

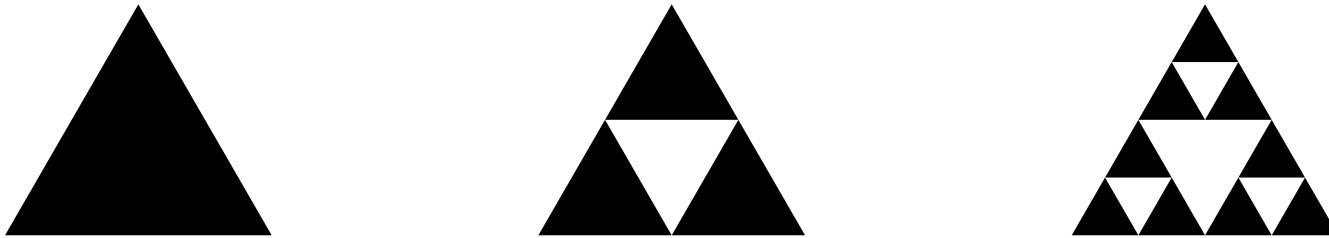
Lecture #7: Tree Recursion

Announcements:

- Hog contest to be released today (I think). It is optional.
- Also watch for HW #2.

Subproblems and Self-Similarity

- Recursive routines arise when solving a problem naturally involves solving smaller instances of the same problem.
- A classic example where the subproblems are visible is *Sierpinski's Triangle* (aka bit Sierpinski's Gasket).
- This triangle may be formed by repeatedly replacing a figure, initially a solid triangle, with three quarter-sized images of itself ($1/2$ size in each dimension), arranged in a triangle:



- Or we can think creating a "triangle of order N and size S " by drawing either
 - a solid triangle with side S if $N = 0$, or
 - three triangles of size $S/2$ and order $N - 1$ arranged in a triangle.

The Gasket in Python

- We can describe this as a recursive Python program that produces Postscript output.

```
sin60 = sqrt(3) / 2
def make_gasket(x, y, s, n, output):
    """Write Postscript code for a Sierpinski's gasket of order N
    with lower-left corner at (X, Y) and side S on OUTPUT."""
    if n == 0:
        draw_solid_triangle(x, y, s, output)
    else:
        make_gasket(x, y, s/2, n - 1, output)
        make_gasket(x + s/2, y, s/2, n - 1, output)
        make_gasket(x + s/4, y + sin60*s/2, s/2, n - 1, output)

def draw_solid_triangle(x, y, s, output):
    "Draw a solid triangle lower-left corner at (X, Y) and side S."
    print("{0} {1} moveto " # Go x, y
          "{2} 0 rlineto " # Horizontal move by s units
          "-{3} {4} rlineto " # Move up and to left
          "closepath fill" # Close path and fill with black
          .format(x, y, s, s/2, s*sin60), file=output)
```

Aside: Using the Functions

- Just to complete the picture, we can use `make_gasket` to create a standalone Postscript file on a given file.

```
def draw_gasket(n, output=sys.stdout):
    print("%!", file=output)
    make_gasket(100, 100, 400, 8, output)
    print("showpage", file=output)
```

- And just for fun, here's some Python magic to display triangles automatically (uses `gs`, the Ghostscript interpreter for Postscript).

```
from subprocess import Popen, PIPE, DEVNULL
```

```
def make_displayer():
    """Create a Ghostscript process that displays its input (sent in through
    .stdin)."""
    return Popen("gs", stdin=PIPE, stdout=DEVNULL)
```

```
>>> d = make_displayer()
>>> draw_gasket(5, d.stdin)
>>> draw_gasket(10, d.stdin)
```

Aside: The Gasket in Pure Postscript

- One can also perform the logic to generate figures in Postscript directly, which is itself a full-fledged programming language:

```
%! 
```

