

In [ ]:

1

In [ ]:

```

1 Matrix
2 int[][]
3
4 vector<vector<int> >

```

In [2]:

```

1 matrix = [
2     [1 , 1, 1, 1],
3     [0 , 1, 0, 1],
4     [0 , 1, 1, 1],
5     [0 , 1, 1, 1],
6 ]
7
8 print(matrix)
9
10
11 rows = len(matrix)
12 cols = len(matrix[0])
13 for i in range(rows): # for (i=0;i<len(matrix);i++)
14     for j in range(cols):
15         print(matrix[i][j])
16

```

```
[[1, 1, 1, 1], [0, 1, 0, 1], [0, 1, 1, 1], [0, 1, 1, 1]]
```

```

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

```

In [ ]:

```

1 4 Neighbour
2 8 Neighbour

```

In [ ]:

1

## V1: Rat in a Maze D=(R,D), Return a Boolean

Given a maze of size  $N \times N$  represented in the form of 0s and 1s.  
Where 1 denotes a cell that can be visited and 0 denotes a blocker/s  
tone/obstacle

Given a starting point  $(0,0)$  and an ending point  $(N-1, N-1)$ , return  
whether there exists a path or not.

At a time the Rat can move only one step in Right or Down directions  
only.

```

    0  1  2  3
[
0    1  1  1  1
1    0  1  0  1
2    0  1  1  1
3    0  1  1  1
]
```

```

In [10]: 1 def is_solvable(grid):
2         return is_solvable_util(grid, 0, 0)
3
4 def is_solvable_util(grid, r, c):
5     # boundary check
6     print((r,c), end=',')
7     if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
8         return False
9
10    if grid[r][c] == 0:
11        return False
12
13    if (r == len(grid) - 1 and c == len(grid[0]) - 1):
14        return True
15
16    # # right
17    # if is_solvable_util(grid, r, c+1):
18    #     return True
19
20    # # down
21    # if is_solvable_util(grid, r+1, c):
22    #     return True
23    # return False
24    return is_solvable_util(grid, r, c+1) or is_solvable_util(grid, r+1, c)
25 matrix = [
26     # 0  1  2  3
27     [1 , 1,  1,  1],
28     [0 , 1,  0,  0],
29     [0 , 1,  1,  1],
30     [0 , 1,  1,  1],
31 ]
32 print(is_solvable(matrix))
33
34
35

```

```

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

```

In [14]:

```

1
2 matrix = [
3     [1 , 1, 1, 1],
4     [0 , 1, 0, 0],
5     [0 , 1, 0, 1],
6     [0 , 1, 0, 1],
7     ]
8 print(is_solvable(matrix))
9
10
11 print()
12 matrix = [
13     [1 , 1, 1, 1, 1],
14     [0 , 1, 0, 0, 1],
15     [0 , 1, 0, 1, 1],
16     [0 , 1, 0, 1, 0],
17     [0 , 1, 0, 1, 1],
18 ]
19 print(is_solvable(matrix))
20
21
22 print()
23 matrix = [
24     [1 , 1, 1, 1],
25     [1 , 1, 1, 1],
26     [1 , 1, 1, 1],
27     [1 , 1, 1, 0],
28 ]
29 print(is_solvable(matrix))

```

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

(0, 0),(0, 1),(0, 2),(0, 3),(0, 4),(0, 5),(1, 4),(1, 5),(2, 4),(2, 5),(3, 4),  
(1, 3),(1, 2),(1, 1),(1, 2),(2, 1),(2, 2),(3, 1),(3, 2),(4, 1),(4, 2),(5, 1),  
(1, 0),False

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

In [ ]:

1

In [18]:

```

1  def is_solvable(grid):
2      visited = set()
3      res = is_solvable_util(grid, 0, 0, visited)
4      print("visited set=", visited)
5      return res
6
7  def is_solvable_util(grid, r, c, visited):
8      # boundary check
9      print((r,c), end=',')
10     if (r < 0 or r >= len(grid) or c < 0 or c >= len(grid[0])):
11         return False
12
13     if grid[r][c] == 0:
14         return False
15
16     # check if the current position is already seen in the past
17     if (r,c) in visited ):
18         return False
19
20     if (r == len(grid) - 1 and c == len(grid[0]) - 1):
21         return True
22
23     visited.add( (r,c) )
24     if is_solvable_util(grid, r, c+1, visited) or is_solvable_util(grid,
25         return True
26
27     return False
28
29 matrix = [
30     # 0 1 2 3
31     [1 , 1, 1, 1],
32     [0 , 1, 0, 0],
33     [0 , 1, 1, 1],
34     [0 , 1, 1, 1],
35 ]
36 print(is_solvable(matrix))
37
38 print()
39 matrix = [
40     [1 , 1, 1, 1],
41     [1 , 1, 1, 1],
42     [1 , 1, 1, 1],
43     [1 , 1, 1, 0],
44 ]
45 print(is_solvable(matrix))
46
47 # n = row/col
48 # TC: O(n^2)
49 # SC: O(n^2) + O(2n) => O(n^2)
50 #      set      stack

