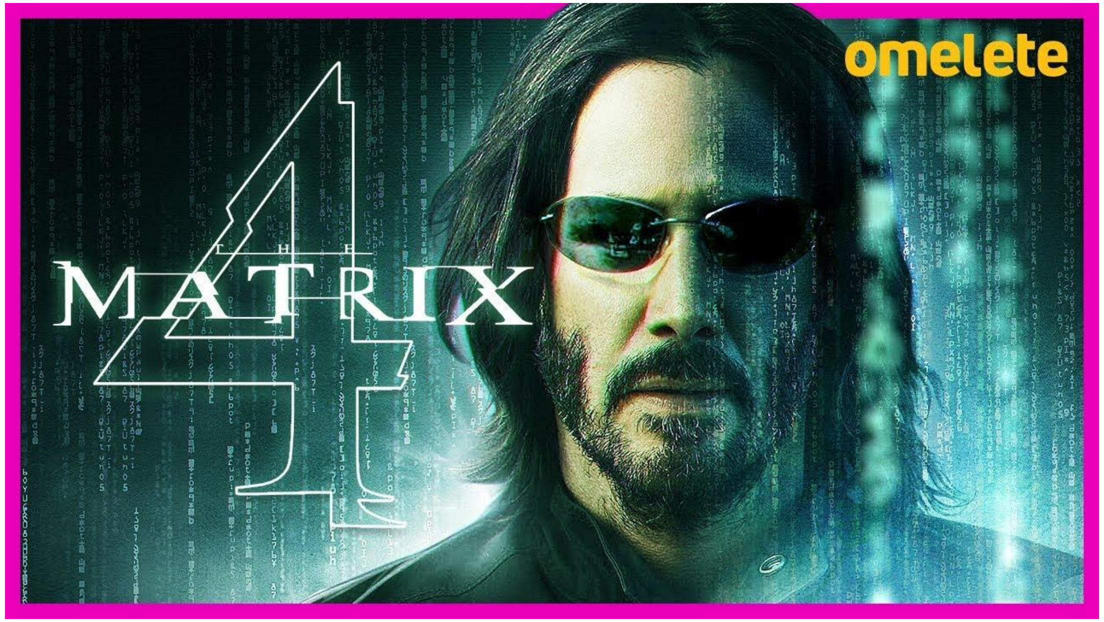
Week 6

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 helper functions provided in the lecture
  - `createZeroMatrix()`, `mTightPrint()`, etc.
- The solutions may be found online but try to code them by yourself
- The package `numpy` (and some other packages) provides these functionalities but we want to learn how to code them ourselves

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)` that transforms an `r` × `c` matrix into 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?

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)` that transforms an `r` × `c` matrix into 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 [optional]: Try a list comprehension one-liner?

Multiplication

- 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} involves row 2 in A and column 1 in B
However, our row and column numbers start with 0 in Python

- Such that

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

Multiplication

- 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?

Multiplication

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

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

Multiplication

- Given two matrices A and B

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

- 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

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

Minor Matrix

- Write a function `minorMatrix(m,i,j)` that returns the matrix 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 minor is the determinant of our definition of the minor matrix

Minor Matrix

- Try?

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

For each row
except row i

Add that row
without column j
to the output

Determinant

- Assume the input is a square matrix
- No matter how big the matrix is:
  - For each element of the first row
  - Find the determinant of its corresponding minor matrix

$$
\begin{aligned}
|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}
&= a \begin{vmatrix} \square & \square & \square \\ \square & e & f \\ \square & h & i \end{vmatrix}
- b \begin{vmatrix} \square & \square & \square \\ d & \square & f \\ g & \square & i \end{vmatrix}
+ c \begin{vmatrix} \square & \square & \square \\ d & e & \square \\ g & h & \square \end{vmatrix} \\
&= a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\
&= aei + bfg + cdh - ceg - bdi - afh.
\end{aligned}
$$

If the minor matrix is not 2 x 2

↓

Recursion!

Determinant

- Assume the input is a square matrix
- No matter how big the matrix is:
  - For each element of the first row
  - Find the determinant of its corresponding minor matrix

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

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

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

If the minor matrix is not 2 x 2

↓

Recursion!

