY] = None

var nearestEnemySlave: 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

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 < 2) -1000 else 0

case \_ => 0.0

}

val direction45 = cellRelPos.toDirection45

directionValue(direction45) += value

}

}

(directionValue, nearestEnemyMaster, nearestEnemySlave)

}

}

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

// 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)

}

}

def BFS() = {

var X = Array(0, 1, 1, 1, 0, -1, -1, -1)

var Y = Array(-1, -1, 0, 1, 1, 1, 0, -1)

var QueueV = scala.collection.mutable.Queue[XY]()

val listOfcells = cells.grouped(size).toList

var visited = scala.collection.mutable.Map[XY, Boolean]()

for(i<- 0 to size)

for(j<- 0 to size)

visited += (XY(i, j) -> false)

QueueV.enqueue(center)

var foodItem = false;

var v = center;

while(!foodItem && !QueueV.isEmpty){

v = QueueV.dequeue

if (listOfcells(v.y) (v.x) == 'P' || listOfcells(v.y)(v.x)=='B')

foodItem = true;

if (!foodItem) {

var i = 0;

while(i != 7)

{

i = i+1;

if (v.x + X(i) > 0 && v.x + X(i) < size && v.y + Y(i) > 0 && v.y + Y(i) < size && visited(XY(v.x + X(i), v.y + Y(i))) == false) {

visited(XY(v.x + X(i), v.y + Y(i))) = true

QueueV.enqueue(XY(v.x + X(i), v.y + Y(i)))

}

}

}

}

if (foodItem)

Some(XY(v.x - size / 2, v.y - size / 2))

else

None

}

}

# F# (L3)

## Darbo užduotis

10127 Ones

Given any integer 0 ≤ n ≤ 10000 not divisible by 2 or 5, some multiple of n is a number which in decimal notation is a sequence of 1’s. How many digits are in the smallest such a multiple of n?

Input

A file of integers at one integer per line.

Output

Each output line gives the smallest integer x > 0 such that p = ∑x−1 i=0 1 × 10i = a × b, where a is the corresponding input integer, and b is an integer greater than zero.

Sample Input

3

7

9901

Sample Output

3

6

12

## Programos tekstas

let findSmallestMultipleWithOnesSeq n =

let rec findSmallestMultipleWithOnesSeq' remainder length =

match remainder with

| 0 -> length

| \_ ->

let nextRemainder = (remainder \* 10 + 1) % n

findSmallestMultipleWithOnesSeq' nextRemainder (length + 1)

findSmallestMultipleWithOnesSeq' 1 1

let inputLines = System.IO.File.ReadAllLines("input.txt")

let outputLines =

inputLines

|> Array.map int

|> Array.map findSmallestMultipleWithOnesSeq

|> Array.map string

System.IO.File.WriteAllLines("output.txt", outputLines)

## Pradiniai duomenys ir rezultatai

input.txt tekstas

3

7

9901

Output.txt tekstas

3

6

12

# Prolog (L4)