```

```
(0, 0),(0, 1),(0, 2),(0, 3),(0, 4),(1, 3),(1, 2),(1, 1),(1, 2),(2, 1),(2, 2),
(2, 3),(2, 4),(3, 3),visited set= {(0, 1), (2, 1), (0, 0), (1, 1), (0, 3),
(2, 3), (0, 2), (2, 2)}
```

True

```
(0, 0),(0, 1),(0, 2),(0, 3),(0, 4),(1, 3),(1, 4),(2, 3),(2, 4),(3, 3),(1, 2),
(1, 3),(2, 2),(2, 3),(3, 2),(3, 3),(4, 2),(1, 1),(1, 2),(2, 1),(2, 2),(3, 1),
(3, 2),(4, 1),(1, 0),(1, 1),(2, 0),(2, 1),(3, 0),(3, 1),(4, 0),visited set=
{(0, 1), (1, 2), (2, 1), (0, 0), (3, 1), (1, 1), (0, 3), (2, 0), (3, 0), (2,
3), (0, 2), (2, 2), (1, 0), (3, 2), (1, 3)}
```

False

```
In [19]: 1 l = [(0, 0),(0, 1),(0, 2),(0, 3),(0, 4),(1, 3),(1, 4),(2, 3),(2, 4),(3, 3)
2         ]
3         print(len(l))
```

31

In [ ]:

1

## V2: Rat in a Maze D=(R,D), Return a list of coordinates which denotes the path in maze

Given a maze of size N\*N represented in the form of 0s and 1s.

Where 1 denotes a cell that can be visisted and 0 denotes a blocker/s tone/obstacle

Given a starting point (0,0) and and ending point (N-1, N-1), Return a list of coordinates which denotes the path in maze if there exists a path.

At a time the Rat can move only one step in Right or Down directions only.

```

    0  1  2  3
[
0    1  1  1  1
1    0  1  0  1
2    0  1  1  1
3    0  1  1  1
]
```

In [23]:

```

1  def is_solvable(grid):
2      visited = set()
3      result = []# array ##### NEW
4      res = is_solvable_util(grid, 0, 0, visited, result)
5      return result
6
7  def is_solvable_util(grid, r, c, visited, result):
8      # boundary check
9      if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
10         return False
11
12         if grid[r][c] == 0:
13             return False
14
15         # check if the current position is already seen in the past
16         if (r,c) in visited ):
17             return False
18
19         result.append( (r,c) ) # add to the end ### NEW
20         if (r == len(grid) - 1 and c == len(grid[0]) - 1):
21             return True
22
23         visited.add( (r,c) )
24         if is_solvable_util(grid, r, c+1, visited, result) or is_solvable_util(
25             return True
26
27         result.pop() # remove from behind ### NEW
28
29         return False
30
31 matrix = [
32 #   0   1   2   3
33     [1 , 1,  1,  1],
34     [0 , 1,  0,  0],
35     [0 , 1,  1,  1],
36     [0 , 1,  1,  1],
37 ]
38 print(is_solvable(matrix))
39
40 print()
41 matrix = [
42     [1 , 1,  1,  1],
43     [1 , 1,  1,  1],
44     [1 , 1,  1,  1],
45     [1 , 1,  1,  0],
46 ]
47 print(is_solvable(matrix))

```

```
[(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (3, 3)]
```

```
[]
```

```

In [33]: 1 def is_solvable(grid):
2         result = []# array ##### NEW
3         is_solvable_util(grid, 0, 0, result)
4
5     def is_solvable_util(grid, r, c, result):
6         # boundary check
7         if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
8             return
9
10        if grid[r][c] == 0:
11            return
12
13        result.append( (r,c) ) # add to the end ### NEW
14
15        if (r == len(grid) - 1 and c == len(grid[0]) - 1):
16            print(result)
17            result.pop()
18            return
19
20        is_solvable_util(grid, r, c+1, result)
21        is_solvable_util(grid, r+1, c, result)
22
23        result.pop() # remove from behind ### NEW
24
25    matrix = [
26    # 0  1  2  3
27    [1 , 1,  1,  1],
28    [0 , 1,  0,  0],
29    [0 , 1,  1,  1],
30    [0 , 1,  1,  1],
31    ]
32    print(is_solvable(matrix))
33
34    print()
35    matrix = [
36    [1 , 1,  1,  1],
37    [1 , 1,  1,  1],
38    [1 , 1,  1,  1],
39    [1 , 1,  1,  0],
40    ]
41    print(is_solvable(matrix))