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
```

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

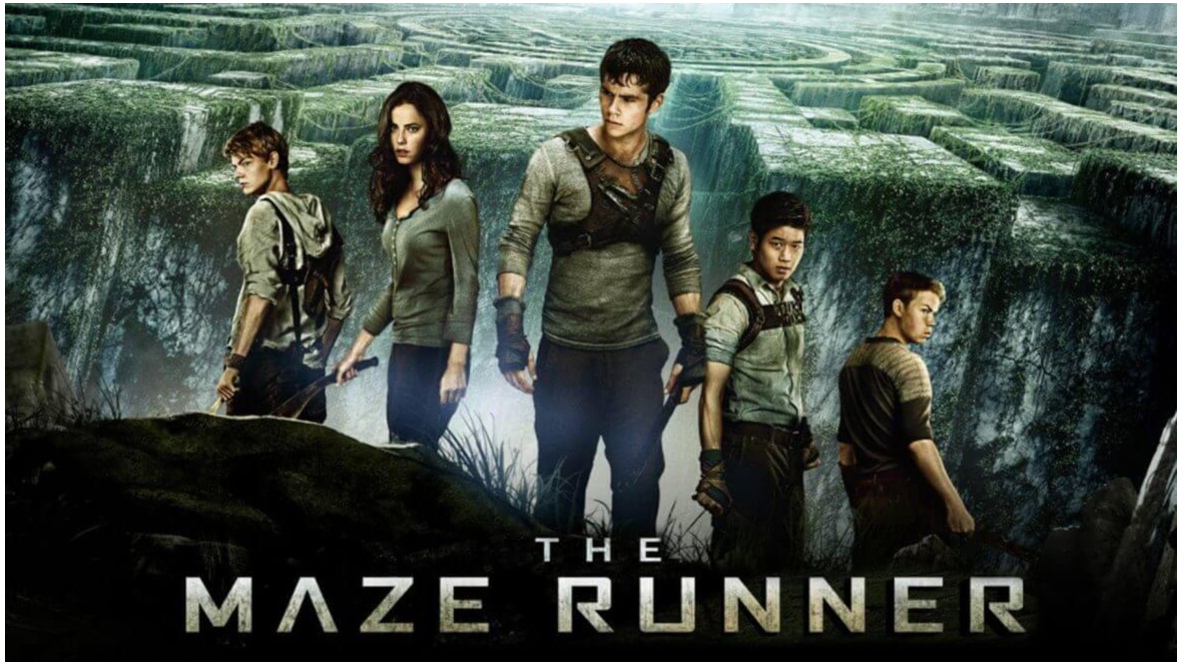
- Try?

Determinant

```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



Maze

- A maze is an r x c grid such that for each space,
  - empty = 0
  - blocked = 1

```
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011010101000100110000
011000111000111000000100001100
101101100110100001010000011101
111101000110010000000100001100
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

Random Maze

- Write a function `createRandomMaze(r,c)` to generate a maze where every space has an equal random chance of being empty (0) or blocked (1)

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

How to Solve a Maze?

- A maze is solvable if it is possible to go from `(0,0)` to `(r-1,c-1)`.
- You can only move horizontally or vertically, not diagonally.

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

How to Solve a Maze?

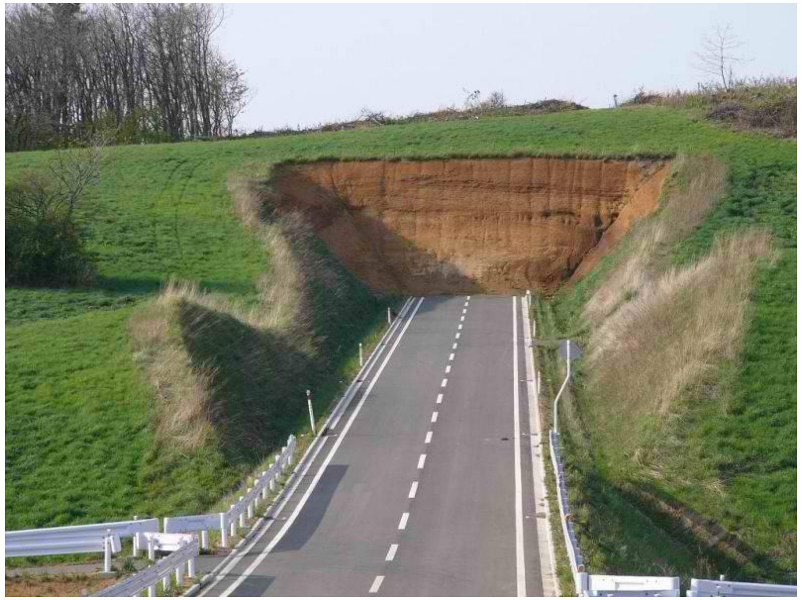
- A maze is solvable if it is possible to go from `(0,0)` to `(r-1,c-1)`.
- You can only move horizontally or vertically, not diagonally.

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

How to Solve a Maze?

- A maze is NOT solvable if it is not possible to go from `(0,0)` to `(r-1,c-1)`.

