Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Основы алгоритмизации и программирования (ОАиП)

ОТЧЕТ

по лабораторной работе №5

Тема работы: Работа с многомерными массивами

Выполнил

студент: гр. 251003 Панкратьев Е.С.

Проверил: Фадеева Е.П.

Минск 2022

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc121595374)

[2 Методика решения 4](#_Toc121595375)

[2.1 Описание оператора try..except..end 4](#_Toc121595376)

[2.2 Условия ввода 4](#_Toc121595377)

[2.3 Проверка введенных данных 4](#_Toc121595378)

[2.3.1 Проверка ввода количества строк и столбцов 4](#_Toc121595379)

[2.3.2 Проверка заполнения лабиринта 0 или 1 4](#_Toc121595380)

[2.3.3 Проверка ввода стартовой точки 5](#_Toc121595381)

[2.4 Краткое описание алгоритма решения задачи 5](#_Toc121595382)

[3 Описание алгоритмов решения задачи 7](#_Toc121595383)

[4 Структура данных 8](#_Toc121595384)

[4.1 Структуры данных программы 8](#_Toc121595385)

[4.2 Структуры данных алгоритма Input 8](#_Toc121595386)

[4.3 Структуры данных алгоритма PathOutput 9](#_Toc121595387)

[4.4 Структуры данных алгоритма FindExitDFS 9](#_Toc121595388)

[5 Схема алгоритма решения задачи по ГОСТ 19.701-90 10](#_Toc121595389)

[5.1 Схема алгоритма Input по ГОСТ 19.701-90 11](#_Toc121595390)

[5.2 Схема алгоритма PathOutput по ГОСТ 19.701-90 12](#_Toc121595391)

[5.3 Схема алгоритма FindExitDFS по ГОСТ 19.701-90 13](#_Toc121595392)

[Приложение А 15](#_Toc121595393)

[Приложение Б 22](#_Toc121595394)

[Приложение С 27](#_Toc121595395)

[Приложение D 40](#_Toc121595396)

[Приложение E 43](#_Toc121595397)

# Постановка задачи

Лабиринт задан массивом *X* размерностью *n×n*, в котором *Xk,m* = 0, если клетка [*k, m*] “проходима”; *Xk,m* = 1, если клетка [*k, m*] “не проходима”. Начальное положение путника задается в проходимой клетке [*i, j*]. Путник может перемещаться из одной клетки в другую, если они имеют общую сторону. Путник выходит из лабиринта, когда попадает в граничную клетку. Написать программу выхода путника из лабиринта. Вывести координаты клеток выхода путника из лабиринта.

# [Методика решения](#_Toc83996305)

## Описание оператора try..except..end

Оператор try..except..end имеет вид:

try  
 операторы;  
except  
  блок обработки исключений;  
end;

Выполнение блока начинается с секции try, при отсутствии исключительных ситуаций только она и выполняется. Секция except получает управление в случае возникновения исключительной ситуации. После обработки выполняются операторы, стоящие после end.

## Условия ввода

Условия ввода:

* количество строк и столбцов в лабиринте должно быть натуральным и принадлежать промежутку [4, 35];
* заполнять лабиринт нужно 0 или 1;
* стартовая точка должна быть в лабиринте в проходимой позиции.

## Проверка введенных данных

### Проверка ввода количества строк и столбцов

Проверка ввода количества строк и столбцов происходит с помощью цикла с предусловием repeat..until, чтобы при вводе некорректных данных пользователь заново заполнял их. В теле цикла с помощью оператора try..except..end (описание оператора см. [главу 2.1](#_Описание_оператора_try..except..end)) проверяем целочисленность; оператором if проверяет принадлежность заданному промежутку.

### Проверка заполнения лабиринта 0 или 1

В с помощью оператора try..except..end (описание оператора см. [главу 2.1](#_Описание_оператора_try..except..end)) проверяем целочисленность; оператором if проверяет заполнение лабиринта 0 или 1. В случае некорректного заполнения лабиринта, выводится совершенная ошибка. Программу необходимо перезапустить.

### Проверка ввода стартовой точки

Проверка ввода стартовой точки происходит с помощью цикла с предусловием repeat..until, чтобы при вводе некорректных данных пользователь заново заполнял их. В теле цикла с помощью оператора try..except..end (описание оператора см. [главу 2.1](#_Описание_оператора_try..except..end)) проверяем целочисленность; оператором if проверяет нахождения точки в проходимой позиции.

## Краткое описание алгоритма решения задачи

Данное решение использует поиск в глубину. Обход начинается со стартовой точки. Затем мы двигаемся в определенном направлении до тех пор, пока не достигнем непроходимой точки или выхода. Если мы достигли непроходимой точки или выхода, то мы возвращаемся назад (к точке разветвления или расхождения путей) и идем по другому маршруту. Рассмотрим пример с графами (серым цветом обозначены пройденные вершины; белым – необнаруженные вершины):



Рисунок – Пример обхода в глубину на графах

Как мы видим, данный алгоритм продвигается по определенному направлению. Если конец пути, то возвращаемся назад и двигаемся по другому пути до тех пор, пока не исследуем все варианты.

Рассмотрим пример с лабиринтом (серым цветом обозначены пройденные клетки; белым – необнаруженные клетки; черным – непроходимые клетки; цифры обозначают номер шага клетки):



Рисунок – Пример обхода в глубину на лабиринте

Как мы видим, алгоритм является рекурсивным. В данном примере мы сначала пытаемся переместиться в правую клетку, затем нижнию, затем левую, затем правую. Еще видим, что данный алгоритм позволяет найти все способы выхода из лабиринта.

# Описание алгоритмов решения задачи

Таблица – Описание алгоритмов решения задачи

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| №  п.п. | Наименование алгоритма | Назначение алгоритма | Формальные параметры | Предполагаемый тип реализации |
| 1 | Основной алгоритм | Вызывает следующие подпрограммы:  Input,  FindExitВFS(CoordI, CoordJ) |  |  |
| 2 | Input | Ввод SizeI, SizeJ, Lab[1..SizeI, 1..SizeJ], StrStartCoords | Не предъявляется | Процедура |
| 3 | FindExit  DFS(CoordI, CoordJ) | Ищет выходы из лабиринта с помощью алгоритма поиска в глубину.  Выводит на экран координаты клеток выхода из лабиринта. | CoordI – получает от фактического параметра адрес  CoordJ – получает от фактического параметра адрес | Процедура |

# Структура данных

## Структуры данных программы

Таблица – Структура данных основной программы

|  |  |  |
| --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение |
| Convert | String | Набор символов для перевода символа в его численное значение и наоборот. |
| MinSizes | Byte | Минимальный размер лабиринта |
| MaxSizes | Byte | Максимальный размер лабиринта |
| Lab | Array [1..MaxSizes, 1..MaxSizes] of Byte | Хранит исходный лабиринт |
| ExitCoords | Array [1..MaxSizes, 1..MaxSizes] of Byte | Хранит координаты выхода из лабиринта |
| SizeI | Byte | Количество строк у лабиринта |
| SizeJ | Byte | Количество столбцов у лабиринта |
| StartI | Byte | Начальная координата строки |
| StartJ | Byte | Начальная координата столбца |
| CurrNumStep | Byte | Текущий номер шага |
| k | Byte | Счетчик цикла |
| flag | Boolean | Индикатор проверки правильности ввода |
| IsPathFound | Boolean | Индикатор существования выхода из лабиринта |

