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](#_Toc121595335)

[2 Методика решения 4](#_Toc121595336)

[2.1 Описание оператора try..except..end 4](#_Toc121595337)

[2.2 Условия ввода 4](#_Toc121595338)

[2.3 Проверка введенных данных 4](#_Toc121595339)

[2.3.1 Проверка ввода количества строк и столбцов 4](#_Toc121595340)

[2.3.2 Проверка заполнения лабиринта 0 или 1 4](#_Toc121595341)

[2.3.3 Проверка ввода стартовой точки 5](#_Toc121595342)

[2.4 Краткое описание алгоритма решения задачи 5](#_Toc121595343)

[3 Описание алгоритмов решения задачи 7](#_Toc121595344)

[4 Структура данных 8](#_Toc121595345)

[4.1 Структуры данных программы 8](#_Toc121595346)

[4.2 Структуры данных алгоритма Input 8](#_Toc121595347)

[4.3 Структуры данных алгоритма PathOutput 9](#_Toc121595348)

[4.4 Структуры данных алгоритма FindExitBFS 9](#_Toc121595349)

[5 Схема алгоритма решения задачи по ГОСТ 19.701-90 10](#_Toc121595350)

[5.1 Схема алгоритма Input по ГОСТ 19.701-90 11](#_Toc121595351)

[5.2 Схема алгоритма PathOutput по ГОСТ 19.701-90 12](#_Toc121595352)

[5.3 Схема алгоритма FindExitBFS по ГОСТ 19.701-90 13](#_Toc121595353)

[Приложение А 19](#_Toc121595354)

[Приложение Б 28](#_Toc121595355)

[Приложение С 33](#_Toc121595356)

[Приложение D 48](#_Toc121595357)

# 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 ширину. Обход начинается со стартовой точки. Затем алгоритм посещает соседей этой вершины. За ними - соседей соседей, и так далее. Тем самым образуется очередь из вершин для посещения. Рассмотрим пример с графами (серым цветом обозначены пройденные вершины; голубым – очередь; белым – необнаруженные вершины):



Рисунок 1 – Пример обхода в ширину на графах

При посещении очередной вершины в очередь добавляются все её соседи, которые ещё не были посещены. Если в очереди находятся две одинаковые вершины, то посещать данную вершину будем один раз (второй раз будем игнорировать её). Также в очереди будут находится два уровня вершин: при посещения всех вершин одного уровня следующий уровень сдвигается влево.

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



Рисунок 2 – Пример обхода в ширину на лабиринте

Как мы видим, данный алгоритм позволяет найти кратчайший путь в каждый выход. Также очевидно как восстановить путь: начиная от граничной клетки, нужно отнимать единицу от текущего уровня и идти по клеткам, соответствующие найденному уровню.

# Описание алгоритмов решения задачи

Таблица 1 – Описание алгоритмов решения задачи

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| №  п.п. | Наименование алгоритма | Назначение алгоритма | Формальные параметры | Предполагаемый тип реализации |
| 1 | Основной алгоритм | Вызывает следующие подпрограммы:  Input,  FindExitBFS(StartI, StartJ) |  |  |
| 2 | Input | Ввод SizeI, SizeJ, Lab[1..SizeI, 1..SizeJ], StrStartCoords | Не предъявляется | Процедура |
| 3 | PathOutput (CoordI, CoordJ) | Выводит на экран координаты клеток выхода из лабиринта | CoordI – получает от фактического параметра адрес  CoordJ – получает от фактического параметра адрес | Процедура |
| 4 | FindExitBFS(StartI, StartJ) | Ищет выходы из лабиринта с помощью алгоритма поиска в ширину. Вызывает подпрограмму PathOutput (CoordI, CoordJ) | StartI – получает от фактического параметра адрес  StartJ – получает от фактического параметра адрес | Процедура |

# Структура данных

## Структуры данных программы

Таблица 2 – 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 |
| Way | Array [1..MaxSizes, 1..MaxSizes] of Byte | Хранит пройденные позиции |
| SizeI | Byte | Количество строк у лабиринта |
| SizeJ | Byte | Количество столбцов у лабиринта |
| StartI | Byte | Начальная координата строки |
| StartJ | Byte | Начальная координата столбца |
| 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 PathOutput

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

|  |  |  |  |
| --- | --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение | Тип параметра |
| CoordI | Byte | Координата строки | Формальный |
| CoordJ | Byte | Координата столбца | Формальный |
| PrevNumStep | Byte | Предыдущий номер шага | Локальный |

