Problem 1: Geography Problem

Description of problems:

You have a set of words. A list of words in legal if the last letter of each word is the same as the first letter of the following word. The problem is to find a legal list of words using all the words exactly once. Assume that initially the first word is there, and you just need to complete it.

The components for solving the problem:

1. **State:**

   In code, the state represents the current word in the word chain. It's a word from the list of words, and the goal is to form a chain of words where each word's last letter matches the first letter of the following word.

2. **Initial State:**

   The initial state is the starting word for list of word chain. It's typically the first word in list of words.

3. **Goal Test:**

   The goal test checks if a valid word chain has been formed, i.e., if the length of the word chain is equal to the total number of words in the list.

4. **Successor Function:**

   The successor function generates the next possible states (words) that can follow the current word in the word chain. In your code, it looks for words that start with the last letter of the current word.

Code

Now, let's discuss the output and the order in which nodes (states) are visited during the search.

This order depends on the specifics of the search algorithm (in this case, depth-first search).

```python
def find_word_chain(original_list_of_words):
    # Depth-first search function to find a word chain
    def solve_geograpgy_problem_using_dfs(word, recursion_depth, steps):
        if len(word_chain) == len(original_list_of_words):
            return word_chain

        last_letter = word[-1]
        match_list = graph_of_first_letter.get(last_letter)
        # If there are no words starting with the last letter, return None
        if match_list is None:
            return None
        recursion_depth += 1


        for next_word in match_list:
            if next_word not in used_words:
                word_chain.append(next_word)
                used_words.add(next_word)

                steps += 1
                # Print the state (word) as it is visited
                print("Recursion depth: {depth}, steps: {steps} and visiting {word}".format
                    depth=recursion_depth,
                    steps=steps,
                    word=next_word,
                ))

                # Recursively search for the next word in the chain
                result = solve_geograpgy_problem_using_dfs(next_word, recursion_depth, steps
                if result:
                    return result

                # Backtrack by removing the current word
                word_chain.pop()
                used_words.remove(next_word)

    # Create a graph_of_first_letter where each word is a node with outgoing edges to words
    graph_of_first_letter = {}
    for word in original_list_of_words:
        if word[0] not in graph_of_first_letter:
            graph_of_first_letter[word[0]] = []
        graph_of_first_letter[word[0]].append(word)

    initial_word = original_list_of_words[0]
    used_words = set([initial_word])
    word_chain = [initial_word]

    recursion_depth = 0
    steps = 0

    return solve_geograpgy_problem_using_dfs(initial_word, recursion_depth, steps)

# Example list_of_words to form a word chain
list_of_words = ["ABC", "CDE", "CFG", "EHE", "EIJ", "GHK", "GLC"]
# list_of_words = ["apple", "lion", "nut", "elephant", "tiger", 'redpoll']
```

```python
# Find and print the word sequence
sequence_of_chain_words = find_word_chain(list_of_words)

if sequence_of_chain_words:
    sequence_of_chain_words = ' -> '.join(sequence_of_chain_words)
    print(sequence_of_chain_words)
else:
    print("No valid word sequence found.")
```

Explanation of code

Here's explaination of the code for finding a word chain from a list of words using depth-first search (DFS):

Explanation:

1. `solve_geography_problem_using_dfs` is a depth-first search (DFS) function used to find a word chain.
   - It takes the current `word`, `recursion_depth`, and `steps` as input.
   - It checks if the length of the `word_chain` is equal to the length of `original_list_of_words`. If they match, a valid word sequence is found, and it returns the `word_chain`.
   - It identifies the `last_letter` of the current word and looks for words in `match_list` that start with this last letter. If none are found, it returns None.
   - It iterates through words in `match_list` and recursively explores them, adding them to the `word_chain` and marking them as used.
   - It prints the recursion depth and steps taken as it visits words.
   - If a valid sequence is found during recursion, it returns the result. Otherwise, it backtracks by removing the last added word from the `word_chain`.
2. The `graph_of_first_letter` dictionary is created to represent the graph of words where each word is a node, and there are edges from words ending with the same letter to others.
3. The initial state is set to the first word in the list, and it's marked as used in the `used_words` set.
4. The code initiates the search by calling `solve_geography_problem_using_dfs` with the initial state.
5. Finally, the code prints the resulting word sequence if found, or indicates that no valid sequence was found.

