Recursive Functions

Recursion in Familiar Contexts

Thinking Recursively

Eric S. Roberts

Recursion in Familiar Contexts

Cat in the Hat Comes Back

Dr. Seuss

Recursion in Familiar Contexts

And then Little Cat A
Told the hat off my head.
It is good I have some one
To help me," he said.
"This is Little Cat B.
And I sleep like ashes,
And when I used help.
Then I let him come out."
Recursion in Familiar Contexts

Recursion

- Recursion is the process of solving a problem by dividing it into smaller subproblems of the same form. The italicized phrase is the essential characteristic of recursion; without it, all you have is a description of the more familiar strategy of decomposition or stepwise refinement.

- The fact that recursive decomposition generates subproblems that have the same form as the original problem means that recursive programs will use the same function or method to solve subproblems at different levels of the solution. In terms of the structure of the code, the defining characteristic of recursion is having functions that call themselves, directly or indirectly, as the decomposition process proceeds.

A Simple Illustration of Recursion

- Suppose that you are the national fundraising director for a national campaign and need to raise $1,000,000.
- One possible approach is to find a wealthy donor and ask for a single $1,000,000 contribution. The problem with that strategy is that individuals with the necessary combination of means and generosity are difficult to find. Donors are much more likely to make contributions in the $100 range.
- Another strategy would be to ask 10,000 friends for $100 each. Unfortunately, most of us don’t have 10,000 friends.
- There are, however, more promising strategies. You could, for example, find ten regional coordinators and ask each one to raise $100,000. Those regional coordinators could in turn delegate the task to local coordinators, each with a goal of $10,000, continuing until the process reached the $100 level.

A Pseudocode Fundraising Strategy

- If you were to implement the fundraising strategy in the form of a Python function, it would look something like this:

```python
def collectContributions(n):
    if n <= 100:
        Collect the money from a single donor
    else:
        Find 10 volunteers.
        Get each volunteer to collect n/10 dollars.
        Combine the money raised by the volunteers.
```

- What makes this strategy recursive is that the line

```
get each volunteer to collect n/10 dollars
```

will be implemented using the following recursive call:

```
collectContributions(n / 10);
```

Recursive Functions

- The easiest examples of recursion to understand are functions in which the recursion is clear from the definition. As an example, consider the factorial function, which can be defined in either of the following ways:

\[
 n! = n \cdot (n-1)! \quad \text{if } n \geq 1
\]

\[
 n! = \begin{cases} 
 1 & \text{if } n = 0 \\
 n \cdot (n-1)! & \text{otherwise}
\end{cases}
\]

- The second definition leads directly to the following code, which is shown in simulated execution on the next slide:

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```
The Fibonacci Function

• In 1202, the Italian mathematician Leonardo Fibonacci proposed the following problem: How many pairs of rabbits exist after $n$ months if one newborn pair is introduced in the first month and rabbits reproduce according to these rules:
  - Each pair of rabbits produces a new pair each month.
  - Rabbits become fertile in the second month of life.
  - Old rabbits never die.
• The number of rabbits over the course of ten months is then:

<0 1 1 2 3 5 8 13 21 34>
<0 1 2 3 4 5 6 7 8 9>

• This sequence of numbers is called the Fibonacci sequence.

Efficiency of the Recursive Solution

• Unfortunately, the recursive solution on the preceding slide is highly inefficient because the computation recalculates the same value many times:

- It is critical to understand that the problem does not lie in the fact that the solution is recursive but in how recursion is used.

Solving a More General Problem

• As it happens, you can implement the Fibonacci function more efficiently by solving a more general problem.
  - Define an additive sequence as one in which the first two terms are $t_0$ and $t_1$ and every subsequent term is the sum of the previous two. The $n$th term in any additive sequence is simply the $(n-1)^{th}$ in the additive sequence that begins one term later, which leads to the following function definition:

- The Fibonacci sequence is then an additive sequence in which the values $t_0$ and $t_1$ are 0 and 1, respectively.
Exercise: The Cannonball Problem

- Although more efficient packing strategies are possible, you can pack cannonballs in square layers that decrease in size as you move upward.

- A stack of height 3 consists of:
  - A 3 × 3 square of 9 cannonballs
  - A 2 × 2 square of 4 cannonballs
  - A 1 × 1 square of 1 cannonball
  for a total of 14 cannonballs.

- Write a function `cannonball(n)` that uses recursion to determine the number of cannonballs in a stack of size n.