Multi-dimensional Arrays

Part 1: Matrix

A Matrix

- We can represent a matrix by a list of lists
  - E.g. a 4 x 10 matrix

```
>>> pprint(m)
[[1, 1, 1, 0, 1, 0, 0, 1, 0, 1],
 [1, 0, 1, 0, 0, 0, 1, 0, 0, 0],
 [0, 0, 1, 1, 0, 0, 0, 1, 1, 0],
 [0, 1, 1, 1, 1, 0, 0, 0, 1, 1]]
```

Matrix Exercises

- You can assume all the entries are integers
- You can use the functions provided in the lecture
  - `createZeroMatrix()`, `mTightPrint()`, etc.
- There may be a lot of the code online, but you may want to try to code by yourself
- And the package numpy (and some other packages) does have these functionalities. But we want to learn how to code these

Task 1: Transpose

$$\begin{bmatrix} 6 & 4 & 24 \\ 1 & -9 & 8 \end{bmatrix}^T = \begin{bmatrix} 6 & 1 \\ 4 & -9 \\ 24 & 8 \end{bmatrix}$$

- Write a function `transpose(m)` to transform an r × c matrix to a c × r matrix

```
>>> pprint(m)
[[1, 1, 1, 0, 1, 0, 0, 1, 0, 1],
 [1, 0, 1, 0, 0, 0, 1, 0, 0, 0],
 [0, 0, 1, 1, 0, 0, 0, 1, 1, 0],
 [0, 1, 1, 1, 1, 0, 0, 0, 1, 1]]
>>> pprint(transpose(m))
[[1, 1, 0, 0],
 [1, 0, 0, 1],
 [1, 1, 1, 1],
 [0, 0, 1, 1],
 [1, 0, 0, 1],
 [0, 0, 0, 0],
 [0, 1, 0, 0],
 [1, 0, 1, 0],
 [0, 0, 1, 1],
 [1, 0, 0, 1]]
```

- Try?

Task 1: Transpose

$$\begin{bmatrix} 6 & 4 & 24 \\ 1 & -9 & 8 \end{bmatrix}^T = \begin{bmatrix} 6 & 1 \\ 4 & -9 \\ 24 & 8 \end{bmatrix}$$

- Write a function `transpose(m)` to transform an r × c matrix to a c × r matrix

```
def transpose(m):
    r = len(m)
    c = len(m[0])
    output = createZeroMatrix(c,r)
    for i in range(r):
        for j in range(c):
            output[j][i] = m[i][j]
    return output
```

- Challenge: Try one line list comprehension code?

Task 2

- Given two matrices A and B

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

- Compute the multiplication C = AB

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

E.g. c_{21} involve the row 2 in A and column 1 in B However, our row and column numbers start with 0 in Python

- Such that

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

Task 2

- Write a function to multiply two matrices.

```
>>> m1 = [[1,2,3],[5,6,7],[9,10,11],[13,14,15]]
>>> m2 = [[4,3,2,1,8,1],[1,2,3,4,3,1],[5,6,7,8,1,2]]
>>> pprint(matMul(m1,m2))
[[21, 25, 29, 33, 17, 9],
 [61, 69, 77, 85, 65, 25],
 [101, 113, 125, 137, 113, 41],
 [141, 157, 173, 189, 161, 57]]
```

- Try?

Task 2

Check the matrix dimensions

Calculate each c_{ij}

```
def matMul(m1,m2):
    r1 = len(m1)
    c1 = len(m1[0])
    r2 = len(m2)
    c2 = len(m2[0])
    if c1 != r2:
        print("Matrices not match")
        return
    output = createZeroMatrix(r1,c2)
    for i in range(r1):
        for j in range(c2):
            cij = 0
            for k in range(c1):
                cij += m1[i][k]*m2[k][j]
            output[i][j] = cij
    return output
```

Sum by k in range(c1) (== r2)

- Such that

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

Task 2

- Given two matrices A and B

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

k

k

- Compute the multiplication C = AB

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

- Such that

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

Task 3 Minor Matrix

- Write a function `minorMatrix(m,i,j)` to find the minor matrix of m ***without*** row `i` and column `j`.

```
>>> pprint(m2)
[[4, 3, 2, 1, 8, 1],
 [1, 2, 3, 4, 3, 1],
 [5, 6, 7, 8, 1, 2],
 [4, 3, 2, 1, 8, 1],
 [1, 2, 3, 4, 3, 1],
 [5, 6, 7, 8, 1, 2]]
>>> pprint(minorMatrix(m2,2,4))
[[4, 3, 2, 1, 1],
 [1, 2, 3, 4, 1],
 [4, 3, 2, 1, 1],
 [1, 2, 3, 4, 1],
 [5, 6, 7, 8, 2]]
```

- Actually there is no formal definition of "minor matrix" but only *minors*
  - A minors is the determinant of our definition of minor matrix

Task 3 Minor Matrix

- Try?

