Leetcode 79.
Word Search

#Array #String #Backtracking #Matix

Problem Description - Word Search

Given an **m x n grid** of characters board and a string word, return **true** if word **exists** in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring.

The same letter cell may not be used more than once.

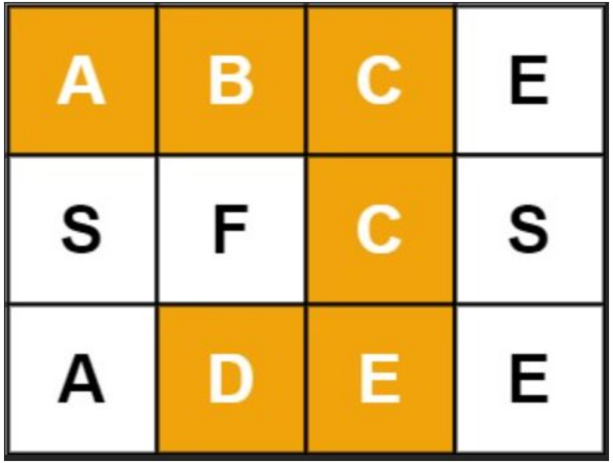
Example 1:

Input:

board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]],

word = "ABCCED"

Output: true



A	B	C	E
S	F	C	S
A	D	E	E

* Problem Description - Word Search

Given an **m x n grid** of characters board and a string word, return **true** if word **exists** in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring.

The same letter cell may not be used more than once.

Example 2:

Input:

board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"

Output: true

A	B	C	E
S	F	C	S
A	D	E	E

Example 3:

Input:

board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"

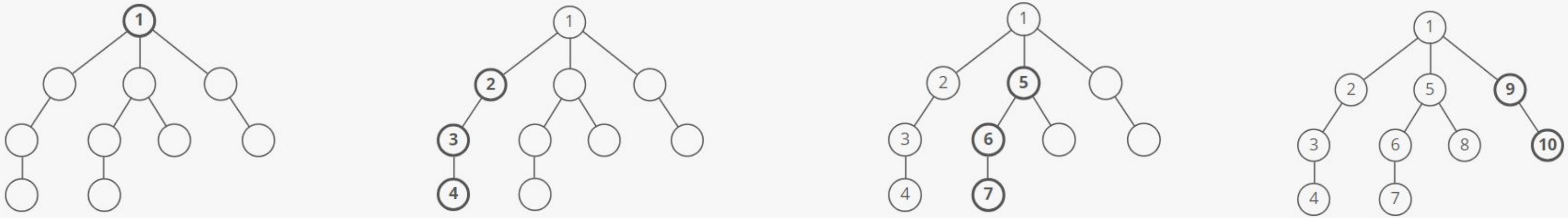
Output: false

A	B	C	E
S	F	C	S
A	D	E	E

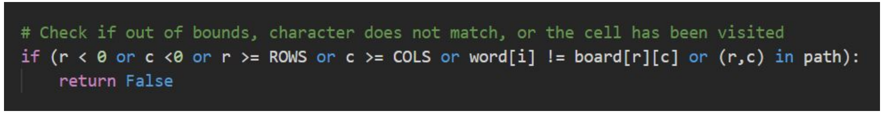
Depth-First Search(DFS)

In a DFS, you **go as deep as possible down** one path before backing up and trying a different one.

Depth-first search is like walking through a corn maze.
You explore one path, **hit a dead end**, and **go back and try a different one**.



Brute Force Approach -



```python
# Check if out of bounds, character does not match, or the cell has been visited
if (r < 0 or c <0 or r >= ROWS or c >= COLS or word[i] != board[r][c] or (r,c) in path):
    return False
```