## Структуры данных алгоритма FindExitBFS

Таблица 5 – Структура данных алгоритма FindExitBFS(StartI, StartJ)

|  |  |  |  |
| --- | --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение | Тип параметра |
| StartI | Byte | Стартовая координата строки | Формальный |
| StartJ | Byte | Стартовая координата столбца | Формальный |
| Queue | Array [1..MaxSizes\*4, 1..2] of Byte | Очередь позиций, которые нужно пройти | Локальный |
| CoordI | Byte | Текущая координата строки | Локальный |
| CoordJ | Byte | Текущая координата столбца | Локальный |
| AmountInQueue | Byte | Количество позиций в очереди | Локальный |
| AmountSameStep | Byte | Количество позиций, имеющие одинаковый номер шага | Локальный |
| CurrPosQueue | Byte | Текущий индекс в очереди | Локальный |
| CurrNumStep | Byte | Текущий номер шага | Локальный |
| i | Byte | Счетчик цикла | Локальный |

# Схема алгоритма решения задачи по ГОСТ 19.701-90



Рисунок 3 – Схема алгоритма решения задачи по ГОСТ 19.701-90

## Схема алгоритма Input по ГОСТ 19.701-90



Рисунок 4 – Схема алгоритма Input по ГОСТ 19.701-90

## Схема алгоритма PathOutput по ГОСТ 19.701-90



Рисунок 5 – Схема алгоритма PathOutput по ГОСТ 19.701-90

## Схема алгоритма FindExitBFS по ГОСТ 19.701-90



Рисунок 6 – Схема алгоритма FindExitBFS по ГОСТ 19.701-90 (часть 1)



Рисунок 7 – Схема алгоритма FindExitBFS по ГОСТ 19.701-90 (часть 2)



Рисунок 8 – Схема алгоритма FindExitBFS по ГОСТ 19.701-90 (часть 3)



Рисунок 9 – Схема алгоритма FindExitBFS по ГОСТ 19.701-90 (часть 4)



Рисунок 10 – Схема алгоритма FindExitBFS по ГОСТ 19.701-90 (часть 5)



Рисунок 11 – Схема алгоритма FindExitBFS по ГОСТ 19.701-90 (часть 6)

Приложение А

(обязательное)

Исходный код программы

Program BFS;

{

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 path to each еxit.

}

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

SizeI, SizeJ, StartI, StartJ: Byte;

flag, IsPathFound : Boolean;

//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

//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 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 FindExitBFS(StartI, StartJ: Byte);

var

Queue : array [1..MaxSizes\*4, 1..2] of Byte;

CoordI, CoordJ : Byte;

AmountInQueue, AmountSameStep, CurrPosQueue, CurrNumStep,

i : Byte;

//Queue - queue cell

//CoordI - current coordinates by i

//CoordJ - current coordinates by j

//AmountInQueue - amount in the queue

//AmountSameStep - amount in the queue that have the same

//step number

//CurrPosQueue - current position in the queue

//CurrNumStep - current number step in the Way

//i - cycle counter

begin

//Initializing the variables

CurrNumStep:= 1;

CurrPosQueue:= 1;

Queue[CurrPosQueue,1]:= StartI;

Queue[CurrPosQueue,2]:= StartJ;

AmountInQueue:= 1;

AmountSameStep:= 1;

//Cycle to review all cells

while AmountInQueue <> 0 do

begin

//Checking if this cell was taken before

for i := CurrPosQueue-1 downto 1 do

if (Queue[CurrPosQueue,1] = Queue[i,1]) and

(Queue[CurrPosQueue,2] = Queue[i,2]) then

begin

Queue[CurrPosQueue,1]:= 0;

Queue[CurrPosQueue,2]:= 0;

end;

//If wasnt taken before, work with it

if (Queue[CurrPosQueue,1] <> 0) and

(Queue[CurrPosQueue,2] <> 0) then

begin

//Assign the coordinates with which work

CoordI:= Queue[CurrPosQueue,1];

CoordJ:= Queue[CurrPosQueue,2];

//On the way, assign the step number to the current

//cell

Way[CoordI,CoordJ]:= CurrNumStep;

//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);

//Turn to the procedure PathOutput to writing the

//path

PathOutput(CoordI, CoordJ);

Writeln;

//The way is found

IsPathFound:= True;

end

//Else looking for an neighboring, available and

//untraveled cell.

