Q0: Recursion on 2D Arrays

1. Problem Understanding

- Recursion on 2D arrays (matrix) involves processing elements row by row or column by column.
- Useful for:
    - Traversal / Printing
    - Sum / Count
    - Searching elements
    - Pathfinding (like maze, rat in a grid)
    - Backtracking problems (all paths, subsets)
- Typically uses two indices: i for row, j for column.

---

2. Constraints

- Array size = m x n
- Indices should stay within bounds ($0 \le i < m$, $0 \le j < n$)
- Base case depends on:
    - End of row/column
    - Reaching last cell
- Stack depth = O(m * n) in full traversal

---

3. Edge Cases

- Empty matrix (m = 0 or n = 0)
- Single row or single column
- Diagonal-only traversal (for specialized problems)
- Obstacles (for pathfinding/backtracking)

---

4. Examples

```
Input: [[1,2],[3,4]] → Output (traverse): 1 2 3 4

Input: [[1,2],[3,4]] → Sum = 10

Input: Maze with 0/1 → Print all paths from top-left to bottom-right
```

---

5. Approaches

Approach 1: Full Traversal (Row-wise)

Idea:

- Traverse row by row, column by column.

Java Code:

```
void traverse(int[][] mat, int i, int j) {
    int m = mat.length, n = mat[0].length;
    if (i == m) return;
    if (j == n) {
        traverse(mat, i + 1, 0);
        return;
    }
    System.out.print(mat[i][j] + " ");
    traverse(mat, i, j + 1);
}
```

Complexity (Time & Space):

- Time: O(m * n)
- Space: O(m * n)

Approach 2: Sum of Elements

Idea:

- Add current element + recursion on rest of matrix.

Java Code:

```
int sumMatrix(int[][] mat, int i, int j) {
    int m = mat.length, n = mat[0].length;
    if (i == m) return 0;
    if (j == n) return sumMatrix(mat, i + 1, 0);
    return mat[i][j] + sumMatrix(mat, i, j + 1);
}
```

Complexity (Time & Space):

- Time: O(m * n)
- Space: O(m * n)

Approach 3: Search Element

Idea:

- Return true if element is found in recursion.

Java Code:

```java
boolean searchMatrix(int[][] mat, int i, int j, int key) {
    int m = mat.length, n = mat[0].length;
    if (i == m) return false;
    if (j == n) return searchMatrix(mat, i + 1, 0, key);
    if (mat[i][j] == key) return true;
    return searchMatrix(mat, i, j + 1, key);
}
```

Complexity (Time & Space):

- Time: O(m * n)
- Space: O(m * n)

Approach 4: Row-wise Maximum

Idea:

- Find max in current row recursively, compare with rest.

Java Code:

```java
int rowMax(int[][] mat, int i, int j) {
    int n = mat[0].length;
    if (i == mat.length) return Integer.MIN_VALUE;
    if (j == n) return rowMax(mat, i + 1, 0);
    int maxInRest = rowMax(mat, i, j + 1);
    return Math.max(mat[i][j], maxInRest);
}
```

Complexity (Time & Space):

- Time: O(m * n)
- Space: O(m * n)

Approach 5: Column-wise / Diagonal Traversal

Idea:

- Similar to row-wise but adjust indices.
- Column-wise: Swap i and j in recursion logic
- Diagonal traversal: move i+1, j+1 per step
- Time: O(m * n)
- Space: O(m * n)

Approach 6: Pathfinding (Maze / Rat in Grid)

Idea:

- Explore all 4 directions using backtracking.

Java Code:

```
void findPaths(int[][] maze, int i, int j, String path) {
    int m = maze.length, n = maze[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n || maze[i][j] == 1) return;
    if (i == m - 1 && j == n - 1) {
        System.out.println(path);
        return;
    }
    maze[i][j] = 1; // mark visited
    findPaths(maze, i + 1, j, path + "D");
    findPaths(maze, i, j + 1, path + "R");
    findPaths(maze, i - 1, j, path + "U");
    findPaths(maze, i, j - 1, path + "L");
    maze[i][j] = 0; // unmark
}
```

Complexity (Time & Space):

- Time: O(4^(m * n)) worst-case
- Space: O(m * n) recursion stack

Approach 7: Print All Paths from Top-left to Bottom-right

Idea:

- Only move right or down.