For each row
except the row i

```
def minorMatrix(m,i,j):
    output = []
    for row in (m[:i]+m[i+1:]):
        output.append(row[:j]+row[j+1:])
    return output
```

Add that row to
output *without*
column j

Task 4 Determinant

- Assume the input A is a square matrix
- No matter how big the matrix is
  - You take the element of the first row, and find the determinants of all their minor matrices

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} \square & \square & \square \\ \square & e & f \\ \square & h & i \end{vmatrix} - b \begin{vmatrix} \square & \square & \square \\ d & \square & f \\ g & \square & i \end{vmatrix} + c \begin{vmatrix} \square & \square & \square \\ d & e & \square \\ g & h & \square \end{vmatrix}$$

$$= a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

$$= aei + bfg + cdh - ceg - bdi - afh.$$

If the minor is not 2 x 2

↓

Recursion!

Task 4 Determinant

- Assume the input A is a square matrix
- No matter how big the matrix is
  - You take the element of the first row, and find the determinants of all their minor matrices

$$\det\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}\right) = a\det\left(\begin{bmatrix} e & f \\ h & i \end{bmatrix}\right) - b\det\left(\begin{bmatrix} d & f \\ g & i \end{bmatrix}\right) + c\det\left(\begin{bmatrix} d & e \\ g & h \end{bmatrix}\right)$$

$$= a(ei - fh) - b(di - fg) + c(dh - eg)$$

$$= aei + bfg + cdh - afh - bdi - ceg$$

If the minor is not 2 x 2

↓

Recursion!

Task 4 Determinant

- Sample output:

```
>>> m = [[6,1,1],[4,-2,5],[2,8,7]]
>>> det(m)
-306
>>> m = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
>>> det(m)
0
```

$$\begin{vmatrix} 6 & 1 & 1 \\ 4 & -2 & 5 \\ 2 & 8 & 7 \end{vmatrix}$$

- Try?

```python
def det(m):
    if len(m) == 1:
        return m[0][0]
    if len(m) == 2:
        return m[0][0]*m[1][1]-m[0][1]*m[1][0]
    output = 0
    for i in range(len(m)):
        output += ((-1)**i) * m[0][i] * det(minorMatrix(m,0,i))
    return output
```

$$
\begin{aligned}
\det\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}\right) &= a\det\left(\begin{bmatrix} e & f \\ h & i \end{bmatrix}\right) - b\det\left(\begin{bmatrix} d & f \\ g & i \end{bmatrix}\right) + c\det\left(\begin{bmatrix} d & e \\ g & h \end{bmatrix}\right) \\
&= a(ei - fh) - b(di - fg) + c(dh - eg) \\
&= aei + bfg + cdh - afh - bdi - ceg
\end{aligned}
$$

Part 2 Maze

A Maze is a n x m Grid such that

- empty = 0
- blocked = 1
- We can generate a maze with a half and half chance of empty or blocked

```
>>> maze = createRandomMaze(10,30)
>>> mTightPrint(maze)
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011010101000100110000
011000111000111000000100001100
101101100110100001010000011101
111101000110010000001000011000
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

Task 1

- Write a function `createRandomMaze(n,m)` to generate such a maze

```
>>> maze = createRandomMaze(10,30)
>>> mTightPrint(maze)
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011010101000100110000
011000111000111000000100000110
101101100110100001010000011101
111101000110010000000100001100
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

Task 2: Solving a Maze

- A Maze is Solvable if we can go from `(0,0)` to `(n-1,m-1)`.

```
>>> maze = createRandomMaze(10,30)
>>> mTightPrint(maze)
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011010101000100110000
011000111000111000000100001100
101101100110100001010000011101
111101000110010000000100011000
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

Solving a Maze

- A Maze is Solvable if we can go from `(0,0)` to `(n-1,m-1)`.