exploring every possible path in the grid to find the target word

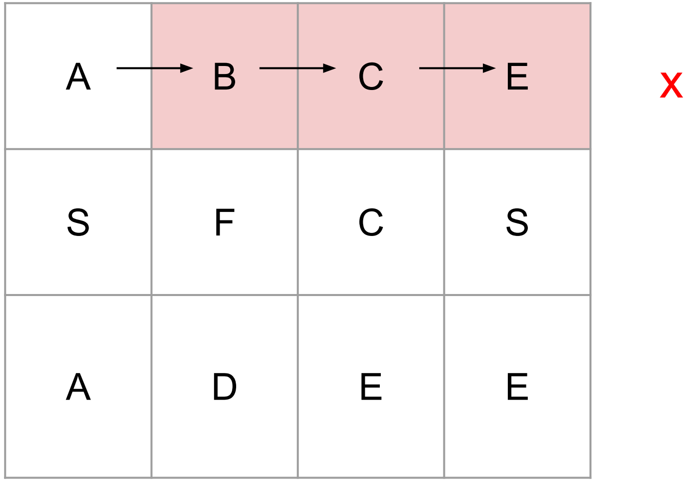
Start at the 1st cell of the grid in the first row

Explore as far as possible along each row and column **before backtracking**

Keep track of which **element** have been **visited**

board = [["**A**","B","C","E"],
["S","F","C","S"],
["A","D","E","E"]]

word = "**SEE**"

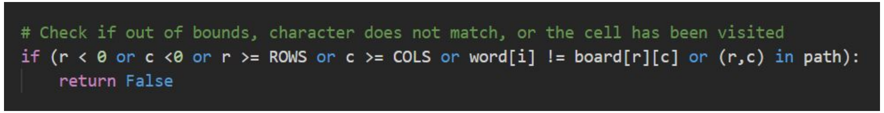


A	B	C	E
S	F	C	S
A	D	E	E

r, c, path: 0 0 set()
r, c, path: 0 1 set()
r, c, path: 0 2 set()
r, c, path: 0 3 set()

Search For “**S**”

* Brute Force Approach -



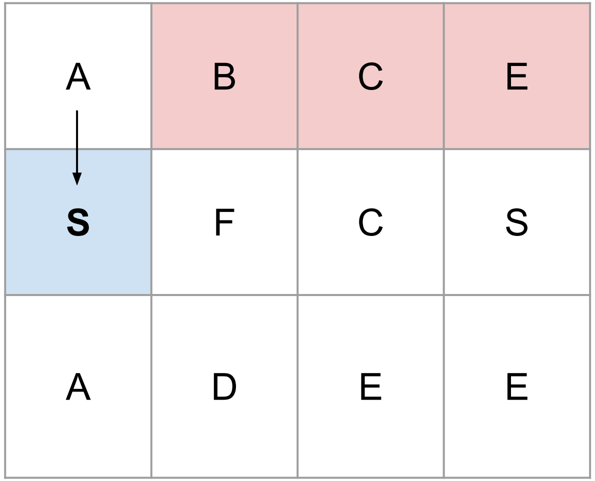
```
# Check if out of bounds, character does not match, or the cell has been visited
if (r < 0 or c <0 or r >= ROWS or c >= COLS or word[i] != board[r][c] or (r,c) in path):
    return False
```

exploring every possible path in the grid to find the target word

Start at the 1st cell of the grid in the first row

Explore as far as possible along each row and column **before backtracking**

Keep track of which **element** have been **visited**



A	B	C	E
S	F	C	S
A	D	E	E

X

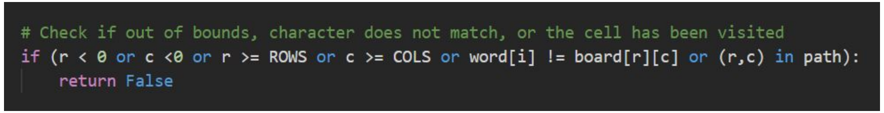
board = [["**A**","B","C","E"],
["S","F","C","S"],
["A","D","E","E"]]

word = "**SEE**"

cur path: {(1, 0)}

Search For “**S**”

