BFS, DFS 算法总结

BFS, DFS 作为算法题中一种常见题型，其解题方式相对固定，但其运算思想很巧妙，先总结与此。

LeetCode 207. Course Schedule

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

Example 1:

```
Input: 2, [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0. So it is
possible.
```

Example 2:

```
Input: 2, [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0, and to take
course 0 you should also have finished course 1. So it is impossible.
```

Note:

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

思路1：**BFS**

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) ->
bool:
        indegrees = [0 for _ in range(numCourses)]
        adjacency = [[] for _ in range(numCourses)]
        queue = []

        for cur, pre in prerequisites:
            indegrees[cur] += 1
            adjacency[pre].append(cur)

        queue = [node for node in range(len(indegrees)) if not indegrees[i]]
        while queue:
            pre = queue.pop(0)
            numCourses -= 1
            for cur in adjacency[pre]:
                indegrees[cur] -= 1
                if not indegrees[cur]:
                    queue.append(cur)
        return not numCourses
```

思路2： **DFS**

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) ->
bool:
        def dfs(i, adjacency, flags):
            if flags[i] == -1:
                return True
            if flags[i] == 1:
                return False
            flags[i] = 1
            for j in adjacency[i]:
                if not dfs(j, adjacency, flags):
                    return False
            flags[i] = -1
            return True


        adjacency = [[] for _ in range(numCourses)]
        flags = [0 for _ in range(numCourses)]
        for cur,pre in prerequisites:
            adjacency[pre].append(cur)
        for i in range(numCourses):
            if not dfs(i, adjacency, flags):
                return False
        return True
```

思路3：**DFS**

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) ->
bool:
        res = []
        graph = [[] for _ in range(numCourses)]
        for cur_node, pre_node in prerequisites:
            graph[pre_node].append(cur_node)

        visited = set()
        def dfs_v2(node, being_visited):
            if node in being_visited:
                return False
            if node in visited:
                return True
            being_visited.add(node)
            for nxt_node in graph[node]:
                if not dfs_v2(nxt_node, being_visited):
                    return False
            being_visited.remove(node)
            visited.add(node)
            res.append(node)
            return True
```

LeetCode 210. Course Schedule II

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

Example 1:

```
Input: 2, [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should
have finished
             course 0. So the correct course order is [0,1] .
```

Example 2:

```
Input: 4, [[1,0],[2,0],[3,1],[3,2]]
Output: [0,1,2,3] or [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should
have finished both
             courses 1 and 2. Both courses 1 and 2 should be taken after you
finished course 0.
             So one correct course order is [0,1,2,3]. Another correct ordering
is [0,2,1,3] .
```

Note:

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

思路1：**BFS**

```
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) ->
List[int]:
        graph = collections.defaultdict(list)
        degree = [0] * numCourses
        for cur_cls, pre_cls in prerequisites:
            graph[pre_cls].append(cur_cls)
            degree[cur_cls] += 1
        res = []
        queue = [node for node in range(numCourses) if degree[node] == 0]
        for node in queue:
            res.append(node)
            for nxt_node in graph[node]:
                degree[nxt_node] -= 1
                if degree[nxt_node] == 0:
                    queue.append(nxt_node)
        return res if len(res) == numCourses else []
```

思路2：**DFS**

```
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) ->
List[int]:
        res = []

        graph = collections.defaultdict(list)
        for cur_cls, pre_cls in prerequisites:
            graph[pre_cls].append(cur_cls)
        mark = [0] * numCourses

        def dfs_v2(node, mark):
            if mark[node] == 1:
                return False
            if mark[node] == -1:
                return True
            mark[node] = 1

            for nxt_node in graph[node]:
                if not dfs_v2(nxt_node, mark):
                    return False
            mark[node] = -1
            res.append(node)
            return True

        for node in range(numCourses):
            if not dfs_v2(node, mark):
                return []
        return res[::-1]
```

思路3：**DFS**

```
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) ->
List[int]:
        res = []

        graph = collections.defaultdict(list)
        for cur_cls, pre_cls in prerequisites:
            graph[pre_cls].append(cur_cls)
        visited = set()

        def dfs(node, being_visited):
            if node in being_visited:
                return False
            if node in visited:
                return True
            visited.add(node)
            being_visited.add(node)
            for nxt_node in graph[node]:
                if not dfs(nxt_node, being_visited):
                    return False
            being_visited.remove(node)
            res.append(node)
            return True

        for node in range(numCourses):
            if not dfs(node, set()):
                return []
        return res[::-1]
```

LeetCode 127. Word Ladder

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time.
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume *beginWord* and *endWord* are non-empty and are not the same.

Example 1:

```
Input:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

Output: 5

Explanation: As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog"
-> "cog",
return its length 5.
```

Example 2:

```
Input:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

Output: 0

Explanation: The endWord "cog" is not in wordList, therefore no possible
transformation.
```

思路**1**：**BFS**

```python
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) ->
int:
        if beginWord == endWord or endWord not in wordList or not beginWord or
not endWord or not wordList:
            return 0
        length = len(beginWord)
        word_dict = collections.defaultdict(list)
        for word in wordList:
            for i in range(length):
                word_dict[word[:i] + "*" + word[i+1:]].append(word)

        queue = [(beginWord, 1)]
        visited = {beginWord: True}
        while queue:
            cur_word, cur_level = queue.pop(0)
            for i in range(length):
                temp_word = cur_word[:i] + "*" + cur_word[i+1:]
                for word in word_dict[temp_word]:
                    if word == endWord:
                        return cur_level + 1
                    if word not in visited:
                        visited[word] = True
                        queue.append((word, cur_level + 1))
                word_dict[temp_word] = []
        return 0
```

LeetCode 130. Surrounded Regions

Given a 2D board containing `'X'` and `'O'` (**the letter O**), capture all regions surrounded by `'X'`.

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

Example:

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

Explanation:

Surrounded regions shouldn't be on the border, which means that any `'O'` on the border of the board are not flipped to `'X'`. Any `'O'` that is not on the border and it is not connected to an `'O'` on the border will be flipped to `'X'`. Two cells are connected if they are adjacent cells connected horizontally or vertically.

思路**1**：**BFS**

```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        if not board or not board[0]:
            return
        rows = len(board)
        cols = len(board[0])

        def dfs(i, j):
            board[i][j] = "B"
            for x,y in [(-1, 0), (1, 0), (0, 1), (0, -1)]:
                temp_i = i + x
                temp_j = j + y
                if 0 <= temp_i < rows and 0 <= temp_j < cols and board[temp_i]
[temp_j] == "O":
                    dfs(temp_i, temp_j)


        def bfs(i, j):
            queue = []
            queue.append((i, j))
            while queue:
                temp_i, temp_j = queue.pop(0)
                if 0 <= temp_i < rows and 0 <= temp_j < cols and board[temp_i]
[temp_j] == "O":
                    board[temp_i][temp_j] = "B"
                    for x, y in [(-1, 0), (1, 0), (0, 1), (0, -1)]:
                        queue.append((temp_i + x, temp_j + y))


        for j in range(cols):
            if board[0][j] == "O":
                bfs(0, j)
            if board[rows - 1][j] == "O":
                bfs(rows - 1, j)

        for i in range(rows):
            if board[i][0] == "O":
                bfs(i, 0)
            if board[i][cols-1] == "O":
                bfs(i, cols - 1)

        for i in range(rows):
            for j in range(cols):
                if board[i][j] == "O":
                    board[i][j] = "X"
                if board[i][j] == "B":
                    board[i][j] = "O"
```

思路**2**：**DFS**

```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        if not board or not board[0]:
            return
        rows = len(board)
        cols = len(board[0])

        def dfs(i, j):
            board[i][j] = "B"
            for x,y in [(-1, 0), (1, 0), (0, 1), (0, -1)]:
                temp_i = i + x
                temp_j = j + y
                if 0 <= temp_i < rows and 0 <= temp_j < cols and board[temp_i]
[temp_j] == "O":
                    dfs(temp_i, temp_j)

        for j in range(cols):
            if board[0][j] == "O":
                bfs(0, j)
            if board[rows - 1][j] == "O":
                bfs(rows - 1, j)

        for i in range(rows):
            if board[i][0] == "O":
                dfs(i, 0)
            if board[i][cols-1] == "O":
                dfs(i, cols - 1)

        for i in range(rows):
            for j in range(cols):
                if board[i][j] == "O":
                    board[i][j] = "X"
                if board[i][j] == "B":
                    board[i][j] = "O"
```

LeetCode 102. Binary Tree Level Order Traversal

Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).

For example: Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

思路1：BFS

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        res = []
        if not root:
            return res

        cur_level, ans = [root], []
        while cur_level:
            res.append([node.val for node in cur_level])
            cur_level = [kid for node in cur_level for kid in (node.left,
node.right) if kid]
        return res
```