## Структуры данных алгоритма Input

Таблица 3 – Структура данных алгоритма Input

|  |  |  |  |
| --- | --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение | Тип параметра |
| LargerSize | Byte | Наибольших размер лабиринта | Локальный |
| i | Byte | Счетчик цикла | Локальный |
| j | Byte | Счетчик цикла | Локальный |
| StrStartCoords | String[4] | Стартовые координаты, записанные символами | Локальный |

## Структуры данных алгоритма FindExitDFS

Таблица 4 – Структура данных алгоритма FindExitDFS(CoordI, CoordJ)

|  |  |  |  |
| --- | --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение | Тип параметра |
| CoordI | Byte | Координата строки | Формальный |
| CoordJ | Byte | Координата столбца | Формальный |

# Схема алгоритма решения задачи по ГОСТ 19.701-90



Рисунок – Схема алгоритма решения задачи по ГОСТ 19.701-90

## Схема алгоритма Input по ГОСТ 19.701-90



Рисунок – Схема алгоритма Input по ГОСТ 19.701-90

## Схема алгоритма FindExitDFS по ГОСТ 19.701-90



Рисунок 5 – Схема алгоритма FindExitDFS по ГОСТ 19.701-90 (часть 1)



Рисунок 6 – Схема алгоритма FindExitDFS по ГОСТ 19.701-90 (часть 2)

Приложение А

(обязательное)

Исходный код программы

Program DFS;

{

Enter the labyrinth, 0 - the cell is passable, 1 - the cell

is impassable.

Possible to move between cells that have a common side.

Find all possible exits

}

{$APPTYPE CONSOLE}

Uses

System.SysUtils;

Const

Convert = '123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';

MinSizes = 4;

MaxSizes = length(Convert);

//Convert - storing values 1..35 to exchange between

//symbols and their values and vice versa

//MinSizes - minimal allowable sizes in a labyrinth

//MaxSizes - maximum allowable sizes in a labyrinth

Var

Lab : array [1..MaxSizes, 1..MaxSizes] of Byte;

ExitCoords : array [1..(sqr(MaxSizes) div 2), 1..2] of

Byte;

StartI, StartJ, SizeI, SizeJ, CurrNumStep, k : Byte;

flag, IsPathFound : Boolean;

//Lab - an array that stores the entered labyrinth

//ExitCoords - an array which stores exit coordinates

//StartI - start coordinates by lines

//StartJ - start coordinates by columns

//SizeI - entered size by lines

//SizeJ - entered size by columns

//CurrNumStep - current number step in the Way

//k - cycle counter

//flag - flag to confirm the correctness of entering numbers

//IsPathFound - indicator of whether the path is found

//Procedure for writing input data

procedure Input;

var

LargerSize, i, j : Byte;

StrStartCoords : string[4];

//LargerSize - the largest value of the sizes

//i,j - cycle counters

//StrStartCoords - string stores the symbols value of the

//starting coordinates

begin

Writeln('Enter the size of the labyrinth (i j), i and j

belongs to ',MinSizes,'..',MaxSizes);

//Cycle with postcondition for entering correct data.

Repeat

//Initialize the flag

flag:= False;

//Validating the correct input data type

Try

Readln(SizeI, SizeJ);

Except

Writeln('Invalid data type entered');

flag:= True;

End;

//Validate Range

if (not (SizeI in [MinSizes..MaxSizes]) or not (SizeJ in

[MinSizes..MaxSizes])) and not flag then

begin

Writeln('(i j) do not belong to the range!');

flag:= True;

end;

Until not flag;

Writeln;

Writeln('Enter the labyrinth. 0 - the cell is passable, 1

- the cell is impassable.');

Writeln('Possible to move between cells that have a common

side');

//Finding the largest size

if SizeI > SizeJ then

LargerSize:= SizeI

else

LargerSize:= SizeJ;

//If the largest size >= 10, inform the user about the

//replacements

if LargerSize >= 10 then

begin

Writeln;

Writeln('For convenience, numbers consisting of two

digits will be represented as follows:');

for i := 10 to LargerSize do

Writeln(Convert[i],' = ',i);

end;

//Writing columns and boundaries for understanding

Writeln;

Write(' ');

for j := 1 to SizeJ do

Write(Convert[j],' ');

Writeln;

Write(' ');

for j := 1 to SizeJ do

Write('\_\_');

Writeln;

//Cycle for reading a labyrinth (line)

i:= 1;

while (i <= SizeI) and not flag do

begin

//Write the line number

Write(Convert[i],'|');

//Cycle for reading a labyrinth (column)

j:= 1;

while (j <= SizeJ) and not flag do

begin

//Validating the correct input data type

Try

Read(Lab[i,j]);

Except

Writeln('Invalid data type entered! Restart the

program');

flag:= True;

End;

//Validate Range

if (Lab[i,j] <> 0) and (Lab[i,j] <> 1) then

begin

Writeln('Number do not belong to the range! Restart

the program');

flag:= True;

end;

//Modernize j

Inc(j);

end;

//Modernize i

Inc(i);

end;

//Input validation

if not flag then

begin

Writeln;

Writeln('Enter start position (i j). This position must

be in a passable cell (0)');

Readln;

//Cycle with postcondition for entering correct data.

Repeat

//Initialize the flag

flag:= False;

Readln(StrStartCoords);

//Validating the correct input data type

StartI:= Pos(StrStartCoords[1], Convert);

StartJ:= Pos(StrStartCoords[3], Convert);

if (StartI = 0) or (StartJ = 0) or

(length(StrStartCoords) <> 3) then

begin

Writeln('Invalid data type entered!');

flag:= True;

end

//Checking, the position must be in the labyrinth

else

if (StartI > SizeI) or (StartJ > SizeJ) then

begin

Writeln('Position not in the labyrinth!');

flag:= True;

end

//Checking, the position must be in a passable cell

else

if Lab[StartI, StartJ] <> 0 then

begin

Writeln('Position not in a passable cell!');

flag:= True;

end;

Until not flag ;

end;

end;

//Procedure for finding a path

procedure FindExitDFS(CoordI, CoordJ: Byte);

begin

//Increase CurrNumStep and mark the passed cell and add the

//coordinates to ExitCoords

Inc(CurrNumStep);

Lab[CoordI,CoordJ]:= 1;

ExitCoords[CurrNumStep, 1]:= CoordI;

ExitCoords[CurrNumStep, 2]:= CoordJ;

//Сhecking for an exit

if (CoordI = 1) or (CoordJ = 1) or (CoordI = SizeI) or

(CoordJ = SizeJ) then

begin

Writeln;

//Output how many steps was found the exit

Writeln('Number of steps:',CurrNumStep);

//Writing the path

for k := 1 to CurrNumStep do

Write('(',Convert[ExitCoords[k, 1]],',',Convert[Ex

itCoords[k, 2]],') ');

Writeln;

//The way is found

IsPathFound:= True;

end

//Else looking for an neighboring, available and untraveled

//cell. And if found, go into it.

else

begin

if Lab[CoordI, CoordJ+1] = 0 then

FindExitDFS(CoordI, CoordJ+1);

if Lab[CoordI+1, CoordJ] = 0 then

FindExitDFS(CoordI+1, CoordJ);

if Lab[CoordI, CoordJ-1] = 0 then

FindExitDFS(CoordI, CoordJ-1);

if Lab[CoordI-1, CoordJ] = 0 then

FindExitDFS(CoordI-1, CoordJ);

end;

//Next, decrease Dec, reset the current coordinates in the

//Way and exit the current cell to the previous

Dec(CurrNumStep);

Lab[CoordI,CoordJ]:= 0;

end;

Begin

//Call the procedure to write data

Input;

//If the labyrinth is entered correctly, then looking for

//a path

if not flag then

begin

//Initialize the variables and go to the procedure

//FindExitDFS

CurrNumStep:= 0;

IsPathFound:= False;

FindExitDFS(StartI, StartJ);

//Сheck if the path is found

if not IsPathFound then

Writeln('Entered labyrinth has no way out');

end;

Readln;

Readln;

End.

