Public-Service Announcement

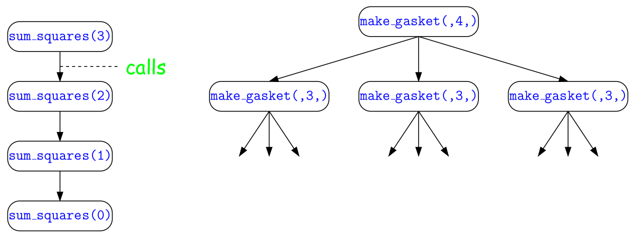
"Autofocus is Berkeley's first mobile photography club. Join us as we build a community of phone photographers at Cal. All you need to be part is an interest in photography and a mobile phone!

Infosessions on 2/2 and 2/7. Details at `tiny.cc/autofocus`"

Lecture #7: Tree Recursion

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* makes at most one recursive call per call.
- A *tail recursion* has at most one recursive call per call, and it is the last thing evaluated.
- A linear recursion such as for `sum_squares` produces the pattern of calls on the left, while `make_gasket` produces the pattern on the right—an instance of what we call a *tree* in computer science.



What About This?

What kind of recursion is this?

```
def find_it(f, y, low, high):
    """Given that F is a nondecreasing function on integers,
    find a value of x between LOW and HIGH inclusive such that
    F(x) == Y.  Return None if there isn't one."""

    if low > high:
        return None
    mid = (low + high) // 2
    val = f(mid)
    return val == y \
           or (val < y and find_it(f, y, low, mid-1)) \
           or (val > y and find_it(f, y, mid+1, high))
```

What About This?

What kind of recursion is this? Tail Recursion

```
def find_it(f, y, low, high):
    """Given that F is a nondecreasing function on integers,
    find a value of x between LOW and HIGH inclusive such that
    F(x) == Y.  Return None if there isn't one."""

    if low > high:
        return None
    mid = (low + high) // 2
    val = f(mid)
    return val == y \
           or (val < y and find_it(f, y, low, mid-1)) \
           or (val > y and find_it(f, y, mid+1, high))
```

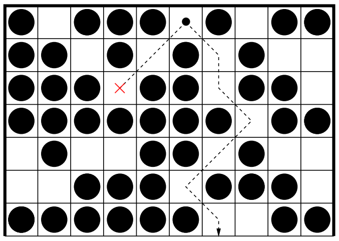
What About This?

What kind of recursion is this? Tree Recursion

```
def find_it(f, y, low, high):
    """Given that F is a nondecreasing function on integers,
    find a value of x between LOW and HIGH inclusive such that
    F(x) == Y.  Return None if there isn't one."""

    if low > high:
        return None
    mid = (low + high) // 2
    val = f(mid)
    return val == y \
           or (val < y and find_it(f, y, low, mid-1)) \
           or (find_it(f, y, mid+1, high))
```

Finding a Path

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

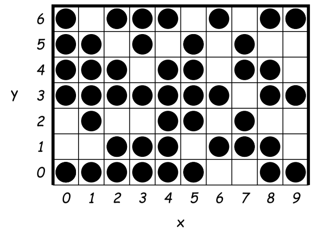


- From a given starting square, one can move down one level and up to one column left or right on each step, as long as the square moved to is unoccupied.
- Problem is to find a path to the bottom layer.
- Diagram shows one path that runs into a dead end and one that escapes.

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 first and
    last) are unoccupied.  BLOCKED is a predicate such that BLOCKED(x, y)
    is true iff the grid square at (x, y) is occupied or off the edge.
    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 (I)

```
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 first and
    last) are unoccupied.  BLOCKED is a predicate such that BLOCKED(x, y)
    is true iff the grid square at (x, y) is occupied or off the edge.
    Each step of a path goes down one row and 1 or 0 columns left or right."""
    if ____________:

        return ____________

    elif ____________:

        return ____________
    else:

        return __________________________________________________
```

is_path Solution (II)

```
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 first and
    last) are unoccupied.  BLOCKED is a predicate such that BLOCKED(x, y)
    is true iff the grid square at (x, y) is occupied or off the edge.
    Each step of a path goes down one row and 1 or 0 columns left or right."""
    if ____________:

        return False

    elif ____________:

        return True
    else:

        return __________________________________________________
```

is_path Solution (III)

```
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 first and
    last) are unoccupied.  BLOCKED is a predicate such that BLOCKED(x, y)
    is true iff the grid square at (x, y) is occupied or off the edge.
    Each step of a path goes down one row and 1 or 0 columns left or right."""
    if blocked(x0, y0):

        return False

    elif ____________:

        return True
    else:

        return __________________________________________________
```

is_path Solution (IV)

```
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 first and
    last) are unoccupied.  BLOCKED is a predicate such that BLOCKED(x, y)
    is true iff the grid square at (x, y) is occupied or off the edge.
    Each step of a path goes down one row and 1 or 0 columns left or right."""
    if blocked(x0, y0):

        return False

    elif y0 == 0:

        return True
    else:

        return __________________________________________________
```

