Step-by-Step Execution of Recursive Permutation Algorithm:

Let's use the array [1, 2, 3, 4] as an example to demonstrate the step-by-step execution of the recursive permutation algorithm described earlier. The algorithm uses a recursive helper function that swaps elements to generate all possible permutations.

Initial Setup:

- We start with the array A = [1, 2, 3, 4].
- We call helper(A, 0) to start generating permutations from the first element.

Step-by-Step Execution:

1. First Call to helper(A, 0):
    - Array: [1, 2, 3, 4]
    - Index i = 0
    - The loop runs from j = 0 to 3.

    First Loop Iteration (j = 0):

    - Swap A[0] with A[0] (no change).
    - Array remains [1, 2, 3, 4].
    - Recursively call helper(A, 1).

2. Second Call to helper(A, 1):
    - Array: [1, 2, 3, 4]
    - Index i = 1

- The loop runs from j = 1 to 3.

First Loop Iteration (j = 1):

- Swap A[1] with A[1] (no change).
- Array remains [1, 2, 3, 4].
- Recursively call helper(A, 2).

3. Third Call to helper(A, 2):
   - Array: [1, 2, 3, 4]
   - Index i = 2
   - The loop runs from j = 2 to 3.

   First Loop Iteration (j = 2):

   - Swap A[2] with A[2] (no change).
   - Array remains [1, 2, 3, 4].
   - Recursively call helper(A, 3).

4. Fourth Call to helper(A, 3):
   - Array: [1, 2, 3, 4]
   - Index i = 3
   - The loop runs from j = 3 to 3.

   First Loop Iteration (j = 3):

   - Swap A[3] with A[3] (no change).
   - Array remains [1, 2, 3, 4].
   - Recursively call helper(A, 4).

5. Base Case: Call to helper(A, 4):

   - Array: [1, 2, 3, 4]

   - Index i = 4 (equals size(A)).

   - Print [1, 2, 3, 4].

   Backtrack: Return to the previous call helper(A, 3).

   - Swap back A[3] with A[3] (no change).

   End of Loop in helper(A, 3): Return to the previous call helper(A, 2).

6. Back to helper(A, 2):

   - Array: [1, 2, 3, 4]

   - Continue loop for j = 3.

   Second Loop Iteration (j = 3):

   - Swap A[2] with A[3].

   - Array becomes [1, 2, 4, 3].

   - Recursively call helper(A, 3).

   Proceed with similar steps as before to generate [1, 2, 4, 3]:

   - Recursively call helper(A, 4) and print [1, 2, 4, 3].

   - Backtrack to helper(A, 3) and helper(A, 2), swapping back to restore [1, 2, 3, 4].

   End of Loop in helper(A, 2): Return to helper(A, 1).

7. Back to helper(A, 1):

   - Array: [1, 2, 3, 4]

   - Continue loop for j = 2 and j = 3.

   Continue with similar steps to generate:

   - [1, 3, 2, 4], [1, 3, 4, 2], [1, 4, 3, 2], and [1, 4, 2, 3].

   End of Loop in helper(A, 1): Return to helper(A, 0).

8. Back to helper(A, 0):

   - Array: [1, 2, 3, 4]

   - Continue loop for j = 1, j = 2, and j = 3.

   Continue with similar steps to generate:

   - [2, 1, 3, 4], [2, 1, 4, 3], [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 3, 1], [2, 4, 1, 3], and so on...

Summary:

By following this approach, the algorithm systematically generates all possible permutations of [1, 2, 3, 4]. Each step involves swapping elements, recursively generating the next level of permutations, and then backtracking by swapping the elements back to their original positions. This ensures that all permutations are covered without repetition.

Complete List of Permutations Generated:

1. [1, 2, 3, 4]
2. [1, 2, 4, 3]
3. [1, 3, 2, 4]

4. [1, 3, 4, 2]

5. [1, 4, 3, 2]

6. [1, 4, 2, 3]

7. [2, 1, 3, 4]

8. [2, 1, 4, 3]

9. [2, 3, 1, 4]

10. [2, 3, 4, 1]

11. [2, 4, 3, 1]

12. [2, 4, 1, 3]

13. [3, 2, 1, 4]

14. [3, 2, 4, 1]

15. [3, 1, 2, 4]

16. [3, 1, 4, 2]

17. [3, 4, 1, 2]

18. [3, 4, 2, 1]

19. [4, 2, 3, 1]

20. [4, 2, 1, 3]

21. [4, 3, 2, 1]

22. [4, 3, 1, 2]

23. [4, 1, 3, 2]

24. [4, 1, 2, 3]