**Problem Statement: Hashset Maze Solver**

**Scenario:** You're tasked with developing a maze-solving algorithm that utilizes hashsets for efficient pathfinding. The maze is represented as a grid of cells where some cells are walls, and others are open paths. You need to determine if there exists a path from the start cell to the end cell.

**Input Format:**

1. The first line contains two integers, m and n, representing the number of rows and columns in the maze, respectively.
2. The next m lines contain n characters each, where each character is either:
   * . representing an open path,
   * # representing a wall,
   * S representing the start cell,
   * E representing the end cell.

**Output Format:** Print "YES" if there exists a path from the start cell to the end cell. Otherwise, print "NO".

**Constraints:**

* 1 <= m, n <= 1000
* There is exactly one S and one E in the maze.

**Sample Input:**

5 5

S...#

.#.#.

.#...

..#.E

#####

**Sample Output:**

YES

**Explanation:** There is a path from the start cell S to the end cell E in the given maze.

**Extra Test Cases:**

**Test Case 1:**

makefile

Input:

3 3

S.#

#.#

#E.

Output:

NO

**Test Case 2:**

makefile

Input:

4 4

S.#.

.#.#

.#.E

..#.

Output:

YES

**Test Case 3:**

makefile

Input:

6 6

S.#...

.#..#.

#..#..

#..#.#

.#..#E

######

Output:

YES

**Test Case 4:**

shell

Input:

5 5

S####

#....

#.#.#

#....

###E#

Output:

NO

**Test Case 5:**

makefile

Input:

7 7

S......

.#.#...

..#....

.#.#.#.

..#.#..

.#.#...

.....E#

Output:

YES

**Notes:**

* The solution should efficiently handle large mazes up to the maximum constraint.
* Use hashsets to store visited cells to ensure that each cell is processed only once, reducing time complexity.
* Consider edge cases such as mazes with narrow paths and large open areas.

4o

add solution with this

ChatGPT

Here is the solution to the problem using a breadth-first search (BFS) algorithm to ensure efficient pathfinding:

python

from collections import deque

def solve\_maze(m, n, maze):

# Directions for moving in the maze (right, down, left, up)

directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

# Finding the start and end positions

start = None

end = None

for i in range(m):

for j in range(n):

if maze[i][j] == 'S':

start = (i, j)

elif maze[i][j] == 'E':

end = (i, j)

# If start or end is not found, return "NO"

if not start or not end:

return "NO"

# Initialize queue for BFS and a set for visited positions

queue = deque([start])

visited = set([start])

# BFS

while queue:

x, y = queue.popleft()

# If we reach the end, return "YES"

if (x, y) == end:

return "YES"

# Explore all possible directions

for dx, dy in directions:

nx, ny = x + dx, y + dy

# Check if the new position is within bounds and is an open path

if 0 <= nx < m and 0 <= ny < n and maze[nx][ny] != '#' and (nx, ny) not in visited:

visited.add((nx, ny))

queue.append((nx, ny))

# If the end is not reachable, return "NO"

return "NO"

# Reading input

m, n = map(int, input().split())

maze = [input().strip() for \_ in range(m)]

# Solving the maze

result = solve\_maze(m, n, maze)

print(result)

**Explanation:**

1. **Input Parsing**:
   * Read the dimensions m and n.
   * Read the maze grid line by line.
2. **Finding Start and End**:
   * Loop through the maze to find the positions of 'S' (start) and 'E' (end).
3. **Breadth-First Search (BFS)**:
   * Initialize a queue with the start position and a set to keep track of visited cells.
   * Perform BFS to explore all possible paths from the start.
   * For each cell, explore its neighbors (right, down, left, up).
   * If the end is reached, print "YES".
   * If the queue is exhausted without reaching the end, print "NO".

**Sample Run:**

For the given sample input:

5 5

S...#

.#.#.

.#...

..#.E

#####

The output will be:

YES