Recursive Functions

- A function that calls itself is said to be recursive
- If a function f1 is also recursive if it calls a function f2 which under some condition calls f1, creating a cycle in the sequence of calls
- Reason to write a recursive function is to pass different parameter values to the same function and obtain different results
- Recursive functions are not very efficient because of the number of function calls you need to make but very useful in writing solutions to certain problems

Recursive Functions

- Recursive algorithms follow a general rule:

  if this is a simple case

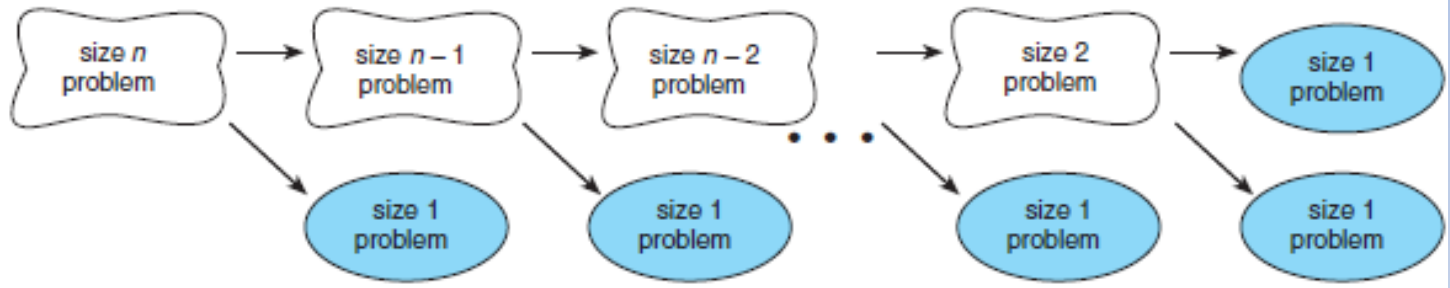
  solve it

  else

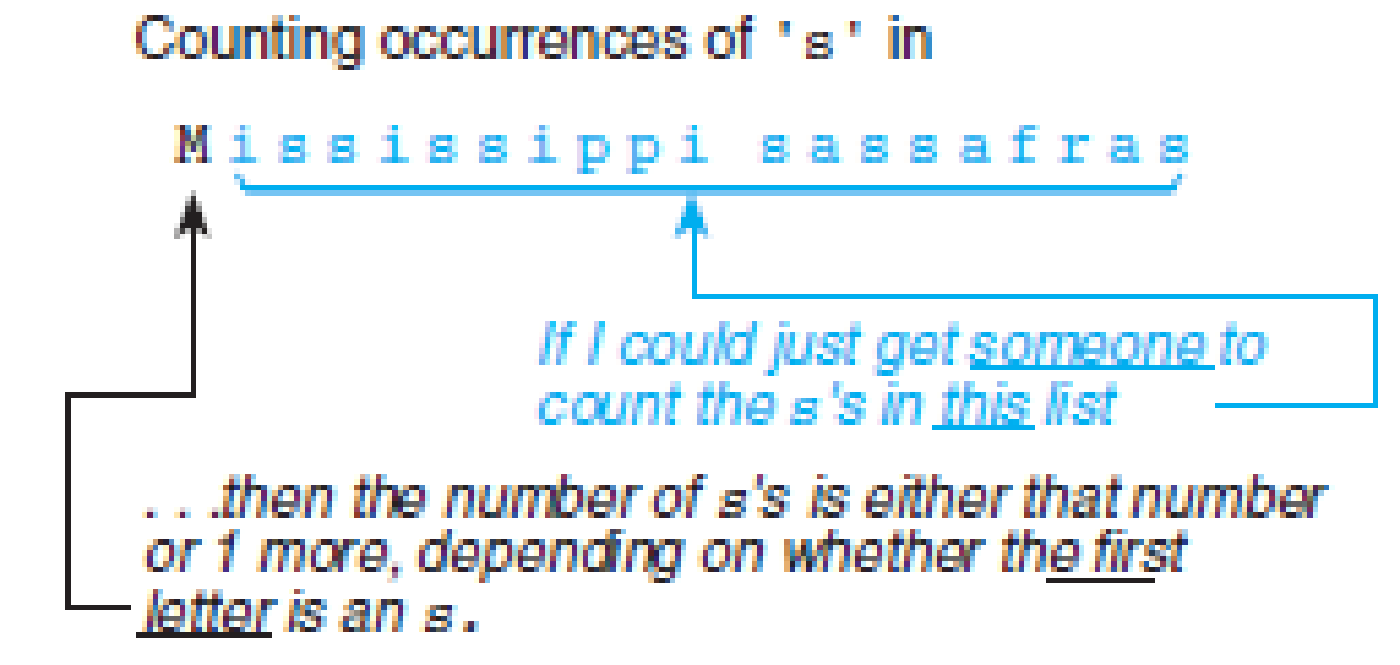
  redefine the problem using recursion

- Example:
  - For a particular problem of n size, we can split the problem into:
    - A problem of size 1
    - A problem of size (n-1)
  - For a particular problem of (n-1) size, we can split the problem into:
    - A problem of size 1
    - A problem of size (n-2)
  - Eventually we will end up with n problems of size 1 that we can solve one at a time

Splitting a Problem into Smaller Problems



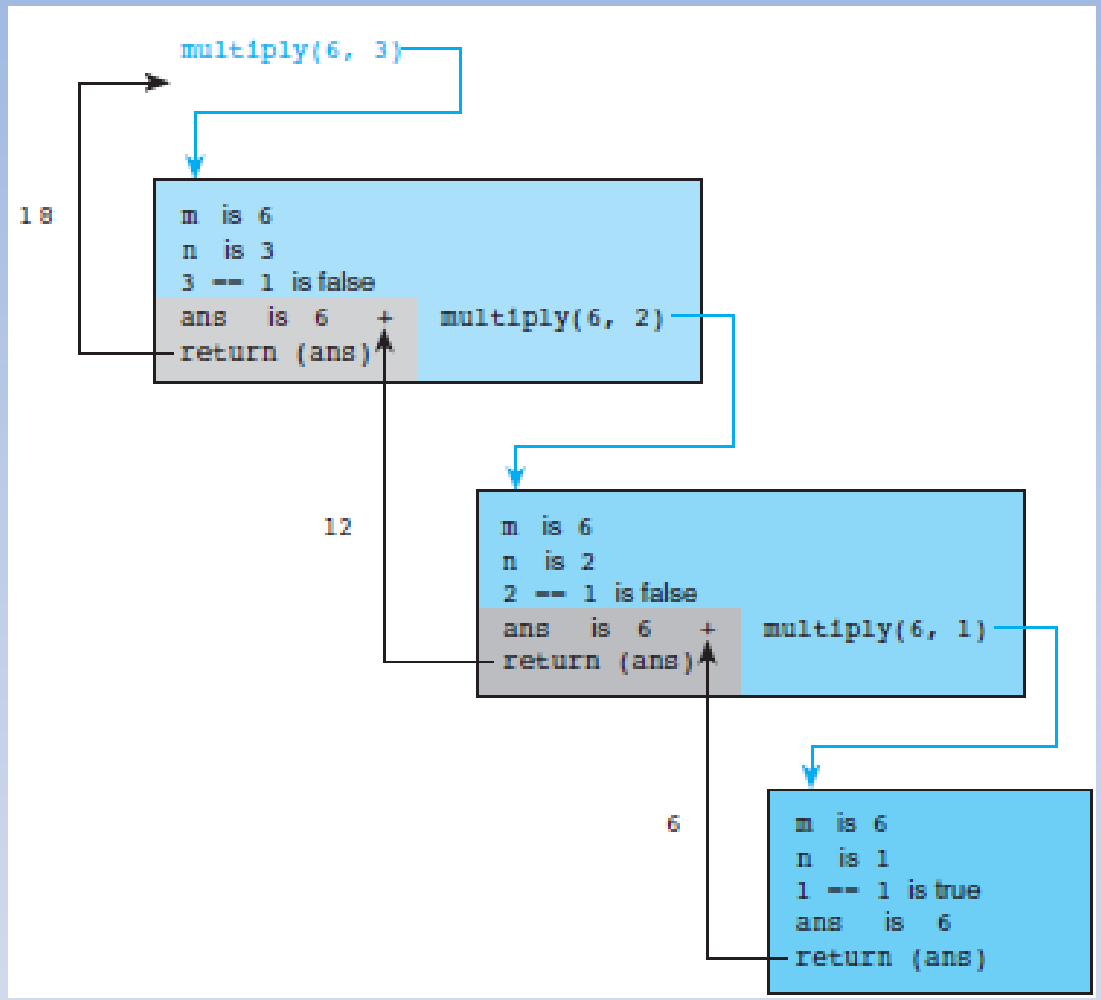
Thought Process of a Recursive Algorithm Developer



