```
Graph notes

Every backtracking problem can be solved by the following strategy:
```

	Use an iterator to list all the possible starting points for our recursion.
"choose"	Select one number by adding it to a stack that holds the current branch.
"explore"	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.
"unchoose"	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.

DFS

```
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
```

BFS

```
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
```

1	1	0	0	0
1	1	0	0	0
0	0	1	0	0
0	0	0	1	1

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.

DFS

```
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
```

BFS

```
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
```

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

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;
}
```

DFS

```
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.

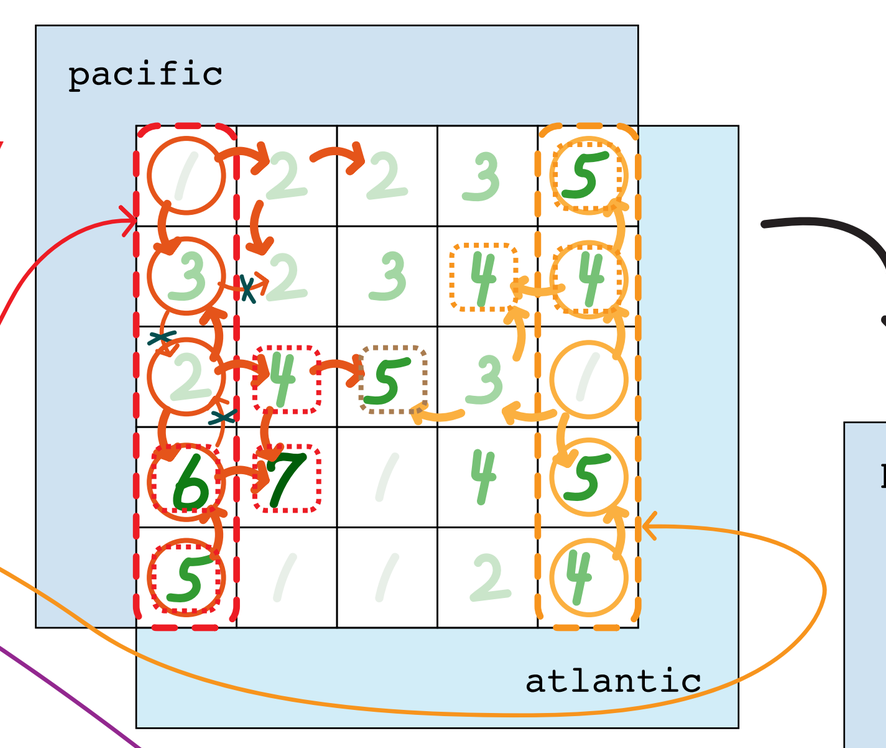
Return a 2D list of grid coordinates result where result[i] = [ri, ci] denotes that rain water can flow from cell (ri, ci) to both the Pacific and Atlantic oceans.