```
>>> maze = createRandomMaze(10,30)
>>> mTightPrint(maze)
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011011101000100110000
011000111000111000001000001100
101101100110101001010000011101
111101000110011000001000011000
111010100001001111010101011011
011100111000111101000011000001
100101010110001110100000011000
```

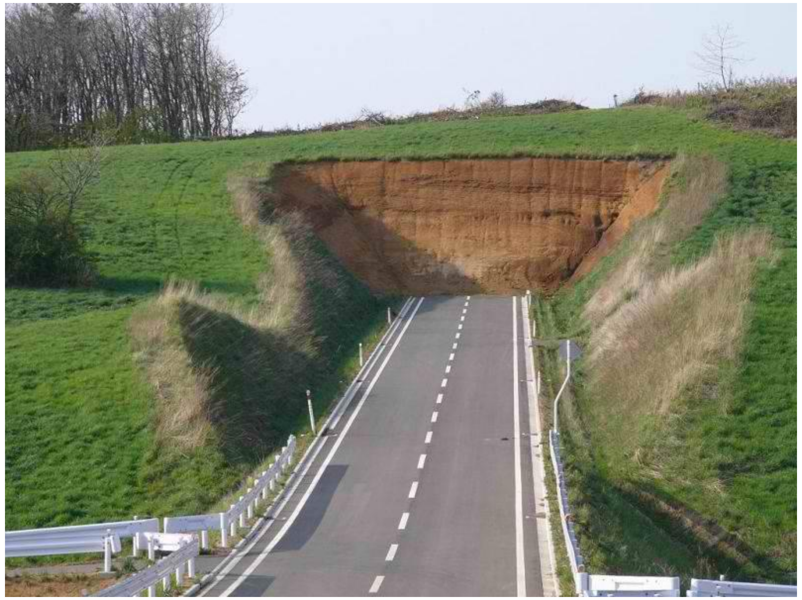


How to Solve a Maze?

- A maze is NOT solvable if it is not possible to go from `(0,0)` to `(r-1,c-1)`.

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

```
>>> maze = createRandomMaze(10,30)
>>> mTightPrint(maze)
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011011101000100110000
011000111000111000001000001100
101101100110101001010000011101
111101000110011000001000011000
111010100001001111010101011011
011100111000111101000011000001
100101010110001110100000011000
```



How to Solve a Maze?

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

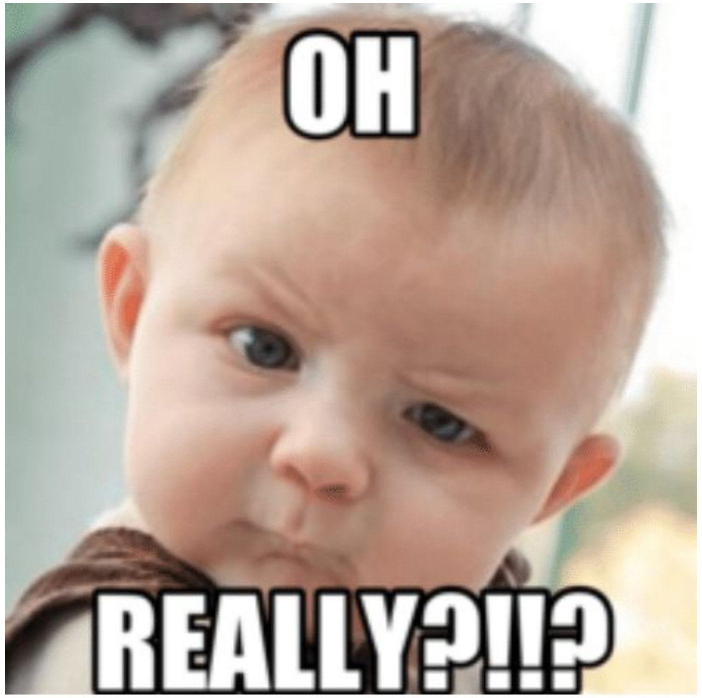
```
>>> maze = createRandomMaze(10,30)
>>> mTightPrint(maze)
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010011
110010100011010101000100110000
011000111000111000001000001100
101101100110100001010000011101
111101000110010000001000011000
111010100001000111010101011011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

Yes! It’s “anyhow go”!