Приложение Б

(обязательное)

Тестовые наборы к основному коду программы

**Тестовая ситуация: некорректный ввод данных**

Тест 1

Исходные данные: Некорректный ввод размера лабиринта

Ожидаемый результат: Повторная попытка

Полученный результат:

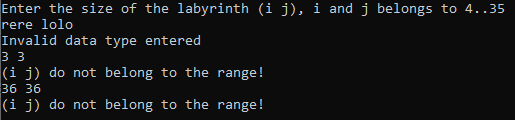


Рисунок 8 – Результаты расчетов

Тест 2

Исходные данные: Некорректный ввод лабиринта

Ожидаемый результат: Просьба перезапустить программу

Полученный результат:

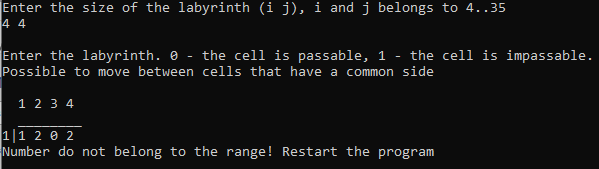


Рисунок 9 – Результаты расчетов

Тест 3

Исходные данные: Некорректный ввод лабиринта

Ожидаемый результат: Просьба перезапустить программу

Полученный результат:

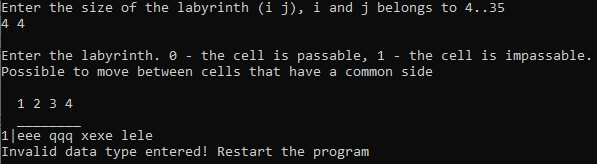


Рисунок – Результаты расчетов

Тест 4

Исходные данные: Некорректный ввод стартовой точки

Ожидаемый результат: Просьба перезапустить программу

Полученный результат:

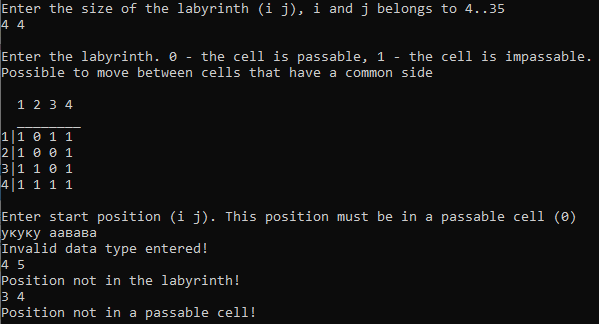


Рисунок – Результаты расчетов

Тест 5

Исходные данные: Неккоректный ввод лабиринта

Ожидаемый результат: Просьба перезапустить программу

Полученный результат:

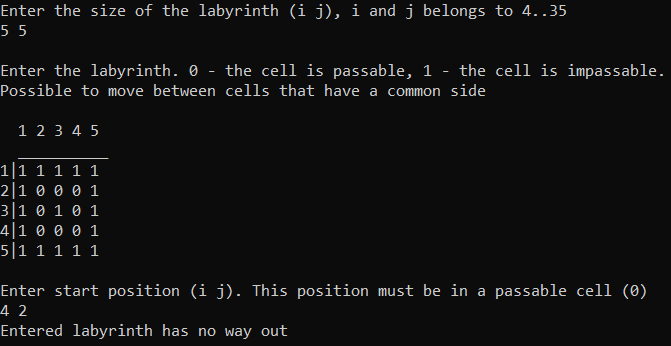


Рисунок – Результаты расчетов

**Тестовая ситуация: корректный ввод данных**

Тест 6

Исходные данные: Лабиринт размерности 55. Стартовая точка (4, 3).

Полученный результат:

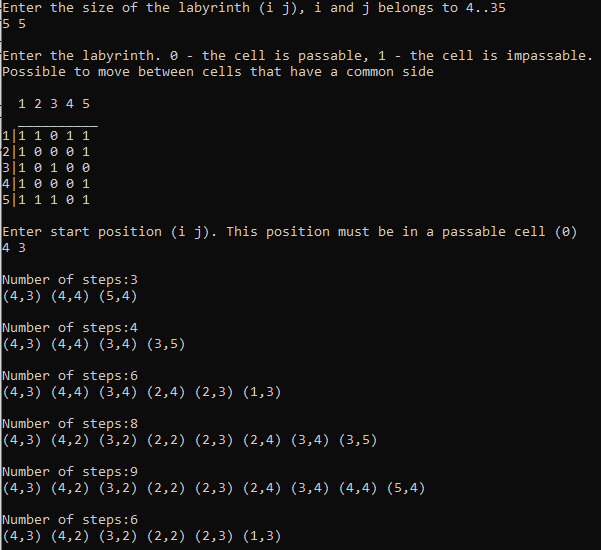


Рисунок – Результаты расчетов

Тест 7

Исходные данные: Лабиринт размерности 66. Стартовая точка (5, 2).

Полученный результат:

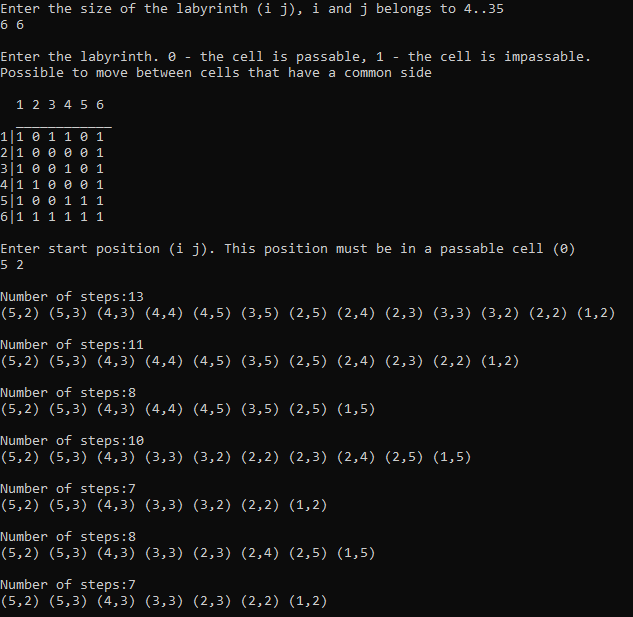


Рисунок – Результаты расчетов

Тест 8

Исходные данные: Лабиринт размерности 1010. Стартовая точка (9, 9).