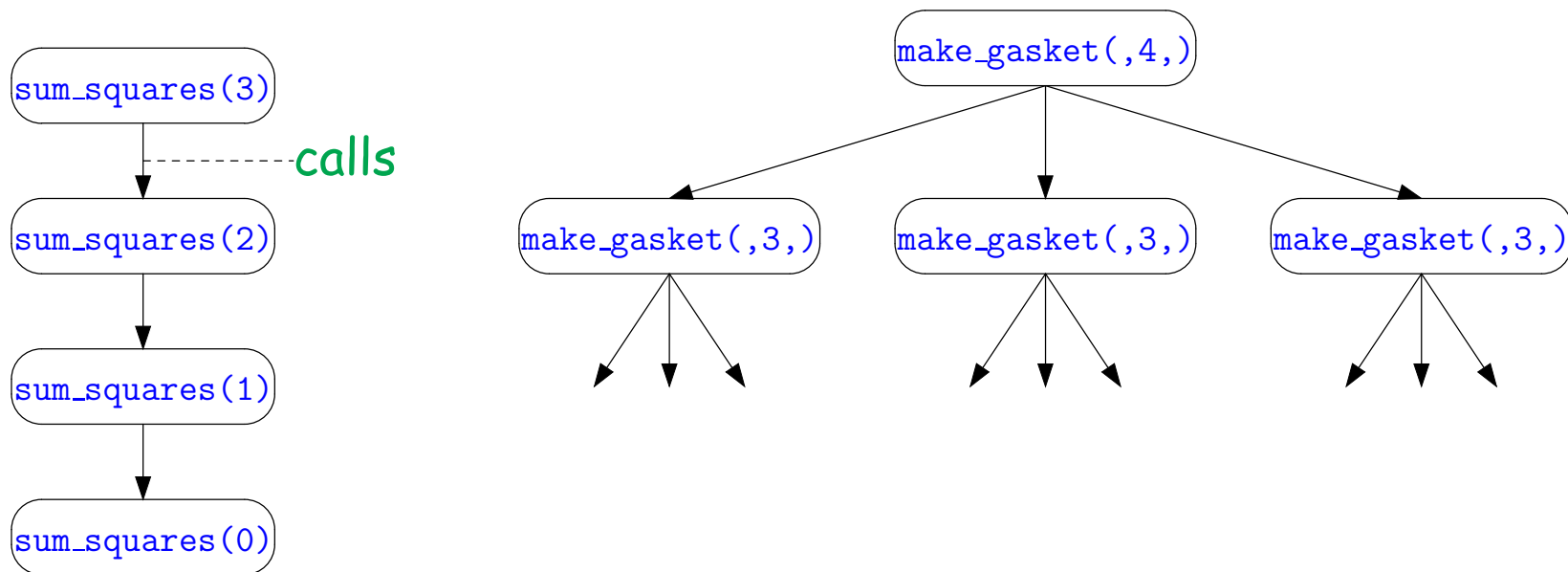
```
/sin60 3 sqrt 2 div def
```

```
/make_gasket {  
  dup 0 eq {  
    3 index 3 index moveto 1 index 0 rlineto 0 2 index rlineto  
    1 index neg 0 rlineto closepath fill  
  } {  
    3 index 3 index 3 index 0.5 mul 3 index 1 sub make_gasket  
    3 index 2 index 0.5 mul add 3 index 3 index 0.5 mul  
    3 index 1 sub make_gasket  
    3 index 2 index 0.25 mul add 3 index 3 index 0.5 mul add  
    3 index 0.5 mul 3 index 1 sub make_gasket  
  } ifelse  
  pop pop pop pop  
} def
```

```
100 100 400 8 make_gasket showpage
```