//And if found, add to the queue

else

begin

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

CoordJ+1] = 0) then

begin

Inc(AmountInQueue);

Queue[AmountInQueue,1]:= CoordI;

Queue[AmountInQueue,2]:= CoordJ+1;

end;

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

CoordJ] = 0) then

begin

Inc(AmountInQueue);

Queue[AmountInQueue,1]:= CoordI+1;

Queue[AmountInQueue,2]:= CoordJ;

end;

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

CoordJ-1] = 0) then

begin

Inc(AmountInQueue);

Queue[AmountInQueue,1]:= CoordI;

Queue[AmountInQueue,2]:= CoordJ-1;

end;

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

CoordJ] = 0) then

begin

Inc(AmountInQueue);

Queue[AmountInQueue,1]:= CoordI-1;

Queue[AmountInQueue,2]:= CoordJ;

end;

end;

end;

//Decrease AmountSameStep

Dec(AmountSameStep);

//Checking if all cells of the same step have been taken

if AmountSameStep = 0 then

begin

//Shift all next step cells to the left

for i := CurrPosQueue+1 to AmountInQueue do

begin

Queue[i-CurrPosQueue, 1]:= Queue[i, 1];

Queue[i-CurrPosQueue, 2]:= Queue[i, 2];

end;

//Сorrecting all variables after the shift

AmountInQueue:= AmountInQueue - CurrPosQueue;

AmountSameStep:= AmountInQueue;

CurrPosQueue:= 1;

//Increasing the current number step

Inc(CurrNumStep);

end

//Else increase CurrPosQueue

else

Inc(CurrPosQueue);

end;

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

//FindExitBFS

IsPathFound:= False;

FindExitBFS(StartI, StartJ);

//Сheck if the path is found

if not IsPathFound then

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

end;

Readln(SizeI);

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:

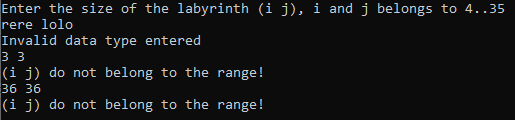


Рисунок 12 – Результаты расчетов

Тест 2

Исходные данные: Некорректный ввод лабиринта

Ожидаемый результат: Просьба перезапустить программу

Полученный результат:

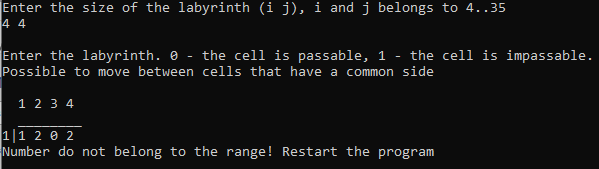


Рисунок 13 – Результаты расчетов

Тест 3

Исходные данные: Некорректный ввод лабиринта

Ожидаемый результат: Просьба перезапустить программу

Полученный результат:

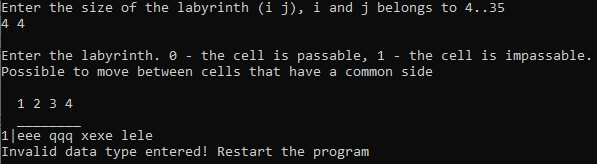


Рисунок 14 – Результаты расчетов

Тест 4

Исходные данные: Некорректный ввод стартовой точки

Ожидаемый результат: Просьба перезапустить программу

Полученный результат:

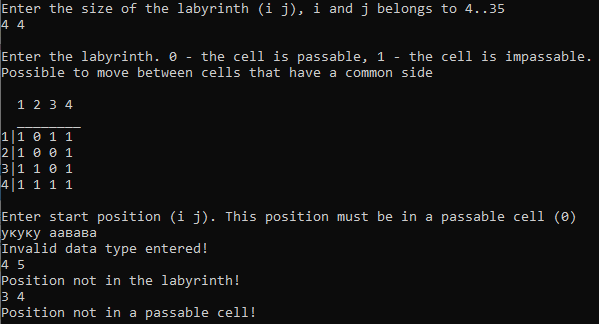


Рисунок 15 – Результаты расчетов

Тест 5

Исходные данные: Неккоректный ввод лабиринта

Ожидаемый результат: Просьба перезапустить программу

Полученный результат:

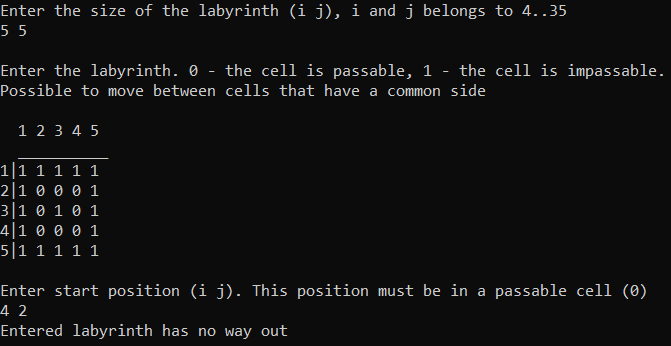


Рисунок 16 – Результаты расчетов

**Тестовая ситуация: корректный ввод данных**

Тест 6

Исходные данные: Лабиринт размерности 87. Стартовая точка (4, 5).

Ожидаемый результат: Первый выход с 4 шагами: (4,5) (4,6) (3,6) (3,7); второй выход с 6 шагами: (4,5) (4,6) (3,6) (2,6) (2,5) (1,5); третий выход с 8 шагами: (4,5) (4,4) (4,3) (4,2) (5,2) (6,2) (7,2) (8,2).