Полученный результат:

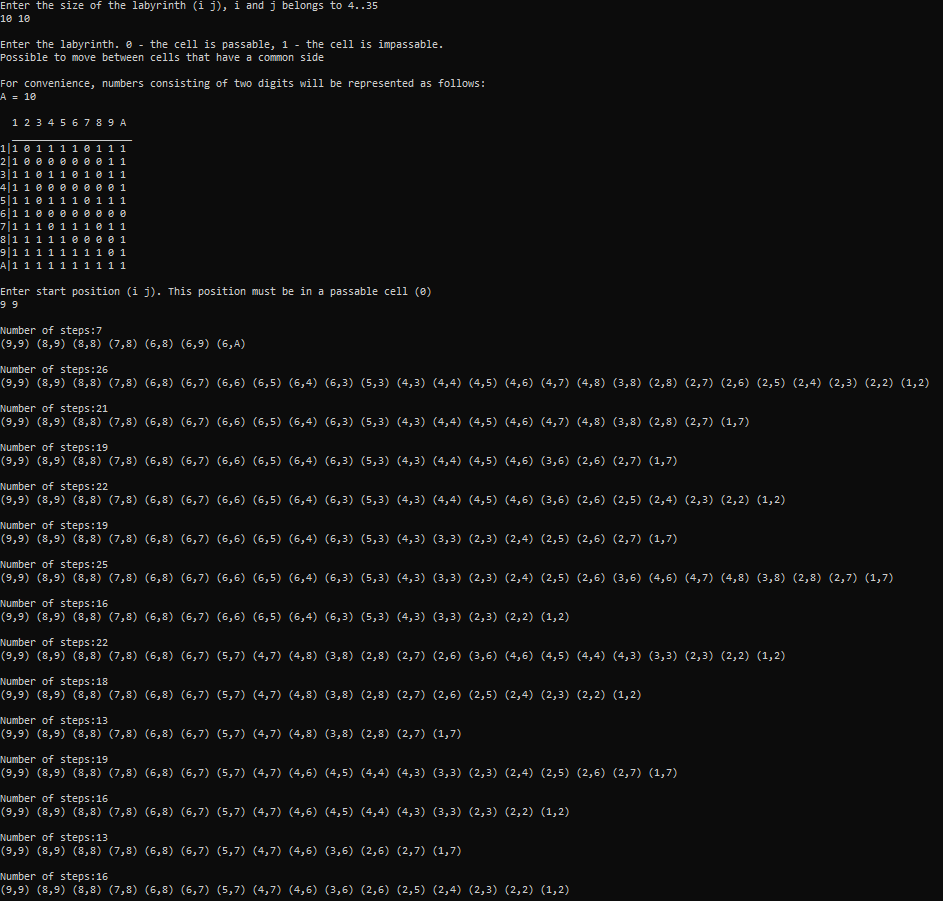


Рисунок – Результаты расчетов

Приложение С

(обязательное)

Код с генератором лабиринта

Program DFSrandom;

{

Enter the labyrinth, 0 - the cell is passable, 1 -

the cell is impassable.

Possible to move between cells that have a common

side. Find all possible exits

}

{$APPTYPE CONSOLE}

uses

System.SysUtils;

Const

Convert = '123456789abcdefghijklmnopqrstuvwxyzA

BCDEFGHIJKLMNOPQRSTUVWXYZ';

MinSizes = 4;

MaxSizes = length(Convert);

//Convert - storing values 1..35 to exchange between

//symbols and their values and vice versa

//MinSizes - minimal allowable sizes in a labyrinth

//MaxSizes - maximum allowable sizes in a labyrinth

Var

Lab: array [1..MaxSizes, 1..MaxSizes] of Byte;

ExitCoords : array [1..(sqr(MaxSizes) div 2), 1..2]

of Byte;

StartI, StartJ, SizeI, SizeJ, CurrNumStep, k : Byte;

//Lab - an array that stores the entered labyrinth

//ExitCoords - an array which stores exit coordinates

//StartI - start coordinates by lines

//StartJ - start coordinates by columns

//SizeI - entered size by lines

//SizeJ - entered size by columns

//CurrNumStep - current number step in the Way

//k - cycle counter

//Procedure for generating a labyrinth

procedure Generator;

var

LargerSize, i, j, PickResult, CoordI, CoordJ,

AmountRotations, AmountStep : Byte;

flag, isBorder: Boolean;

//LargerSize - the largest value of the sizes

//i,j - cycle counters

//PickResult a random number that decides which step

//to take

//CoordI - current position in i

//CoordJ - current position in j

//AmountRotations - amount of rotations to complete

//the labyrinth

//AmountStep - amount of steps to be taken along a

//given line

//flag - flag to confirm the correctness of entering

//numbers

//isBorder - indicator of hitting the border

begin

Writeln('Enter the size of the labyrinth (i j), i and

j belongs to ',MinSizes,'..',MaxSizes);

//Cycle with postcondition for entering correct data.

Repeat

//Initialize the flag

flag:= False;

//Validating the correct input data type

Try

Readln(SizeI, SizeJ);

Except

Writeln('Invalid data type entered');

flag:= True;

End;

//Validate Range

if ((SizeI < MinSizes) or (SizeJ < MinSizes) or

(SizeI > MaxSizes) or (SizeJ > MaxSizes)) and

not flag then

begin

Writeln('(i j) do not belong to the range!');

flag:= True;

end;

Until not flag;

//Finding the largest size

if SizeI > SizeJ then

LargerSize:= SizeI

else

LargerSize:= SizeJ;

//If the largest size >= 10, inform the user about

//the replacements

if LargerSize >= 10 then

begin

Writeln;

Writeln('For convenience, numbers consisting of two

digits will be represented as follows:');

for i := 10 to LargerSize do

Writeln(Convert[i],' = ',i);

end;

//Fill the whole labyrinth with impassable cells

for i := 1 to SizeI do

for j := 1 to SizeJ do

Lab[i,j]:= 1;

Randomize;

//Generate which side will be the first finish

PickResult:= 1 + Random(4);

//Define

case PickResult of

//Up side

1:

begin

//Generating a passable cell on the up side

CoordI:= 1;

CoordJ:= Random(SizeJ-2)+2;

//The first direction in which will go - down

PickResult:= 2;

end;

//Left side

2:

begin

//Generating a passable cell on the left side

CoordI:= Random(SizeI-2)+2;

CoordJ:= 1;

//The first direction in which will go - right

PickResult:= 1;

end;

//Down side

3:

begin

//Generating a passable cell on the down side

CoordI:= SizeI;

CoordJ:= Random(SizeJ-2)+2;

//The first direction in which will go - up

PickResult:= 4;

end;

//Right side

4:

begin

//Generating a passable cell on the right side

CoordI:= Random(SizeI-2)+2;

CoordJ:= SizeJ;

//The first direction in which will go - left

PickResult:= 3;

end;

end;

//Make the first finish passable

Lab[CoordI, CoordJ]:= 0;

//Generate the amount of rotations to complete the

//labyrinth

AmountRotations:= SizeI + SizeJ + Random(SizeI +

SizeJ);

//Going AmountRotations times

for i := 1 to AmountRotations do

begin

//Reset isBorder

isBorder:= False;

//Defining the rotation

case PickResult of

//Right

1:

begin

//Generate amount of steps

AmountStep:= 1 + Random(SizeJ div 2);

//Going AmountStep times or until hit the

//border

j:= 1;

while (j <= AmountStep) and not isBorder do

begin

//Going one step to the right

CoordJ:= CoordJ + 1;

//Check if the border is reached

if CoordJ = SizeJ then

begin

if Random(15) = 0 then

Lab[CoordI, CoordJ]:= 0;

CoordJ:= CoordJ - 1;

isBorder:= True;

end

//Checking if the cell is impassable now

else

if Lab[CoordI, CoordJ] = 1 then

begin

//check whether there are passable cells

//above or below (so as not to mold

//passable cells). If there is, change

//the coordinates and check for reaching

//the border

if Lab[CoordI+1, CoordJ] = 0 then

begin

CoordI:= CoordI - 1;

CoordJ:= CoordJ - 1;

if CoordI = 1 then

begin

CoordI:= CoordI + 1;

isBorder:= True;

end;

end

else

if Lab[CoordI-1, CoordJ] = 0 then

begin

CoordI:= CoordI + 1;