Recursive Functions

- Example: Multiply 6 by n but do not use the multiplication operator
  - The simplest way to do this is to add 6 to the result n times, each time the addition getting added on to the previous result (starting with a value of 0)
  - If we want to multiply 6 by 3:
    - 6 + 0 = 6 6 (Multiply 6 by 1), add to result
    - 6 + 6 = 12 6 (Multiply 6 by 1), add to result
    - 6 + 12 = 18 6 (Multiply 6 by 1), add to result
  - Algorithmically, we write this as:
    - 1. Multiply 6 by 3
      - 1.1 Multiply 6 by 2
        - » 2.1 Multiply 6 by 1
          3.1 Add 6 to result
        - » 2.2 Add 6 to result
      - 1.2 Add 6 to result

Trace of Function multiply



Recursive Functions

```
def multiply(m, n):
    if n == 1:
        answer = m
    else:
        answer = m + multiply(m, n-1)
    return(answer)
```

- Parameter and Local Variables are stored in a special data structure called ***stack***

Recursive Functions - Factorial

- Several mathematical functions are good candidates for recursion
- First let's look at the iterative version of computing the factorial of a number

```
def factorial(n):
    product = 1
    for i in range(1,n+1):
        product = product * i
    return(product)
```

Recursive Functions - Factorial

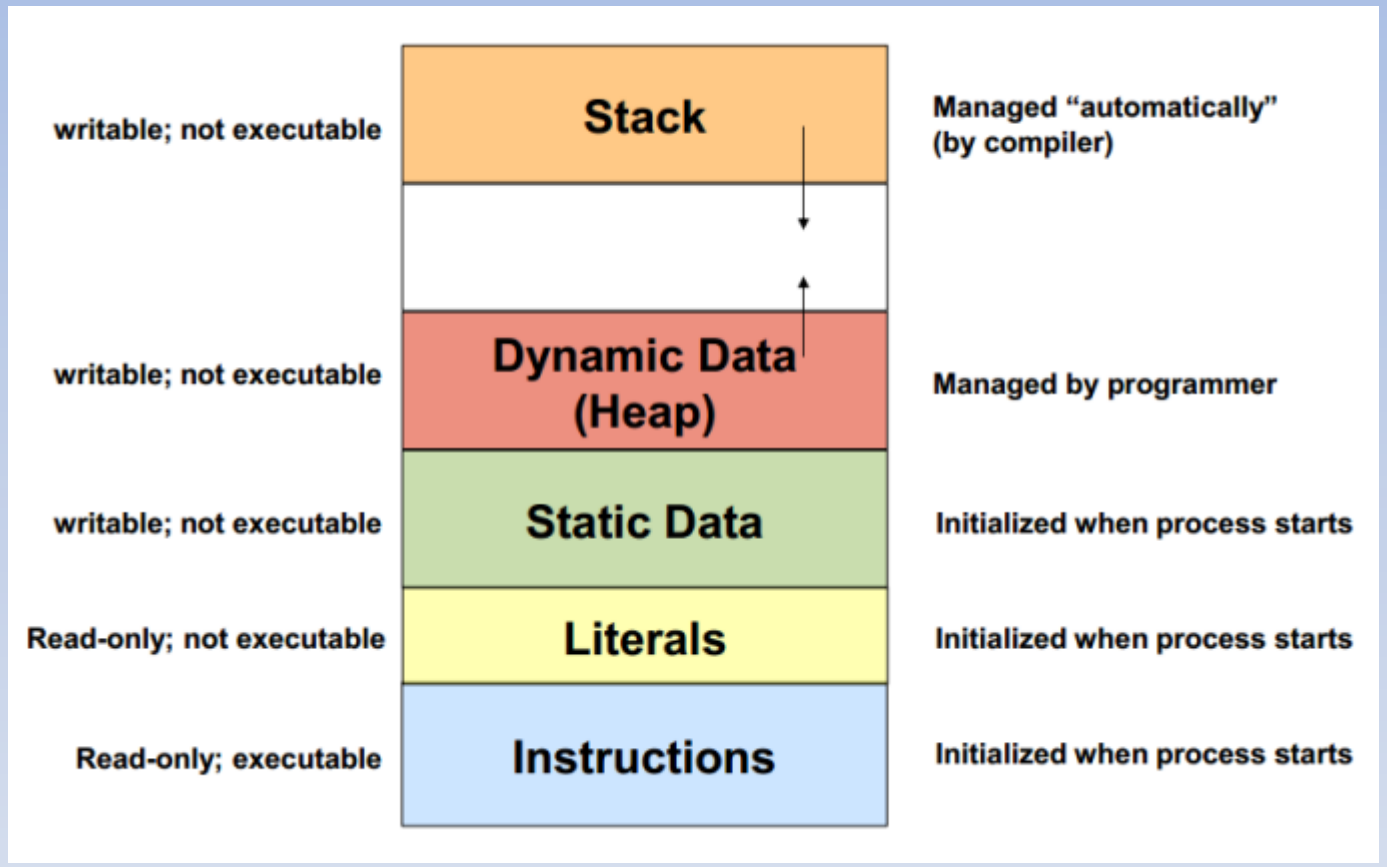
- Compute the factorial of a number using a recursive function

```
def factorial(n):
    if n == 1:
        answer = 1
    else:
        answer = n * factorial(n-1)
    return(answer)
```

Recursion vs. iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution to a problem, often resulting in shorter, more easily understood source code

Memory Allocation



How do I write a recursive function?

