Lecture #7: Tree Recursion

Announcements

- Hog Contest and Hog Dice Design released today! Exercise your strategy- and artistic-design skills on the Game of Hog.
- Please fill out our Week 3 survey (Piazza note @500) to help us adjust the course effectively.
- There have been questions about what Python features one may use to complete the Hog project (among other things). Generally, you can get points for passing the tests by any means on the Python version used by the autograder. However, you may lose composition points as a result of straying into features we haven’t gotten to yet.
- You can sign up for the Berkeley Programming Contest on 11 February. We use this to choose teams for the ACM International Collegiate Programming Contest, the first round of which is in March. Next week’s contest will be entirely online, and will use the North American Qualifier contest. See Piazza post @536 for details and signup link.
- Ask questions on the Piazza thread for today’s lecture (@575).

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.

A Problem

Try to implement the following:

```python
def find_zero(lowest, highest, func):
    """Return a value v such that LOWEST <= v <= HIGHEST and
    FUNC(v) == 0, or None if there is no such value.
    Assumes that FUNC is a non-decreasing function from integers
to integers (that is, if a < b, then FUNC(a) <= FUNC(b)."
    if ??:
        return None
    ??
```
Try to implement the following:

```python
def find_zero(lowest, highest, func):
    """Return a value v such that LOWEST <= v <= HIGHEST and
    FUNC(v) == 0, or None if there is no such value.
    Assumes that FUNC is a non-decreasing function from integers
    to integers (that is, if a < b, then FUNC(a) <= FUNC(b)."
    
    if lowest > highest:    # Base Case
        return None
    elif func(lowest) == 0:
        return lowest
    else:
        # Inductive (Recursive) Case
        return find_zero(lowest + 1, highest, func)
```

What kind of recursion is this?
A Problem

Try to implement the following:

```python
def find_zero(lowest, highest, func):
    """Return a value v such that LOWEST <= v <= HIGHEST and
    FUNC(v) == 0, or None if there is no such value.
    Assumes that FUNC is a non-decreasing function from integers
    to integers (that is, if a < b, then FUNC(a) <= FUNC(b))."""
    if lowest > highest:
        return None
    elif func(lowest) == 0:
        return lowest
    else:
        return find_zero(lowest + 1, highest, func)
```

What kind of recursion is this?

Tail Recursion

```
# Equivalent iterative solution
while lowest <= highest:
    if func(lowest) == 0:
        return lowest
    lowest += 1
# If we get here, returns None
```

Problem, Take 2

Can make it faster by using the fact that the function is non-decreasing.

```python
def find_zero(lowest, highest, func):
    ...
    if lowest > highest:
        return None
    middle = (lowest + highest) // 2
    if func(middle) == 0:
        return middle
    elif func(middle) < 0:
        # Guess is too low, result must be > middle
        return middle
    ??
```

Problem, Take 2

Can make it faster by using the fact that the function is non-decreasing.

```python
def find_zero(lowest, highest, func):
    ...
    if lowest > highest:
        return None
    middle = (lowest + highest) // 2
    if func(middle) == 0:
        return middle
    elif func(middle) < 0:
        # Guess is too low, result must be > middle
        return middle
    ??
```
Problem, Take 2

Can make it faster by using the fact that the function is non-decreasing.

```python
def find_zero(lowest, highest, func):
    ...
    if lowest > highest:
        return None
    middle = (lowest + highest) // 2
    if func(middle) == 0:
        return middle
    elif func(middle) < 0:
        return find_zero(middle + 1, highest, func)
    else:
        # Guess is too high, result must be < middle
        return ??
```

What kind of recursion is this?

Tail Recursion:
Two calls, but only one executed.

Side Trip: Base Cases Without If

Can you do this without an if statement (just and/or)?

```python
def is_a_zero(lowest, highest, func):
    """Return true iff there is a value v such that LOWEST <= v <= HIGHEST
    and FUNC(v) == 0. Assumes that FUNC is a non-decreasing function
    from integers to integers."""
    middle = (lowest + highest) // 2
    if func(middle) == 0:
        return True
    else:
        return False
```
def is_a_zero(lowest, highest, func):
    """Return true iff there is a value v such that LOWEST <= v <= HIGHEST and FUNC(v) == 0. Assumes that FUNC is a non-decreasing function from integers to integers."""
    middle = (lowest + highest) // 2
    return lowest <= highest \
        and (func(middle) == 0 \
            or (func(middle) < 0 and is_a_zero(middle + 1, highest, func)) \
            or (func(middle) > 0 and is_a_zero(lowest, middle - 1, func)))

What kind of recursion is this?
Side Trip: Base Cases Without If

Can you do this without an if statement (just and/or)?

```python
def is_a_zero(lowest, highest, func):
    """Return true iff there is a value v such that LOWEST <= v <= HIGHEST
    and FUNC(v) == 0. Assumes that FUNC is a non-decreasing function
    from integers to integers.""
    middle = (lowest + highest) // 2
    return lowest <= highest \
        and (func(middle) == 0 \
            or (func(middle) < 0 and is_a_zero(middle + 1, highest, func)) \
            or (func(middle) > 0 and is_a_zero(lowest, middle - 1, func)))
```

What kind of recursion is this? Linear Recursion

Only one of the two calls to is_a_zero can happen, but if the first one evaluates to False, we still have to evaluate func(middle)>0. Thus the recursive call is not the last thing executed.

