1.WORD BREAK PROBLEM

```
def word_break(s, word_dict):
    word_set = set(word_dict)
    n = len(s)

    # dp[i] is True if s[0:i] can be segmented into words in the word_dict
    dp = [False] * (n + 1)
    dp[0] = True  # Base case: empty string

    for i in range(1, n + 1):
        for j in range(i):
            if dp[j] and s[j:i] in word_set:
                dp[i] = True
                break

    return dp[n]

s = "leetcode"
word_dict = ["leet", "code"]
print(word_break(s, word_dict))  # Output: True

s = "applepenapple"
word_dict = ["apple", "pen"]
print(word_break(s, word_dict))  # Output: True

s = "catsandog"
word_dict = ["cats", "dog", "sand", "and", "cat"]
print(word_break(s, word_dict))
```

OUTPUT: False

2.WORD TRAP PROBLEM

```
def word_trap(s, word_dict):
```

```
    word_set = set(word_dict)
    memo = {}

    def helper(sub_s):
        if sub_s in memo:
            return memo[sub_s]
        if not sub_s:
            return [[]]

        res = []
        for end in range(1, len(sub_s) + 1):
            word = sub_s[:end]
            if word in word_set:
                for r in helper(sub_s[end:]):
                    res.append([word] + r)

        memo[sub_s] = res
        return res

    result = helper(s)
    return [" ".join(words) for words in result]

s = "catsanddog"
word_dict = ["cat", "cats", "and", "sand", "dog"]
print(word_trap(s, word_dict))
# Output: ['cats and dog', 'cat sand dog']

s = "pineapplepenapple"
word_dict = ["apple", "pen", "applepen", "pine", "pineapple"]
print(word_trap(s, word_dict))
```

OUTPUT: ['pine apple pen apple', 'pineapple pen apple', 'pine applepen apple']

```
s = "catsandog"
word_dict = ["cats", "dog", "sand", "and", "cat"]
print(word_trap(s, word_dict))
```

OUTPUT: []

3.OBST

```
def optimal_bst(keys, freq, n):
    cost = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        cost[i][i] = freq[i]

    for length in range(2, n + 1):  # length of the chain
        for i in range(n - length + 1):
            j = i + length - 1
            cost[i][j] = float('inf')

            total_freq = sum(freq[i:j + 1])

            for r in range(i, j + 1):
                left_cost = cost[i][r - 1] if r > i else 0
                right_cost = cost[r + 1][j] if r < j else 0
                total_cost = left_cost + right_cost + total_freq
                if total_cost < cost[i][j]:
                    cost[i][j] = total_cost

    return cost[0][n - 1]
keys = [10, 12, 20]
freq = [34, 8, 50]
n = len(keys)
print("Cost of Optimal BST is", optimal_bst(keys, freq, n))
```

4.FLOYD ALGORITHM

```
def floyd_warshall(graph):
```

```
    """
    Floyd-Warshall algorithm to find the shortest path between all pairs of nodes.

    :param graph: 2D list representing the adjacency matrix of the graph
                  where graph[i][j] is the weight of the edge from i to j
                  or float('inf') if there is no edge.
    :return: 2D list representing the shortest distances between all pairs of nodes
    """
    # Number of vertices in the graph
    V = len(graph)

    dist = [[graph[i][j] for j in range(V)] for i in range(V)]

    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

inf = float('inf')
graph = [
    [0, 3, inf, inf],
    [2, 0, inf, inf],
    [inf, 7, 0, 1],
    [6, inf, inf, 0]
]

shortest_paths = floyd_warshall(graph)
for row in shortest_paths:
```

```
print(row)
```