```
>>> maze = createRandomMaze(10,30)
>>> mTightPrint(maze)
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011010101000100110000
011000111000111000001000001100
101101100110100001010000011101
111101000110010000001000011000
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

Solving a Maze

- A Maze is NOT Solvable if we cannot go from `(0,0)` to `(n-1,m-1)`.

```
>>> maze = createRandomMaze(10,30)
>>> mTightPrint(maze)
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011011101000100110000
011000111000111000000100000100
101101100110101001010000011101
1111010001100110000001000011000
111010100001001111010101011011
011100111000111010000011000001
100101010110001110100000011000
```

Solving a Maze

- A Maze is NOT Solvable if we cannot go from `(0,0)` to `(n-1,m-1)`.

- All the reachable space from `(0,0)`.

```
>>> maze = createRandomMaze(10,30)
>>> mTightPrint(maze)
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011011101000100110000
011000111000111000000100000110
101101100110101001010000011101
111101000110011000000100001100
111010100001001111010101011011
011100111000111101000011000001
100101010110001110100000011000
```

How to Solve a Maze?

- From (0,0), anyhow go?
  - Any Idea?

```
>>> maze = createRandomMaze(10,30)
>>> mTightPrint(maze)
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011010101000100110000
011000111000111000001000001100
101101100110100001010000011101
111101000110010000001000011000
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

Yes! It’s “Anyhow go”!

How to Solve a Maze?

- When you are in a certain position. You anyhow try to go to your neighbor cells
- If you are in position `(i,j)`, what are the positions you can go?
  - `(i-1,j), (i+1,j), (i,j-1), (i,j+1)`

```
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011010101000100110000
011000111000111000001000001100
101101100110100001010000011101
111101000110010000001000011000
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

How to Solve a Maze?

- When you are in a certain position. You anyhow try to go to your neighbor cell
- What are the possibilities that you CANNOT go to your four neighbors?

```
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011010101000100110000
011000111000111000001000001100
101101100110100001010000011101
111101000110010000001000011000
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

How to Solve a Maze?

- When you are in a certain position. You anyhow try to go to your neighbor cell
- What are the possibilities that you CANNOT go to your four neighbors?
  - Blocked
  - Out of Bound

```
010010100110101111001010001101
000000001111101010000101010110
1010101010010010000001100100100?
110010100011010101000100110000
011000111000111000001000001100
101101100110100001010000011101
111101000110010000001000011000
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

How to Solve a Maze?

- When you are in a certain position. You anyhow try to go to your neighbor cell
- What are the possibilities that you CANNOT go to your four neighbors?
  - Blocked
  - Out of Bound
- If the neighbor has a coordinates `(a,b)`, what is the conditions for `(a,b)` that you CANNOT go?
  - `maze[a][b] == 1`
  - `a < 0 or b < 0 or a >= n or b > = m`

Possible Neighbors

- Write a function `possibleNeighbors(m,i,j)` to return a list of neighbors coordinates such that they are possible
  - E.g. i = 2, j = 29
  - Possible neighbors should be `[[1,29],[3,29]]` only

```
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010010
110010100011010101000100110000
011000111000111000001000001100
101101100110100001010000011101
111101000110010000001000011000
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

Possible Neighbors

- Write a function `possibleNeighbors(m,i,j)` to return a list of neighbors coordinates such that they are possible

```
>>> mTightPrint(maze)
00001001100111000001
10111000100001100100 1
00110010101011101111
10010011101010100010
00000011000000000110
10001010001110000110
01100100111101000000
00110001111111000010
>>> possibleNeighbors(maze,7,4)
[[6, 4], [7, 5]]
```

Possible Neighbors