CoordJ:= CoordJ - 1;

if CoordI = SizeI then

begin

CoordI:= CoordI - 1;

isBorder:= True;

end;

end;

//Make the current position to the

//passable

Lab[CoordI, CoordJ]:= 0;

end;

//Modernize j

Inc(j);

end;

end;

//Down

2:

begin

//Generate amount of steps

AmountStep:= 1 + Random(SizeI div 2);

//Going AmountStep times or until hit the

//border

j:= 1;

while (j <= AmountStep) and not isBorder do

begin

//Going one step to the down

CoordI:= CoordI + 1;

//Check if the border is reached

if CoordI = SizeI then

begin

if Random(15) = 0 then

Lab[CoordI, CoordJ]:= 0;

CoordI:= CoordI - 1;

isBorder:= True;

end

//Checking if the cell is impassable now

else

if Lab[CoordI, CoordJ] = 1 then

begin

//check whether there are passable cells

//left or right (so as not to mold

//passable cells). If there is, change

//the coordinates and check for reaching

//the border

if Lab[CoordI, CoordJ+1] = 0 then

begin

CoordI:= CoordI - 1;

CoordJ:= CoordJ - 1;

if CoordJ = 1 then

begin

CoordJ:= CoordJ + 1;

isBorder:= True;

end;

end

else

if Lab[CoordI, CoordJ-1] = 0 then

begin

CoordI:= CoordI - 1;

CoordJ:= CoordJ + 1;

if CoordJ = SizeJ then

begin

CoordJ:= CoordJ - 1;

isBorder:= True;

end;

end;

//Make the current position to the

//passable

Lab[CoordI, CoordJ]:= 0;

end;

//Modernize j

Inc(j);

end;

end;

//Left

3:

begin

//Generate amount of steps

AmountStep:= 1 + Random(SizeJ div 2);

//Going AmountStep times or until hit the

//border

j:= 1;

while (j <= AmountStep) and not isBorder do

begin

//Going one step to the left

CoordJ:= CoordJ - 1;

//Check if the border is reached

if CoordJ = 1 then

begin

if Random(15) = 0 then

Lab[CoordI, CoordJ]:= 0;

CoordJ:= CoordJ + 1;

isBorder:= True;

end

//Checking if the cell is impassable now

else

if Lab[CoordI, CoordJ] = 1 then

begin

//check whether there are passable cells

//above or below (so as not to mold

//passable cells). If there is, change

//the coordinates and check for reaching

//the border

if Lab[CoordI+1, CoordJ] = 0 then

begin

CoordI:= CoordI - 1;

CoordJ:= CoordJ + 1;

if CoordI = 1 then

begin

CoordI:= CoordI + 1;

isBorder:= True;

end;

end

else

if Lab[CoordI-1, CoordJ] = 0 then

begin

CoordI:= CoordI + 1;

CoordJ:= CoordJ + 1;

if CoordI = SizeI then

begin

CoordI:= CoordI - 1;

isBorder:= True;

end;

end;

//Make the current position to the

//passable

Lab[CoordI, CoordJ]:= 0;

end;

//Modernize j

Inc(j);

end;

end;

//Up

4:

begin

//Generate amount of steps

AmountStep:= 1 + Random(SizeI div 2);

//Going AmountStep times or until hit the

//border

j:= 1;

while (j <= AmountStep) and not isBorder do

begin

//Going one step to the up

CoordI:= CoordI - 1;

//Check if the border is reached

if CoordI = 1 then

begin

if Random(15) = 0 then

Lab[CoordI, CoordJ]:= 0;

CoordI:= CoordI + 1;

isBorder:= True;

end

//Checking if the cell is impassable now

else

if Lab[CoordI, CoordJ] = 1 then

begin

//check whether there are passable cells

//left or right (so as not to mold

//passable cells). If there is, change

//the coordinates and check for reaching

//the border

if Lab[CoordI, CoordJ+1] = 0 then

begin

CoordI:= CoordI + 1;

CoordJ:= CoordJ - 1;

if CoordJ = 1 then

begin

CoordJ:= CoordJ + 1;

isBorder:= True;

end;

end

else

if Lab[CoordI, CoordJ-1] = 0 then

begin

CoordI:= CoordI + 1;

CoordJ:= CoordJ + 1;

if CoordJ = SizeJ then

begin

CoordJ:= CoordJ - 1;

isBorder:= True;

end;

end;

//Make the current position to the

//passable

Lab[CoordI, CoordJ]:= 0;

end;

//Modernize j

Inc(j);

end;

end;

end;

//Choosing the next rotation

PickResult:= 1 + Random(4);

end;

//Make the last point the starting point

StartI:= CoordI;

StartJ:= CoordJ;

Writeln('Starting position - (',Convert[StartI],','

,Convert[StartJ],')');

//Writing columns and boundaries for understanding

Writeln;

Write(' ');

for j := 1 to SizeJ do

Write(Convert[j],' ');

Writeln;

Write(' ');

for j := 1 to SizeJ do

Write('\_\_');

Writeln;

//Displaying the labyrinth

for i := 1 to SizeI do

begin

Write(Convert[i],'|');

for j := 1 to SizeJ do

Write(Lab[i,j],' ');

Writeln;

end;

end;

//Procedure for finding a path

procedure FindExitDFS(CoordI, CoordJ: Byte);

begin

//Increase CurrNumStep and mark the passed cell and

//add the coordinates to ExitCoords

Inc(CurrNumStep);

Lab[CoordI,CoordJ]:= 1;

ExitCoords[CurrNumStep, 1]:= CoordI;

ExitCoords[CurrNumStep, 2]:= CoordJ;

//Сhecking for an exit

if (CoordI = 1) or (CoordJ = 1) or (CoordI = SizeI)

or (CoordJ = SizeJ) then

begin

Writeln;

//Output how many steps was found the exit

Writeln('Number of steps:',CurrNumStep);

//Writing the path

for k := 1 to CurrNumStep do

Write('(',Convert[ExitCoords[k, 1]],',',Con

vert[ExitCoords[k, 2]],') ');

Writeln;

end

//Else looking for an neighboring, available and

//untraveled cell. And if found, go into it.

else

begin

if Lab[CoordI, CoordJ+1] = 0 then

FindExitDFS(CoordI, CoordJ+1);

if Lab[CoordI+1, CoordJ] = 0 then

FindExitDFS(CoordI+1, CoordJ);

if Lab[CoordI, CoordJ-1] = 0 then

FindExitDFS(CoordI, CoordJ-1);

if Lab[CoordI-1, CoordJ] = 0 then

FindExitDFS(CoordI-1, CoordJ);

end;

//Next, decrease Dec, reset the current coordinates

//and exit the current cell to the previous

Dec(CurrNumStep);

Lab[CoordI,CoordJ]:= 0;

end;

Begin

//Call the procedure to generate labyrinth

Generator;

//Initialize the variables and go to the procedure

//FindExitDFS

CurrNumStep:= 0;

FindExitDFS(StartI, StartJ);

Readln(SizeI);

End.