Brute Force Approach -



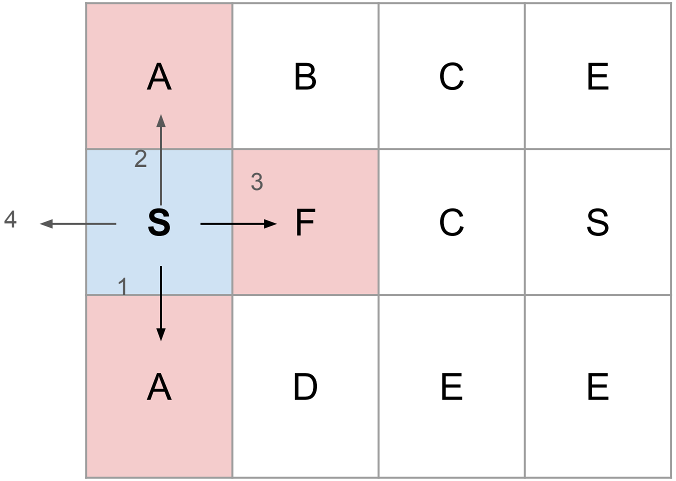
```
# Check if out of bounds, character does not match, or the cell has been visited
if (r < 0 or c <0 or r >= ROWS or c >= COLS or word[i] != board[r][c] or (r,c) in path):
    return False
```

exploring every possible path in the grid to find the target word

Start at the 1st cell of the grid in the first row

Explore as far as possible along each row and column **before backtracking**

Keep track of which **element** have been **visited**



A	B	C	E
S	F	C	S
A	D	E	E

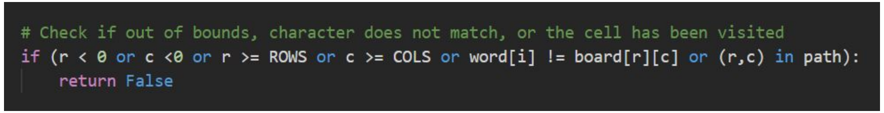
board = [["**A**","B","C","E"],
["S","F","C","S"],
["A","D","E","E"]]

word = "**SEE**"

cur path: ~~{(1, 0)}~~

Search For “**E**”

* Brute Force Approach -



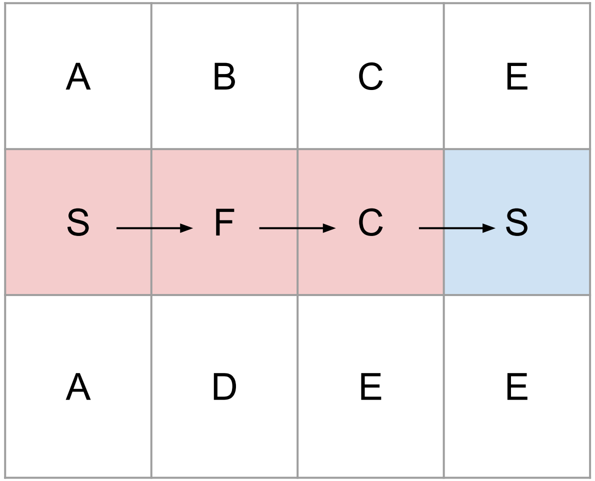
```
# Check if out of bounds, character does not match, or the cell has been visited
if (r < 0 or c <0 or r >= ROWS or c >= COLS or word[i] != board[r][c] or (r,c) in path):
    return False
```

exploring every possible path in the grid to find the target word

Start at the 1st cell of the grid in the first row

Explore as far as possible along each row and column **before backtracking**

Keep track of which **element** have been **visited**



A	B	C	E
S →	F →	C →	S
A	D	E	E

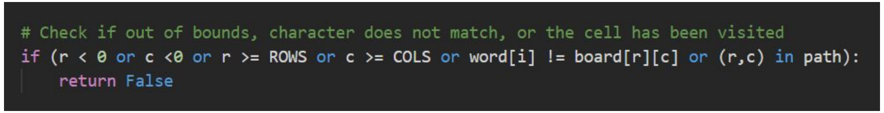
board = [["**A**","B","C","E"],
["S","F","C","S"],
["A","D","E","E"]]

word = "**SEE**"