Полученный результат:

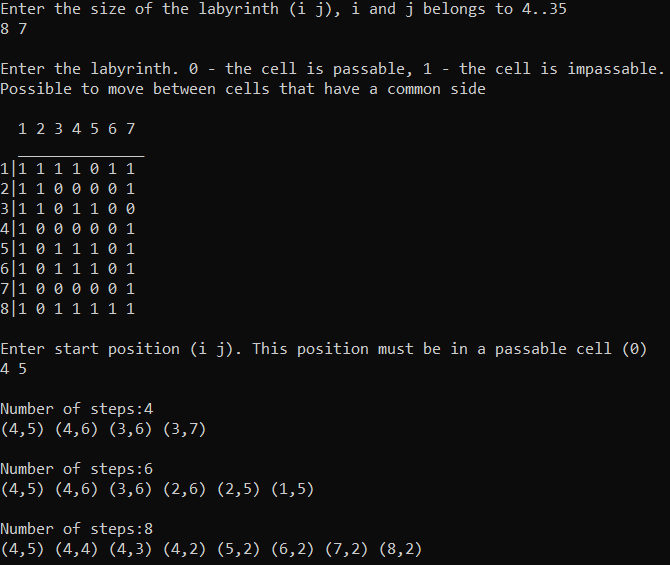


Рисунок 17 – Результаты расчетов

Тест 7

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

Ожидаемый результат: Первый выход с 7 шагами: (5,2) (5,3) (4,3) (3,3) (2,3) (2,2) (1,2); второй выход с 8 шагами: (5,2) (5,3) (4,3) (3,3) (2,3) (2,4) (2,5) (1,5).

Полученный результат:

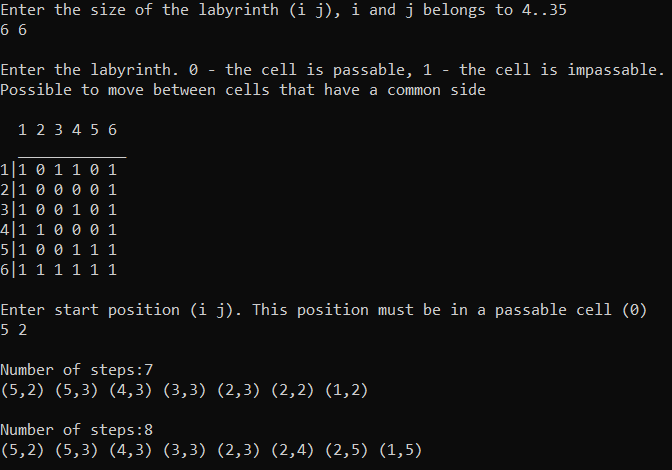


Рисунок 18 – Результаты расчетов

Тест 8

Исходные данные: Лабиринт размерности 1010. Стартовая точка (9, 8).

Ожидаемый результат: Первый выход с 11 шагами: (9,8) (9,9) (8,9) (7,9) (6,9) (5,9) (4,9) (3,9) (3,8) (2,8) (1,8); второй выход с 17 шагами: (9,8) (9,9) (8,9) (7,9) (6,9) (5,9) (4,9) (3,9) (3,8) (3,7) (3,6) (2,6) (2,5) (2,4) (2,3) (2,2) (1,2).

Полученный результат:

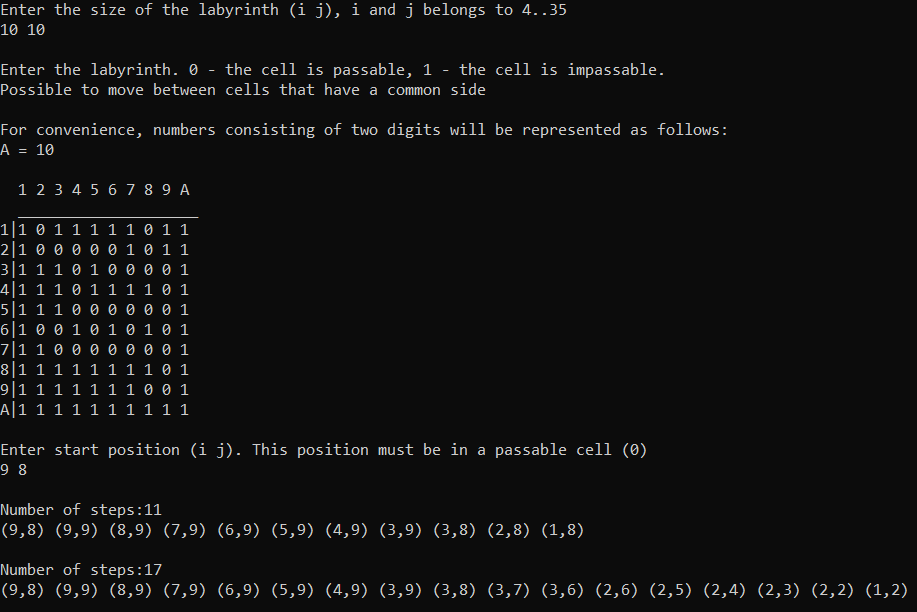


Рисунок 19 – Результаты расчетов

Тест 9

Исходные данные: Лабиринт размерности 77. Стартовая точка (6, 4).

