Problem

Write a program that prompts the user for a number of *segments* (a non-negative integer) and a *size* (a positive integer) and then draws a Y-tree as follows:

- If `segments == 0`, it draws nothing.
- If `segments == 1`, it draws a trunk (a line) of length `size` (a somewhat degenerate tree).
- If `segments == 2`, it draws a little tree, consisting of a trunk (a line) of length `size` that splits into two branches (two lines) of length `size/2`. The tree is symmetric and there is a right angle between the two branches.
- If `segments == 3`, it draws a tree with four more branches. The tree is the same as for `segments == 2`, but with additional splits at the ends of the branches of length `size/2`, where each of the new branches is of length `size/4`.
- For each greater value of `segments`, the program draws trees with yet more branches. Each tree is similar to the previous tree, but with additional splits, where each of the new branches is half the size of the branch from which it splits.
Note that \texttt{segments} counts the number of line segments from the base of the tree to the end of any branch.

\section*{Problem Analysis and Solution Design}

\subsection*{Development}

How do we solve the problem for \texttt{segments == 1}? This seems simple.

\begin{verbatim}
def drawTree1(size):
    forward(size)
\end{verbatim}

How do we solve the problem for \texttt{segments == 2}? This also seems simple.

\begin{verbatim}
def drawTree2(size):
    forward(size)
    left(45)
    forward(size/2)
    forward(-size/2)
    right(45)
    right(45)
    forward(size/2)
\end{verbatim}

These solutions work, but we’ve previously seen that it is good for figure-drawing functions to return to their initial position and orientation after drawing.

Using that idea we solve the problem and return to the initial position and orientation for \texttt{segments == 1} like this:

\begin{verbatim}
def drawTree1(size):
    forward(size)
    forward(-size)
\end{verbatim}

And we might solve the problem and return to the initial position and orientation for \texttt{segments == 2} this way:

\begin{verbatim}
def drawTree2(size):
    forward(size)
    left(45)
    forward(size/2)
    forward(-size/2)
    right(45)
    right(45)
    forward(size/2)
    forward(-size/2)
    left(45)
    forward(-size)
\end{verbatim}
We should notice that there is some repetition in the solution for `segments == 2` and that the repeated code is similar to the solution for `segments == 1`. We can replace the repetitive code by combining it with the solution for `segments == 1`.

```python
def drawTree2(size):
    forward(size)
    left(45)
    drawTree1(size/2)  # reuse the other function
    right(45)
    right(45)
    drawTree1(size/2)  # again
    left(45)
    forward(-size)
```

How do we solve the problem and return to the initial position and orientation for `segments == 3` (and reuse our solution for `segments == 2`)?

```python
def drawTree3(size):
    forward(size)
    left(45)
    drawTree2(size/2)
    right(45)
    right(45)
    drawTree2(size/2)
    left(45)
    forward(-size)
```

How do we solve the problem and return to the initial position and orientation for `segments == 4` (and reuse our solution for `segments == 3`)?

```python
def drawTree4(size):
    forward(size)
    left(45)
    drawTree3(size/2)
    right(45)
    right(45)
    drawTree3(size/2)
    left(45)
    forward(-size)
```

**Evolution to a Recursive Function Solution**

We observe much similarity between `drawTree2`, `drawTree3`, `drawTree4`, .... If we use a `segments` parameter to distinguish the different `drawTree#` functions:

```python
def drawTree(segments, size):
    if segments == 1:
        forward(size)
        forward(-size)
```
else:
    forward(size)
    left(45)
    drawTree(segments-1, size/2)
    right(45)
    right(45)
    drawTree(segments-1, size/2)
    left(45)
    forward(-size)

How do we solve the problem for segments == 0?

def drawTree0(size):
    pass

Note that pass is the Python command to do nothing.

We can incorporate this into our drawTree function as follows:

def drawTree(segments, size):
    if segments == 0:
        pass
    elif segments == 1:
        forward(size)
        forward(-size)
    else:
        forward(size)
        left(45)
        drawTree(segments-1, size/2)
        right(45)
        right(45)
        drawTree(segments-1, size/2)
        left(45)
        forward(-size)

Finally, we notice that the case segments == 1 need not be treated specially — drawing a tree with segments == 1 corresponds to drawing the trunk, then “drawing” two trees with segments == 0.

def drawTree1(size):
    forward(size)
    left(45)
    drawTree0(size/2)
    right(45)
    right(45)
    drawTree0(size/2)
    left(45)
    forward(-size)
def drawTree(segments, size):
    if segments == 0:
        pass
    else:
        forward(size)
        left(45)
        drawTree(segments -1, size /2)
        right(45)
        right(45)
        drawTree(segments -1, size /2)
        left(45)
        forward(-size)

This solution uses something called recursion. Recursion solutions to a problem depend on solutions to smaller instances of the same problem. In our case, the steps for drawing the branches for each of the segments is identical. Because of this similarity, we are able to generalize the code to develop our final drawTree function.