cur path: {(1, 3)}

Search For “S” again

* Brute Force Approach -



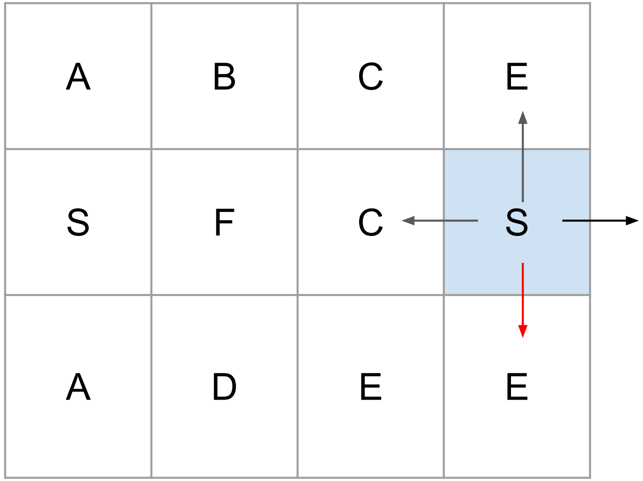
```
# Check if out of bounds, character does not match, or the cell has been visited
if (r < 0 or c <0 or r >= ROWS or c >= COLS or word[i] != board[r][c] or (r,c) in path):
    return False
```

exploring every possible path in the grid to find the target word

Start at the 1st cell of the grid in the first row

Explore as far as possible along each row and column **before backtracking**

Keep track of which **element** have been **visited**



A	B	C	E
S	F	C	S
A	D	E	E

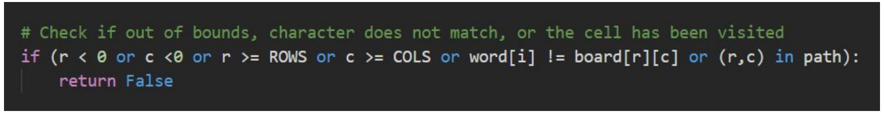
board = [["**A**","B","C","E"],
["S","F","C","S"],
["A","D","E","E"]]

word = "**SEE**"

cur path: {(1, 3)}

Search For “**E**”

Brute Force Approach -



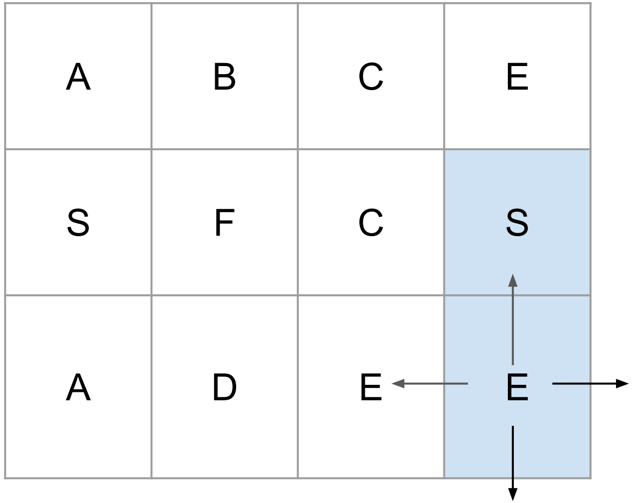
```
# Check if out of bounds, character does not match, or the cell has been visited
if (r < 0 or c <0 or r >= ROWS or c >= COLS or word[i] != board[r][c] or (r,c) in path):
    return False
```

exploring every possible path in the grid to find the target word

Start at the 1st cell of the grid in the first row

Explore as far as possible along each row and column **before backtracking**

Keep track of which **element** have been **visited**



A	B	C	E
S	F	C	S
A	D	E	E

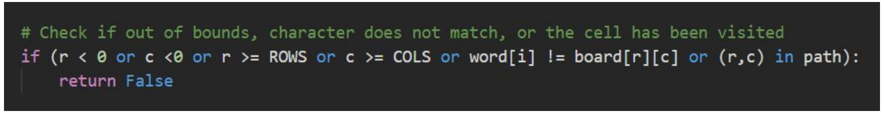
board = [["**A**","B","C","E"],
["S","F","C","S"],
["A","D","E","E"]]

word = "**SEE**"

cur path: {(2, 3), (1, 3)}

Search for the second **“E”**

Brute Force Approach -



```
# Check if out of bounds, character does not match, or the cell has been visited
if (r < 0 or c <0 or r >= ROWS or c >= COLS or word[i] != board[r][c] or (r,c) in path):
    return False
```