Ожидаемый результат: Первый выход с 8 шагами: (6,4) (6,3) (5,3) (4,3) (3,3) (2,3) (2,2) (1,2).

Полученный результат:

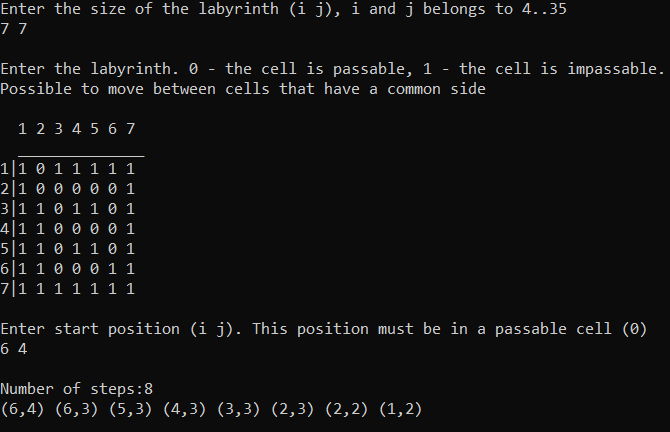


Рисунок 20 – Результаты расчетов

Приложение С

(обязательное)

Код с генератором лабиринта

Program BFSrandom;

{

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 path to each еxit

}

{$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: 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

//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 FindExitBFS(StartI, StartJ: Byte);

var

Queue : array [1..MaxSizes\*4, 1..2] of Byte;

CoordI, CoordJ : Byte;

AmountInQueue, AmountSameStep, CurrPosQueue,

CurrNumStep, i : Byte;

//Queue - queue cell

//CoordI - current coordinates by i

//CoordI - current coordinates by j

//AmountInQueue - amount in the queue

//AmountSameStep - amount in the queue that have the

//same step number

//CurrPosQueue - current position in the queue

//CurrNumStep - current number step in the Way

//i - cycle counter

begin

//Initializing the variables

CurrNumStep:= 1;

CurrPosQueue:= 1;

Queue[CurrPosQueue,1]:= StartI;

Queue[CurrPosQueue,2]:= StartJ;

AmountInQueue:= 1;

AmountSameStep:= 1;

//Cycle to review all cells

while AmountInQueue <> 0 do

begin

//Checking if this cell was taken before

for i := CurrPosQueue-1 downto 1 do

if (Queue[CurrPosQueue,1] = Queue[i,1]) and

(Queue[CurrPosQueue,2] = Queue[i,2]) then

begin

Queue[CurrPosQueue,1]:= 0;

Queue[CurrPosQueue,2]:= 0;

end;

//If wasnt taken before, work with it

if (Queue[CurrPosQueue,1] <> 0) and

(Queue[CurrPosQueue,2] <> 0) then

begin

//Assign the coordinates with which work

CoordI:= Queue[CurrPosQueue,1];

CoordJ:= Queue[CurrPosQueue,2];

//On the way, assign the step number to the

//current cell

Way[CoordI,CoordJ]:= CurrNumStep;

//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);

//Turn to the procedure PathOutput to writing

//the path

PathOutput(CoordI, CoordJ);

Writeln;

end

//Else looking for an neighboring, available and

//untraveled cell. And if found, add to the queue

else

begin

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

CoordJ+1] = 0) then

begin

Inc(AmountInQueue);

Queue[AmountInQueue,1]:= CoordI;

Queue[AmountInQueue,2]:= CoordJ+1;

end;

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

(Way[CoordI+1, CoordJ] = 0) then

begin

Inc(AmountInQueue);

Queue[AmountInQueue,1]:= CoordI+1;

Queue[AmountInQueue,2]:= CoordJ;

end;

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

CoordJ-1] = 0) then

begin

Inc(AmountInQueue);

Queue[AmountInQueue,1]:= CoordI;

Queue[AmountInQueue,2]:= CoordJ-1;

end;

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

(Way[CoordI-1, CoordJ] = 0) then

begin

Inc(AmountInQueue);

Queue[AmountInQueue,1]:= CoordI-1;

Queue[AmountInQueue,2]:= CoordJ;

end;

end;

end;

//Decrease AmountSameStep

Dec(AmountSameStep);