Приложение D

(обязательное)

Тестовые наборы к коду с генератором лабиринта

Тест 1

Полученный результат:

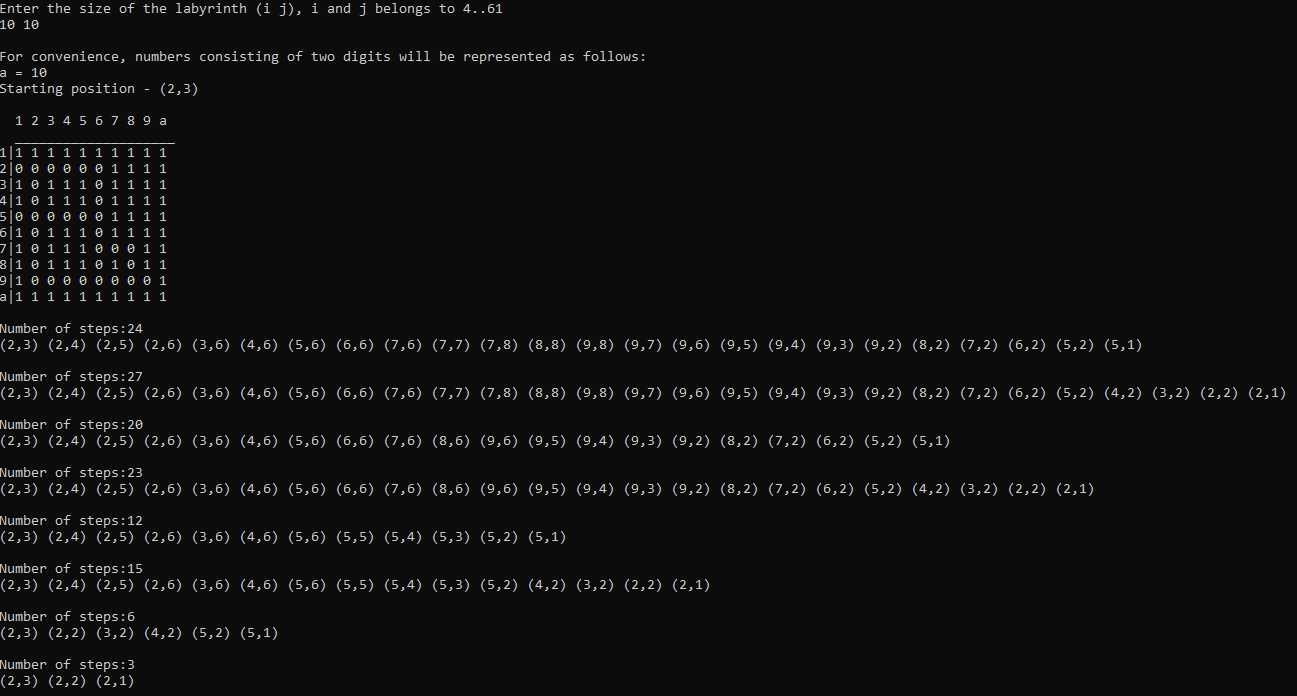


Рисунок 16 – Результаты расчетов

Тест 2

Полученный результат:

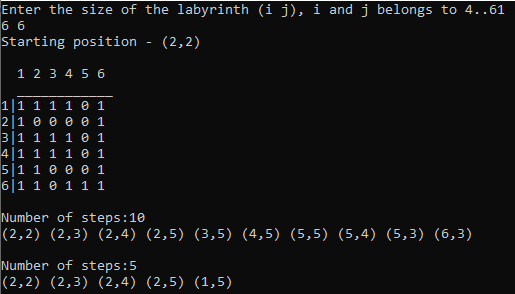


Рисунок 17 – Результаты расчетов

Тест 3

Полученный результат:

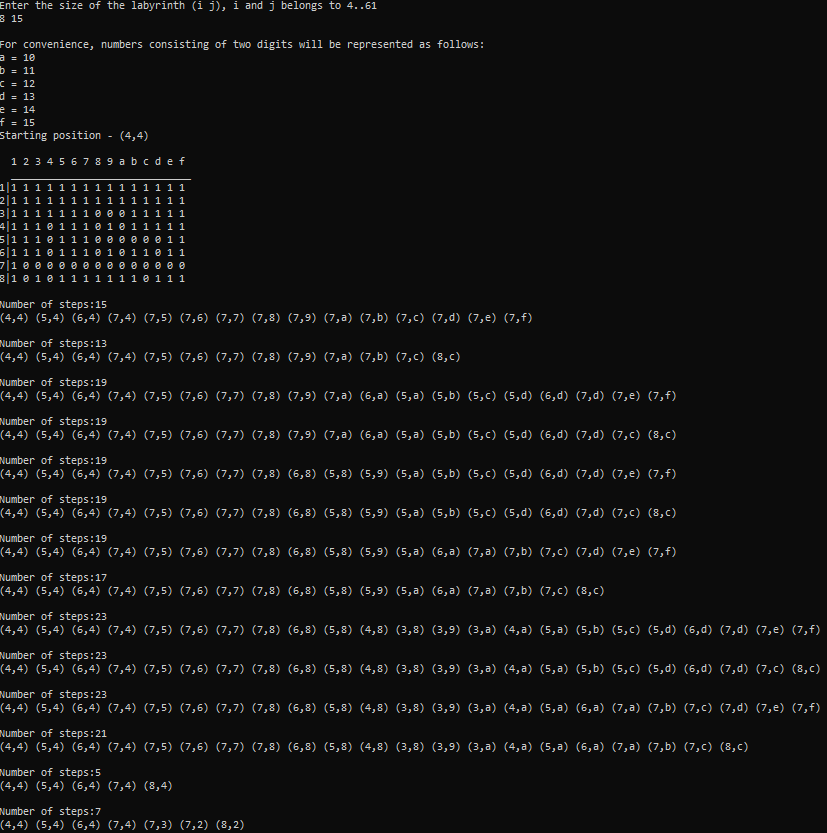


Рисунок 18 – Результаты расчетов

Тест 4

Полученный результат:

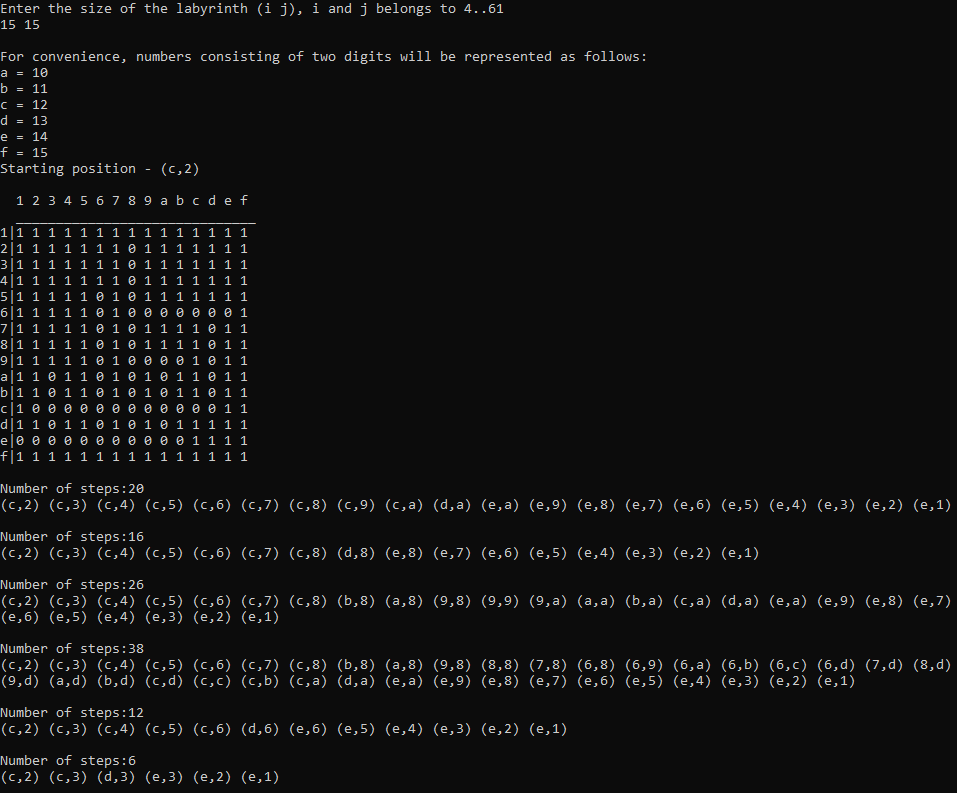


Рисунок 19 – Результаты расчетов

Приложение E

(обязательное)

Код с использованием алгоритма Дейкстры

Program DijkstraRandom;

{

Enter the labyrinth, 0 - the cell is passable, 1 –

the cell is impassable.

