1. Understand Recursive Algorithms:

o Explain the concept of recursion and how it can simplify certain problems.

Concept of Recursion:

Recursion is a technique where a function calls itself to solve a problem. It typically involves solving a smaller instance of the same problem and combining the results to solve the larger problem.

Base Case: The condition under which the recursion terminates. It defines when the function should stop calling itself.

Recursive Case: The condition under which the function calls itself to solve a smaller problem.

Benefits of Recursion:

Simplifies the problem-solving process by breaking a complex problem into simpler subproblems.

Often leads to cleaner and more readable code for problems that have a natural recursive structure (e.g., tree traversal, factorial computation).

Drawbacks:

Can lead to excessive computation and high memory usage if not managed properly, especially if the recursion depth is large.

4. Analysis:

o Discuss the time complexity of your recursive algorithm.

Time Complexity:

Time Complexity: O(n) – The recursive algorithm makes n calls where n is the number of years. Each call performs constant time operations.

Space Complexity: O(n) – The call stack grows with the number of recursive calls, leading to linear space complexity.

o Explain how to optimize the recursive solution to avoid excessive computation.

Optimization:

Memoization: Store previously computed values to avoid redundant calculations. This can reduce the time complexity to O(n) and prevent excessive recomputation.

Iterative Approach: Convert the recursive approach to an iterative one to avoid the overhead of recursive calls and manage memory usage better