Java Code:

```
void printAllPaths(int[][] mat, int i, int j, String path) {
    int m = mat.length, n = mat[0].length;
    if (i >= m || j >= n) return;
    if (i == m - 1 && j == n - 1) {
        System.out.println(path + mat[i][j]);
        return;
    }
    printAllPaths(mat, i, j + 1, path + mat[i][j] + " ");
    printAllPaths(mat, i + 1, j, path + mat[i][j] + " ");
}
```

Complexity (Time & Space):

- Time: O(2^(m+n))
- Space: O(m+n)

Approach 8: Count Paths (Top-left to Bottom-right)

Idea:

- Count number of unique paths using recursion.Count number of unique paths using recursion.

Java Code:

```
int countPaths(int i, int j, int m, int n) {
    if (i == m - 1 && j == n - 1) return 1;
    if (i >= m || j >= n) return 0;
    return countPaths(i + 1, j, m, n) + countPaths(i, j + 1, m, n);
}
```

Complexity (Time & Space):

- Time: O(2^(m+n))
- Space: O(m+n)

Approach 9: Flood Fill / DFS on Grid

Idea:

- Mark visited cells and recursively fill adjacent cells.

Java Code:

```
void floodFill(int[][] grid, int i, int j, int oldColor, int newColor) {
    int m = grid.length, n = grid[0].length;
    if (i < 0 || j < 0 || i >= m || j >= n || grid[i][j] != oldColor) return;
    grid[i][j] = newColor;
    floodFill(grid, i + 1, j, oldColor, newColor);
    floodFill(grid, i - 1, j, oldColor, newColor);
    floodFill(grid, i, j + 1, oldColor, newColor);
    floodFill(grid, i, j - 1, oldColor, newColor);
}
```

Complexity (Time & Space):

- Time: O(m*n)
- Space: O(m*n)

Approach 10: Spiral / Zig-Zag Traversal (Advanced)

Idea:

- Maintain boundaries (top, bottom, left, right) and recurse layer by layer.
- Time: O(m * n)
- Space: O(m*n) recursion stack

---

6. Justification / Proof of Optimality

- Recursion simplifies handling of matrix traversal.
- Essential for backtracking problems in grids.
- Can be extended to DP (memoization) for optimization.

---

7. Variants / Follow-Ups

- Recursion on 3D arrays
- Recursion with constraints (obstacles, walls)
- Combined with subsequence / subset generation per row

---

8. Tips & Observations

- Base case is often reaching last row/column.
- Use visited matrix for cycles in grid problems.
- Visualize recursion as tree with branching directions.
- Backtracking requires state restoration after recursion.
- Optimize by reducing string concatenations in path storage.

Q76: Print All Maze Paths

---

1. Problem Understanding

- You are at the top-left cell (1,1) of an N×M grid and must reach the bottom-right cell (N,M).
- At each step, you can only move:
  - 'h' → horizontally to the right (→)
  - 'v' → vertically downward (↓)
- You must print all possible paths to reach the destination.

---

2. Constraints

- 1 ≤ N, M ≤ 10
- Moves allowed: only h (right) and v (down)
- Recursive solution expected

---

3. Edge Cases

- If N = 1 → only horizontal moves ("h" * (M-1))
- If M = 1 → only vertical moves ("v" * (N-1))
- If N = 1 and M = 1 → already at destination, path = ""

---

4. Examples

```
Example 1
Input:
2
2
Output:
hv
vh
Explanation:
Move right → down → "hv"
Move down → right → "vh"
```

---

5. Approaches

Approach 1: Recursion

Idea:

- At each cell (i, j):
- If not at the destination, make recursive calls:
  - Move horizontally → (i, j + 1)
  - Move vertically → (i + 1, j)
- Append 'h' or 'v' to the path string.

Steps:

- Base Case: if (i == n && j == m), print the current path string.
- If j < m, make a horizontal move ('h').
- If i < n, make a vertical move ('v').
- Recursive exploration ensures all possible paths are printed.