exploring every possible path in the grid to find the target word

Start at the 1st cell of the grid in the first row

Explore as far as possible along each row and column **before backtracking**

Keep track of which **element** have been **visited**

A	B	C	E
S	F	C	S
A	D	E	E

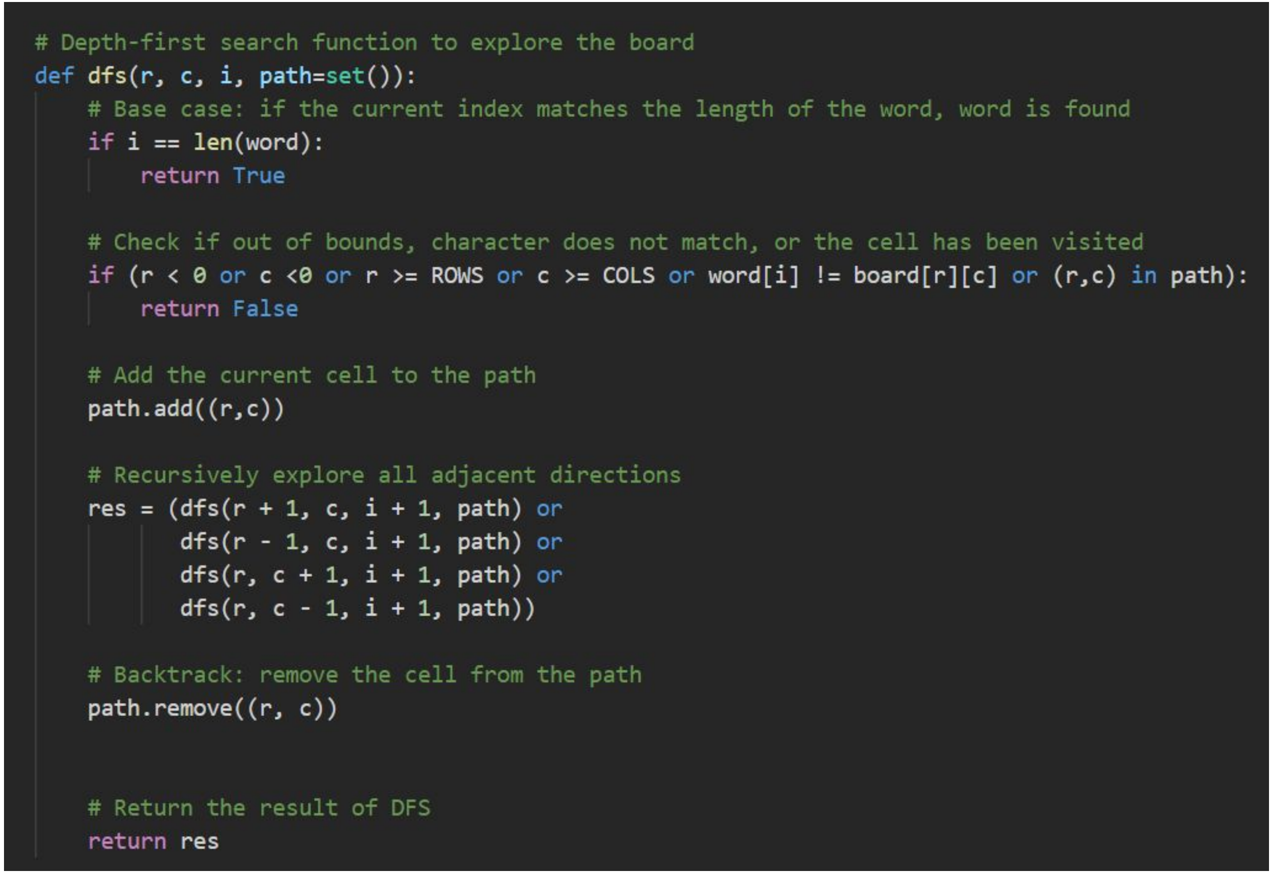
board = [["**A**","B","C","E"],
["S","F","C","S"],
["A","D","E","E"]]

word = "**SEE**"

cur path: {(2, 3), (1, 3), (2, 2)}

Search for the second “**E**”