```
def possibleNeighbors(m,i,j):
    h = len(m)
    w = len(m[0])
    allCandidates = [[i-1,j],[i+1,j],[i,j-1],[i,j+1]]
    output = []
    for c in allCandidates:
        if 0 <= c[0] < h:
            if 0 <= c[1] < w:
                if m[c[0]][c[1]] != BLOCKED:
                    output.append(c)
    return output
```

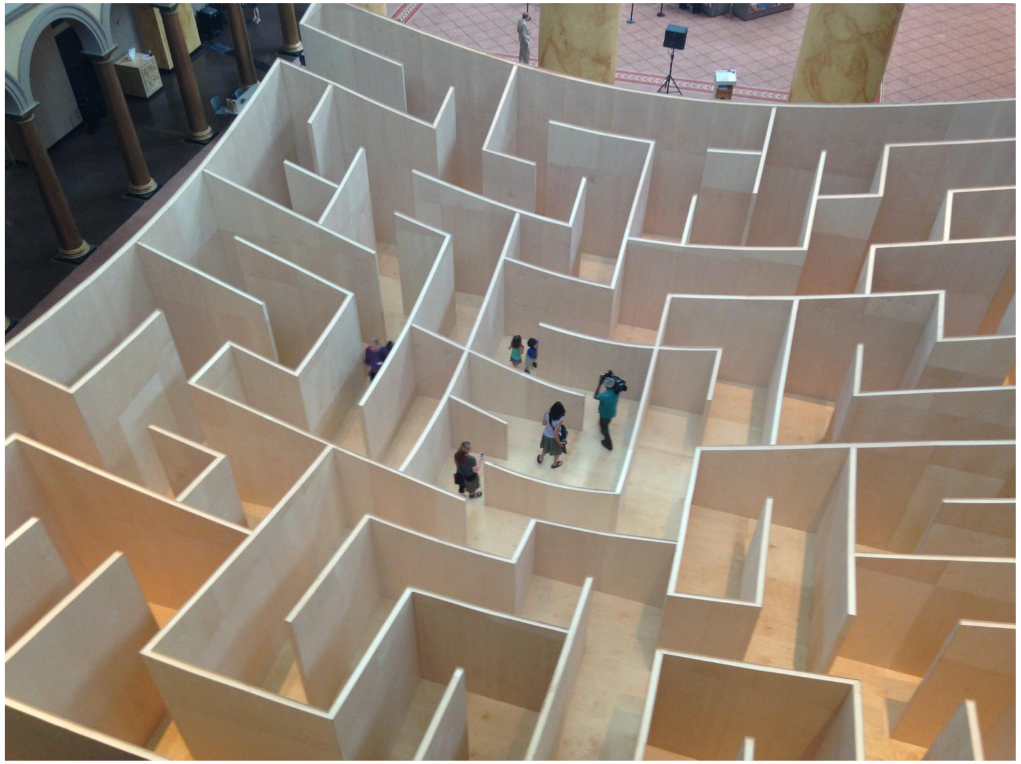
How to Solve a Maze?

- Ok, if can go, then what?
  - Any idea?

```
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010010
110010100011010101000100110000
011000111000111000000100000001100
101101100110100001010000011101
111101000110010000001000011000
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

