Developing a Recursive Program: Listing Permutations

Step #1: Goal and General Algorithm Idea

Scenario: A number of people each need a unique PIN of length \( n \), made up of the digits 1 \ldots n.

Goal: Write a program that generates all possible PINs of length \( n \), made up of the digits 1 \ldots n.

Subgoal: Write a program to generate all permutations of the digits 1 \ldots n.

Let’s begin by looking at the permutations of the digits 1, 2, and 3:

\[
\begin{align*}
1 & \ 2 & \ 3 \\
2 & \ 1 & \ 3 \\
3 & \ 2 & \ 1 \\
1 & \ 3 & \ 2 \\
3 & \ 2 & \ 1 \\
2 & \ 1 & \ 3 \\
\end{align*}
\]

Notice a pattern here: pick the first digit 1, permute the other two, and prepend the 1; then pick the second digit 2, permute the other two, and prepend the 2; and finally, pick the third digit 3, permute the other two, and prepend the 3.

More generally, we pick the \( i \)th digit, permute all the others, and then prepend that \( i \)th digit.

This algorithm suggests recursion. It has a base case, where the recursion stops. Specifically, the permutation of 0 digits is empty, and the permutation of 1 digit is that digit itself. And it has an induction step, namely permuting all but the \( i \)th digit and then prepending that.

Now that we have the general idea, let’s design the program.

Step #2: Data Representation and Program Structure

Part #1: Data Structures:

Represent the sequence of digits as a list; so the sequence 1, 2, 3 would be treated as a list \( L \).

Represent each permutation as an element of another list \( I \).

Part #2: Functions

And now we write the function suggested by the above. Let’s call it:

\[
\text{function perm}(L) \rightarrow \text{returns list of permutations of elements of } L
\]

First, the base case, when there is no recursion and a value is simply returned. This should happen when the list \( L \) contains exactly 1 element. We can also add an error check. \( L \) should never be the empty list, but we can easily check, and so we do:

\[
\begin{align*}
\text{if length of } L \text{ is 0:} & \quad \text{return empty list} \\
\text{if length of } L \text{ is 1:} & \quad \text{return list containing } L
\end{align*}
\]

Next, we have to create the list \( I \) for the list of permutations. Initially, it’s empty:

\[
I \text{ is empty}
\]

Now for the recursion. We want to loop through \( L \), extracting the elements successively. After each extraction, we create a new list without it but with all the other elements. We then permute that list, prepend the extracted element, and continue until we are done with the list:

\[
\begin{align*}
\text{for each element in } L: & \quad \text{remove that element (call it } L[i]) \\
& \quad \text{rest of list is } L[0 \text{ up to } i] + L[\text{everything after } i]; \text{ call this } R
\end{align*}
\]

\[
\begin{align*}
\text{for each element in } \text{perm}(R): & \quad \text{prepend } L[i]; \text{ call the result } P \\
& \quad \text{append } P \text{ to } I
\end{align*}
\]

Now we have the list of permutations in \( I \). So we return it.
And that’s it!

**Step #3: Put It into Python**

We can translate the function above almost line for line:

```python
def perm(L):
    # base cases: if list is empty or
    # has 1 element, return it as a list
    # so it can be appended to the list
    # of permutations
    if len(L) == 0:
        return []
    if len(L) == 1:
        return [L]
    # this will hold the permuted lists of L
    I = []
    # move each element in the list to the front
    # and permute the rest of the list; for each
    # permutation, prepend the front element and
    # save the result in the list of permutations
    for i in range(len(L)):
        # drop the i-th element; this gives you
        # the rest of the list to be permuted
        R = L[:i] + L[i+1:]
        # generate the permutations of the rest
        # for each permutation, prepend the one
        # you held back and add it to the list of
        # permutations
        for e in perm(R):
            P = [L[i]] + e
            I.append(P)
    # return the list of permutations
    return I
```

**Step #4: The Program**

To print the permutations, we just print all the elements in the list that `perm` returns:

```python
# this is the data to permute
# here, it’s numbers, but it can be anything
data = [1, 2, 3, 4]

# get the list of permutations and print it
for p in perm(data):
    print(p)
```