Code Solution



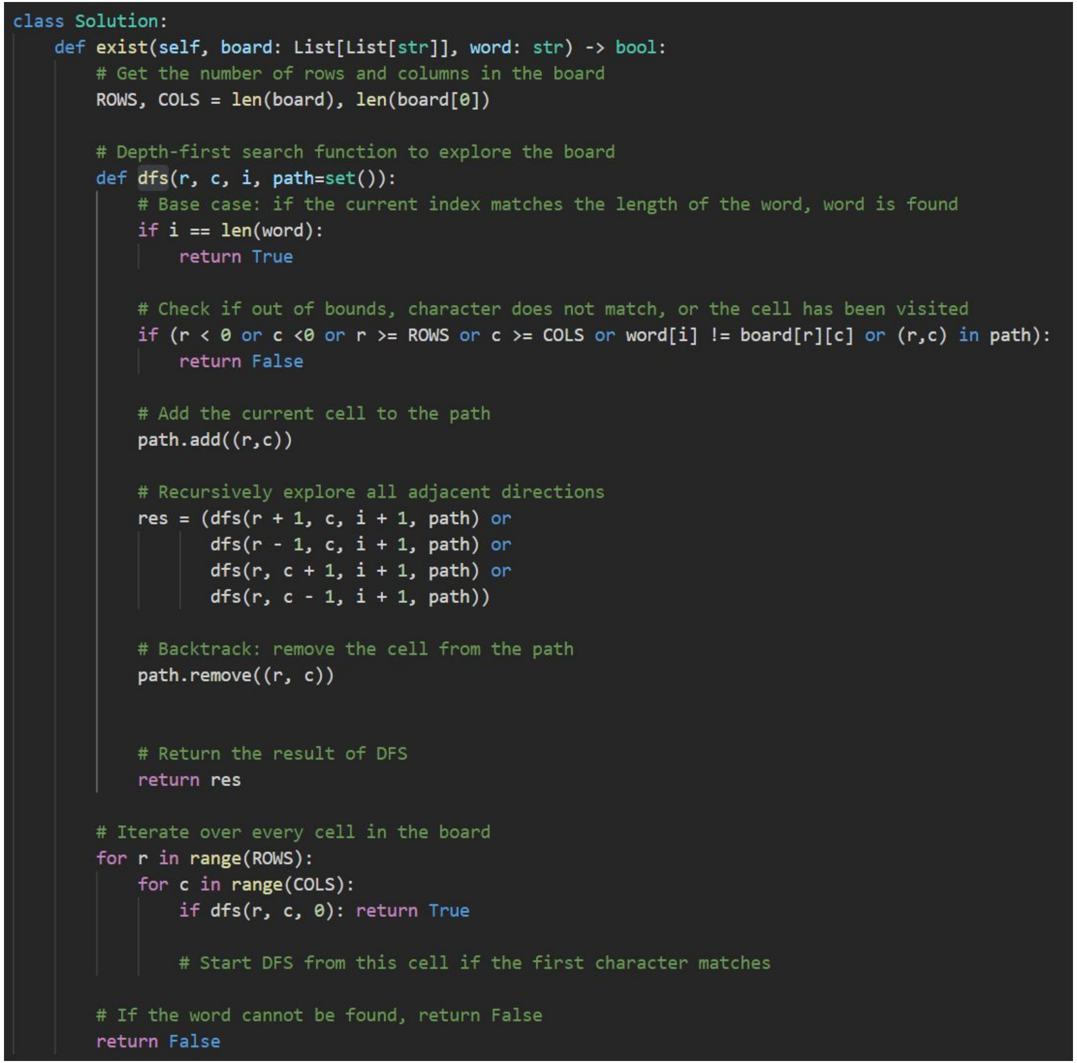
```python
# Depth-first search function to explore the board
def dfs(r, c, i, path=set()):
    # Base case: if the current index matches the length of the word, word is found
    if i == len(word):
        return True

    # Check if out of bounds, character does not match, or the cell has been visited
    if (r < 0 or c <0 or r >= ROWS or c >= COLS or word[i] != board[r][c] or (r,c) in path):
        return False

    # Add the current cell to the path
    path.add((r,c))

    # Recursively explore all adjacent directions
    res = (dfs(r + 1, c, i + 1, path) or
           dfs(r - 1, c, i + 1, path) or
           dfs(r, c + 1, i + 1, path) or
           dfs(r, c - 1, i + 1, path))

    # Backtrack: remove the cell from the path
    path.remove((r, c))


    # Return the result of DFS
    return res
```

Code Solution - Backtracking & dfs

Time Complexity:
O(M * N * 4 * 3^(L-1))