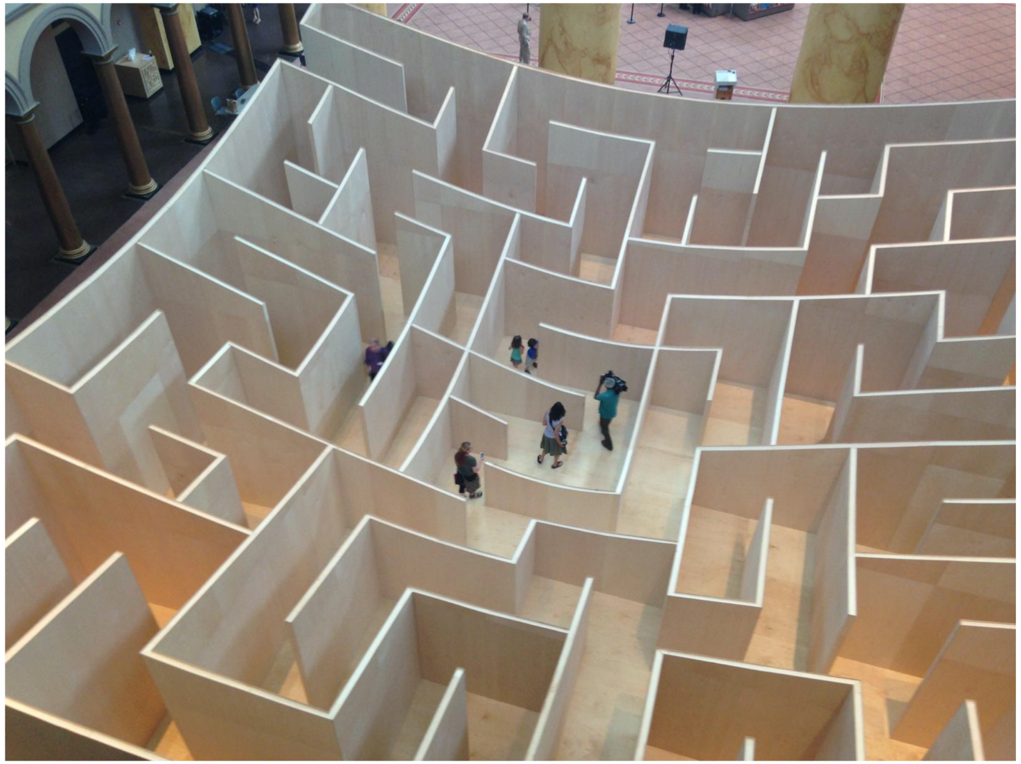
Algorithm 1

- Any how go to any of the possible neighbors
  - With some luck, you can go to the exit
- What if the maze is not solvable?
  - How do you know you cannot reach the exit?
  - Ideas?
    - Limit number of steps?



Algorithm 2

- I collect all the possible neighbors in a collection S and I go to try them
  - When I go to a new place with new neighbors, I keep adding neighbors to S
    - Except that if some of those neighbors are visited before
    - Need to keep track of the visited



Algorithm 2

```
def isSolvable(maze):
    if maze[0][0] == 1:
        return False
    visited = [[0,0]] #record those positions that are visited before
    S = [[0,0]] #all possible neighbors that we want to try
    while S:
        pos = S.pop()
        if pos[0] == (len(maze)-1) and pos[1] == (len(maze[0])-1):
            #Exit reached!
            return True
        pospn = possibleNeighbors(maze,pos[0],pos[1])
        for newpos in pospn:
            if newpos not in visited:
                visited.append(newpos)
                S.append(newpos)
    #After trying every possible move
    #There is no more new neighbor we can try
    return False
```

```python
def isSolvable(maze):
    if maze[0][0] == 1:
        return False
    visited = [[0,0]] #record those positions that are visited before
    S = [[0,0]] #all possible neighbors that we want to try
    while S:
        pos = S.pop()
        print(f'Current pos = {pos}')
        if pos[0] == (len(maze)-1) and pos[1] == (len(maze[0])-1):
            #Exit reached!
            return True
        pospn = possibleNeighbors(maze,pos[0],pos[1])
        print(f'Possible Ngb = {pospn}')
        for newpos in pospn:
            if newpos not in visited:
                visited.append(newpos)
                S.append(newpos)
        print(f'New S = {S}')
    #After trying every possible move
    #There is no more new neighbor we can try
    return False
```

Sample Run

visited

```
0011
1010
0000
```

```
Current pos = [0, 0]
Possible Ngb = [[0, 1]]
New S = [[0, 1]]
```

```
Current pos = [0, 1]
Possible Ngb = [[1, 1], [0, 0]]
New S = [[1, 1]]
```

```
Current pos = [1, 1]
Possible Ngb = [[0, 1], [2, 1]]
New S = [[2, 1]]
```

```
Current pos = [2, 1]
Possible Ngb = [[1, 1], [2, 0], [2, 2]]
New S = [[2, 0], [2, 2]]
```

```
Current pos = [2, 2]
Possible Ngb = [[2, 1], [2, 3]]
New S = [[2, 0], [2, 3]]
```

```
Current pos = [2, 3]
```

Flooding Algorithm

- Idea: Expand the neighborhood
    - Do not repeat the neighbors that are visited

Home Challenge

- Write a simple loop to find a maze that is solvable
- Visualize the path

```
>>> mTightPrint(solve(maze))
S0000001011000001110001O1
EEES11100110EEES110010100
101EES1EES11N01S001011001
00001EEN1S10N01S110101110
100100010S11N10EES1000001
001100011EEEN1111S1011001
0001100100000101S1001001
0000111000101110 1EEES1100
1111110100010000001EEES0
0011100111110111001001E0
```

- Find the shortest path

Today

- A Glimpse of “Algorithm”