Finding Array Permutations Using Recursion

The first example displays code that implements a recursive algorithm to find all possible permutations of a given array. For an array of n elements, there are n! possible permutations.

Recursion in Simple

- A form of decrease and conquer in algorithmic programming.
- Reduces a large problem into a smaller magnitude until reaching the base case.
- The base case has a terminating point and a definitive pivot.

Key Components

The `flat()` Function

- THis function isp purely cosmetic and formats nested lists.
- Flattens nested lists into a single-level list
- Iterates through elements, checking if each is a nested list
- Returns a new flattened array

The `permutations()` Function

1. **Base Case**:

   - If array length is 1, returns the array itself.
   - This terminates the recursive chain.
2. **Recursive Step**:

   - For each element in array:
     A. Creates a new array excluding current element.
     B. Recursively finds permutations of remaining elements.
     C. Combines current element with each sub-permutation.

Example Flow

For array `[1,2,3]`:

1. First iteration: pick 1
   - Recursively permute [2,3]
2. Second level: pick 2

        - Recursively permute [3]
3. Builds solutions by combining elements from each level

Time Complexity

- O(n!) where n is the length of input array.
- Each element generates n! permutations.
- Program takes one iteration for each permutation.

In [ ]:

```
# PROBLEM: Finding all the permutations of a small array of numbers.
# For example array [1,2,3] can be permuted as 123, 132, 213, 231, 312, 321.

def flat(arr): # Converts nested lists to a flat list
        new = []
        for element in arr:
                if type(element) is list:
                        for item in element:
                                new.append(item)
                else:
                        new.append(element)
        return new

def permutations(arr): # Finds all permutations of an array
    if len(arr) == 1: # Base case: One element
        return arr
    result = []
    for i in range(len(arr)):
        nor = arr[:i] + arr[i+1:]
        for p in permutations(nor): # Iterative step: Permutations of the rest
            result.append([arr[i], p])
    return result

test = [1, 2, 3, 4] # Test array
for perm in permutations(test):
    while len(perm) < len(test):
        perm = flat(perm) # Flatten the list to remove all nests
    print(perm)
```

Recursion in Pseudocode

The structured pseudocode example follows a very similar approach to the Python3 program, but some functions like the `flat()` function were ommitted because there was no need for formatting.

The Python3 program was described first to better paint a picture of the differences between a high-level programming language and natural language in pseudocode.

```
Algorithm
    Define permutate(array): // Base case
        If length of array = 1 Then
```

```
            Output array
        End If

        result ← [] // Iterating through an array
        For i from 1 to length of array Do
            remaining ← array without array[i]
            For each p in permutate(remaining) Do // Recursive step
                append array[i], p to result
            End For
        End For

        Output each permutation in result
End Algorithm
```