Problem of Project 8

- Make the computer find some simple paths through a maze.
- Input: $c$ by $r$ lattice of squares, some “filled in” and the rest empty. Assume the squares at $(sx, sy)$ and $(gx, gy)$ are empty.
- Output: All (simple) paths from square $(sx, sy)$ to square $(gx, gy)$ that use adjacent empty squares.
Here is the maze, row by row...

Here is the 5 by 5 shape, start 0 0, and goal 4 4

(Use the above code to read the input!)
1. A **Square** (spot in the maze) is a pair \((x, y)\) of column and row coordinates.

2. A **Path** is a sequence of **Squares**. Let’s implement a **Path** with a singly-linked list of pair data \((x, y)\).

3. 2-dimensional (dynamically allocated) Array

4. To start, declare fields \((A, c, r, sx, sy, gx, gy)\) to be shared by the methods of class **Maze**. Construct:
   
   ```java
   A = new int[c][r];
   ```

5. Declare another field \((M)\) for a 2-dim array of (character) “Marks”
   
   ```java
   M = new char[c][r];
   ```
   
   to be used by more advanced and efficient search algorithms.

6. **EARLY!!** Make sure to write methods to print the used parts of the maze \(A\) and Marks \(M\) arrays.
Another important class of objects: The PathNode class

Let’s implement a Path with a singly-linked list of pair data \((x,y)\).

```java
public class PathNode {
    int x; int y; PathNode prev;
    PathNode(int px,int py,PathNode pprev) {
        x = px; y = py; prev = pprev; }
    void printForwards() {
        if(prev!=null) {
            prev.printForwards();
            System.out.print(",");
        }
        printOne();
    }
    void printOne() {
        System.out.print("(+x+","+y+)");
    }
}
```
public static void main(String[] zzzz)
{
  PathNode n1 = new PathNode(0,0,null);
  PathNode n2 = new PathNode(0,1,n1);
  PathNode n3 = new PathNode(1,1,n2);
  PathNode n4 = new PathNode(2,1,n3);
  System.out.println("Test Forwards printing:");
  n4.printForwards();
}

Output

Test Forwards printing:
(0,0), (0,1), (1,1), (2,1)


Image of a DISCONNECTED Maze

Graph Representation

Vertices: Array positions containing 0’s.

Arcs or Edges: position PAIRS which are adjacent.
A tree is also a graph
A tree is a connected graph with NO CYCLES.

What is a cycle?
A cycle is a piece of a graph that looks like a circle. (Sequence of edges of the form \((v_1, v_2), (v_2, v_3), \ldots, (v_i, v_{i+1}), \ldots (v_k, v_1)\).)
The graph of a maze usually has cycles, so finding paths in mazes is harder than in trees.

How to READ the ABOVE details

- The second vertex \(v_{i+1}\) of each pair \((v_i, v_{i+1})\) is the SAME AS the first vertex of the next pair \((v_{i+1}, v_{i+2})\).
- The second vertex \(v_1\) of the LAST PAIR \((v_k, v_1)\) is the SAME AS the first vertex in \((v_1, v_2)\).
Here is the rule for telling which array positions are neighbors (form an edge):

\[(i, j-1)\]
\[(i-1, j)\] — [\((i, j)\) — [\((i+1, j)\]
\[(i, j+1)\]
Table Based Coding of our graph adjacency rule:

```
int Rule[][] = { {1, 0}, {0, 1}, {0,-1}, {-1,0} };
//Position (i,j) is adjacent to (x,y)
//if and only if:
//(1) 0 <= i < c, 0 <= j < r, 0 <= x < c AND 0 <= y < r
//and (2) A[i][j]==0 AND A[x][y]==0
//and (3) x == i + Rule[q][0] AND y == j + Rule[q][1]
// for ONE OF THE q=0, 1, 2, or 3.
```

```
( x==i+(0) ,  y=j+(-1) )
```

```
(i-1,j)---(i,j)---(i+1,j)
```

```
(i,j+1)
```
This idea is flexible! Let’s do a hexagonal maze.