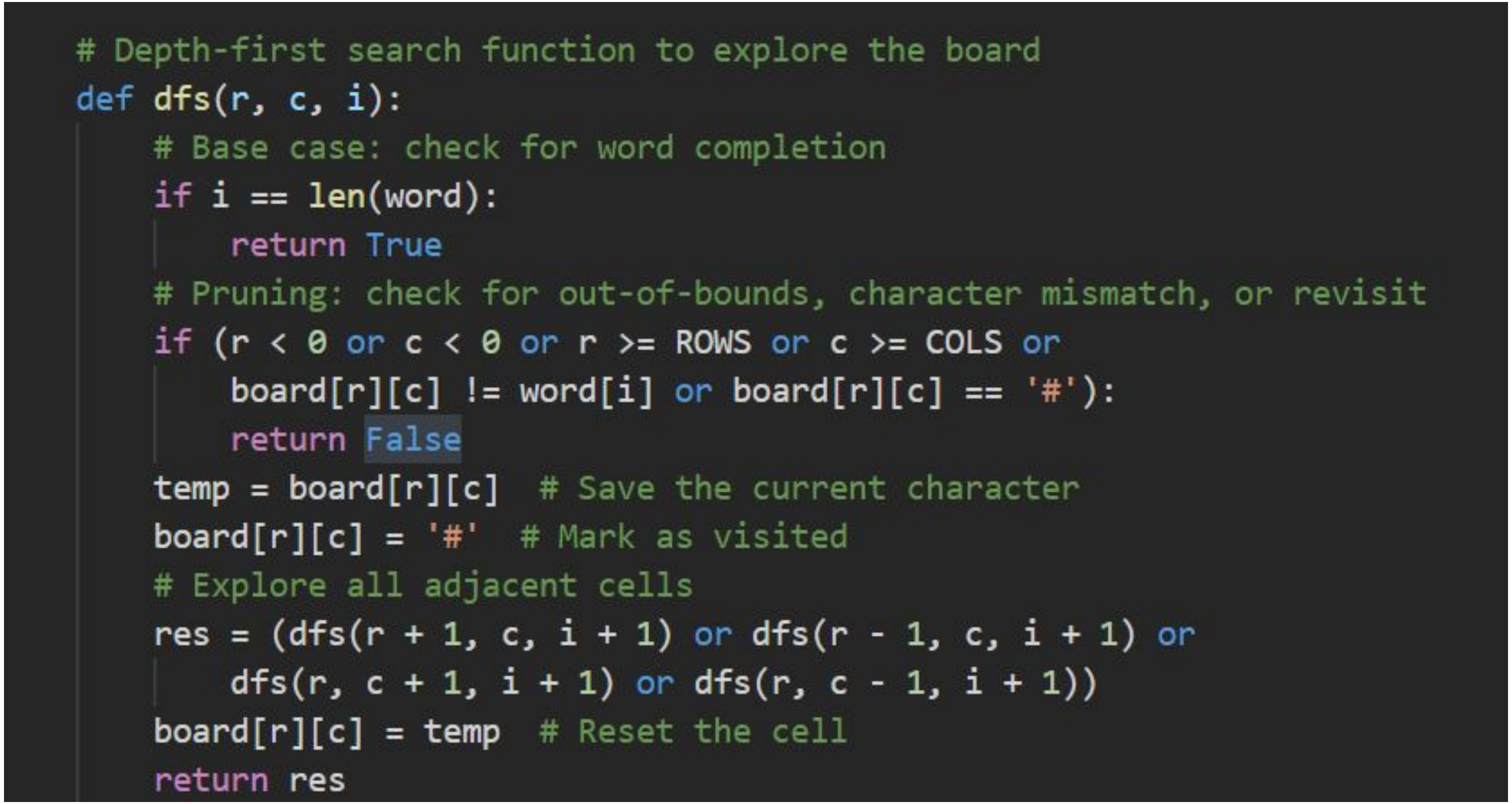
→ where L: length of the word, 3 choices at each subsequent letter, M*N grid given

