**Recursion** is a problem-solving technique where a function calls itself to solve a smaller version of the same problem. This approach can be particularly elegant and efficient for certain problems.

Recursion often provides a more intuitive and concise solution to problems that have a recursive structure. This means the problem can be broken down into smaller, similar subproblems. By solving these subproblems recursively, the original problem can be solved efficiently.

For example, calculating factorials, Fibonacci numbers, and tree traversals

The time complexity of a recursive algorithm depends on the number of recursive calls and the amount of work done in each call. In the provided code, the calculateFutureValue function has a time complexity of O(n), where n is the number of years. This is because it makes n recursive calls

While recursion can be elegant, it's essential to avoid unnecessary computations to prevent performance issues. Here are some optimization techniques:

1. **Memoization:** Store the results of function calls in a data structure (often a hash table) to avoid recalculating the same values. This is particularly effective for problems with overlapping subproblems.
2. **Tail Recursion:** In some languages, compilers can optimize tail-recursive functions into iterative code, improving performance. Tail recursion occurs when the recursive call is the last operation in the function.
3. **Base Case Optimization:** Ensure the base case is reached efficiently to minimize the number of recursive calls.
4. **Avoid Unnecessary Recursion:** Sometimes, iterative solutions might be more efficient than recursive ones. Analyze the problem carefully to determine the best approach.