This code uses DFS to systematically explore possible word sequences and backtracks when needed to find a valid word chain. The recursion depth and steps are printed for each visited word to track the progress of the search.

Output:

Output from the Program:

```
list_of_words = ["apple", "lion", "nut", "elephant", "tiger", 'redpoll']

Recursion depth: 1, steps: 1 and visiting elephant
Recursion depth: 2, steps: 2 and visiting tiger
Recursion depth: 3, steps: 3 and visiting redpoll
Recursion depth: 4, steps: 4 and visiting lion
Recursion depth: 5, steps: 5 and visiting nut
apple -> elephant -> tiger -> redpoll -> lion -> nut
```

```
list_of_words = ["ABC", "CDE", "CFG", "EHE", "EIJ", "GHK", "GLC"]

Recursion depth: 1, steps: 1 and visiting CDE
Recursion depth: 2, steps: 2 and visiting EHE
Recursion depth: 3, steps: 3 and visiting EIJ
Recursion depth: 2, steps: 3 and visiting EIJ
Recursion depth: 1, steps: 2 and visiting CFG
Recursion depth: 2, steps: 3 and visiting GHK
Recursion depth: 2, steps: 4 and visiting GLC
Recursion depth: 3, steps: 5 and visiting CDE
Recursion depth: 4, steps: 6 and visiting EHE
Recursion depth: 5, steps: 7 and visiting EIJ
Recursion depth: 4, steps: 7 and visiting EIJ
No valid word sequence found.
```

Problem 2: PCP Problem

Description of problems:

You have a set of dominoes. Each one contains a string on the top and a string on the bottom. The problem is to find a sequence of dominoes such that the concatenation of the strings on the top gives the same string as the concatenation of all the strings on the bottom. Dominoes may be used as many times as you want. Assume that the first domino is there.

let's define the components for solving the problem:

1. **State:**

   In this problem, a state represents the current sequence of dominoes that you have placed, which forms a chain. The state changes as you add more dominoes to the sequence.

2. **Initial State:** The initial state represents the starting point of the sequence. It includes the possible matching dominoes.

3. **Goal Test:**

   The goal test checks whether the current sequence of dominoes forms a chain where the concatenation of the top strings matches the concatenation of the bottom strings. If the top and bottom strings of the entire chain are the same, the goal test is satisfied, indicating that you have successfully solved the problem.

4. **Successor Function:** The successor function generates the next possible states by considering which dominoes can be added to the current sequence. For each state, the successor function provides a list of valid next states based on the dominoes that can be added to the chain. These valid next states represent the different possible extensions of the current chain.

In summary, this problem involves finding a sequence of dominoes, starting with the initial state, where the top strings of the dominoes form the same string as the concatenation of all the bottom strings. The successor function helps explore possible sequences, and the goal test checks if the current sequence satisfies the problem's condition.

Code

Now, let's discuss the output and the order in which nodes (states) are visited during the search.

This order depends on the specifics of the search algorithm (in this case, depth-first search).

```python
def match_top_bottom_domino(dominos):

    def solve_pcp_problem_using_dfs(all_possible_paths, recursion_depth, steps):
        # Base case: If there are no more paths to explore, return None
        if not all_possible_paths:
            return None

        new_possible_paths = []
        for path in all_possible_paths:
            for domino in dominos:
                new_path = path.copy()
                new_path.append(domino)

                steps += 1

                # Concatenate top and bottom of the dominoes in the path
                top_domino = "".join([d[0] for d in new_path])
                bottom_domino = "".join([d[1] for d in new_path])

                # Check if top and bottom are equal
                if top_domino == bottom_domino:
                    return [top_domino, bottom_domino]

                # Check for partial match and add to new_possible_paths
                min_len = min(len(top_domino), len(bottom_domino))
                if top_domino[:min_len] == bottom_domino[:min_len]:
                    new_possible_paths.append(new_path)

        recursion_depth += 1
        print("Recursion depth: {depth}, paths: {paths}, steps: {steps}".format(
            depth=recursion_depth,
            paths=len(all_possible_paths),
            steps=steps
        ))
        # Continue the search with new_possible_paths
        return solve_pcp_problem_using_dfs(new_possible_paths, recursion_depth, steps)

    recursion_depth = 0
    steps = 0

    # Initialize all_possible_paths with dominoes having an initial partial match
    all_possible_paths = []
    for domino in dominos:
        top_domino, bottom_domino = domino
        min_len = min(len(top_domino), len(bottom_domino))

        if top_domino[:min_len] == bottom_domino[:min_len]:
            all_possible_paths.append([domino,])

    # Start solving the PCP problem using depth-first search
    solution = solve_pcp_problem_using_dfs(all_possible_paths, recursion_depth, steps)

    return solution

# Example dominoes
dominoes = [
    ("bba", "b"),
    ("ba", "baa"),
    ("ba", "aba"),
    ("ab", "bba"),
]

# Find a solution for the PCP problem
solution = match_top_bottom_domino(dominoes)

if solution:
    print(f"Solution found: {solution[0]} domino top is equal to {solution[1]} domino bottor
else:
    print("No solution found.")
```