//Checking if all cells of the same step have been

//taken

if AmountSameStep = 0 then

begin

//Shift all next step cells to the left

for i := CurrPosQueue+1 to AmountInQueue do

begin

Queue[i-CurrPosQueue, 1]:= Queue[i, 1];

Queue[i-CurrPosQueue, 2]:= Queue[i, 2];

end;

//Сorrecting all variables after the shift

AmountInQueue:= AmountInQueue - CurrPosQueue;

AmountSameStep:= AmountInQueue;

CurrPosQueue:= 1;

//Increasing the current number step

Inc(CurrNumStep);

end

//Else increase CurrPosQueue

else

Inc(CurrPosQueue);

end;

end;

Begin

//Call the procedure to generate labyrinth

Generator;

//Initialize the variables and go to the procedure

//FindExitBFS

FindExitBFS(StartI, StartJ);

Readln;

End.

Приложение D

(обязательное)

Тестовые наборы к коду с генератором лабиринта

Тест 1

Полученный результат:

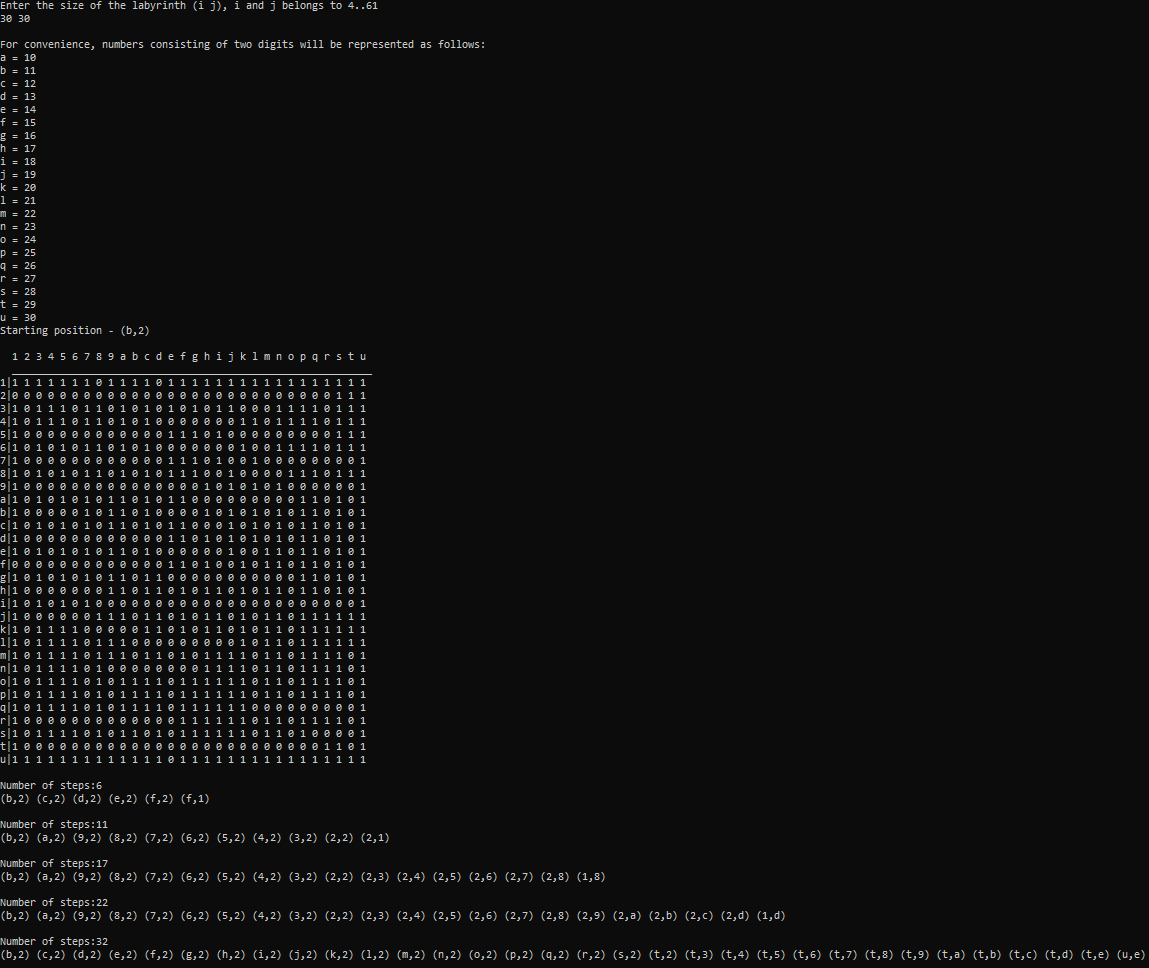


Рисунок 21 – Результаты расчетов

Тест 2

Полученный результат:

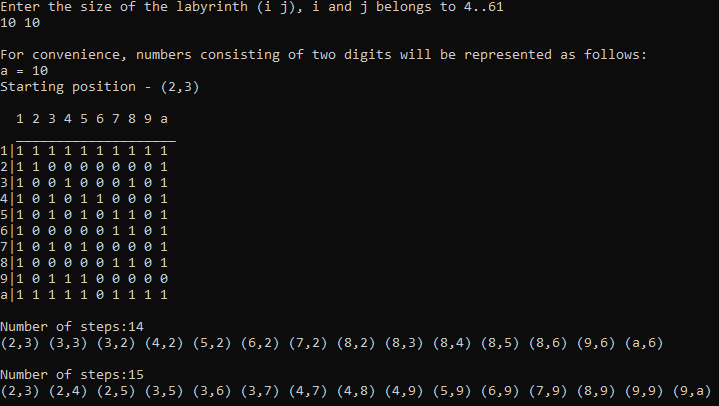


Рисунок 22 – Результаты расчетов

Тест 3

Полученный результат:

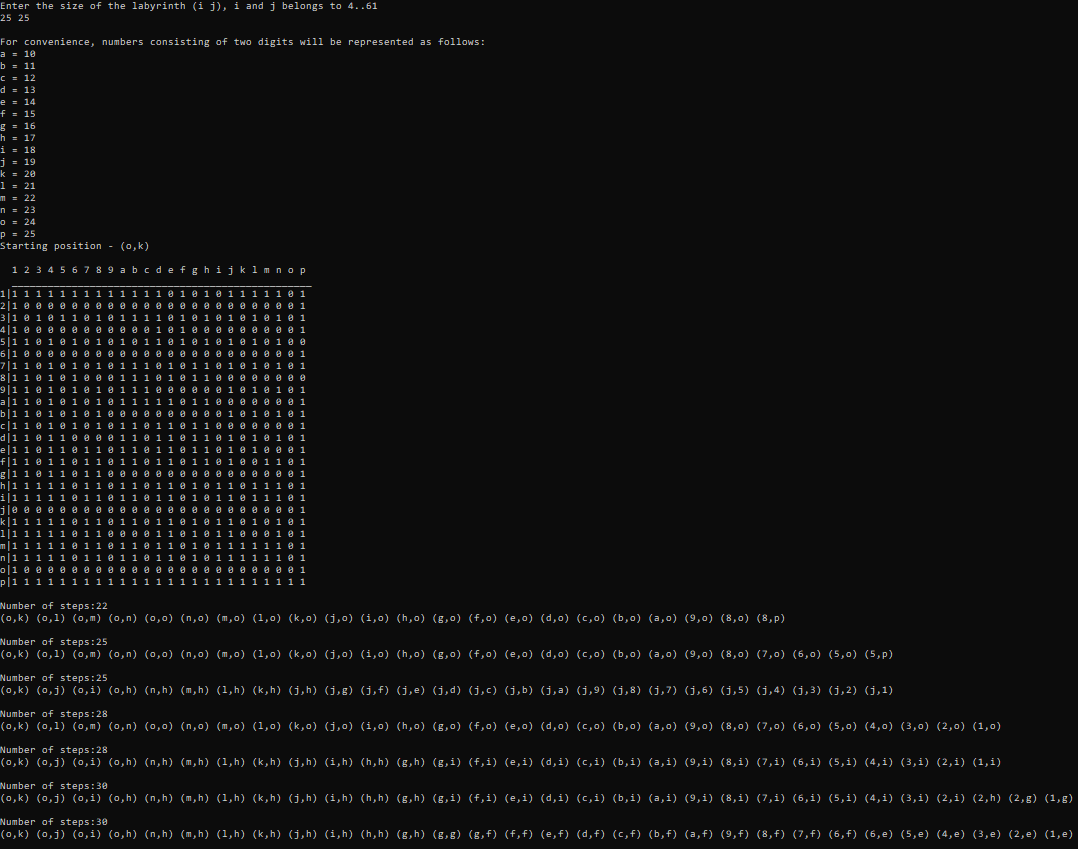


Рисунок 23 – Результаты расчетов

Тест 4

Полученный результат:

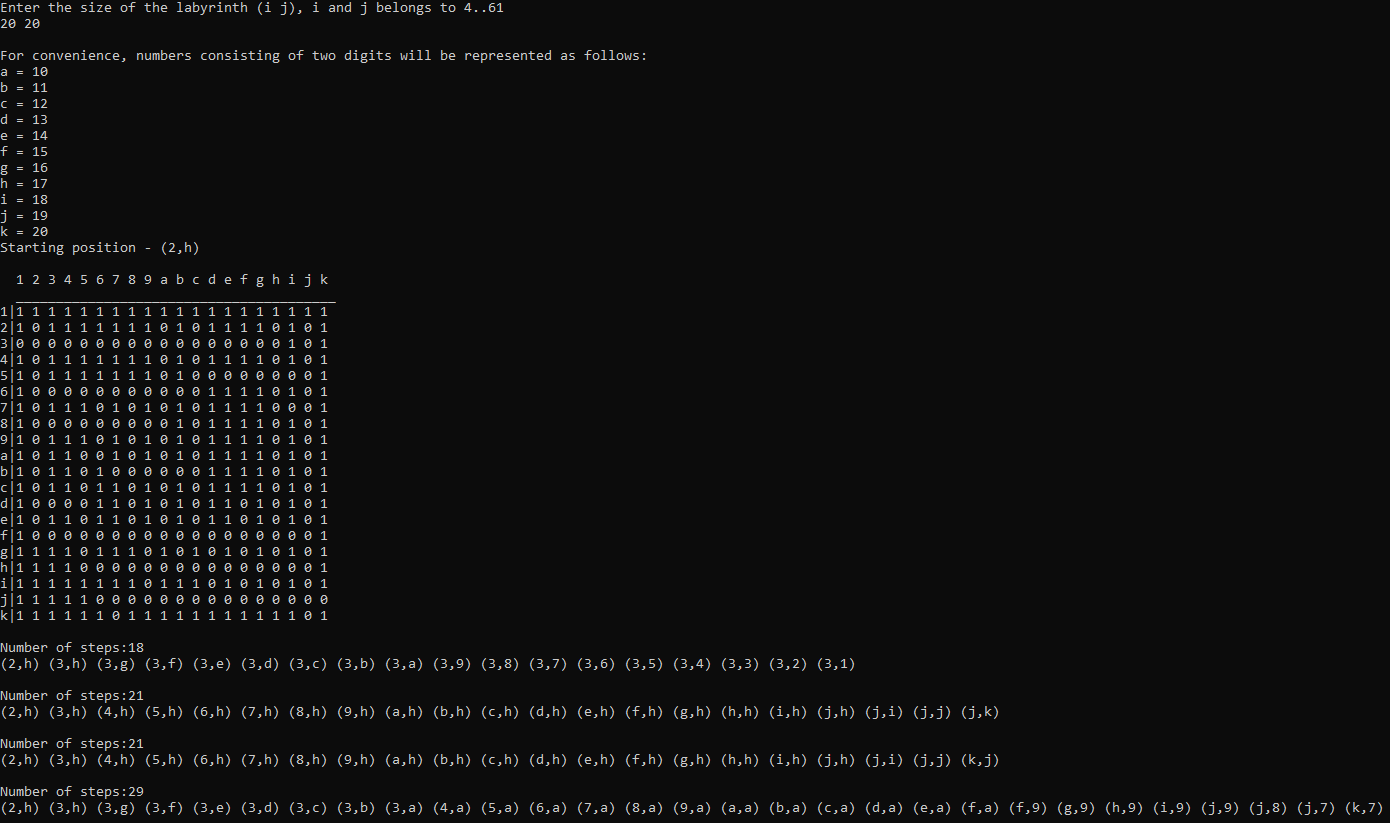


Рисунок 24 – Результаты расчетов

Тест 5

Полученный результат:

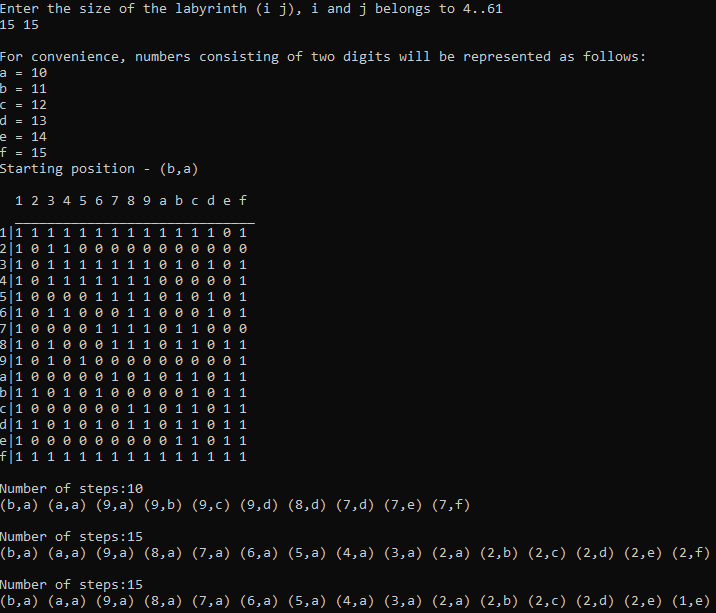


Рисунок 25 – Результаты расчетов