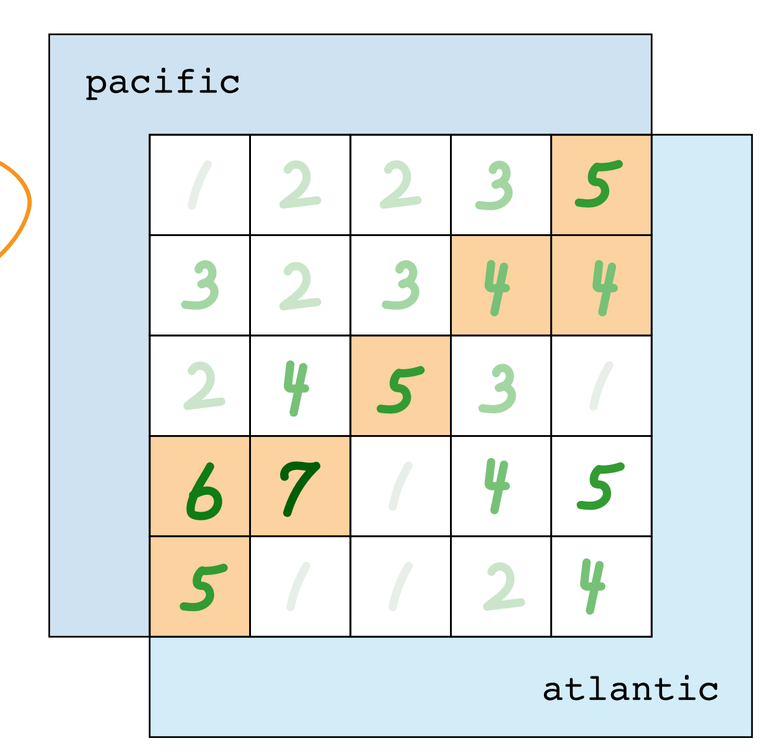
DFS

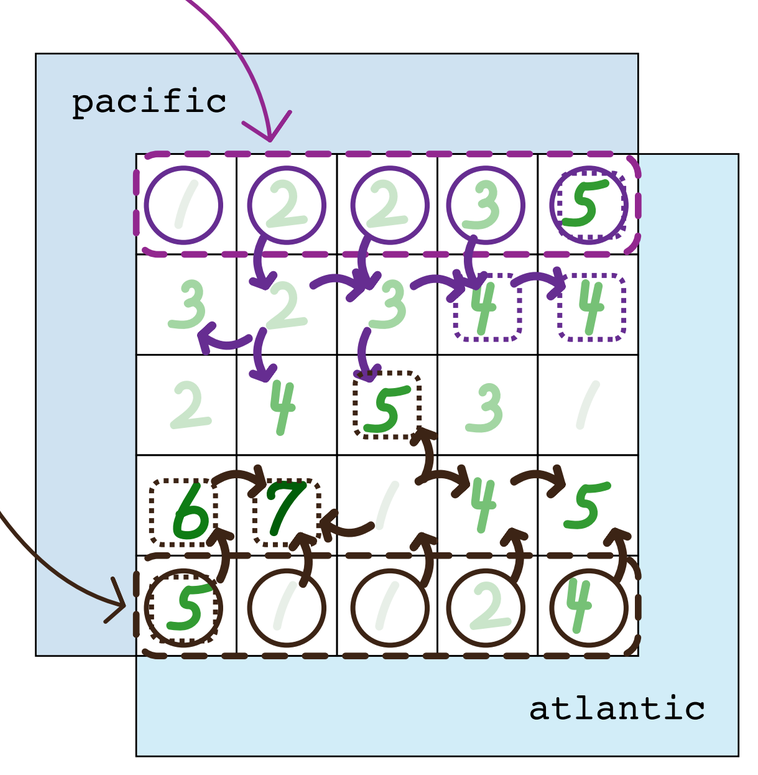
```
class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:

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

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

        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
```

if the curr height < prevHeight, then return

always going up in value







Lastly, visit each (row, col) in the matrix, and if (row,col) exists in both pacific set and atlantic set, add to answer.

```
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]]
```

[1] create two sets:
- pacific = set()
- atlantic = set()

[2] starting at the edge of the matrix,
- run the dfs (or bfs) and explore adjacent cells, particularly if the **previous cell value < current cell value**
- Check to see if starting from the edge of one side reaches the other side and:
  - is saved in the pacific set
  - is saved in the atlantic set

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.

DFS

```
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.

BFS

```
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
```

2	1	1
1	1	0
0	1	2

[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
```

Course Schedule (LC 207)

207. Course Schedule
Medium

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
Return true if you can finish all courses. Otherwise, return false.