Memory Complexity: O(L)
→ where L: depth of the recursion stack



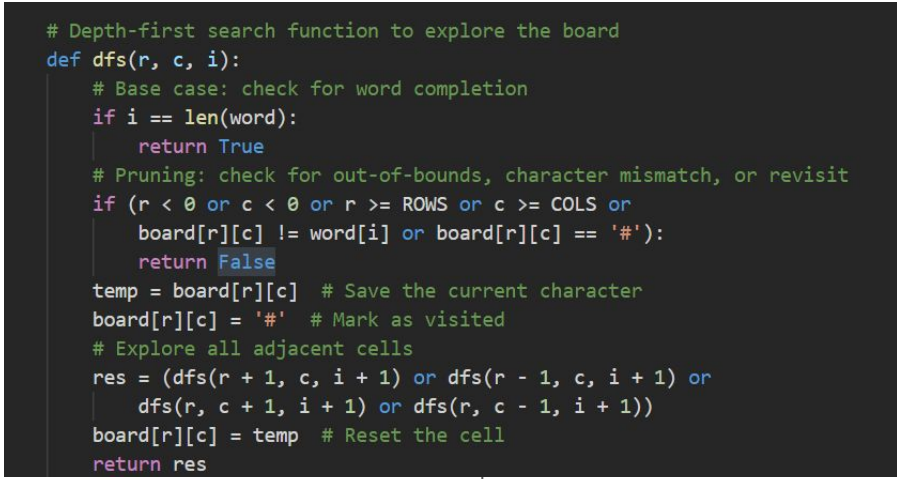
```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        # Get the number of rows and columns in the board
        ROWS, COLS = len(board), len(board[0])

        # Depth-first search function to explore the board
        def dfs(r, c, i, path=set()):
            # Base case: if the current index matches the length of the word, word is found
            if i == len(word):
                return True

            # Check if out of bounds, character does not match, or the cell has been visited
            if (r < 0 or c <0 or r >= ROWS or c >= COLS or word[i] != board[r][c] or (r,c) in path):
                return False

            # Add the current cell to the path
            path.add((r,c))

            # Recursively explore all adjacent directions
            res = (dfs(r + 1, c, i + 1, path) or
                   dfs(r - 1, c, i + 1, path) or
                   dfs(r, c + 1, i + 1, path) or
                   dfs(r, c - 1, i + 1, path))

            # Backtrack: remove the cell from the path
            path.remove((r, c))


            # Return the result of DFS
            return res

        # Iterate over every cell in the board
        for r in range(ROWS):
            for c in range(COLS):
                if dfs(r, c, 0): return True

                # Start DFS from this cell if the first character matches

        # If the word cannot be found, return False
        return False
```

Code Solution - Marking the Board Directly when tracking the visited cells



```python
# Depth-first search function to explore the board
def dfs(r, c, i):
    # Base case: check for word completion
    if i == len(word):
        return True
    # Pruning: check for out-of-bounds, character mismatch, or revisit
    if (r < 0 or c < 0 or r >= ROWS or c >= COLS or
        board[r][c] != word[i] or board[r][c] == '#'):
        return False
    temp = board[r][c]  # Save the current character
    board[r][c] = '#'  # Mark as visited
    # Explore all adjacent cells
    res = (dfs(r + 1, c, i + 1) or dfs(r - 1, c, i + 1) or
        dfs(r, c + 1, i + 1) or dfs(r, c - 1, i + 1))
    board[r][c] = temp  # Reset the cell
    return res
```

* Code Solution



```python
# Depth-first search function to explore the board
def dfs(r, c, i):
    # Base case: check for word completion
    if i == len(word):
        return True
    # Pruning: check for out-of-bounds, character mismatch, or revisit
    if (r < 0 or c < 0 or r >= ROWS or c >= COLS or
        board[r][c] != word[i] or board[r][c] == '#'):
        return False
    temp = board[r][c]  # Save the current character
    board[r][c] = '#'  # Mark as visited
    # Explore all adjacent cells
    res = (dfs(r + 1, c, i + 1) or dfs(r - 1, c, i + 1) or
        dfs(r, c + 1, i + 1) or dfs(r, c - 1, i + 1))
    board[r][c] = temp  # Reset the cell
    return res
```

A	B	C	E
S	F	C	S
A	D	E	E

A	B	C	E
#	F	C	S
A	D	E	E

A	B	C	E
S	F	C	#
A	D	#	#

감사합니다!

THANK YOU