Java Code:

```java
static void printMazePaths(int i, int j, int n, int m, String psf) {
    // Base case: reached destination
    if (i == n && j == m) {
        System.out.println(psf);
        return;
    }

    // Horizontal move (right)
    if (j < m) {
        printMazePaths(i, j + 1, n, m, psf + "h");
    }

    // Vertical move (down)
    if (i < n) {
        printMazePaths(i + 1, j, n, m, psf + "v");
    }
}
```

```
▨ Initial call:
printMazePaths(1, 1, N, M, "");

ex
printMazePaths(1, 1, 3, 3, "")

(1,1,"")
           /                  \
      h→(1,2,"h")          v→(2,1,"v")
      /        \            /         \
 h→(1,3,"hh") v→(2,2,"hv") h→(2,2,"vh") v→(3,1,"vv")
    |           |            |            |
 v→(2,3,"hhv") v→(2,3,"hvv") h→(2,3,"vhh") v→(3,2,"vvh")
    |           |            |            |
 v→(3,3,"hhvv") v→(3,3,"hv v") h→(3,3,"vhhv") v→(3,3,"vvhv")

Output
hhvv
hvhv
hvvh
vhhv
vhhv
vvhh
```

Complexity (Time & Space):

- Time Complexity: O(2^(N+M)) → each move branches into two possibilities
- Space Complexity: O(N + M) → recursion stack depth

---

6. Justification / Proof of Optimality

- Every path represents a combination of horizontal and vertical moves.
- For a grid of size (N, M), total paths = C((N-1)+(M-1), (N-1)) (combinatorial count).
- Recursion naturally explores all paths.

---

7. Variants / Follow-Ups

- Print paths with diagonal moves ('d') — extend recursion with one more call.
- Return list of paths instead of printing.
- Count total paths (return int instead of printing).
- Blocked maze — skip moves through blocked cells.

---

8. Tips & Observations

- This is a template for many grid problems (rat in a maze, unique paths, etc.).
- Always make horizontal call before vertical, as required.
- For counting or storing paths, modify the base case to accumulate results.

Q77: Maze Paths with Jumps

1. Problem Understanding

- You are in the top-left cell (1, 1) of an n × m grid and must reach the bottom-right cell (n, m).
- In each move, you can jump:
- Horizontally → to the right, by 1 to m - j steps (h1, h2, ...).
- Vertically → downward, by 1 to n - i steps (v1, v2, ...).
- Diagonally → to the bottom-right, by 1 to min(n - i, m - j) steps (d1, d2, ...).
- You must print all possible paths to reach the destination.

---

2. Constraints

- 1 ≤ N, M ≤ 5
- Allowed moves: multiple jumps (h, v, d)
- Recursive solution expected (no loops for path generation except for jump range)

---

3. Edge Cases

- If already at destination (i == n && j == m), print the path.
- If n = 1 and m = 1, only one possible path — empty string.
- Smallest jump allowed = 1 step.
- Largest jump allowed = remaining steps in that direction.

---

4. Examples

```
Input:
2
2
Output:
h1v1
v1h1
d1
Explanation:
From (1,1) to (2,2):
Move right 1 → down 1 → "h1v1"
Move down 1 → right 1 → "v1h1"
Move diagonally 1 → "d1"
```

---

5. Approaches

Approach 1: Recursive Backtracking

Idea:

- At each position (i, j):
- Try all horizontal jumps (from 1 to m - j).
- Try all vertical jumps (from 1 to n - i).
- Try all diagonal jumps (from 1 to min(n - i, m - j)).
- Each recursive call appends the move (h, v, or d + jump size) to the path string.

Steps:

- Base Case:
- If (i == n && j == m), print the path string.
- Recursive Exploration:
- For h → loop through all possible jump sizes (1 to m - j)
- For v → loop through (1 to n - i)
- For d → loop through (1 to min(n - i, m - j))
- Recurse to new position after jump.

