Documentation

1. Problem Statement

In the **Frog Jump** problem, the goal is to determine if a frog can cross a river by landing on stones placed at specific positions. The frog starts at the first stone and must make jumps according to specific rules. Initially, the frog's first jump must be exactly one unit. If the frog's last jump was k units, then the next jump must be either k-1, k, or k+1 units. The frog can only move forward, and it must land on stones (not water). The task is to determine whether it is possible for the frog to reach the last stone in the series of stones provided.

2. Intuition

The frog can attempt various jumps from each stone, but not all jumps are valid due to the stone positions. By maintaining a record of which jumps can lead to which stones, we can use a dynamic programming approach to efficiently calculate whether the frog can reach the last stone. The challenge lies in considering all possible jump combinations while ensuring that the frog only lands on valid stones. Instead of checking all possible paths exhaustively, we aim to track possible jumps at each stone to efficiently determine the solution.

3. Key Observations

Several key observations help simplify the problem:

- The frog can only land on the stones that exist in the given list.
- Each stone can be reached by different jump sizes from the previous stone.
- The frog can jump in increments of k-1, k, or k+1, where k is the previous jump size. This means the frog has three options for every jump (if valid).
- The gap between two stones must be small enough for the frog to make a jump to the next stone. If the gap is too large, the frog cannot continue.
- Using dynamic programming (DP), we can maintain a table of stones and track the possible jump sizes that can land on each stone.

4. Approach

The approach involves using a dictionary (dp) where each key represents a stone, and the corresponding value is a set of possible jump sizes that can reach that stone. Initially, at stone 0, the frog can make a jump of size 0. For each subsequent stone, we check the possible jumps (i.e., k-1, k, and k+1) and see if any of them lead to a valid next stone. If so, we store these jump sizes in the DP table for that stone. Finally, if the DP table for the last stone contains any valid jump sizes, it means the frog can reach the last stone.

5. Edge Cases

Several edge cases need to be considered:

- ***Minimum Input Case:*** The smallest input is when there are only two stones (stones = [0,1]), in which the frog can easily jump to the last stone.
- ***Large Jump Gaps:*** If the stones are placed with large gaps (e.g., stones = [0,1,3,6,10]), the frog may not be able to jump across, and the solution should handle these cases appropriately.
- ***Already Sorted Stones:*** Since the problem guarantees that the stones are sorted in ascending order, there is no need to sort them again.
- ***Maximum Constraints:*** When the number of stones is large (up to 2000), the solution must efficiently handle such inputs within the provided time limits.

6. Complexity Analysis

The time complexity of the solution is $\mathbf{O(N^2)}$, where N is the number of stones. This is because for each stone, we potentially check up to three jump sizes, and for each jump size, we check if it leads to a valid stone (which takes constant time due to the hash set lookup). The space complexity is also $\mathbf{O(N^2)}$ in the worst case, as each stone can potentially have up to N jump sizes stored in the DP table.

7. Alternative Approaches

An alternative approach to solving this problem is using **Depth-First Search (DFS)** with memoization. The idea is to recursively try each jump size and memoize the results to avoid recomputation. However, this approach can have an exponential time complexity of $\mathbf{O(3^N)}$ in the worst case, making it inefficient for large inputs. Another approach is **Breadth-First Search (BFS)**, which explores all possible jumps level by level. While BFS can solve the problem, its implementation is more complex compared to dynamic programming and may be slower due to the overhead of managing the queue.

8. Code Implementation

The code implementation uses dynamic programming to solve the problem efficiently. It maintains a DP table where each stone position maps to a set of valid jump sizes. Starting from the first stone, it iterates through each stone and explores all possible jumps that can lead to other stones. At the end, it checks whether there is a valid jump that reaches the last stone.

9. Test Cases

Several test cases should be considered to ensure the solution works for various scenarios:

1. ***Basic Test Case:*** stones = [0,1,3,5,6,8,12,17] → The frog can reach the last stone, so the output should be True.
2. ***Gap Too Large:*** stones = [0,1,2,3,4,8,9,11] → The frog cannot jump past the large gap between stones 4 and 8, so the output should be False.
3. ***Minimum Stones:*** stones = [0,1] → The frog can easily jump from the first stone to the second, so the output should be True.
4. ***Large Stone List:*** A case where the frog must make many jumps across a large list of stones, ensuring that the solution handles the maximum constraints efficiently.

10. Final Thoughts

The dynamic programming approach is the most efficient and optimal solution for the Frog Jump problem. By using a dictionary to track valid jump sizes at each stone, the solution avoids unnecessary computations and can handle large inputs within the time constraints. The complexity analysis shows that the solution is efficient enough for inputs of size up to 2000. Although other approaches such as DFS or BFS are possible, they may be less efficient and harder to implement. Therefore, the dynamic programming solution is the best choice for this problem.