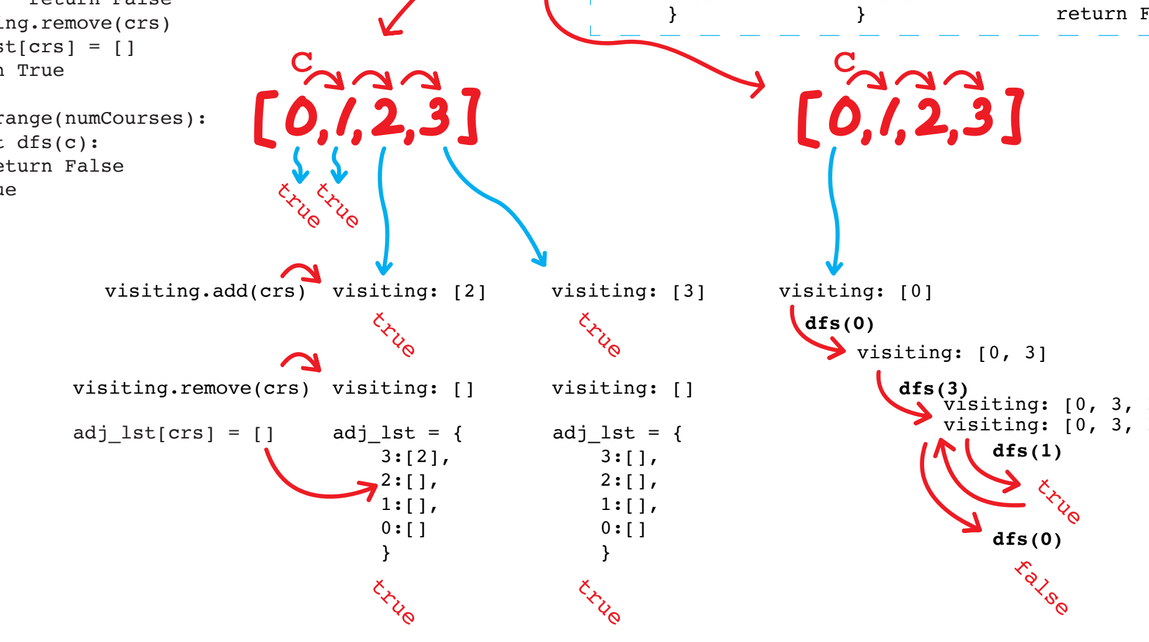
[1] create an adjacency list of interrelated nodes

[2] use dfs() to visit each node starting from range of 0 to numCourses. Keep track of visited nodes using a set.

DFS

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        adj_lst = {i: [] for i in range(numCourses)}

        for crs, prereq in prerequisites:
            adj_lst[crs].append(prereq)

        visiting = set()

        def dfs(crs):
            if crs in visiting:
                return False
            if adj_lst[crs] == []:
                return True

            visiting.add(crs)
            for prereq in adj_lst[crs]:
                if not dfs(prereq):
                    return False
            visiting.remove(crs)
            adj_lst[crs] = []
            return True

        for c in range(numCourses):
            if not dfs(c):
                return False
        return True
```

```
adj_lst = {      adj_lst = {      Example 1:
    3:[],            3:[2],       input:
    2:[],            2:[1,0],         numCourses = 4
    1:[],            1:[],            prerequisites = [[3,2],[2,1],[2,0]]
    0:[]             0:[]         output:
    }                }                return True

