**1. Difference Between Recursion and Iteration:**

* **Recursion**: A function calls itself to solve a smaller instance of the problem. Each recursive call creates a new stack frame, where local variables and return addresses are stored.
* **Iteration**: A loop repeats a set of instructions until a condition is met. It doesn't create new function calls and is generally more memory-efficient.

**Pros and Cons**:

* **Recursion**: Easier to understand for problems that naturally follow recursive patterns (e.g., tree traversals, Fibonacci numbers). However, it can lead to high memory usage due to the call stack and risks stack overflow if the recursion depth is too large.
* **Iteration**: Usually more memory-efficient since it doesn't involve the overhead of multiple function calls. However, for problems like tree traversal, an iterative approach can be less intuitive.

**2. What is a Stack, and How Does It Support Recursion?**

* A **stack** is a data structure that follows the Last-In-First-Out (LIFO) principle. It holds function calls in a **call stack**, storing return addresses, local variables, and the function state.
* **Recursion relies on the call stack** to manage function calls. Each time a recursive function is called, the current state is saved on the stack until the function reaches its base case and the state is popped off the stack.
* **Stack Overflow** occurs when the call stack exceeds its limit, typically when recursion is too deep or an infinite recursion occurs.

**3. What is the “Call Stack” in Recursion?**

* The **call stack** is used by the system to manage function calls. When a function calls another function, the address to return to (the point after the function call) is saved along with any local variables.
* For recursion, each recursive call creates a new stack frame in the call stack. When a function reaches its base case, it "unwinds," returning values from the stack, thus managing the function execution step by step.

**4. Why Might Recursion Lead to Stack Overflow, and How Can This Be Avoided?**

* **Stack Overflow** happens when the function calls exceed the stack size, which can happen if recursion depth is too deep (e.g., when solving problems without a proper base case or using excessive recursion depth).
* **Prevention Methods**:
  + **Tail Recursion**: When the recursive call is the last action in the function, compilers can optimize it by reusing the current stack frame, preventing the creation of a new frame.
  + **Refactor to Iteration**: Some problems can be solved iteratively, avoiding recursion completely.

**5. What is Tail Recursion and Its Advantages?**

* **Tail Recursion**: A form of recursion where the recursive call is the last action in the function. In this case, the function's current stack frame can be reused by the compiler, thus saving memory.
* **Advantages**: This optimization avoids the overhead of creating a new stack frame, reducing memory usage and preventing stack overflow for deep recursion.

**6. How Does a Stack-Based Approach Help in Converting Recursion to Iteration?**

* Using an **explicit stack** in iterative algorithms simulates the function calls that would otherwise be handled by the system's call stack.
* For example, a **Depth-First Search (DFS)** that uses recursion can be rewritten to use an explicit stack to manage nodes instead of relying on the system's call stack.

**7. Performance Implications of Replacing Recursion with Iteration Using a Stack**

* **Time Complexity**: The time complexity typically remains the same when converting recursion to iteration, as both approaches explore the problem in a similar manner. However, the constant factors might differ.
* **Space Complexity**: Iteration can reduce space complexity, especially for problems with deep recursion, as the stack frames in recursion can consume significant memory. Iteration using a stack may be more memory-efficient.
* **Memory Overhead**: Iteration usually consumes less memory since it doesn't require the overhead of multiple function calls.

**8. Simulating Recursive Function Calls Using an Explicit Stack**

* To simulate recursion using an **explicit stack**:
  + **Push state onto the stack**: Each recursive call’s state, such as local variables and parameters, is stored in the stack.
  + **Process stack**: In the loop, pop the stack and simulate the recursive calls by handling each "state" step by step, just as the recursive function would.

**9. Base Case in Recursion and Its Importance**

* **Base Case**: The condition that stops the recursion. Without a base case, recursion will continue indefinitely, leading to stack overflow.
* The base case ensures that recursion eventually terminates, returning control to the previous recursive calls.