LeetCode 103. Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example: Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

思路1： **BFS**

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
        res = []
        if not root:
            return res
        layer_nodes, cur = [root], None
        layer_count = 1
        while layer_nodes:
            if layer_count % 2 == 1:
                res.append([node.val for node in layer_nodes])
            else:
                res.append([node.val for node in layer_nodes[::-1]])
            layer_count += 1
            layer_nodes = [kid for node in layer_nodes for kid in (node.left,
node.right) if kid]
        return res
```

LeetCode 104. Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Note: A leaf is a node with no children.

Example:

Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its depth = 3.

思路**1**：**BFS**

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        # if not root:
        #     return 0
        # return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1
        if not root:
            return 0

        layer_count = 0
        layer_nodes = [root]
        while layer_nodes:
            layer_count += 1
            layer_nodes = [kid for node in layer_nodes for kid in [node.left, 
node.right] if kid]
        return layer_count
```

LeetCode 107. Binary Tree Level Order Traversal II

Given a binary tree, return the *bottom-up level order* traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example: Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its bottom-up level order traversal as:

```
[
  [15,7],
  [9,20],
  [3]
]
```

思路**1：BFS**

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def levelOrderBottom(self, root: TreeNode) -> List[List[int]]:
        res = []
        if not root:
            return res
        layer_nodes = [root]
        while layer_nodes:
            res.append([node.val for node in layer_nodes])
            layer_nodes = [kid for node in layer_nodes for kid in [node.left, 
node.right] if kid]
        return res[::-1]
```

LeetCode 199. Binary Tree Right Side View

Given a binary tree, imagine yourself standing on the *right* side of it, return the values of the nodes you can see ordered from top to bottom.

Example:

```
Input: [1,2,3,null,5,null,4]
Output: [1, 3, 4]
Explanation:

   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
```

思路1：**BFS**

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def rightSideView(self, root: TreeNode) -> List[int]:
        res = []
        if not root:
            return res
        # queue = []
        # queue.append(root)
        layer_nodes = [root]
        res = []
        while layer_nodes:
            res.append(layer_nodes[-1].val)
            layer_nodes = [kid for node in layer_nodes for kid in [node.left,
node.right] if kid]
        return res
```

LeetCode 310. Minimum Height Trees

For an undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Format The graph contains `n` nodes which are labeled from `0` to `n - 1`. You will be given the number `n` and a list of undirected `edges` (each edge is a pair of labels).

You can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]` and thus will not appear together in `edges`.

Example 1 :

```
Input: n = 4, edges = [[1, 0], [1, 2], [1, 3]]

        0
        |
        1
       / \
      2   3

Output: [1]
```

Example 2 :

```
Input: n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]

     0  1  2
      \ | /
        3
        |
        4
        |
        5

Output: [3, 4]
```

Note:

- According to the definition of tree on Wikipedia: “a tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree.”
- The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

思路**1**：**BFS**

```
class Solution:
    def findMinHeightTrees(self, n: int, edges: List[List[int]]) -> List[int]:
        if n == 1:
            return [0]
        adj = [[] for i in range(n)]
        for i, j in edges:
            adj[i].append(j)
            adj[j].append(i)

        leaves = []
        for i in range(n):
            if len(adj[i]) == 1:
                leaves.append(i)

        while n > 2:
            n -= len(leaves)
            new_leaves = []
            for node in leaves:
                adj_node = adj[node][0]
                adj[adj_node].remove(node)
                if len(adj[adj_node]) == 1:
                    new_leaves.append(adj_node)
            leaves = new_leaves

        return leaves
```

LeetCode 322. Coin Change

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

Example 1:

```
Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

Example 2:

```
Input: coins = [2], amount = 3
Output: -1
```

思路**1**:

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        if not coins or amount < 0:
            return -1
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0
        for coin in coins:
            for i in range(coin, amount + 1):
                dp[i] = min(dp[i - coin] + 1, dp[i])
        return dp[amount] if dp[amount] != amount + 1 else -1
```