adj_lst = {      adj_lst = {      Example 2:
    3:[],            3:[2],       input:
    2:[],            2:[1,0],         numCourses = 4
    1:[],            1:[],            prerequisites = [[3,2],[2,1],[2,0],[0,3]]
    0:[]             0:[3]        output:
    }                }                return False
```



Course Schedule II (LC 210)

210. Course Schedule II
Medium

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

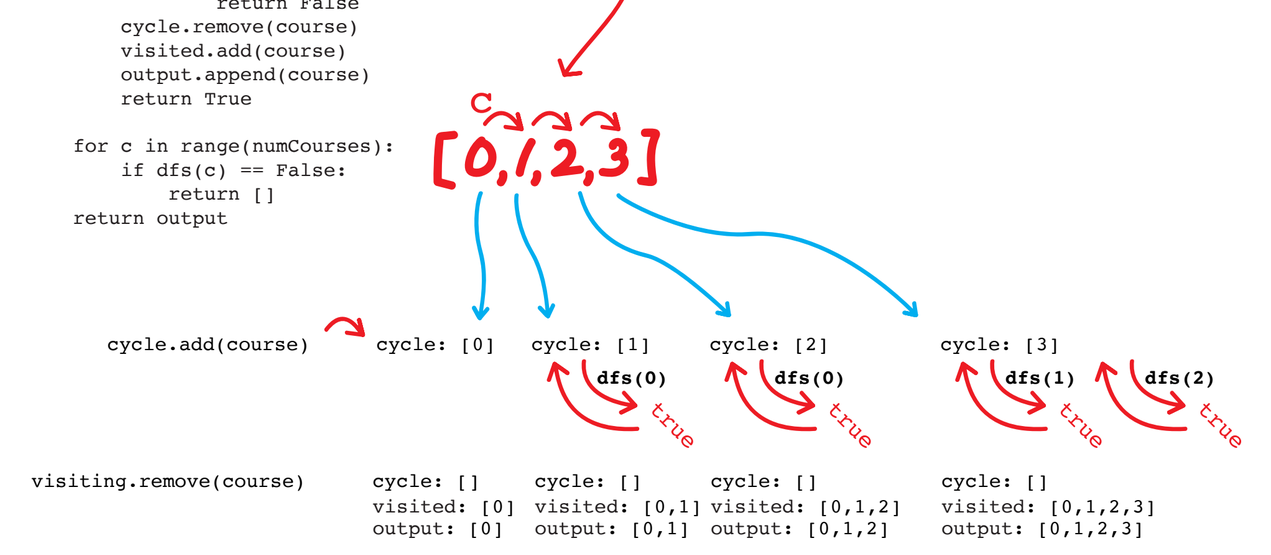
[1] create an adjacency list of interrelated nodes

[2] use dfs() to visit each node starting from range of 0 to numCourses.
Keep track of visited nodes using a set.
Keep track of a cycle using a set.

DFS

```
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        adj_lst = {k:[] for k in range(numCourses)}

        for crs, prereq in prerequisites:
            adj_lst[crs].append(prereq)

        output = []
        visited, cycle = set(), set()

        def dfs(course):
            if course in cycle:
                return False
            if course in visited:
                return True

            cycle.add(course)
            for prereq in adj_lst[course]:
                if dfs(prereq) == False:
                    return False
            cycle.remove(course)
            visited.add(course)
            output.append(course)
            return True

        for c in range(numCourses):
            if dfs(c) == False:
                return []
        return output
```

