**Understanding- Recursion**

Recursion is when a function calls itself to break a big problem into smaller versions of the same problem. Each call keeps reducing the problem until it reaches something simple enough to solve directly, called the base case. Recursion especially helpful when dealing with problems that naturally have a repetitive or nested structure or when you're dividing tasks into smaller chunks.

**Analysis-**

Time Complexity: O(n)

Space Complexity: O(n)

So, if you're calling a function n times, both time and space will grow linearly.

Even if the time is acceptable, the space used on the call stack can be a problem. Each recursive call takes memory, so for input size n, the space complexity is also often O(n). This can lead to a stack overflow for deep recursion.

How to optimize:

Tail Recursion: This is where the recursive call is the last thing in the function. Some languages (like Python or JavaScript with limits) optimize this to save stack space, but Java doesn’t support it natively.

Iteration: Using loops instead of recursion is usually more efficient and avoids stack overflows entirely. It's often a better choice when performance is key.

Hybrid Solutions:Some problems benefit from combining recursion with iteration. For example, merge sort uses recursion to divide, but merging can be optimized with iterative techniques.

Caching: For recursive problems with overlapping subproblems (like Fibonacci or dynamic programming), storing previously computed results can cut down redundant work.