In [ ]:
```

```

In [ ]:
```
Matrix
int[][]

vector<vector<int> >
```

In [2]:
```
matrix =  [
   [1 , 1,  1,  1],
   [0 , 1,  0,  1],
   [0 , 1,  1,  1],
   [0 , 1,  1,  1],
   ]

print(matrix)


rows = len(matrix)
cols = len(matrix[0])
for i in range(rows): # for (i=0;i<len(matrix);i++)
    for j in range(cols):
        print(matrix[i][j])

```

```
[[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 [ ]:
```
4 Neighbour
8 Neighbour
```

In [ ]:
```

```

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

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/stone/obstacle

Given a starting point (0,0) and 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 [10]:

```
def is_solvable(grid):
    return is_solvable_util(grid, 0, 0)

def is_solvable_util(grid, r, c):
    # boundary check
    print((r,c), end=',')
    if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
        return False

    if grid[r][c] == 0:
        return False

    if (r == len(grid) - 1 and c == len(grid[0]) - 1):
        return True

#     # right
#     if is_solvable_util(grid, r, c+1):
#         return True

#     # down
#     if is_solvable_util(grid, r+1, c):
#         return True
#     return False
    return is_solvable_util(grid, r, c+1) or is_solvable_util(grid, r+1, 
matrix =  [
#   0   1   2   3
   [1 , 1,  1,  1],
   [0 , 1,  0,  0],
   [0 , 1,  1,  1],
   [0 , 1,  1,  1],
]
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),
(2, 3),(2, 4),(3, 3),True
```

In [14]:
```
matrix =  [
   [1 , 1,  1,  1],
   [0 , 1,  0,  0],
   [0 , 1,  0,  1],
   [0 , 1,  0,  1],
   ]
print(is_solvable(matrix))


print()
matrix =  [
   [1 , 1,  1,  1, 1],
   [0 , 1,  0,  0, 1],
   [0 , 1,  0,  1, 1],
   [0 , 1,  0,  1, 0],
   [0 , 1,  0,  1, 1],
]
print(is_solvable(matrix))


print()
matrix =  [
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  0],
]
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 [ ]:
```

```

In [18]:

```python
def is_solvable(grid):
    visited = set()
    res = is_solvable_util(grid, 0, 0, visited)
    print("visited set=", visited)
    return res

def is_solvable_util(grid, r, c, visited):
    # boundary check
    print((r,c), end=',')
    if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
        return False

    if grid[r][c] == 0:
        return False

    # check if the current position is already seen in the past
    if ( (r,c) in visited ):
        return False

    if (r == len(grid) - 1 and c == len(grid[0]) - 1):
        return True

    visited.add( (r,c) )
    if is_solvable_util(grid, r, c+1, visited) or is_solvable_util(grid, 
        return True

    return False

matrix =  [
#   0   1   2   3
   [1 , 1,  1,  1],
   [0 , 1,  0,  0],
   [0 , 1,  1,  1],
   [0 , 1,  1,  1],
]
print(is_solvable(matrix))

print()
matrix =  [
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  0],
]
print(is_solvable(matrix))

# n = row/col
# TC: O(n^2)
# SC: O(n^2) + O(2n) => O(n^2)
#      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]:
```
l = [(0, 0),(0, 1),(0, 2),(0, 3),(0, 4),(1, 3),(1, 4),(2, 3),(2, 4),(3, 3
]
print(len(l))
```

```
31
```

In [ ]:
```

```

# 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/stone/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]:

```
def is_solvable(grid):
    visited = set()
    result = []# array  ##### NEW
    res = is_solvable_util(grid, 0, 0, visited, result)
    return result

def is_solvable_util(grid, r, c, visited, result):
    # boundary check
    if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
        return False

    if grid[r][c] == 0:
        return False

    # check if the current position is already seen in the past
    if ( (r,c) in visited ):
        return False

    result.append( (r,c) ) # add to the end ### NEW
    if (r == len(grid) - 1 and c == len(grid[0]) - 1):
        return True

    visited.add( (r,c) )
    if is_solvable_util(grid, r, c+1, visited, result) or is_solvable_uti
        return True

    result.pop() # remove from behind ### NEW

    return False

matrix =  [
#   0   1   2   3
   [1 , 1,  1,  1],
   [0 , 1,  0,  0],
   [0 , 1,  1,  1],
   [0 , 1,  1,  1],
]
print(is_solvable(matrix))

print()
matrix =  [
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  0],
]
print(is_solvable(matrix))
```

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

[]
```