```
adj_lst = {      adj_lst = {      Example 1:
    3:[],            3:[1,2],     input:
    2:[],            2:[0],           numCourses = 4
    1:[],            1:[0],           prerequisites = [[1,0],[2,0],[3,1],[3,2]]
    0:[]             0:[]         output:
    }                }                return True
```



Rendundant Connection (LC 684)

684. Redundant Connection
Medium

In this problem, a tree is an undirected graph that is connected and has no cycles.

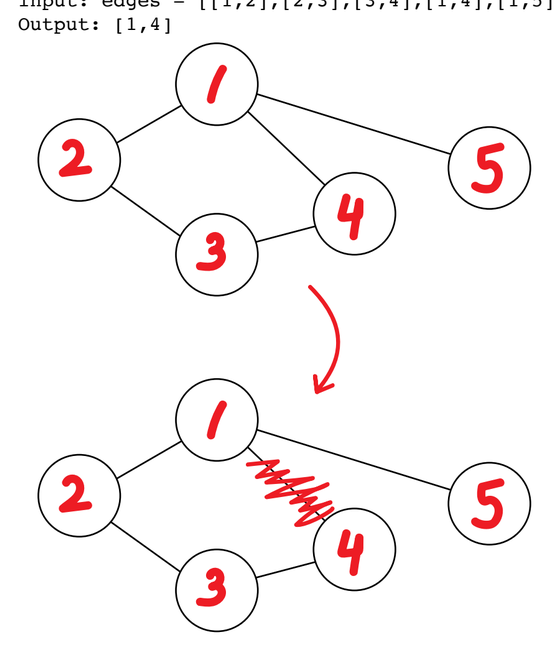
You are given a graph that started as a tree with n nodes labeled from 1 to n, with one additional edge added. The added edge has two different vertices chosen from 1 to n, and was not an edge that already existed. The graph is represented as an array edges of length n where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the graph.

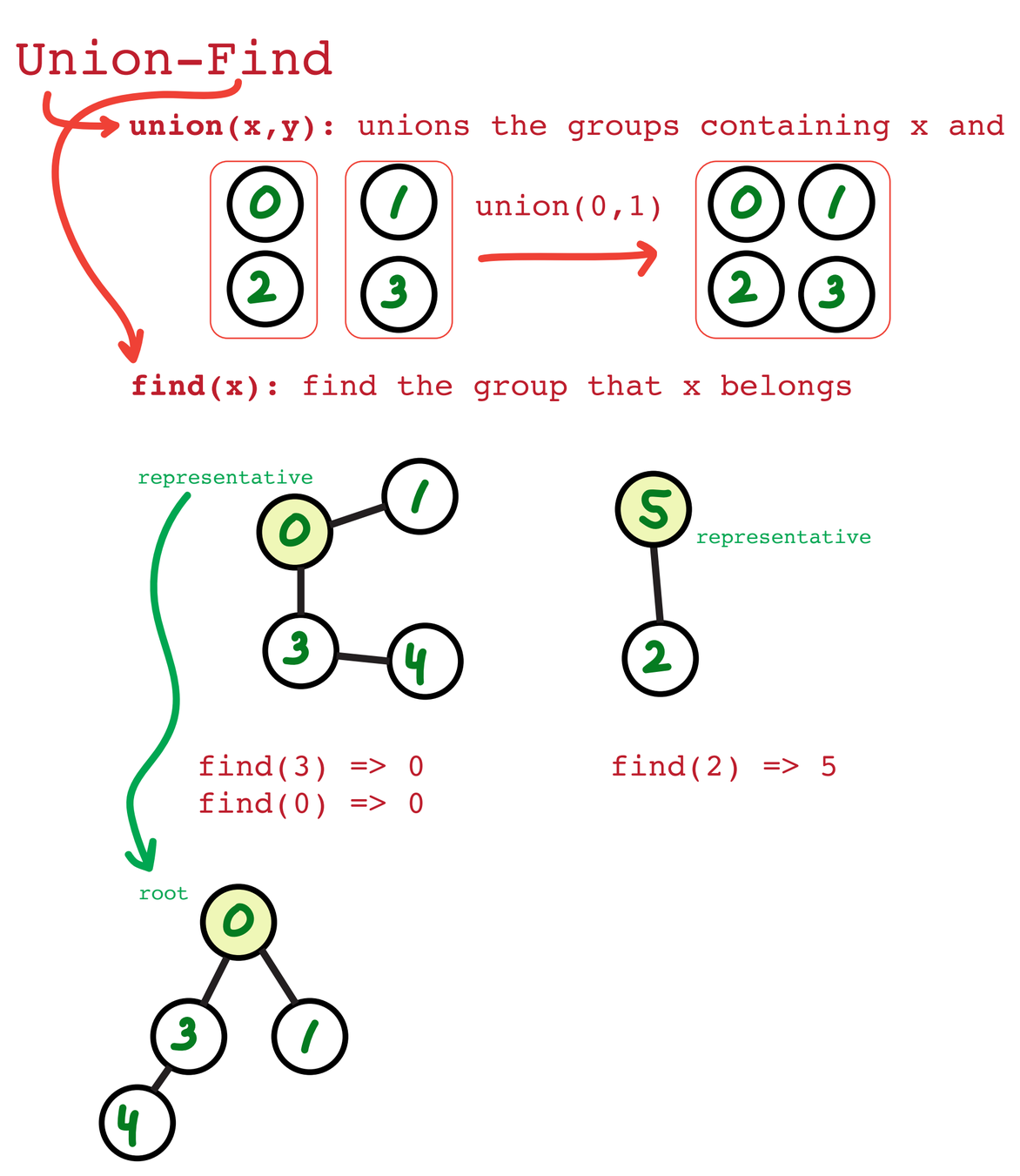
Return an edge that can be removed so that the resulting graph is a tree of n nodes. If there are multiple answers, return the answer that occurs last in the input.