- Determine the size factor
- Determine the base case(s)

  (the one for which you know the answer)
- Determine the general case(s)

  (the one where the problem is expressed as a smaller version of itself)
- Verify the algorithm

  (use the "Three-Question-Method")

Three-Question Verification Method

- The Base-Case Question:
- Is there a non-recursive way out of the function, and does the routine work correctly for this "base" case?
- The Smaller-Caller Question:
- Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?
- The General-Case Question:
- Assuming that the recursive call(s) work correctly, does the whole function work correctly?

Recursive Functions - Fibonacci

- The Fibonacci series is a sequence of numbers of the form: 1, 1, 2, 3, 5, 8, 13, 21, … where each number in the series is the sum of the previous two numbers (except for the first two which are 1)
- Write a recursive to compute the nth Fibonacci number, and a main function which prints the first 20 numbers of the Fibonacci series

```
def fibonacci(n):
    if(n <= 2):
        return(1)
    else:
        return(fibonacci(n-1) + fibonacci(n-2))

for i in range (1, 21):
    print(fibonacci(i), end=" ")
```

```
>>>
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

Example – Reverse order of input

```
def main():
    ## Reverse the order of items entered by the user.
    state = ""
    getState(state)

def getState(state):
    state = input("Enter a state: ")
    if state != "End":
        getState(state)
        print(state)

main()
```

```
Enter a state: Maine
Enter a state: Utah
Enter a state: Wyoming
Enter a state: End
Wyoming
Utah
Maine
```

Towers of Hanoi

- Courtesy:
  http://www.numerit.com/samples/hanoi/doc.htm