Graph notes

Every backtracking problem can be solved by the following strategy:

"choose" → "explore" → "unchoose"

- Use an iterator to list all the possible starting points for our recursion.
- Select one number by adding it to a stack that holds the current branch.
- Recursively call the 'explore_helper' function which will carry on the recursion and pass along the stack which contains the numbers chosen in the current branch.
- Remove the recently added number and go back to step 1 to explore another sub-branch.
  The termination condition that will stop the recursion and add the current branch to the 'results' list is given by the comparison between the length of the branch and the length of the original list to permute.

Number of Islands (LC 200)

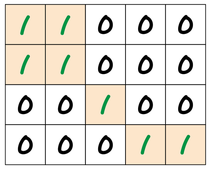
200. Number of Islands
Medium

Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

```python
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid:
            return 0
        rows, cols = len(grid), len(grid[0])

        visit = set()
        islands = 0

        def dfs(r, c):
            if grid[r][c] == "0" or (r,c) in visit:
                return
            visit.add((r, c))
            if 0 <= r + 1 < rows and 0 <= c < cols:
                dfs(r + 1, c)
            if 0 <= r - 1 < rows and 0 <= c < cols:
                dfs(r - 1, c)
            if 0 <= r < rows and 0 <= c + 1 < cols:
                dfs(r, c + 1)
            if 0 <= r < rows and 0 <= c - 1 < cols:
                dfs(r, c - 1)

        for row in range(rows):
            for col in range(cols):
                if (grid[row][col] == "1" and (row, col) not in visit):
                    dfs(row, col)
                    islands += 1

        return islands
```

```python
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid:
            return 0

        rows, cols = len(grid), len(grid[0])

        visit = set()
        islands = 0

        def bfs(r, c):
            q = collections.deque()
            visit.add((r, c))
            q.append((r, c))
            while q:
                rw, cl = q.popleft()
                directions = [[1, 0], [-1, 0], [0, 1], [0, -1]]
                for dr, dc in directions:
                    r, c = rw + dr, cl + dc
                    if (r in range(rows) and
                        c in range(cols) and
                        grid[r][c] == "1" and
                        (r, c) not in visit):
                        q.append((r, c))
                        visit.add((r,c))

        for row in range(rows):
            for col in range(cols):
                if (grid[row][col] == "1" and
                    (row, col) not in visit):
                    bfs(row, col)
                    islands += 1
        return islands
```



Explore and record if the area has not been visited.

```
Example 1:
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
```

Max Area of Island (LC 695)

695. Max Area of Island
Medium

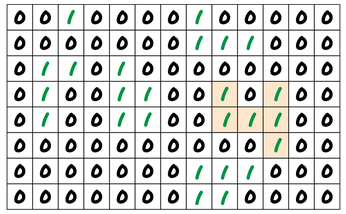
You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The area of an island is the number of cells with a value 1 in the island.

Return the maximum area of an island in grid. If there is no island, return 0.

```python
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        if not grid:
            return 0
        rows, cols = len(grid), len(grid[0])
        visited = set()
        maxArea = 0

        def dfs(r, c):
            if ((r,c) in visited or
                    r not in range(rows) or
                    c not in range(cols) or
                    grid[r][c] == 0 ):
                return 0
            visited.add((r,c))
            return 1 + dfs(r+1,c) + dfs(r-1,c) + dfs(r,c+1) + dfs(r,c-1)

        for row in range(rows):
            for col in range(cols):
                if (grid[row][col] == 1 and
                        (row, col) not in visited):
                    island_area = dfs(row, col)
                    maxArea = max(maxArea, island_area)

        return maxArea
```

```python
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        rows, cols = len(grid), len(grid[0])
        self.max_area = 0
        visited = set()

        def bfs(r, c):
            area = 1
            q = collections.deque()
            q.append((r,c))
            directions = [(1,0), (-1,0), (0,1), (0,-1)]
            visited.add((r, c))
            while q:
                qr, qc = q.popleft()
                for dr, dc in directions:
                    row, col = qr + dr, qc + dc
                    if (row in range(rows) and
                            col in range(cols) and
                            (row, col) not in visited and
                            grid[row][col] == 1):
                        visited.add((row, col))
                        q.append((row, col))
                        area += 1
            self.max_area = max(self.max_area, area)

        for row in range(rows):
            for col in range(cols):
                if grid[row][col] == 1 and (row, col) is not visited:
                    bfs(row, col)

        return self.max_area
```



Explore and record the area of an island, as long as it has not been visited before

```
Example 1:

Input: grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],
               [0,0,0,0,0,0,0,1,1,1,0,0,0],
               [0,1,1,0,1,0,0,0,0,0,0,0,0],
               [0,1,0,0,1,1,0,0,1,0,1,0,0],
               [0,1,0,0,1,1,0,0,1,1,1,0,0],
               [0,0,0,0,0,0,0,0,0,0,1,0,0],
               [0,0,0,0,0,0,0,1,1,1,0,0,0],
               [0,0,0,0,0,0,0,1,1,0,0,0,0]]
Output: 6
Explanation: The answer is not 11, because the
island must be connected 4-directionally.
```

Clone Graph (LC 133)

133. Clone Graph
Medium

Given a reference of a node in a connected undirected graph.

Return a deep copy (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

```python
class Node:
    def __init__(self, val = 0, neighbors = None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []

class Solution:
    def cloneGraph(self, node: Optional['Node']) -> Optional['Node']:
        hashmap = {}

        def dfs(node):
            if node in hashmap:
                return hashmap[node]

            copy = Node(node.val)
            hashmap[node] = copy

            for next_node in node.neighbors:
                copy.neighbors.append(dfs(next_node))
            return copy

        return dfs(node) if node else None
```

Pacific Atlantic Water Flow (LC 417)

417. Pacific Atlantic Water Flow
Medium

There is an m x n rectangular island that borders both the Pacific Ocean and Atlantic Ocean. The Pacific Ocean touches the island's left and top edges, and the Atlantic Ocean touches the island's right and bottom edges.