```
class Solution:
    def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:
        parent = [i for i in range(len(edges) + 1)]
        rank = [1] * (len(edges) + 1)

        def find(n):
            p = parent[n]
            while p != parent[p]:
                parent[p] = parent[parent[p]]
                p = parent[p]
            return p

        # return False if already unioned
        def union(n1, n2):
            p1, p2 = find(n1), find(n2)

            if p1 == p2:
                return False
            if rank[p1] > rank[p2]:
                parent[p2] = p1
                rank[p1] += rank[p2]
            else:
                parent[p1] = p2
                rank[p2] += rank[p1]
            return True

        for n1, n2 in edges:
            if not union(n1, n2):
                return [n1, n2]
```

```
parent = [0, 1, 2, 3, 4, 5]
rank = [1, 1, 1, 1, 1, 1]
```

```
Example 1:

Input: edges = [[1,2],[2,3],[3,4],[1,4],[1,5]]
Output: [1,4]
```





Word Ladder (LC 127)

127. Word Ladder
Hard

BFS

A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

- Every adjacent pair of words differs by a single letter.
- Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
- sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return the number of words in the shortest transformation sequence from beginWord to endWord, or 0 if no such sequence exists.

```
Example 1:

Input: beginWord = "hit", endWord = "cog", wordList =
["hot","dot","dog","lot","log","cog"]
Output: 5
Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot"
-> "dog" -> cog", which is 5 words long.

Example 2:

Input: beginWord = "hit", endWord = "cog", wordList =
["hot","dot","dog","lot","log"]
Output: 0
Explanation: The endWord "cog" is not in wordList, therefore there is no
valid transformation sequence.
```

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:

import collections
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        if endWord not in wordList:
            return 0

        lst_of_neighbors = collections.defaultdict(list)

        wordList.append(beginWord)

        for word in wordList:
            for j in range(len(word)):
                pattern = word[:j] + "*" + word[j + 1 :]
                lst_of_neighbors[pattern].append(word)

        visited = set([beginWord])
        q = deque([beginWord])
        counter = 1

        while q:
            for i in range(len(q)):
                word = q.popleft()
                if word == endWord:
                    return counter
                for j in range(len(word)):
                    pattern = word[:j] + "*" + word[j + 1 :]
                    for neighboring_word in lst_of_neighbors[pattern]:
                        if neighboring_word not in visited:
                            visited.add(neighboring_word)
                            q.append(neighboring_word)

            counter += 1
        return 0
```

if popped word is equal to end word, then the algorithm ends return counter

create pattern

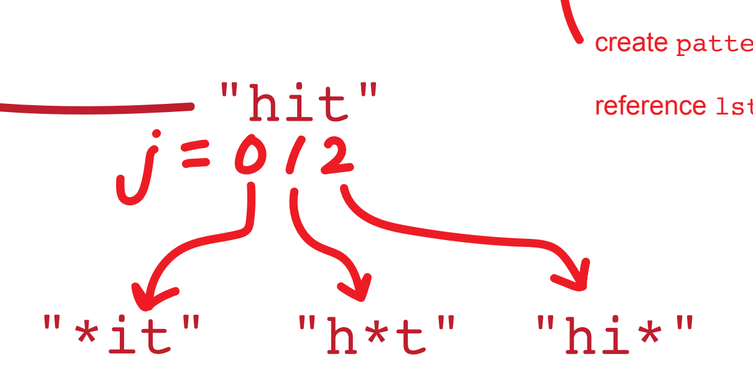
reference lst_of_neighbors[pattern]

```
lst_of_neighbors = {
    '*ot': ['hot', 'dot', 'lot'],
    'h*t': ['hot', 'hit'],
    'ho*': ['hot'],
    'd*t': ['dot'],
    'do*': ['dot', 'dog'],
    '*og': ['dog', 'log', 'cog'],
    'd*g': ['dog'],
    'l*t': ['lot'],
    'lo*': ['lot', 'log'],
    'l*g': ['log'],
    'c*g': ['cog'],
    'co*': ['cog'],
    '*it': ['hit'],
    'hi*': ['hit']
  }
```

[1] Create an adjacency list of possible neighbors patterns

[2] visited -> stores visited combinations
q -> queue for permutations
counter -> result

[3] bfs algorithm



lst_of_neighbors["*it"]

lst_of_neighbors["h*t"]

lst_of_neighbors["hi*"]