```

```

[(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (3, 3)]
[(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (3, 2), (3, 3)]
[(0, 0), (0, 1), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3)]
None

```

```
None
```

```
In [ ]: 1
```

### V3: Rat in a Maze D=(L,R,D,U), Return a list of



## coordinates which denotes the path in maze

Given a maze of size  $N \times N$  represented in the form of 0s and 1s.  
Where 1 denotes a cell that can be visited and 0 denotes a blocker/s  
tone/obstacle

Given a starting point (0,0) and ending point (N-1, N-1), return  
whether there exists a path or not.

At a time the Rat can move only one step in Right or Down directions  
only.

	0	1	2	3
[				
0	1	1	1	1
1	0	1	0	1
2	0	1	1	1
3	0	1	1	1

```

In [34]: 1  ## Try to fit existing code!!! Doesn't work
2
3  def is_solvable(grid):
4      result = []# array ##### NEW
5      is_solvable_util(grid, 0, 0, result)
6
7  def is_solvable_util(grid, r, c, result):
8      # boundary check
9      if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
10         return
11
12     if grid[r][c] == 0:
13         return
14
15     result.append( (r,c) ) # add to the end ### NEW
16
17     if (r == len(grid) - 1 and c == len(grid[0]) - 1):
18         print(result)
19         result.pop()
20         return
21
22     is_solvable_util(grid, r, c+1, result) # right
23     is_solvable_util(grid, r+1, c, result) # down
24     is_solvable_util(grid, r, c-1, result) # left
25     is_solvable_util(grid, r-1, c, result) # up
26
27     result.pop() # remove from behind ### NEW
28
29 matrix = [
30     # 0  1  2  3
31     [1 , 1, 1, 1],
32     [0 , 1, 0, 0],
33     [0 , 1, 1, 1],
34     [0 , 1, 1, 1],
35 ]
36 print(is_solvable(matrix))
37
38 print()
39 matrix = [
40     [1 , 1, 1, 1],
41     [1 , 1, 1, 1],
42     [1 , 1, 1, 1],
43     [1 , 1, 1, 0],
44 ]
45 print(is_solvable(matrix))

```

```

-----
RecursionError                                Traceback (most recent call last)
C:\Users\LEANGA~1\AppData\Local\Temp\ipykernel_22456\988656515.py in <module>
    34     [0, 1, 1, 1],
    35 ]
----> 36 print(is_solvable(matrix))
    37
    38 print()

C:\Users\LEANGA~1\AppData\Local\Temp\ipykernel_22456\988656515.py in is_solva
ble(grid)
    3 def is_solvable(grid):
    4     result = []# array ##### NEW
----> 5     is_solvable_util(grid, 0, 0, result)
    6
    7 def is_solvable_util(grid, r, c, result):

C:\Users\LEANGA~1\AppData\Local\Temp\ipykernel_22456\988656515.py in is_solva
ble_util(grid, r, c, result)
    20         return
    21
----> 22     is_solvable_util(grid, r, c+1, result) # right
    23     is_solvable_util(grid, r+1, c, result) # down
    24     is_solvable_util(grid, r, c-1, result) # left

C:\Users\LEANGA~1\AppData\Local\Temp\ipykernel_22456\988656515.py in is_solva
ble_util(grid, r, c, result)
    20         return
    21
----> 22     is_solvable_util(grid, r, c+1, result) # right
    23     is_solvable_util(grid, r+1, c, result) # down
    24     is_solvable_util(grid, r, c-1, result) # left

... last 2 frames repeated, from the frame below ...

C:\Users\LEANGA~1\AppData\Local\Temp\ipykernel_22456\988656515.py in is_solva
ble_util(grid, r, c, result)
    20         return
    21
----> 22     is_solvable_util(grid, r, c+1, result) # right
    23     is_solvable_util(grid, r+1, c, result) # down
    24     is_solvable_util(grid, r, c-1, result) # left

RecursionError: maximum recursion depth exceeded in comparison