How to Solve a Maze?

- When in a certain position, “anyhow” try to go to your neighbouring cells
- If you are in position `(i,j)`, what are your possible neighbours?
  `(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 in a certain position, “anyhow” try to go to your neighbouring cells
- When will you NOT be able to go to any of your four neighbouring cells?

```
010010100110101111001010001101
000000001111101010000101010110
101010101001001000000110010010
110010100011010101000100110000
011000111000111000000100000110
101101100110100001010000011101
111101000110010000000100001100
111010100000100011101010101101
011100111000110101000001100000
100101010110000011010000000110
```

How to Solve a Maze?

- When in a certain position, “anyhow” try to go to your neighbouring cells
- When will you NOT be able to go to any of your four neighbouring cells?
  - Blocked
  - Out of Bounds

```
010010100110101111001010001101
000000001111101010000101010110
1010101010010010000001100100 10?
110010100011010101000100110000
011000111000111000001000001100
101101100110100001010000011101
111101000110010000001000011000
111010100001000111010101011011
011100111000110101000011000001
100101010110000110100000011000
```

How to Solve a Maze?

- When in a certain position, “anyhow” try to go to your neighbouring cells
- When will you NOT be able to go to any of your four neighbouring cells?
  - Blocked
  - Out of Bounds
- If a neighbouring cell has coordinates `(a,b)`, what are the conditions for `(a,b)` such that you CANNOT go to the cell?
  - `maze[a][b] == 1`
  - `a < 0 or b < 0 or a >= n or b >= m`

Possible Neighbours

- Write a function `possibleNeighbours(m,i,j)` to return a list of the positions (as lists) of the neighbouring cells you can go to
  - E.g. i = 2, j = 29
  - Possible neighbours should be `[[1,29],[3,29]]` only

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

Possible Neighbours

- Write a function `possibleNeighbours(m,i,j)` to return a list of the positions (as lists) of the neighbour cells you can go to

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

Possible Neighbours

```
def possibleNeighbours(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
```

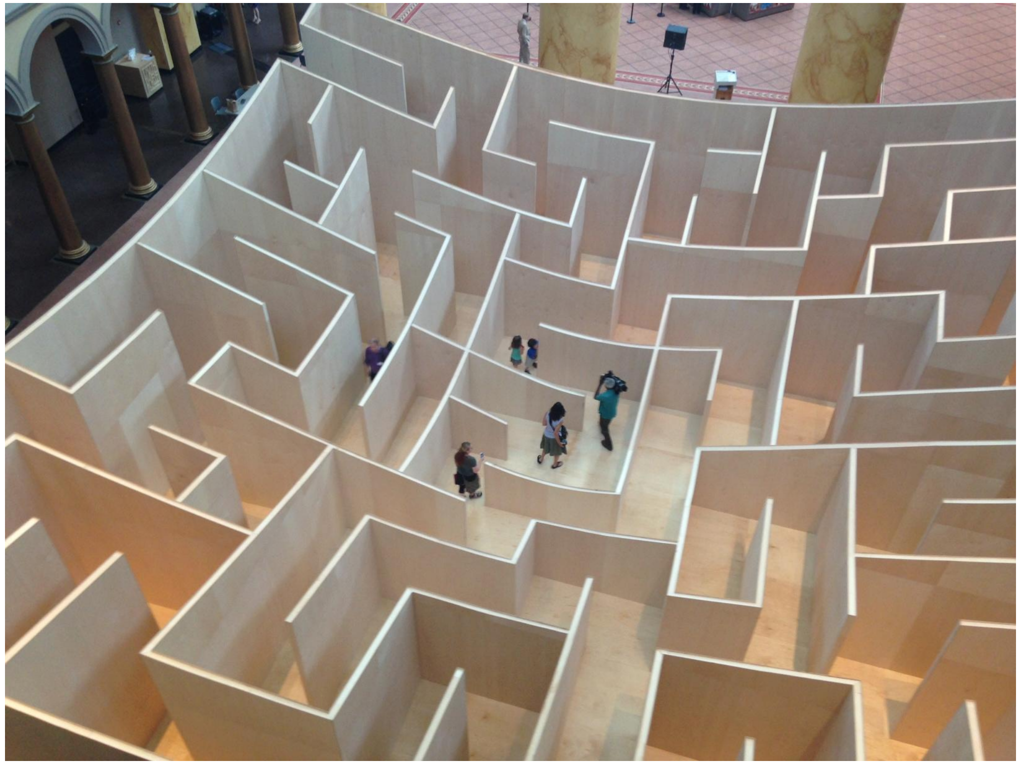
How to Solve a Maze?