Java Code:

```
static void printMazePathsWithJumps(int i, int j, int n, int m, String psf) {
    // Base case: reached destination
    if (i == n && j == m) {
        System.out.println(psf);
        return;
    }

    // Horizontal jumps
    for (int jump = 1; j + jump <= m; jump++) {
        printMazePathsWithJumps(i, j + jump, n, m, psf + "h" + jump);
    }

    // Vertical jumps
    for (int jump = 1; i + jump <= n; jump++) {
        printMazePathsWithJumps(i + jump, j, n, m, psf + "v" + jump);
    }

    // Diagonal jumps
    for (int jump = 1; i + jump <= n && j + jump <= m; jump++) {
        printMazePathsWithJumps(i + jump, j + jump, n, m, psf + "d" + jump);
    }
}
Initial call:

printMazePathsWithJumps(1, 1, n, m, "");

                  (1,1,"")
                /     |     \
          h1→(1,2)   v1→(2,1)  d1→(2,2)
```

```
             /              \
   v1→(2,2,"h1v1")   h1→(2,2,"v1h1")

Prints:
  "h1v1"
  "v1h1"
  "d1"
```

Complexity (Time & Space):

- Time Complexity:
- O(3^(N+M)) → each cell can generate up to 3 recursive branches (h, v, d).
- (Each branch further multiplies by possible jump counts, but grid size ≤ 5 keeps it feasible.)
- Space Complexity:
- O(N + M) → recursion stack depth.

---

6. Justification / Proof of Optimality

- Each recursive call explores all valid jump paths until reaching the destination.
- The use of nested loops ensures every possible jump distance is explored once per direction.

---

7. Variants / Follow-Ups

- Return paths instead of printing — collect paths in a list and return.
- Blocked cells — skip moves through blocked cells.
- Count total paths — return an integer instead of printing.
- Allow only specific move sets (e.g., no diagonals) — adjust recursive branches.

---

8. Tips & Observations

- Use this as a template for problems involving multiple move lengths.
- "Jump" just means moving multiple steps in one direction, not repeated single moves.
- Keep horizontal call before vertical, and vertical before diagonal for consistent order.
- The recursive structure is almost identical to normal maze paths — just extended with loops for jump lengths.

Q80: Count Maze Path

---

1. Problem Understanding

- You have a grid of size N × M.
- You start at the top-left cell (1,1) and want to reach bottom-right cell (N,M).
- Allowed moves:

- Horizontal → move 1 step to the right (h)
- Vertical → move 1 step down (v)
- Task: count the total number of paths (not print them).

---

2. Constraints

- 1 ≤ N ≤ 10
- 1 ≤ M ≤ 10

---

3. Edge Cases

- Single row (N = 1) → only horizontal moves possible → 1 path.
- Single column (M = 1) → only vertical moves possible → 1 path.
- Grid size 1 × 1 → already at destination → 1 path.

---

4. Examples

```
Input:
2
2
Output:
2
Explanation:
Two possible paths: h→v and v→h.
```

---

5. Approaches

Approach 1: Recursive Count

Idea:

- Each cell (i,j) can move:
- Horizontally to (i,j+1)
- Vertically to (i+1,j)
- Count paths recursively from (i,j) to destination (N,M).
- Base case: reached destination → return 1.

Java Code:

```
static int countMazePath(int sr, int sc, int dr, int dc) {
    // Base case: reached destination
    if (sr == dr && sc == dc) return 1;

    int count = 0;
```

```
    // Move horizontally if within bounds
    if (sc + 1 <= dc) count += countMazePath(sr, sc + 1, dr, dc);

    // Move vertically if within bounds
    if (sr + 1 <= dr) count += countMazePath(sr + 1, sc, dr, dc);

    return count;
}
System.out.println(countMazePath(1, 1, N, M));
Initial call: (1,1) → destination (2,2)
               (1,1)
             /       \
        h→(1,2)      v→(2,1)
           |              |
      h not possible    v not possible
           |              |
        (2,2)           (2,2)
Count: 2 paths
```

Complexity (Time & Space):

- Time Complexity: O(2^(N+M)) → each step can branch into horizontal or vertical move.
- Space Complexity: O(N+M) → recursion stack depth.

---

6. Justification / Proof of Optimality

- Each cell is a decision point: move right or down.
- Base case ensures counting only valid paths.
- Recursive addition of counts gives total number of paths.
- Works correctly for edge cases (1×N, N×1 grids).