Possible to move between cells that have a common

side. Find closest way

}

{$APPTYPE CONSOLE}

uses

System.SysUtils;

Const

Convert = '123456789abcdefghijklmnopqrstuvwxyzA

BCDEFGHIJKLMNOPQRSTUVWXYZ';

MinSizes = 4;

MaxSizes = length(Convert);

//Convert - storing values 1..35 to exchange between

//symbols and their values and vice versa

//MinSizes - minimal allowable sizes in a labyrinth

//MaxSizes - maximum allowable sizes in a labyrinth

Var

Lab, Way : array [1..MaxSizes, 1..MaxSizes] of Byte;

SizeI, SizeJ, StartI, StartJ, i, j : Byte;

CurrNumStep, CoordExitI, CoordExitJ : Byte;

//Lab - an array that stores the entered labyrinth

//Way - an array that stores path to the exit

//SizeI - entered size by lines

//SizeJ - entered size by columns

//StartI - start coordinates by lines

//StartJ - start coordinates by columns

//i,j - cycle counters

//CurrNumStep - current number step in the Way

//CoordExitI - nearest exit coordinate (i)

//CoordExitJ - nearest exit coordinate (j)

//Procedure for generating a labyrinth

procedure Generator;

var

LargerSize, i, j, PickResult, CoordI, CoordJ,

AmountRotations, AmountStep : Byte;

flag, isBorder: Boolean;

//LargerSize - the largest value of the sizes

//i,j - cycle counters

//PickResult a random number that decides which step

//to take

//CoordI - current position in i

//CoordJ - current position in j

//AmountRotations - amount of rotations to complete

//the labyrinth

//AmountStep - amount of steps to be taken along a

//given line

//flag - flag to confirm the correctness of entering

//numbers

//isBorder - indicator of hitting the border

begin

Writeln('Enter the size of the labyrinth (i j), i and

j belongs to ',MinSizes,'..',MaxSizes);

//Cycle with postcondition for entering correct data.

Repeat

//Initialize the flag

flag:= False;

//Validating the correct input data type

Try

Readln(SizeI, SizeJ);

Except

Writeln('Invalid data type entered');

flag:= True;

End;

//Validate Range

if ((SizeI < MinSizes) or (SizeJ < MinSizes) or

(SizeI > MaxSizes) or (SizeJ > MaxSizes)) and

not flag then

begin

Writeln('(i j) do not belong to the range!');

flag:= True;

end;

Until not flag;

//Finding the largest size

if SizeI > SizeJ then

LargerSize:= SizeI

else

LargerSize:= SizeJ;

//If the largest size >= 10, inform the user about

//the replacements

if LargerSize >= 10 then

begin

Writeln;

Writeln('For convenience, numbers consisting of two

digits will be represented as follows:');

for i := 10 to LargerSize do

Writeln(Convert[i],' = ',i);

end;

//Fill the whole labyrinth with impassable cells

for i := 1 to SizeI do

for j := 1 to SizeJ do

Lab[i,j]:= 1;

Randomize;

//Generate which side will be the first finish

PickResult:= 1 + Random(4);

//Define

case PickResult of

//Up side

1:

begin

//Generating a passable cell on the up side

CoordI:= 1;

CoordJ:= Random(SizeJ-2)+2;

//The first direction in which will go - down

PickResult:= 2;

end;

//Left side

2:

begin

//Generating a passable cell on the left side

CoordI:= Random(SizeI-2)+2;

CoordJ:= 1;

//The first direction in which will go - right

PickResult:= 1;

end;

//Down side

3:

begin

//Generating a passable cell on the down side

CoordI:= SizeI;

CoordJ:= Random(SizeJ-2)+2;

//The first direction in which will go - up

PickResult:= 4;

end;

//Right side

4:

begin

//Generating a passable cell on the right side

CoordI:= Random(SizeI-2)+2;

CoordJ:= SizeJ;

//The first direction in which will go - left

PickResult:= 3;

end;

end;

//Make the first finish passable

Lab[CoordI, CoordJ]:= 0;

//Generate the amount of rotations to complete the

//labyrinth

AmountRotations:= SizeI + SizeJ + Random(SizeI +

SizeJ);

//Going AmountRotations times

for i := 1 to AmountRotations do

begin

//Reset isBorder

isBorder:= False;

//Defining the rotation

case PickResult of

//Right

1:

begin

//Generate amount of steps

AmountStep:= 1 + Random(SizeJ div 2);

//Going AmountStep times or until hit the

//border

j:= 1;

while (j <= AmountStep) and not isBorder do

begin

//Going one step to the right

CoordJ:= CoordJ + 1;

//Check if the border is reached

if CoordJ = SizeJ then

begin

if Random(15) = 0 then

Lab[CoordI, CoordJ]:= 0;

CoordJ:= CoordJ - 1;

isBorder:= True;

end

//Checking if the cell is impassable now

else

if Lab[CoordI, CoordJ] = 1 then

begin

//check whether there are passable cells

//above or below (so as not to mold

//passable cells). If there is, change

//the coordinates and check for reaching

//the border

if Lab[CoordI+1, CoordJ] = 0 then

begin

CoordI:= CoordI - 1;

CoordJ:= CoordJ - 1;

if CoordI = 1 then

begin

CoordI:= CoordI + 1;

isBorder:= True;

end;

end

else

if Lab[CoordI-1, CoordJ] = 0 then

begin

CoordI:= CoordI + 1;

CoordJ:= CoordJ - 1;

if CoordI = SizeI then

begin

CoordI:= CoordI - 1;

isBorder:= True;

end;

end;

//Make the current position to the

//passable

Lab[CoordI, CoordJ]:= 0;

end;

//Modernize j

Inc(j);

end;

end;

//Down

2:

begin

//Generate amount of steps

AmountStep:= 1 + Random(SizeI div 2);

//Going AmountStep times or until hit the

//border

j:= 1;

while (j <= AmountStep) and not isBorder do

begin

//Going one step to the down

CoordI:= CoordI + 1;

//Check if the border is reached

if CoordI = SizeI then

begin

if Random(15) = 0 then

Lab[CoordI, CoordJ]:= 0;

CoordI:= CoordI - 1;

isBorder:= True;

end

//Checking if the cell is impassable now

else

if Lab[CoordI, CoordJ] = 1 then

begin

//check whether there are passable cells

//left or right (so as not to mold

//passable cells). If there is, change

//the coordinates and check for reaching

//the border

if Lab[CoordI, CoordJ+1] = 0 then

begin

CoordI:= CoordI - 1;