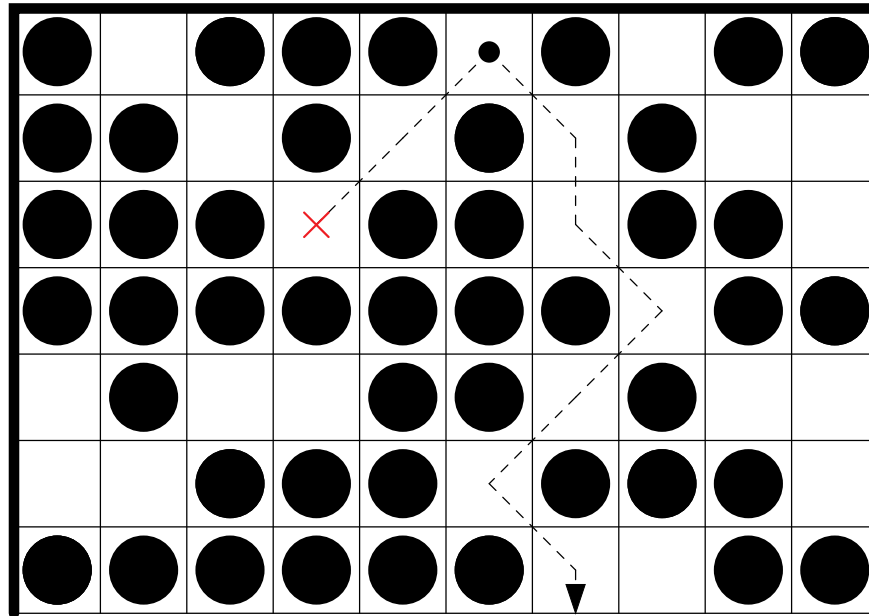
Tree Recursion

- The `make_gasket` function is an example of a *tree recursion*, where each call makes multiple recursive calls on itself.
- A linear recursion such as that on the left (for `sum_squares`) produces a pattern of calls such as that on the left, while `make_gasket` produces the pattern on the right—an instance of what we call a *tree* in computer science.



Finding a Path

- Consider the problem of finding your way through a maze of blocks:

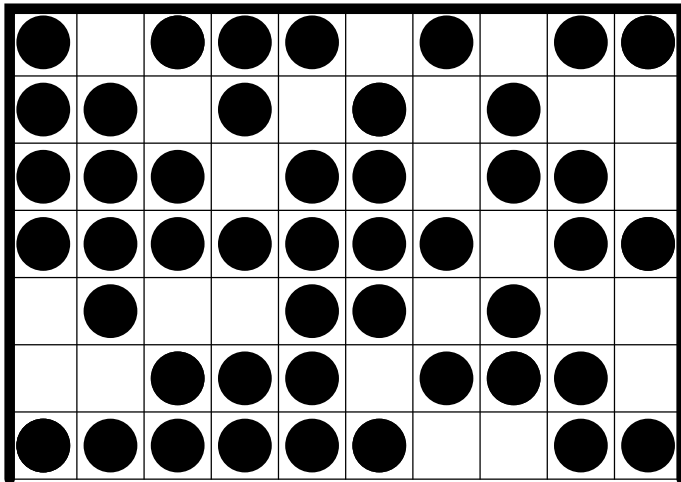


- From a given starting square, one can move down one level and up to one column left or right on each step, as long as the square moved to is unoccupied.
- Problem is to find a path to the bottom layer.
- Diagram shows one path that runs into a dead end and one that escapes.

Path-Finding Program

- Translating the problem into a function specification:

```
def is_path(blocked, x0, y0):  
    """True iff there is a path of squares from (X0, Y0) to some  
    square (x1, 0) such that all squares on the path (including ends)  
    are unoccupied.  BLOCKED is a predicate such that BLOCKED(x, y)  
    is true iff the grid square at (x, y) is occupied. Each step of a  
    path goes down one row and 1 or 0 columns left or right."""
```



This grid would be represented by a predicate M where, e.g, $M(0,0)$, $M(1,0)$, $M(1,2)$, not $M(1, 1)$, not $M(2,2)$.