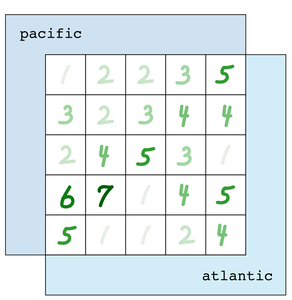
The island is partitioned into a grid of square cells. You are given an m x n integer matrix heights where heights[r][c] represents the height above sea level of the cell at coordinate (r, c).

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is less than or equal to the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return a 2D list of grid coordinates result where result[i] = [ri, ci] denotes that rain water can flow from cell (ri, ci) to both the Pacific and Atlantic oceans.

```python
class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:

        ROWS, COLS = len(heights), len(heights[0])
        pacific, atlantic = set(), set()

        def dfs(r, c, visit, prevHeight):
            if (
                (r, c) in visit
                or r < 0
                or c < 0
                or r == ROWS
                or c == COLS
                or heights[r][c] < prevHeight
            ):
                return
            visit.add((r, c))
            dfs(r + 1, c, visit, heights[r][c])
            dfs(r - 1, c, visit, heights[r][c])
            dfs(r, c + 1, visit, heights[r][c])
            dfs(r, c - 1, visit, heights[r][c])

        for c in range(COLS):
            dfs(0, c, pacific, heights[0][c])
            dfs(ROWS - 1, c, atlantic, heights[ROWS - 1][c])

        for r in range(ROWS):
            dfs(r, 0, pacific, heights[r][0])
            dfs(r, COLS - 1, atlantic, heights[r][COLS - 1])

        result = []
        for r in range(ROWS):
            for c in range(COLS):
                if (r, c) in pacific and (r, c) in atlantic:
                    result.append([r, c])
        return result
```

```
Example 1:

Input: heights = [[1,2,2,3,5],
                  [3,2,3,4,4],
                  [2,4,5,3,1],
                  [6,7,1,4,5],
                  [5,1,1,2,4]]
Output: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]
```



Surrounded Regions (LC 130)

130. Surrounded Regions
Medium

Given an m x n matrix board containing 'X' and 'O', capture all regions that are 4-directionally surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

```python
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        rows, cols = len(board), len(board[0])
        o_s = set()
        def dfs(r, c, visited):
            if (r not in range(rows) or
                    c not in range(cols) or
                    board[r][c] == "X" or
                    (r, c) in visited):
                return

            visited.add((r,c))
            directions = [(1,0),(-1,0),(0,1),(0,-1)]
            for dr, dc in directions:
                row, col = r + dr, c + dc
                dfs(row, col, visited)

        for row in range(rows):
            dfs(row, 0, o_s)
            dfs(row, cols-1, o_s)
        for col in range(cols):
            dfs(0, col, o_s)
            dfs(rows-1, col, o_s)

        for row in range(rows):
            for col in range(cols):
                if (row, col) not in o_s and board[row][col] == "O":
                    board[row][col] = "X"
```

Rotting Oranges (LC 994)

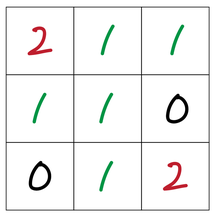
994. Rotting Oranges
Medium

You are given an m x n grid where each cell can have one of three values:

0 representing an empty cell,
1 representing a fresh orange, or
2 representing a rotten orange.
Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1.

```python
class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        rows, cols = len(grid), len(grid[0])
        q_rotten = collections.deque()
        fresh = 0
        time = 0

        for r in range(rows):
            for c in range(cols):
                if grid[r][c] == 1:
                    fresh += 1
                if grid[r][c] == 2:
                    q_rotten.append((r,c))

        directions = [(0,1),(0,-1),(1,0),(-1,0)]
        while fresh > 0 and q_rotten:
            length_q = len(q_rotten)
            for i in range(length_q):
                r, c = q_rotten.popleft()
                for dr, dc in directions:
                    row, col = r + dr, c + dc
                    if (row in range(rows) and
                            col in range(cols) and
                            grid[row][col] == 1):
                        grid[row][col] = 2
                        q_rotten.append((row, col))
                        fresh -= 1
            time += 1
        return time if fresh == 0 else -1
```



[1] Visit each cell, and record
- locations of rotten (in a queue)
- number of fresh oranges

```
q_rotten = [(0,0), (2,2)]
fresh = 5
```

[2] Because q_rotten stores the coordinates of the rotten(2) orange, start from those points.
- loop through until
  q_rotten it is empty, AND
  until fresh count is 0.
- for the number of rotten orange coordinates are inside q_rotten, loop through that number at a time.
- if an adjacent apple is fresh (1), change it to rotten(2), and add it to q_rotten.
- meanwhile decrement fresh by 1

```
Example 1:
[2,1,1]   [2,2,1]   [2,2,2]   [2,2,2]   [2,2,2]
[1,1,0] -> [2,1,0] -> [2,2,0] -> [2,2,0] -> [2,2,0]
[0,1,1]   [0,1,1]   [0,1,1]   [0,2,1]   [0,2,2]
Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4

Example 2:
[2,1,1]   [2,2,1]   [2,2,2]
[1,1,0] -> [2,1,0] -> [2,2,0]
[0,1,2]   [0,2,2]   [0,2,2]
Input: grid = [[2,1,1],[1,1,0],[0,1,2]]
Output: 2
```