Explanation of code

Here's an explanation of the provided Python code for solving the Post Correspondence Problem (PCP):

1. `match_top_bottom_domino` Function:
    - This is the main function that takes a list of dominoes as input and attempts to find a sequence of dominoes that match the top and bottom strings.
2. `solve_pcp_problem_using_dfs` Function:
    - This function is a recursive depth-first search (DFS) function that explores all possible sequences of dominoes to solve the PCP.
    - It takes three parameters:
        - `all_possible_paths`: A list of lists, where each inner list represents a sequence of dominoes to be explored.
        - `recursion_depth`: An integer that keeps track of the depth of recursion.
        - `steps`: An integer that counts the number of steps taken in the search.
3. Base Case:
    - If `all_possible_paths` is empty (i.e., no more paths to explore), the function returns `None`, indicating that no solution was found.
4. Iterating Through Paths:
    - The function iterates through each path in `all_possible_paths`.
    - For each path, it tries to append each domino from the provided `dominos` list.
5. Combining Dominoes:
    - It concatenates the top and bottom strings of the dominoes in the path to form `top_domino` and `bottom_domino`.
6. Matching Conditions:
    - It checks two conditions:
        - If `top_domino` is equal to `bottom_domino`, it means a valid sequence is found, and it returns the matching strings.
        - If there is a partial match (the first characters of `top_domino` and `bottom_domino` match), it adds the new path to `new_possible_paths` for further exploration.
7. Recursion Depth and Steps:
    - The `recursion_depth` and `steps` variables are updated to keep track of the depth of recursion and the number of steps taken.
8. Printing Progress:
    - It prints information about the current recursion depth, the number of paths, and the number of steps taken during each recursive call.

9. Recursive Call:
    - The function makes a recursive call with `new_possible_paths` to continue the search.
10. Initialization and Start:
    - The `recursion_depth` and `steps` variables are initialized to 0.
    - An initial set of `all_possible_paths` is created based on dominoes with initial partial matches.
11. Starting the Search:
    - The PCP problem-solving process starts by calling `solve_pcp_problem_using_dfs` with the initial parameters.
12. Solution or No Solution:
    - If a solution is found, it prints the matching strings and their size.
    - If no solution is found, it indicates that no solution exists.

Overall, this code implements a depth-first search to systematically explore all possible sequences of dominoes to solve the PCP problem. It reports progress along the way and returns a solution if one is found.

Output of code:

Output from the Program:

```
dominoes = [
    ("bba", "b"),
    ("ba", "baa"),
    ("ba", "aba"),
    ("ab", "bba"),
]

...
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Visiting: ('ab', 'bba')
```

```
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Recursion depth: 1, paths: 2, steps: 8
...
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Recursion depth: 64, paths: 6394, steps: 237572
...
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')
Visiting: ('ab', 'bba')
Visiting: ('bba', 'b')
Visiting: ('ba', 'baa')
Visiting: ('ba', 'aba')

Solution found: baabbaababbabbabaabbaabbaababbaababbabbaababbabbabaabbbabbabaababababbabbababb
```

```
dominoes = [
    ("MOM", "M"),
    ("O", "OMOMO"),
]
```

```
Visiting: ('MOM', 'M')
```

```
Visiting: ('O', 'OMOMO')
Visiting: ('MOM', 'M')
Visiting: ('O', 'OMOMO')
Recursion depth: 1, paths: 2, steps: 4
Visiting: ('MOM', 'M')

Solution found: MOMOMOM domino top is equal to MOMOMOM domino bottom with a size of 7 each.
```