Here, $is_path(M, 5, 6)$ is true; $is_path(M, 1, 6)$ and $is_path(M, 6, 6)$ are false.

is_path Solution

```
def is_path(blocked, x0, y0):
    """True iff there is a path of squares from (X0, Y0) to some
    square (x1, 0) such that all squares on the path (including ends)
    are unoccupied. BLOCKED is a predicate such that BLOCKED(x, y)
    is true iff the grid square at (x, y) is occupied. Each step of a
    path goes down one row and 1 or 0 columns left or right."""

    if _____:

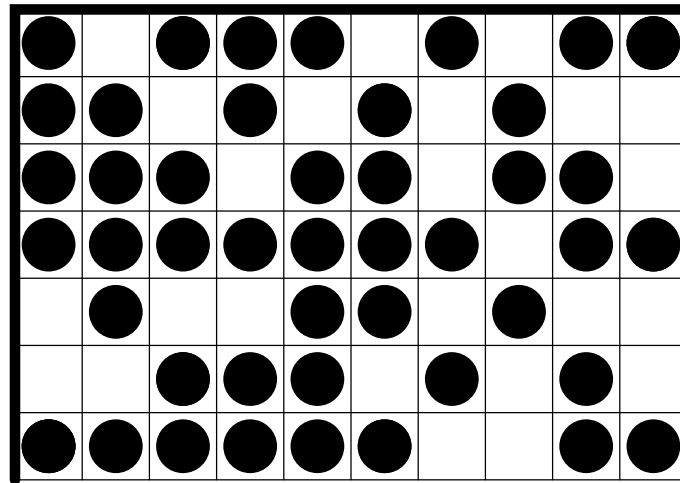
        return _____
    elif _____:

        return _____
    else:

        return _____
```

Variation I

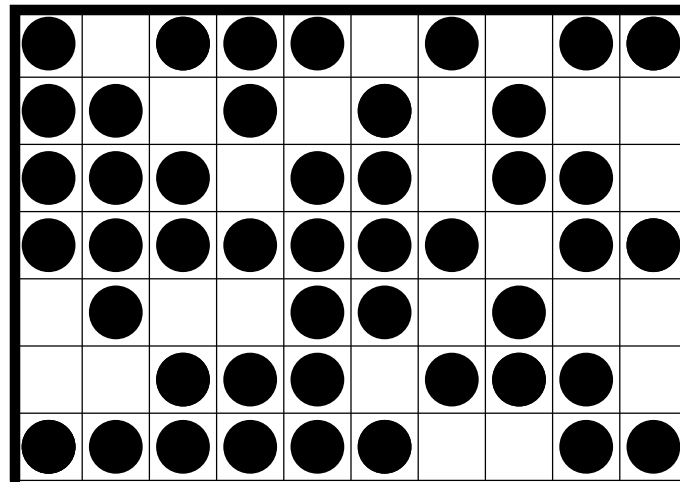
```
def num_paths(blocked, x0, y0):  
    """Return the number of paths that run from  
    (X0, Y0) to some unoccupied square (x1, 0).  
    BLOCKED is a predicate such that BLOCKED(x, y) is  
    true iff the grid square at (x, y) is occupied. """
```



Result of `num_paths(M, 5, 6)` is 1 (original M) If M2 is the maze above (missing (7, 1)), then result of `num_paths(M2, 5, 6)` is 5.

Variation II

```
def find_path(blocked, x0, y0):  
    """Return a string containing the steps in a path  
    from (X0, Y0) to some unoccupied square (x1, 0),  
    or None if not is_path(BLOCKED, X0, Y0). BLOCKED is a  
    predicate such that BLOCKED(x, y) is true iff the  
    grid square at (x, y) is occupied. """
```



Possible result of `find_path(M, 5, 6)`:

```
"(5, 6) (6, 5) (6, 4) (7, 3) (6, 2) (5, 1) (6, 0)"
```

find_path Solution

```
def find_path(blocked, x0, y0):  
    """Return a string containing the steps in a path  
    from (X0, Y0) to some unoccupied square (x1, 0),  
    or None if not is_path(BLOCKED, X0, Y0). BLOCKED is a  
    predicate such that BLOCKED(x, y) is true iff the  
    grid square at (x, y) is occupied. """
```

A Change in Problem

- Suppose we changed the definition of “path” for the maze problems to allow paths to go left or right without going down.
- And suppose we changed solutions in the obvious way, adding clauses for the $(x_0 - 1, y_0)$ and $(x_0 + 1, y_0)$ cases.
- Will this work? What would happen?

And a Little Analysis

- All our linear recursions took time proportional (in some sense) to the size of the problem.
- What about `is_path`?