Here is the rule for telling which array positions are neighbors (form an edge):

\[(i, j-1) \quad (i+1, j+1)\]
\[(i-1, j) \quad (i, j) \quad (i+1, j)\]
\[(i-1, j-1) \quad (i, j+1)\]
Table Based Coding of our HEXAGONAL graph adjacency rule:

```c
int Rule[] []
    = { {-1, 0}, {-1, 1}, {0,-1}, {0,1}, {1,-1}, {1,0}};
//Position (i,j) is adjacent to (x,y)
//if and only if:
//(1) 0 <= i < c, 0 <= j < r, 0 <= x < c AND 0 <= y < r
//and (2) A[i][j]==0 AND A[x][y]==0
//and (3) x == i + Rule[q][0] AND y == j + Rule[q][1]
// for ONE OF THE q=0, 1, 2, 3, 4, or 5.
```

```
(i,j-1)   (i+1,j-1)

(i-1,j)  (i,j)  (i+1,j)

(i-1,j+1)  (i,j+1)
```
Graphs Modelling Adjacencies: EXTREMELY Flexible Idea!

1. Your graph can be completely imaginary, not based on any kind of grid.

2. The graph can represent Cities (by vertices) and Direct Airline Flights (by edges). The problem of finding a sequence of flights from given origin Cities to a Destination Cities is just like our maze problem.

3. **Social Networks:** Vertices represent People and Edges represent Friendships.

4. The Internet: Hosts and Direct Connections.

**Current Networking Challenge—Wireless/Mobile**

The network CHANGES FREQUENTLY. How is the routing information that mobile stations and routers need maintained efficiently?
Problem of Project 8:

First, make the computer find one simple path through a planar square lattice based maze.

Input: c by r lattice of squares, some “filled in” and the rest empty. Assume the squares at (sx, sy) and (gx, gy) are empty.

Output: One simple path from square (sx, sy) to square (gx, gy) that use adjacent empty squares.
Sample input:

5 5
0 0
4 4
0 0 1 1 1
1 0 1 0 0
0 0 0 0 0
0 1 0 1 0
1 0 0 0 0

Sample output (first, reprint input maze):

Input maze:
0 0 1 1 1
1 0 1 0 0
0 0 0 0 0
0 1 0 1 0
1 0 0 0 0
Then, a print a solution

Input maze:

0 0 1 1 1
1 0 1 0 0
0 0 0 0 0
0 1 0 1 0
1 0 0 0 0

Solution

(0,0), (1,0), (1,1), (1,2), (2,2), (3,2), (3,1), (4,1), (4,2), (4,3), (4,4)

(visualized better:)

(0,0) (1,0)
    (1,1)   (3,1) (4,1)
    (1,2) (2,2) (3,2) (4,2)
    (4,3)
    (4,4)
OR, (FOR MORE POINTS) print a solution picture!

One solution:

A B 1 1 1
1 C 1 G H
0 D E F I
0 1 0 1 J
1 0 0 0 K
Solution Strategy I

1. Input the maze dims $c, r$, row and column coordinates of the start and goal squares, and then $c \times r$ 0’s or 1’s to represent the filled and empty squares of the maze. (DONE)

2. Print the maze; to verify it was formed correctly and to help develop code to print the path pictures.

3. “Every journey begins with the first step.” The key procedure will be called with parameter “path containing first square ($sx, sy$) only.”

4. What should the key procedure do? **Print one path that ends at the goal square ($gx, gy$), and return true; otherwise return false**

5. What data does the key procedure need? It needs access to the whole maze, at least.

6. Apply the recursion/divide and conquer pattern: Make the key procedure recursive. Let’s name it PrintSolution(...).
7. The solution path must be simple, which means it must not use the same square twice. (Otherwise, there might be an infinite number of solutions).

8. So, `PrintSolution(...)` MUST be able to tell which squares are in the path found SO FAR. This data varies with WHAT SQUARES ARE IN THE PATH FOUND SO FAR.

9. So, let's try making the PATH FOUND SO FAR be a parameter to `PrintSolution(...)`.

10. (But we may also need to keep “marks” somewhere to mark when a square is visited. So, declare an array for the marks in the Maze class. But for a slow version, we don't need it.)
11. Draft algorithm pseudo-code for

```java
boolean PrintSolution( PathNode pLast )
```

//PRECONDITION: pLast points to a node holding the most currently found square in the path found so far. This node starts a (linked) list holding ALL the squares in the path found so far.

//POSTCONDITION: A path, if any, beginning with the given path so far and continuing to the goal square $(gx,gy)$ is printed. It DOES NOT USE any square in the path found so far, nor does it use a ‘‘filled’’ square or a square outside the c by r maze.