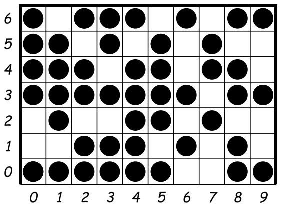
is_path Solution (V)

```
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 first and
   last) are unoccupied.  BLOCKED is a predicate such that BLOCKED(x, y)
   is true iff the grid square at (x, y) is occupied or off the edge.
   Each step of a path goes down one row and 1 or 0 columns left or right."""
   if blocked(x0, y0):

      return False

   elif y0 == 0:

      return True
   else:

      return is_path(blocked, x0-1, y0-1) or is_path(blocked, x0, y0-1) \
             or is_path(blocked, x0+1, y0-1)
```

Variation I

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

For the previous predicate `M`, the result of `num_paths(M, 5, 6)` is 1.
For the predicate `M2`, denoting this grid (missing (7, 1)):



the result of `num_paths(M2, 5, 6)` is 5.

num_paths Solution (I)

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

   if blocked(x0, y0):

      return ________________________

   elif y0 == 0:

      return ________________________
   else:

      return _________________________________________________
```

num_paths Solution (II)

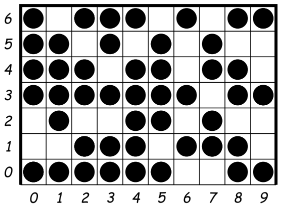
```
def num_paths(blocked, x0, y0):
   """Return the number of unoccupied paths that run from (X0, Y0)
   to some square (x1, 0). BLOCKED is a predicate such that BLOCKED(x, y)
   is true iff the grid square at (x, y) is occupied or off the edge. """

   if blocked(x0, y0):

      return 0

   elif y0 == 0:

      return 1
   else:

      return _________________________________________________
```

num_paths Solution (III)

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

   if blocked(x0, y0):

      return 0

   elif y0 == 0:

      return 1
   else:

      return num_paths(blocked, x0-1, y0-1) + num_paths(blocked, x0, y0-1)
             + num_paths(blocked, x0+1, y0-1)
```

Variation II

```
def find_path(blocked, x0, y0):
   """Return a string containing the steps in an unoccupied
   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 or off the edge. """
```



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 (I)

```
def find_path(blocked, x0, y0):
   """Return a string containing the steps in an unoccupied
   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 or off the edge. """

   if blocked(x0, y0):

       return ____________________

   elif y0 == 0:

       return ____________________
   else:

       return ________________________________________
```

find_path Solution (II)

```
def find_path(blocked, x0, y0):
   """Return a string containing the steps in an unoccupied
   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 or off the edge. """

   step = "({}, {})".format(x0, y0)
   # Alternative: step = str((x0, y0))
   if blocked(x0, y0):
       return None
   elif y0 == 0:
       return step
   else:
       return ________________________________________
```

find_path Solution (III)

```
def find_path(blocked, x0, y0):
   """Return a string containing the steps in an unoccupied
   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 or off the edge. """

   step = "({}, {})".format(x0, y0)
   if blocked(x0, y0):
       return None
   elif y0 == 0:
       return step
   else:
       p = find_path(blocks, x0-1, y0-1)
       if p is not None: return p + " " + step
       p = find_path(blocks, x0, y0-1)
       if p is not None: return p + " " + step
       p = find_path(blocks, x0+1, y0-1)
       if p is not None: return step + " " + p
       return None
```

find_path Solution (IV)

```
def find_path(blocked, x0, y0):
   """Return a string containing the steps in an unoccupied
   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 or off the edge. """

   step = "({}, {})".format(x0, y0)
   if blocked(x0, y0):
       return None
   elif y0 == 0:
       return step
   else:
       for x in (x0-1, x0, x0+1):
           p = find_path(blocks, x, y0-1)
           if p is not None: return p + " " + step
       return None
```

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?

And a Little Analysis

- All our linear recursions took time proportional (in some sense) to the size of the problem.
- What about `is_path`? Each call spawns up to three others, up to y0 "generations." That means the number of possible calls could be as many as 3^{y0}—exponential growth.