Final Program Solution

For the final program, in addition to the drawTree function developed in the previous section, we need an initialization function. Further, it is often worthwhile to have an entry-point function that does initialization and then calls the function of interest. All the functions are also carefully commented.

def drawTree( segments, size ):
    """
    drawTree recursively draws the tree.
    segments -- NonNegInteger;
        number of line segments from the base of the tree to
        the end of any branch should be integral and non-negative.
    size -- PosNumber;
        length of tree "trunk" to draw should be (strictly) positive.
    pre-conditions: segments >= 0, size > 0.
        turtle is at base of tree,
        turtle is facing along trunk of tree,
        turtle is pen-down.
    post-conditions: a segments-level tree was drawn on the canvas,
        turtle is at base of tree,
        turtle is facing along trunk of tree,
        turtle is pen-down.
    """
    if segments == 0:
        # base case: draw nothing
        pass
    else:
        # recursive case: draw trunk and two sub-trees
        forward( size )
        left( 45 )
def initWorld(size):
    """
    initWorld initializes the drawing by establishing its pre-conditions.
    
    size -- PosNumber;
    length of tree trunk to draw should be positive.
    
    pre-conditions:
    post-conditions: coordinate system
    is (-2*|size|,-2*|size|) at lower-left
    to (2*|size|, 2*|size|) at upper-right.
    turtle is at origin,
    turtle is facing North,
    turtle is pen-down.
    """

setup(600, 600)
# provide a canvas boundary of 2
setworldcoordinates(-2*abs(size) - 2, -2*abs(size) - 2, \
    2*abs(size) + 2, 2*abs(size) + 2)
home()  # turtle is at origin, facing east, pen-down
left(90)  # turtle is facing North
down()  # turtle is pen-down
pensize(2)

def initWorldAndDrawTree(segments, size):
    """
    initWorldAndDrawTree prints a message, initializes the world,
    draws an instance of the recursive tree, and waits for ENTER.
    
    segments -- NonNegInteger;
    number of line segments from the base of the tree to
    the end of any branch should be integral and non-negative.
    size -- PosNumber;
    length of tree "trunk" to draw should be (strictly) positive.
    message -- String;
    message to display
    """
    print("Drawing recursive tree with", (segments, size))
    initWorld(size)
    drawTree(segments, size)
    update()
done()

Note that the drawTree expects the turtle’s position and orientation to be at the base of the tree and facing along the trunk. This is called a pre-condition because a specific state is assumed to have been established before the function begins its execution.
Also, note that we are careful to return the turtle to its initial position and orientation at the base of the tree and facing along the trunk after drawing the tree. The turtle must be at exactly the same position and in exactly the same orientation as at the beginning of the drawing. Suppose otherwise: after drawing the left smaller tree, the turtle would not be at the crotch of the tree (the base of the smaller tree) and we would draw the right tree starting somewhere other than the crotch. (See what happens if you remove the “move by length -size units” from the pseudocode of drawTree.) This is called a post-condition because a specific state is guaranteed to have been established after the function finishes its execution.

We should always document pre- and post-conditions for functions. This allows other programmers to know what needs to be done before calling the function and what can be expected after calling the function. Beware: if you call a function without satisfying its pre-conditions, all bets are off!

**Execution Diagram**

We can visualize the execution of the drawTree function with an execution diagram.

![Execution Diagram](image)

**Implementation**

See tree.py.
Testing (Test Cases, Procedures, etc.)

We need to test the base case(s) and the recursive case(s).

For the base case that draws nothing, we should test segments == 0 and a variety of values for size (e.g., 10, 100, 1, 0, possibly also -10).

For the simple recursive case that draws a trunk and recurses to draw nothing, we should test segments == 1 and a variety of values for size (e.g., 10, 100, 1, 0, possibly also -10).

For the complex recursive cases that draw a trunk and recurse to draw something, we need to test segments > 1 (e.g., 2, 3, 5) and a variety of values for size.

NOTE: The variety of values for size only matter if we do not adjust the world coordinates to the size.