CoordJ:= CoordJ - 1;

if CoordJ = 1 then

begin

CoordJ:= CoordJ + 1;

isBorder:= True;

end;

end

else

if Lab[CoordI, CoordJ-1] = 0 then

begin

CoordI:= CoordI - 1;

CoordJ:= CoordJ + 1;

if CoordJ = SizeJ then

begin

CoordJ:= CoordJ - 1;

isBorder:= True;

end;

end;

//Make the current position to the

//passable

Lab[CoordI, CoordJ]:= 0;

end;

//Modernize j

Inc(j);

end;

end;

//Left

3:

begin

//Generate amount of steps

AmountStep:= 1 + Random(SizeJ div 2);

//Going AmountStep times or until hit the

//border

j:= 1;

while (j <= AmountStep) and not isBorder do

begin

//Going one step to the left

CoordJ:= CoordJ - 1;

//Check if the border is reached

if CoordJ = 1 then

begin

if Random(15) = 0 then

Lab[CoordI, CoordJ]:= 0;

CoordJ:= CoordJ + 1;

isBorder:= True;

end

//Checking if the cell is impassable now

else

if Lab[CoordI, CoordJ] = 1 then

begin

//check whether there are passable cells

//above or below (so as not to mold

//passable cells). If there is, change

//the coordinates and check for reaching

//the border

if Lab[CoordI+1, CoordJ] = 0 then

begin

CoordI:= CoordI - 1;

CoordJ:= CoordJ + 1;

if CoordI = 1 then

begin

CoordI:= CoordI + 1;

isBorder:= True;

end;

end

else

if Lab[CoordI-1, CoordJ] = 0 then

begin

CoordI:= CoordI + 1;

CoordJ:= CoordJ + 1;

if CoordI = SizeI then

begin

CoordI:= CoordI - 1;

isBorder:= True;

end;

end;

//Make the current position to the

//passable

Lab[CoordI, CoordJ]:= 0;

end;

//Modernize j

Inc(j);

end;

end;

//Up

4:

begin

//Generate amount of steps

AmountStep:= 1 + Random(SizeI div 2);

//Going AmountStep times or until hit the

//border

j:= 1;

while (j <= AmountStep) and not isBorder do

begin

//Going one step to the up

CoordI:= CoordI - 1;

//Check if the border is reached

if CoordI = 1 then

begin

if Random(15) = 0 then

Lab[CoordI, CoordJ]:= 0;

CoordI:= CoordI + 1;

isBorder:= True;

end

//Checking if the cell is impassable now

else

if Lab[CoordI, CoordJ] = 1 then

begin

//check whether there are passable cells

//left or right (so as not to mold

//passable cells). If there is, change

//the coordinates and check for reaching

//the border

if Lab[CoordI, CoordJ+1] = 0 then

begin

CoordI:= CoordI + 1;

CoordJ:= CoordJ - 1;

if CoordJ = 1 then

begin

CoordJ:= CoordJ + 1;

isBorder:= True;

end;

end

else

if Lab[CoordI, CoordJ-1] = 0 then

begin

CoordI:= CoordI + 1;

CoordJ:= CoordJ + 1;

if CoordJ = SizeJ then

begin

CoordJ:= CoordJ - 1;

isBorder:= True;

end;

end;

//Make the current position to the

//passable

Lab[CoordI, CoordJ]:= 0;

end;

//Modernize j

Inc(j);

end;

end;

end;

//Choosing the next rotation

PickResult:= 1 + Random(4);

end;

//Make the last point the starting point

StartI:= CoordI;

StartJ:= CoordJ;

Writeln('Starting position - (',Convert[StartI],','

,Convert[StartJ],')');

//Writing columns and boundaries for understanding

Writeln;

Write(' ');

for j := 1 to SizeJ do

Write(Convert[j],' ');

Writeln;

Write(' ');

for j := 1 to SizeJ do

Write('\_\_');

Writeln;

//Displaying the labyrinth

for i := 1 to SizeI do

begin

Write(Convert[i],'|');

for j := 1 to SizeJ do

Write(Lab[i,j],' ');

Writeln;

end;

end;

//Procedure to writing the path

procedure PathOutput(CoordI, CoordJ: Byte);

var

PrevNumStep: Byte;

//PrevNumStep - previous number step in the Way

begin

//Find the previous number step in the Way

//to find previous coordinates in the path

PrevNumStep:= Way[CoordI, CoordJ] - 1;

//Looking for a path to the starting cell

if (CoordI <> StartI) or (CoordJ <> StartJ) then

begin

if Way[CoordI, CoordJ-1] = PrevNumStep then

PathOutput(CoordI, CoordJ-1)

else

if Way[CoordI-1, CoordJ] = PrevNumStep then

PathOutput(CoordI-1, CoordJ)

else

if Way[CoordI, CoordJ+1] = PrevNumStep then

PathOutput(CoordI, CoordJ+1)

else

if Way[CoordI+1, CoordJ] = PrevNumStep then

PathOutput(CoordI+1, CoordJ);

end;

//Write coordinates

Write('(',Convert[CoordI],',',Convert[CoordJ],') ');

end;

//Procedure for finding a path

procedure DijkstraClosestWay(CoordI, CoordJ: Byte);

var

NextNumStep: Byte;

//NextNumStep - next number step in the Way

begin

//Increase CurrNumStep and add it to the array Way at

//the current coordinates

Inc(CurrNumStep);

NextNumStep:= CurrNumStep + 1;

Way[CoordI,CoordJ]:= CurrNumStep;

//Сhecking for a nearest exit

if ((CoordI = 1) or (CoordJ = 1) or (CoordI = SizeI)

or (CoordJ = SizeJ)) and (CurrNumStep <

Way[CoordExitI,CoordExitJ]) then

begin

CoordExitI:= CoordI;

CoordExitJ:= CoordJ;

end

//Else looking for an neighboring, available cell.

//Also look for the shortest path to the cell. And if

//found, go into it

else

begin

if (Lab[CoordI, CoordJ+1] = 0) and (Way[CoordI,

CoordJ+1] > NextNumStep) then

DijkstraClosestWay(CoordI, CoordJ+1);

if (Lab[CoordI+1, CoordJ] = 0) and (Way[CoordI+1,

CoordJ] > NextNumStep) then

DijkstraClosestWay(CoordI+1, CoordJ);

if (Lab[CoordI, CoordJ-1] = 0) and (Way[CoordI,

CoordJ-1] > NextNumStep) then

DijkstraClosestWay(CoordI, CoordJ-1);

if (Lab[CoordI-1, CoordJ] = 0) and (Way[CoordI-1,

CoordJ] > NextNumStep) then

DijkstraClosestWay(CoordI-1, CoordJ);

end;

//Next, decrease Dec and exit the current cell to the

//previous

Dec(CurrNumStep);

end;

Begin

//Call the procedure to generate labyrinth

Generator;

//According to Dijkstra's algorithm, assume that

//initially all cells can be reached by an infinitely

//long path

for i := 1 to SizeI do

for j := 1 to SizeJ do

Way[i,j]:= 255;

//Also assume that initially the exit coordinates are

//1,1 (since this cell in the labyrinth does not make

//sense) so that the path length is infinitely large

CoordExitI:= 1;

CoordExitJ:= 1;

//Initialize the variables and go to the procedure

//DijkstraClosestWay

CurrNumStep:= 0;

DijkstraClosestWay(StartI, StartJ);

//Write the result

Writeln('Found the nearest way. Amount of steps:

',Way[CoordExitI,CoordExitJ]);

PathOutput(CoordExitI, CoordExitJ);

Readln(i);

End.