Mathematical Analysis of Recursive and Non Recursive Algorithms

Fibonacci sequence

Recursive Method

The recursive approach for the Fibonacci sequence is a direct implementation of its mathematical definition: $Fn=Fn-1+Fn-2$, with base cases $F0=0$ and $F1=1$. A function for a given n calls itself to find the values for $n-1$ and $n-2$ and then adds them. This process continues until it reaches the base cases.

Algorithmic Flow

1. **Base Case:** If n is 0 or 1, return n.
2. **Recursive Step:** For any $n>1$, return `fibonacci(n-1) + fibonacci(n-2)`.

Characteristics

- **Simplicity and Readability:** The code is often concise and directly mirrors the mathematical formula.
- **Inefficiency:** This method is highly inefficient due to **redundant calculations**. For example, to compute `fibonacci(5)`, it needs to compute `fibonacci(4)` and `fibonacci(3)`. Both of those calls will, in turn, compute `fibonacci(2)` and `fibonacci(1)`, leading to the same subproblems being solved multiple times. This results in an exponential time complexity, approximately $O(2^n)$.
- **Memory Usage:** The repeated function calls consume a significant amount of memory on the call stack, potentially leading to a **stack overflow** for large values of n.

Non-Recursive (Iterative) Method

The non-recursive approach solves the problem from the bottom up, building the sequence one number at a time using a loop. It starts with the first two terms (0 and 1) and iteratively calculates each subsequent term by adding the previous two.

Algorithmic Flow

1. **Initialization:** Start with two variables, say `a = 0` and `b = 1`, to represent the first two terms.
2. **Loop:** Loop from 2 up to n. In each iteration, calculate the next term `c` as `a + b`. Then, update the variables by setting `a = b` and `b = c` to prepare for the next iteration.
3. **Return:** After the loop finishes, the value of `b` will be the n-th Fibonacci number.

Characteristics

- **Efficiency:** This method is far more efficient. It calculates each term only once, resulting in a linear time complexity, O(n).
- **Memory Usage:** It only requires a constant amount of memory to store the previous two numbers, making it highly memory-efficient, O(1).
- **Less Intuitive:** While more efficient, the iterative code can sometimes be slightly less intuitive to read for someone unfamiliar with the concept, as it doesn't directly map to the mathematical formula.

Feature	Recursive Method	Non-Recursive (Iterative) Method
Approach	Top-down (breaks problem into subproblems)	Bottom-up (builds solution from base cases)
Time Complexity	$O(2^n)$ (Exponential)	O(n) (Linear)
Space Complexity	O(n) (due to call stack)	O(1) (constant)
Readability	High, mirrors mathematical definition	Lower, requires understanding the loop logic
Performance	Very inefficient for large n	Highly efficient and scalable

Merge Sort : Divide & Conquer

The core difference between recursive and non-recursive (iterative) merge sort lies in their approach to splitting the input list. Recursive merge sort uses a top-down, divide-and--conquer strategy, while non-recursive merge sort uses a bottom-up approach. Both algorithms have the same time complexity of O(nlogn), but they differ in their implementation, space usage, and practical considerations.

Recursive Merge Sort

This is the classic implementation of merge sort. It's a top-down approach based on the "divide and conquer" paradigm.

Algorithmic Flow

1. **Divide:** The function takes an array and recursively calls itself to split the array in half until it reaches subarrays of size 1. A single-element array is considered sorted by definition, which serves as the **base case** for the recursion.
2. **Conquer:** Once the base cases are reached, the function begins to return. The returned subarrays are merged in a sorted manner.
3. **Merge:** The merge function takes two sorted subarrays and combines them into a single sorted array. This merging process is the heart of the algorithm. It compares elements from both subarrays and places the smaller one into the new array. This is done repeatedly until all elements are merged.

Analysis

- **Time Complexity:** O(nlogn) because at each level of recursion, the merge operation takes O(n) time to process all elements. The number of levels of recursion is logn.
- **Space Complexity:** O(n) due to the temporary array used in the merge step and the space consumed on the call stack for the recursive function calls.
- **Pros:** The code is generally more concise, elegant, and easier to understand conceptually, as it directly follows the divide-and-conquer logic.
- **Cons:** Can lead to **stack overflow** errors for very large inputs due to deep recursion.

Non-Recursive (Iterative) Merge Sort

This is a bottom-up approach that avoids recursion by using a loop. It starts by merging small, sorted subarrays and progressively builds larger ones.

Algorithmic Flow

1. **Iterative Merge:** Instead of splitting, the algorithm starts by considering adjacent pairs of elements as subarrays of size 1. It merges these pairs to create sorted subarrays of size 2.
2. **Progressive Merging:** It then takes these sorted subarrays of size 2 and merges them to create sorted subarrays of size 4. This process continues, doubling the size of the subarrays to be merged in each pass (i.e., subarrays of size 8, 16, etc.) until the entire array is sorted.
3. **Loop Structure:** The algorithm uses a main loop that controls the size of the subarrays to be merged (e.g., size = 1, 2, 4, 8, ...). Inside this loop, a second loop iterates through the array, performing the merge operation on adjacent subarrays of the current size.

Analysis

- **Time Complexity:** O(nlogn), same as the recursive version. The outer loop runs logn times, and the inner loop with the merge operation takes O(n) time.
- **Space Complexity:** O(n) because it still requires an auxiliary array for the merge step. However, it avoids the overhead of the call stack, so it is more space-efficient in practice.
- **Pros: No risk of stack overflow**, making it suitable for very large datasets where recursion depth could be an issue. It can be more efficient in terms of memory overhead because it doesn't use the call stack.
- **Cons:** The implementation is generally more complex and harder to read than the recursive version due to the nested loops and manual management of array indices.

Feature	Recursive Merge Sort	Non-Recursive (Iterative) Merge Sort
Approach	Top-down (Divide and Conquer)	Bottom-up (Progressive Merging)
Logic	Splits first, then merges	Merges first, then splits (logically)
Call Stack	Uses a call stack; potential for stack overflow	Avoids the call stack; no stack overflow risk

Code Simplicity	Easier to write and understand	More complex to implement
Memory	Uses more memory for the call stack	No call stack overhead, but still needs a temporary array
Practical Use	Common for general-purpose sorting	Preferred for extremely large datasets or in environments with limited stack memory