---

7. Variants / Follow-Ups

- Print all paths (instead of counting).
- Allow diagonal moves (h,v,d) → see Maze Paths with Jumps problem.
- Memoization/DP → optimize to O(N×M) time complexity.
- Return paths as list → for further processing instead of just count.

---

8. Tips & Observations

- Decision at each cell: You can either move right or down.
- Base case: When you reach the bottom-right cell (N,M), count as 1.
- Prune early: Do not move out of bounds (beyond row N or column M).
- Observation:
  - Paths in an N×M grid = combination problem → C(N+M-2, N-1) (can be verified via recursion).
  - Recursion explores all paths systematically.
- Edge cases:

- Single row or single column → only 1 path.
  - Grid size 1×1 → only 1 path.
- Optimization hint:
  - Recursive solution can be converted to DP using a 2D table to store intermediate counts for each cell.
- Analogy: Think of recursion like building a tree of decisions at each cell: move right or down, then sum counts from each branch.

Q81: Count Maze Paths – Every Direction

---

1. Problem Understanding

- Grid of size N × M.
- Start at top-left (1,1) and reach bottom-right (N,M).
- Allowed moves: all 8 directions: up, down, left, right, and diagonals.
- Constraint: A cell cannot be visited twice in the same path.
- Task: Count total number of paths from start to end.

---

2. Constraints

- 1 ≤ N, M ≤ 9
- Grid small → recursion feasible.

---

3. Edge Cases

- Start = End → return 1.
- Single row or single column → only forward moves.
- Ensure no cycles using visited matrix.

---

4. Examples

```
Input:
2 2
Output:
5
Explanation: 5 valid paths from (1,1) to (2,2) without repeating cells.
```

---

5. Approaches

Approach 1: Recursion + Backtracking

Idea:

- Use a visited matrix to mark cells and explore all 8 directions recursively. Backtrack after each recursive call.

Steps:

- Base case: if (i,j) is out of bounds or already visited → return 0.
- If (i,j) = destination → return 1.
- Mark (i,j) as visited.
- Recurse into all 8 directions.
- Unmark (i,j) after recursion.
- Sum counts from all directions.

Java Code:

```
static int countAllPath(int n, int m) {
    boolean[][] visited = new boolean[n+1][m+1]; // 1-based indexing
    return helper(1, 1, n, m, visited);
}

static int helper(int i, int j, int n, int m, boolean[][] visited) {
    if (i < 1 || j < 1 || i > n || j > m || visited[i][j]) return 0;
    if (i == n && j == m) return 1;

    visited[i][j] = true;
    int count = 0;

    int[] dirX = {-1, -1, -1, 0, 0, 1, 1, 1};
    int[] dirY = {-1, 0, 1, -1, 1, -1, 0, 1};

    for (int d = 0; d < 8; d++) {
        count += helper(i + dirX[d], j + dirY[d], n, m, visited);
    }

    visited[i][j] = false; // backtrack
    return count;
}
(1,1)
 ├─ (1,2)
 │    └─ (2,2) → count=1
 ├─ (2,1)
 │    └─ (2,2) → count=1
 ├─ (2,2) → count=1
 ├─ (1,0) invalid
 ├─ (0,1) invalid
 ├─ (0,0) invalid
 ├─ (0,2) invalid
 └─ (2,0) invalid
```

Complexity (Time & Space):

- Time Complexity: O(8^(N×M)) → exponential, worst case explores all paths.
- Space Complexity: O(N×M) → recursion stack + visited matrix.

---

6. Justification / Proof of Optimality

- Ensures all possible paths are counted without repeating a cell.
- Visited matrix prevents cycles.
- Base case guarantees only valid paths contribute.
- Backtracking ensures paths are fully explored and state is reset.

---

7. Variants / Follow-Ups

- Print all paths instead of counting.
- Restrict moves to only horizontal, vertical, or diagonal forward.
- Find longest/shortest path using DFS + backtracking.
- Maze with obstacles → skip blocked cells.
- Return all paths as a list of strings.

---

8. Tips & Observations

- Always use visited matrix to avoid cycles.
- For counting only, order of directions does not matter.
- Use arrays for 8 directions to simplify code.
- Works only for small grids due to exponential growth.
