Recursion

Examples of recursive functions

We will start with a simple mathematical function, the factorial. In mathematics, the factorial of a whole number \( n \) (note that whole numbers do not include 0) is defined as:

\[
\text{fact}(n) = n 
\times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1
\]

Thus, according to the definition above we will have:

\[
\begin{align*}
\text{fact}(1) &= 1 \\
\text{fact}(2) &= 2 \times 1 = 2 \\
\text{fact}(3) &= 3 \times 2 \times 1 = 6 \\
&\vdots
\end{align*}
\]

An interesting property of the factorial is that it grows very large for small values of \( n \). For example, \( \text{fact}(6) = 720! \)

BTW, why is the factorial function useful? Well, suppose you like ice cream and you would like to taste six different ice cream samples. However, the taste of each sample may change depending on the previous sample you ate. If you want to try samples in all possible sequences, how many different orders should you make? The answer is ... 720! And 720 is the factorial of 6. And it might take a couple of years to try all the sample sequences (if you try two of them each day)!

The factorial function can be implemented in a straightforward way without using recursion. Look at the following code (and try it):

```python
def factorial(n):
    fact = 1
    for i in range(n, 1, -1):
        fact *= i
    return fact
```

If you look at the (intuitive) definition of the factorial that we wrote above, you can easily see that if you want to define, say, \( \text{fact}(8) \), you would have to write:

\[
\text{fact}(8) = 8 \times 7 \times 6 \times 5 \times \ldots \times 1
\]

Well, we can actually define \( \text{fact}(n) \) in a recursive way as follows (also taking into account the case when \( n \) is 0):

```python
fact(n) =
1 if n = 0
n \times \text{fact}(n-1) \text{ if } n > 0
```

And we can implement our function \( \text{factorial}(n) \) in a recursive way:

```python
# Factorial of a positive integer n.
def factorial(n):
    if n == 0:
        return 1
    else:
        return n \times \text{factorial}(n-1)
```

Let's look at why we can do that. We know that \( \text{fact}(0) = 1 \). Then we can see that \( \text{fact}(1) \) can be “decomposed” as \( 1 \times \text{fact}(0) \), that is \( 1 \times 1 \). \( \text{fact}(2) \) can be “decomposed” as \( 2 \times \text{fact}(1) \), that is \( 2 \times 1 \times 1 \). \( \text{fact}(3) \) can be “decomposed” as \( 3 \times \text{fact}(2) \), that is \( 3 \times 2 \times 1 \times 1 \). And so on. See the recurring “pattern”? Note also the “termination condition” (if \( n = 0 \) return 1) in the code above and how it corresponds to the recursive definition of fact.
In the next section, we will revisit some of the tree traversal algorithm we saw earlier in the course and look at how they can be implemented in Python.

**Encoding Binary trees with Python Lists**

Binary trees can be encoded by using lists in Python. Each node the binary tree is represented as a triple `[data, left, right]`, where `data` is the data/value at a node, `left` is the left child of the node and `right` is the right child of a node. Note that `left` and `right` are integers. A list of such triples for all nodes of the tree constitute the encoding for the entire binary tree. Also note that the `left` and `right` values for each node are indices of the triple for the left child and right child respectively in the entire list of nodes. A node that is represented by the triple `[data, -1, -1]` means that it does not have any left or right children and hence is a leaf node. A `-1` for the left or right link indicates the absence of the respective child.

Let's look at a few examples of how the encoding works.

Consider the following simple binary tree.

```
2
/   \
1     3
```

The root node has value 2, the left child has value 1 and the right child has value 3. The encoding for this tree is given by the list as `[ [2, 1, 2], [1, -1, -1], [3, -1, -1] ]`. The root node is encoded as `[2, 1, 2]`, the value at the root is 2, its left child is the second element in the list, the node represented by `[1, -1, -1]` and its right child is the third element in the list, the node represented by `[3, -1, -1]`.

Now look at the following binary tree and arrive at the encoding for the tree,

```
10
/   \
2   3
/ \
6  7 5  8
```

The above tree can be encoded as, `[ [10, 1, 2], [2, 3, 4], [3, 5, 6], [6, -1, -1], [7, -1, -1], [5, -1, -1], [8, -1, -1] ]`. Let the above tree be encoded as the list `T`. The first element in the list, i.e., `T[0]` is the root node. Since each node is
encoded as a triple, $T[0][0]$ is the value/data at the root, $T[0][1]$ gives the left child of the root and $T[0][2]$ is the right child of the root.

**Tree Traversal**

Once we can correctly encode any binary tree as described above, we can traverse the tree in any order. Let’s look at the pseudocode for recursive tree traversal.

```plaintext
// Input - T, the root node of the tree
PrintBTR(T)
    if (T == NULL)
        return
    PrintBTR(T.left)
    print(T.data)
    PrintBTR(T.right)
```

Using the pseudocode above, we can write a recursive tree traversal function `PrintBTR(T, currNode)` in Python. The function takes two parameters, the list encoding of the tree and the index, in the list, of the current node that we are looking at.

In the above pseudocode, the termination condition for recursion is when we look past a valid node of the tree, i.e., trying to follow a NULL link. The equivalent condition for our list encoding of the binary tree is the case when the link we are trying to follow, i.e., the left or right link is empty. This condition can be expressed as

```
currNode == -1
```

The link to the left child of a node can be extracted as $T[currNode][1]$ and the link to the right child can be extracted as $T[currNode][2]$. $T[currNode][0]$ gives the data/value at the current node.

Using this, the above algorithm can be written in Python as follows,

```python
def PrintBTR(T, currNode):
    if currNode == -1: # Termination condition for recursion
        return
    left = T[currNode][1] // Left child of the current node
    right = T[currNode][2] // Right child of the current node
    if left != -1:
        PrintBTR(T, left) // Recursively traverse the left sub-tree of current node
    print(T[currNode][0]) // Print value at current node
    if right != -1:
        PrintBTR(T, right) // Recursively traverse the right sub-tree of current node
```

sp2012_prelab12.txt · Last modified: 2012/03/29 22:40 by gsurendr