```

```

In [36]: 1 def is_solvable(grid):
2         result = []# array ##### NEW
3         is_solvable_util(grid, 0, 0, result)
4
5 def is_solvable_util(grid, r, c, result):
6     # boundary check
7     if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
8         return
9
10    if grid[r][c] == 0:
11        return
12
13    if (r,c) in result: # O(n) using result as current visited path to avoid
14        return
15
16    result.append( (r,c) ) # add to the end ### NEW
17
18    if (r == len(grid) - 1 and c == len(grid[0]) - 1):
19        print(result)
20        result.pop()
21        return
22
23    is_solvable_util(grid, r, c+1, result) # right
24    is_solvable_util(grid, r+1, c, result) # down
25    is_solvable_util(grid, r, c-1, result) # left
26    is_solvable_util(grid, r-1, c, result) # up
27
28    result.pop() # remove from behind ### NEW
29 # 0,0 0,1 1,1 2,1
30 matrix = [
31 #     0   1   2   3
32     [1 , 1,  1,  1],
33     [0 , 1,  0,  0],
34     [0 , 1,  1,  1],
35     [0 , 1,  1,  1],
36 ]
37 print(is_solvable(matrix))
38
39 print()
40 matrix = [
41     [1 , 1,  1,  1],
42     [1 , 1,  1,  1],
43     [1 , 1,  1,  1],
44     [1 , 1,  1,  0],
45 ]
46 print(is_solvable(matrix))
47
48 # TC:

```

```

[(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (3, 1), (3, 2), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (2, 0), (3, 0), (3, 1),
(3, 2), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (1, 1), (1, 0), (2, 0),
(3, 0), (3, 1), (3, 2), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (2, 2), (2, 3), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (2, 2), (3, 2), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (3, 1), (3, 2), (2, 2),
(2, 3), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (2, 0), (3, 0), (3, 1),
(3, 2), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (2, 0), (3, 0), (3, 1),
(3, 2), (2, 2), (2, 3), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (1, 0), (2, 0), (2, 1), (2, 2),
(2, 3), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (1, 0), (2, 0), (2, 1), (2, 2),
(3, 2), (3, 3)]
[(0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (1, 0), (2, 0), (2, 1), (3, 1),
(3, 2), (3, 3)]

```

In [ ]:

1

```

In [45]: 1 count = 0
2 def is_solvable(grid):
3     result = []# array ##### NEW
4     is_solvable_util(grid, 0, 0, result)
5
6 def is_solvable_util(grid, r, c, result):
7     # boundary check
8     global count
9     if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
10         count += 1
11         return
12
13     if grid[r][c] == 0:
14         count += 1
15         return
16
17     if (r,c) in result: # O(n) using result as current visited path to avoid
18         count += 1
19         return
20
21     result.append( (r,c) ) # add to the end ### NEW
22
23     if (r == len(grid) - 1 and c == len(grid[0]) - 1):
24         print(result)
25         result.pop()
26         count += 1
27         return
28
29     is_solvable_util(grid, r, c+1, result) # right
30     is_solvable_util(grid, r+1, c, result) # down
31     is_solvable_util(grid, r, c-1, result) # left
32     is_solvable_util(grid, r-1, c, result) # up
33
34     result.pop() # remove from behind ### NEW
35
36 matrix = [
37     [1, 1, 1, 1],
38     [1, 1, 1, 1],
39     [1, 1, 1, 1],
40     [1, 1, 1, 1],
41 ]
42 print(is_solvable(matrix))
43 print(count)
44 # TC:
45
46
47 # 16 ....8

```

5/20/23, 11:45 AM

Class\_27\_Backtracking\_20\_May - Jupyter Notebook

```
[ (0, 0), (0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (2, 2), (3, 2), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (2, 2), (2, 1), (3, 1),
(3, 2), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (2, 2), (2, 1), (2, 0),
(3, 0), (3, 1), (3, 2), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (2, 2), (2, 1), (1, 1),
(1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (3, 1), (3, 2), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (2, 0), (3, 0), (3, 1),
(3, 2), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (1, 1), (1, 0), (2, 0),
(3, 0), (3, 1), (3, 2), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (2, 2), (2, 3), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (2, 2), (3, 2), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (3, 1), (3, 2), (2, 2),
(2, 3), (3, 3) ]
[ (0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (2, 1), (3, 1), (3, 2), (2, 2),
(2, 3), (3, 3) ]
```

In [46]:

1

Out[46]:

57.113921245174545

# Permutations

In [ ]:

1

In [ ]:

1

# HW

- Combinations
- <https://leetcode.com/problems/sudoku-solver/> (<https://leetcode.com/problems/sudoku-solver/>)

In [ ]:

1