The return value is true if there is such a path, false otherwise.
12. boolean PrintSolution( PathNode pLast ) {
    if( (pLast.x,pLast.y) is the goal square )
    { print the path from pLast back; return true; }
    else {
        int q; //LOCAL, ACTIVATION REC. VARIABLE!!
        PathNode pNext = new PathNode(??,??,pLast);

        for(q=0; q<4; q++) {
            if( the square adjacent to (pLast.x,pLast.y) according to Rule[q]
                is in the maze, AND is empty,
                AND NOT in the path so far )
            {
                Set pNext.x, pNext.y to this square;
                boolean success = PrintSolution( pNext );
                if(success) /*YOU FIGURE OUT*/
            } } /*for loop finish*/ return ???; }
Remember

class PathNode { public int x, public int y, public PathNode pNext; }

When the if statement in PrintSolution( PathNode pLast ) detects that (pLast.x,pLast.y) is (gx,gy), it should print the whole path starting at pLast back to the start square (sx,sy). What should it do next?

A Recurse and return what the recursion returned.
B Recurse and return true.
C Extend the path in each of 4 directions and recurse each time.
D Try each of 4 directions to extend the path and recurse only under special conditions.
E Recurse and return false.
WHOOPS!! None of the choices are correct!!
Remember

class PathNode { public int x, public int y, public PathNode pNext; }

When the if statement in PrintSolution( PathNode pLast ) detects that (pLast.x,pLast.y) is (gx,gy), it should print the whole path starting at pLast back to the start square (sx,sy). What should it do next?

FOR PART 1: PRINT JUST ONE SOLUTION
It should print the solution and return true.

FOR PART 3: PRINT ALL SOLUTIONS OF MINIMUM LENGTH
It should print the solution and return false (so callers continue to search).
But only steps from distance D to distance D+1 should be tried.
boolean PrintSolution(PathNode pLast) {
    if ((pLast.x, pLast.y) is the goal square) {
        print the path from pLast back; return true;
    }
    else {
        int q; // LOCAL, ACTIVATION REC. VARIABLE!!
        PathNode pNext = new PathNode(??, ??, pLast);

        for (q = 0; q < 4; q++) {
            if (the square adjacent to (pLast.x, pLast.y)
                according to Rule[q]
                is in the maze, AND is empty,
                AND NOT in the path so far)
                {
                Set pNext.x, pNext.y to this square;
                boolean success = PrintSolution( pNext );
                if (success) /* YOU FIGURE OUT */
                } }
        return ???
    }
} /* for loop finish */
Remember

class PathNode { public int x, public int y, public PathNode pNext; }

Under the false branch after the if statement:

A  Recurse and return what the recursion returned.
B  Try each of 4 directions to extend the path and recurse, but return the first time a recursion returns true;
C  Extend the path in each of 4 directions and recurse each time.
D  Try each of 4 directions to extend the path and recurse always when extending the path is possible.
E  Recurse and return false.
When an item is added to an end of a linked list, are the old items, those already in the list, copied?

A  Never
B  Always
C  Sometimes
When an item is added to an extendable, partially filled array (Java ArrayList), are the old items, those already in the array, copied? Suppose the capacity doubling strategy is used.

A Never
B Always
C Sometimes
Suppose an extendable, partially filled array (Java ArrayList) starts EMPTY with a capacity of 4. Suppose 4 items are then added. Suppose the capacity is DOUBLED when more array space is needed. So when the 5th item is given, a new ItemType[8] is done.

When the 5th item is given, HOW MANY items must be copied into the new array?

A 1
B 2
C 4
D 5
E 8
Back to Path Finding in a Maze
iClicker Question!!

Remember

class PathNode { public int x, public int y, public PathNode pNext; }

When the if statement in PrintSolution( PathNode pLast ) detects that (pLast.x,pLast.y) is (gx,gy), it should print the whole path starting at pLast back to the start square (sx,sy). What should it do next?

A  Recurse and return what the recursion returned.
B  Recurse and return true.
C  Extend the path in each of 4 directions and recurse each time.
D  Try each of 4 directions to extend the path and recurse only under special conditions.
E  Recurse and return false.
WHOOPS!! None of the choices are correct!!

Remember

class PathNode { public int x, public int y, public PathNode pNext; } 

When the if statement in PrintSolution( PathNode pLast ) detects that (pLast.x,pLast.y) is (gx,gy), it should print the whole path starting at pLast back to the start square (sx,sy). What should it do next?