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 row 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 (X) and one that escapes.
Path-Finding Program

- Translating the problem into a function specification:

```python
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 __________
```

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)

```python
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)

```python
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)

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

**Counting the Paths**

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.

**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 num_paths(blocked, x0-1, y0-1) or num_paths(blocked, x0, y0-1) or num_paths(blocked, x0+1, y0-1)

**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 ____________

Last modified: Fri Feb 5 01:35:14 2021 CS61A: Lecture #7 29

Last modified: Fri Feb 5 01:35:14 2021 CS61A: Lecture #7 30

Last modified: Fri Feb 5 01:35:14 2021 CS61A: Lecture #7 29

Last modified: Fri Feb 5 01:35:14 2021 CS61A: Lecture #7 30
**num_paths Solution (II)**

```python
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)
```

**num_paths Solution (III)**

```python
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)
```

---

**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, so that instead
of just having recursive calls for the three squares
  \[(x_0 - 1, y_0 - 1), (x_0, y_0 - 1), \text{and } (x_0 - 1, y_0 + 1),\]
we added calls for the two other squares
  \[(x_0 - 1, y_0) \text{ and } (x_0 + 1, y_0).\]
- Will this work? What would happen?

**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, so that instead
of just having recursive calls for the three squares
  \[(x_0 - 1, y_0 - 1), (x_0, y_0 - 1), \text{and } (x_0 - 1, y_0 + 1),\]
we added calls for the two other squares
  \[(x_0 - 1, y_0) \text{ and } (x_0 + 1, y_0).\]
- Will this work? What would happen?

Infinite recursions, such as

\[(8, 2) \to (9, 2) \to (8, 2) \to \cdots\]
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 can spawn three others, for up to $y_0$ "generations." That means the number of possible calls could be as many as $3^{y_0}$—exponential growth.

Another Recursion Problem: Counting Partitions

• I’d like to know the number of distinct ways of expressing an integer as a sum of positive integer "parts."

• To make things more interesting, let’s also limit the size of the integer parts to some given value:

  ```python
def num_partitions(n, k):
    """Returns number of distinct ways to express N as a sum of positive integers each of which is <= K, where K > 0. (Empty sum is 0.)""
```

• Example:

  ```
  6 = 3 + 3
  = 3 + 2 + 1
  = 3 + 1 + 1 + 1
  = 2 + 2 + 2
  = 2 + 2 + 1 + 1
  = 2 + 1 + 1 + 1 + 1
  = 1 + 1 + 1 + 1 + 1 + 1
  so num_partitions(6, 3) is 7.
  ```

Identifying the Problem in the Problem

• Again, consider `num_partitions(6, 3)`.

• Some partitions will contain the maximum size integer, 3, and the rest won’t.

• Those that do contain 3 then have various ways to partition the remaining 3:

  ```
  3 + 3
  3 + 2 + 1
  3 + 1 + 1 + 1
  ```

• While those that do not contain 3 partition 6 using integers no larger than 2:

  ```
  2 + 2 + 2
  2 + 2 + 1 + 1
  2 + 1 + 1 + 1 + 1
  1 + 1 + 1 + 1 + 1 + 1
  ```

• These observations generalize, and lead immediately to a solution.
Counting Partitions: Code (I)

def num_partitions(n, k):
    """Number of distinct ways to express N as a sum of positive integers each of which is <= K, where K > 0. (The empty sum is 0.)""
    if n < 0:
        return 0
    elif k == 1:
        return 1
    else:
        return num_partitions(n - k, k) + num_partitions(n, k - 1)

Counting Partitions: Code (II)

def num_partitions(n, k):
    """Number of distinct ways to express N as a sum of positive integers each of which is <= K, where K > 0. (The empty sum is 0.)""
    if n < 0:
        return 0
    elif k == 1:
        return 1
    else:
        return num_partitions(n - k, k) + num_partitions(n, k - 1)

Counting Partitions: Code (III)

def num_partitions(n, k):
    """Number of distinct ways to express N as a sum of positive integers each of which is <= K, where K > 0. (The empty sum is 0.)""
    if n < 0:
        return 0
    elif k == 1:
        return 1
    else:
        return num_partitions(n - k, k) + num_partitions(n, k - 1)

Counting Partitions: Code (IV)

def num_partitions(n, k):
    """Number of distinct ways to express N as a sum of positive integers each of which is <= K, where K > 0. (The empty sum is 0.)""
    if n < 0:
        return 0
    elif k == 1:
        return 1
    else:
        return num_partitions(n - k, k) + num_partitions(n, k - 1)
Recurrences

- The partition problem is a typical example of a mathematical recurrence relation.
- A familiar one is the Fibonacci sequence, defined by
  \[ \text{fib}(n) = \begin{cases} 
  1, & \text{if } n \in \{0, 1\} \\
  \text{fib}(n-2) + \text{fib}(n-1), & \text{if } n > 1 
  \end{cases} \]
- Which of course translates immediately to:
  ```python
  def fib(n):
      if n == 0 or n == 1:
          return 1
      else:
          return fib(n-2) + fib(n-1)
  ```
- Giving us the sequence (for increasing values of \( n \))
  \[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \]
- Again, this is a tree recursion requiring an exponential amount of computation.
- But as we will see later, both here and in all the examples we've seen so far, dramatic speedup is possible.