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

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

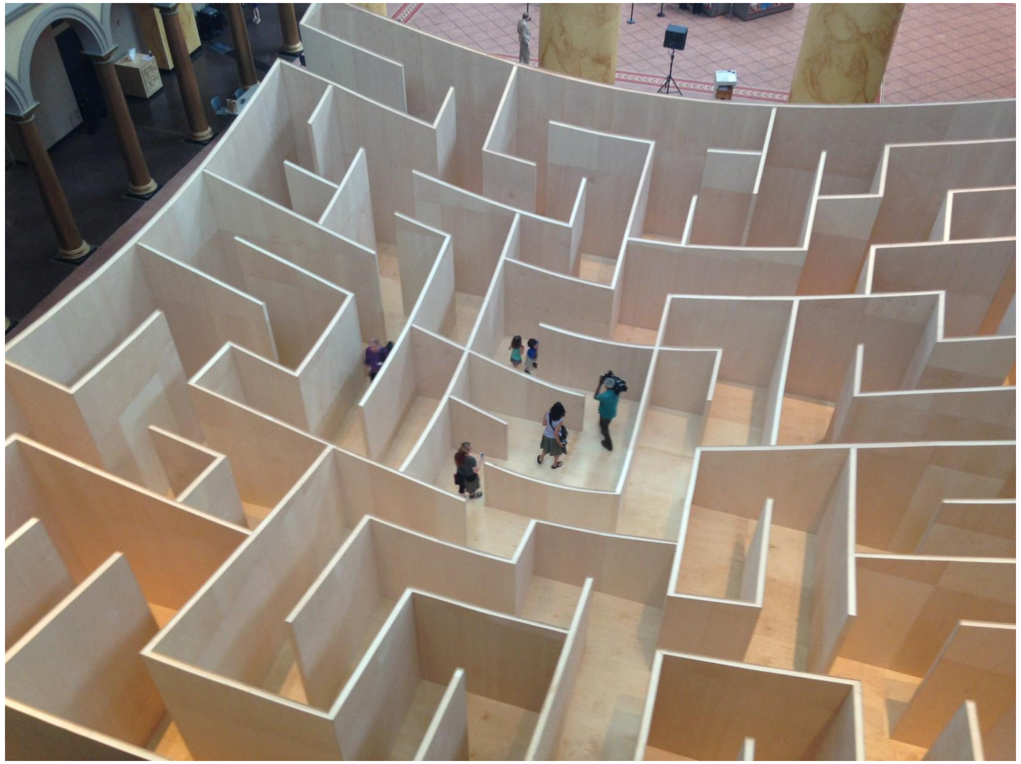
Algorithm 1

- Anyhow go to any possible neighbour
  - With some luck, you can reach the exit
- What if the maze is not solvable?
  - How do you know if you can never ever reach the exit?
  - Ideas?
    - Limit the number of steps?



Algorithm 2

- I collect all possible neighbours in a collection S, and try each of them
  - When I reach a new cell with new neighbours, I add the neighbours to S
    - Except the neighbours that I 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 neighbours that we want to try
    while S:
        pos = S.pop()
        if pos[0] == (len(maze) - 1) and pos[1] == (len(maze[0])-1):
            return True #exit reached!
        pospn = possibleNeighbours(maze,pos[0],pos[1])
        for newpos in pospn:
            if newpos not in visited:
                visited.append(newpos)
                S.append(newpos)
    return False #after trying every possible move there is no more new neighbor to try
```

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 neighbours 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):
            return True #exit reached!
        pospn = possibleNeighbours(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}')
    return False #after trying every possible move there is no more new neighbor to try
```

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 neighbourhood
  - Do not repeat the neighbours that are visited

Extra Tasks

- Write a simple loop to find a maze that is solvable
- *Beautify* your maze presentation
- Visualize the path, e.g.

```
>>> mTightPrint(solve(maze))
S000000101100000111000101
EEES11100110EEES110010100
101EES1EES11N01S001011001
00001EEN1S10N01S110101110
100100010S11N10EES1000001
001100011EEEN1111S1011001
0001100100000101S1001001
0000111000101110 1EEES1100
1111110100010000001EEES0
001111001111101110010 01E0
```

Shortest Path?

Today

- A Glimpse of “Algorithm”