FOR PART 1: PRINT JUST ONE SOLUTION
It should print the solution and return true.

FOR PART 3: PRINT ALL SOLUTIONS OF MINIMUM LENGTH
It should print the solution and return false (so callers CONTINUE TO SEARCH).
But only steps from distance D to distance D+1 should be tried.
How the LOCAL, STACKED q variables coordinate the search

```java
int Rule[][] = { {1, 0}, {0, 1}, {0,-1}, {-1,0} }; 
for(int q=0; q<4; q++) {
    if( the square adjacent to (pLast.x,pLast.y) 
        according to Rule[q] is in the maze, 
        AND is empty, 
        AND NOT in the path so far ) 
    {
        Set pNext.x, pNext.y to this square;
        boolean success = PrintSolution( pNext );
        maybe return;
    }
}
```

s 0 0<-- DEAD END!!! SEARCH DEMONSTRATED ON DOC-CAM.
1 0 1
1 0 g
Java 2-d array shorthand

```java
int Rule[][] = { {1, 0}, {0, 1}, {0,-1}, {-1,0} };

is shorthand for:

int Rule[][] = new int[4][2];
Rule[0][0] = 1;
Rule[0][1] = 0;
Rule[1][0] = 0;
Rule[1][1] = 1;
Rule[2][0] = 0;
Rule[2][1] = -1;
Rule[3][0] = -1;
Rule[3][1] = 0;
```
// Start with the start square marked, and no others.
boolean PrintSolution( PathNode pLast ) {
    if( (pLast.x,pLast.y) is the goal square )
    { print the path from pLast back; return true; }
    else {
        int q; //LOCAL, STACKED VARIABLE!!
        PathNode pNext = new PathNode(????);
        for(q=0; q<4; q++) {
            if( the square adjacent to pLast->data
                according to Rule[q]
                is in the maze, AND is empty,
                AND IS NOT MARKED )
            { MARK THIS SQUARE
                pNext.x,pNext.y = this square;
                success=PrintSolution( pNext );
                if(success) /*YOU FIGURE OUT*/
            }
        }
        return ???; }
}
The next sequence of slides shows that the algorithm that
▶ MARKS each square THE FIRST TIME it finds it, and
▶ does not extend the path-so-far into already marked squares
is more efficient.
These steps are avoided when marking is done.
Path–search steps AVOIDED because of the marks.
Start

The search continues...

Goal

and finds a path to the goal.
Array of things with STATE, two ways

Old way

class Maze {
    int A[][];
    char Marks[][];
    Maze() { A = new int[c][r];
              Marks = new char[c][r]; }
};

if( A[i][j] == 1 )
    Marks[i][j] = 'V';

“Parallel Arrays”

In Fortran or other record-less languages, use Parallel Arrays to simulate records.

New way

class Maze {
    class Square {
        int Filled;
        char Mark; }
    Square A[][];
    Maze() { A = new Square[c][r];
              for i,j A[i][j] = new Square(); }
};

if( A[i][j].Filled == 1 )
    A[i][j].Mark = 'V';

‘‘Array of structs’’
Design Ideas I

1. Design a class named Maze to model one maze, and have methods for reading it, printing it, printing paths, and printing the solution path. Also, add a data member for the labels or marks.

2. Figure out how to get the right value so you can print the first solution square with an A, next B, etc. (NOTE: The alphabet should repeat when you go beyond Z. Use the Java % 26 operator, or subtract 26 when necessary)

3. The 2-dim A array and size variable are **private data members** so they may be accessed by any methods that need them. Such methods include PrintSolution()

4. The Maze class uses some kind of linked list node class type PathNode to implement the linked list of square that is the path found so far.
5. The Maze class uses ANOTHER dynamically allocated array to hold “labels”, so that visited squares can be “marked”.

Alternative:
Maze uses an array of object references. Each object holds both the filled vs. empty information and the label.
Design Ideas III

6. Analyze the pseudo-code in the outline of the `PrintSolution()` method to design helper methods to help implement its operations: For example,

6.1 Test if a given index pair (denoting a square) is in a linked list.
6.2 Test if a given index pair denotes a marked square.
6.3 Test if an index pair is within the $0 \ldots c-1$, $0 \ldots r-1$ range AND it corresponds to an “empty” square.