In [33]:
```
def is_solvable(grid):
    result = []# array  ##### NEW
    is_solvable_util(grid, 0, 0, result)

def is_solvable_util(grid, r, c, result):
    # boundary check
    if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
        return

    if grid[r][c] == 0:
        return

    result.append( (r,c) ) # add to the end ### NEW

    if (r == len(grid) - 1 and c == len(grid[0]) - 1):
        print(result)
        result.pop()
        return

    is_solvable_util(grid, r, c+1, result)
    is_solvable_util(grid, r+1, c, result)

    result.pop() # remove from behind ### NEW

matrix =  [
#   0   1   2   3
   [1 , 1,  1,  1],
   [0 , 1,  0,  0],
   [0 , 1,  1,  1],
   [0 , 1,  1,  1],
]
print(is_solvable(matrix))

print()
matrix =  [
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  0],
]
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*N represented in the form of 0s and 1s.
Where 1 denotes a cell that can be visisted and 0 denotes a blocker/stone/obstacle

Given a starting point (0,0) and 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]:

```
## Try to fit existing code!!! Doesn't work

def is_solvable(grid):
    result = []# array  ##### NEW
    is_solvable_util(grid, 0, 0, result)

def is_solvable_util(grid, r, c, result):
    # boundary check
    if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
        return

    if grid[r][c] == 0:
        return

    result.append( (r,c) ) # add to the end ### NEW

    if (r == len(grid) - 1 and c == len(grid[0]) - 1):
        print(result)
        result.pop()
        return

    is_solvable_util(grid, r, c+1, result) # right
    is_solvable_util(grid, r+1, c, result) # down
    is_solvable_util(grid, r, c-1, result) # left
    is_solvable_util(grid, r-1, c, result) # up

    result.pop() # remove from behind ### NEW

matrix =  [
#   0   1   2   3
   [1 , 1,  1,  1],
   [0 , 1,  0,  0],
   [0 , 1,  1,  1],
   [0 , 1,  1,  1],
]
print(is_solvable(matrix))

print()
matrix =  [
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  0],
]
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]:

```
def is_solvable(grid):
    result = []# array  ##### NEW
    is_solvable_util(grid, 0, 0, result)

def is_solvable_util(grid, r, c, result):
    # boundary check
    if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
        return

    if grid[r][c] == 0:
        return

    if (r,c) in result: # O(n) using result as current visited path to ava
        return

    result.append( (r,c) ) # add to the end ### NEW

    if (r == len(grid) - 1 and c == len(grid[0]) - 1):
        print(result)
        result.pop()
        return

    is_solvable_util(grid, r, c+1, result) # right
    is_solvable_util(grid, r+1, c, result) # down
    is_solvable_util(grid, r, c-1, result) # left
    is_solvable_util(grid, r-1, c, result) # up

    result.pop() # remove from behind ### NEW
# 0,0 0,1 1,1 2,1
matrix =  [
#   0   1   2   3
   [1 , 1,  1,  1],
   [0 , 1,  0,  0],
   [0 , 1,  1,  1],
   [0 , 1,  1,  1],
]
print(is_solvable(matrix))

print()
matrix =  [
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  0],
]
print(is_solvable(matrix))

# 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]:

```
count = 0
def is_solvable(grid):
    result = []# array  ##### NEW
    is_solvable_util(grid, 0, 0, result)

def is_solvable_util(grid, r, c, result):
    # boundary check
    global count
    if (r < 0 or r >=len(grid) or c < 0 or c >= len(grid[0])):
        count += 1
        return

    if grid[r][c] == 0:
        count += 1
        return

    if (r,c) in result: # O(n) using result as current visited path to ava
        count += 1
        return

    result.append( (r,c) ) # add to the end ### NEW

    if (r == len(grid) - 1 and c == len(grid[0]) - 1):
        print(result)
        result.pop()
        count += 1
        return

    is_solvable_util(grid, r, c+1, result) # right
    is_solvable_util(grid, r+1, c, result) # down
    is_solvable_util(grid, r, c-1, result) # left
    is_solvable_util(grid, r-1, c, result) # up

    result.pop() # remove from behind ### NEW

matrix =  [
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
   [1 , 1,  1,  1],
]
print(is_solvable(matrix))
print(count)
# TC:


# 16 ....8
```

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

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
```