These must be member functions of class Maze so they can access the (private) the 2-dim arrays and the size variable.
Continue the Design, build and test INCREMENTALLY

1. Invent (or use our given) names for the data and function members.

2. Start writing the file Maze.java, and try to write pre- and postconditions to document what you want each method to do.

3. Check that the function declarations, method declarations and pre- and postconditions are consistent with each other and with the intentions you have in your mind.

4. TEST OFTEN AND EARLY: Start with REALLY TRIVIAL CASES!!
Part 2:
Use Breadth-First Labelling to find EVERY SQUARE’S distance from the start square.

Distance, or length of a minimum length path, here means “Minimum number of legal steps within the maze” between two squares.

Part 3:

- Use Part 2 results.
- Use backtracking without marking to find ALL minimum length paths from the start to the goal square.
- Restrict steps so we ONLY go from a square with distance $D$ to a square of distance $D + 1$. 
Search versus Traverse

1. **SEARCH:** Find one path from START to GOAL
2. **TRAVERSE:** Find all squares that are reachable from the START square.
The main idea is to put and retain a “mark” on each vertex as soon as we determine that it can be reached from the start vertex. This way, to test if a vertex has been found takes CONSTANT TIME, instead of the $O(\text{length of list})$ time to look in a list.
class Maze {
    // Data Members
    int A[][[]];  int r,c;
    int sx, sy, gx, gy;
    // 2-dim partially filled array
    // plus start and goal coordinates
    // to hold Part1 input data
    char Marks[][[]]; // See Lect. Notes
};

MyMaze

A

n

sx

sy

gx

gy

Marks
Variation: Breadth-first search

1. Label the start vertex 0 and put it in a QUEUE. Initialize the labels of the others so the label means “not labelled yet”

Idea
Use integer labels: $-1$ means “unlabelled” and $0, 1, 2, \text{etc}$ mean distances.

2. While the QUEUE is not empty, remove a square $S$, retrieve its label $D$, and for each unlabelled square adjacent to $S$:
   - Label that square $D + 1$.
   - Put that square in the QUEUE.
Demo

Start

Diagram of a network with start and end points.
The final breadth-first traversal labelling.
Here are the some of the first steps in the Breadth-first labelling algorithm run on this example
Label the start square 0 and put it in the queue.

(0,0)
Remove head square, label and enqueue its neighbors.
Remove head square. It has no unlabelled neighbors.

(1,0) (0,1)
Remove head square, label and enqueue its one unlabelled neighbor.

\[(2,0) \ (1,0)\]
Remove head square, label and enqueue its one unlabelled neighbor.

(3,0) (2,0) (1,0)
Remove head square, label and enqueue its two unlabelled neighbors.

(4,0) (3,1) (3,0)
Remove head square, label and enqueue its one unlabelled neighbor.

((3,2)(4,0)(3,1)(3,0))
See how another square, and then a dead end square, is processed.

(5,0)(3,2)(4,0) (3,1) (3,0)
Processing the next square causes **TWO** more squares to be labelled and enqueued.

What happens when the dead end square is processed?
What happens when the dead end square is processed?

NOTHING! except it’s removed from the queue.
After a few more steps, we have three squares labelled 7 and put in the queue.

(5,2)(4,3)(1,2)(4,2)(2,2)
Part3: All-paths finding algorithm

PrintSolutionS( PathNode pLast ) {
    if( pLast.x,pLast.y is the goal square )
        { print the path found so far; return; }
    else{
        int q;  //LOCAL VARIABLE!!
        PathNode pNext = new PathNode();
        pNext.prev = pLast;  //MUST set data too!!
        for(q=0; q<4; q++) {
            if( the square adjacent to pLast->data
                according to Rule[q] is in the
                maze AND is EMPTY, AND the "D-rule" is true,
                AND it’s NOT in the path so far )
                {
                    pNext.x,pNext.y = this square;
                    PrintSolutionS( pNext );
                } /*for loop finish*/
        } return; }
}
The ACTIVATION RECORD (aka execution) STACK stores which neighbor each activation is up to.

The use of a stack in path-finding is must be explained.
What else?

We want you to print a count of the solutions.
You figure out a way.
Either use a variable that is shared among activations of PrintsolutionS,
or make it return the correct value after calculating it recursively.
Graph Search and Traversal by Labelling/Backtrack Algorithms.
Search and traversal of a maze of squares ABSTRACTS to problems on Graphs.