CS 142
Recursion II

Announcements

Reminders:

- Program 3 now due **2/22** by 11:55pm
- Midterm 1 is now on Wed, Feb. 25th

isPalindrome Function

Write a function that takes in a string and returns True if that string is a palindrome and False otherwise.

Write the function recursively.

Examples:

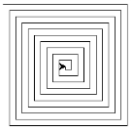
```
isPalindrome("civic") returns True
isPalindrome("123456") returns False
```


3 Laws of Recursion

- A recursive algorithm must have a **base case**.
- A recursive algorithm must change its state and move toward the base case.
- A recursive algorithm must call itself, recursively.

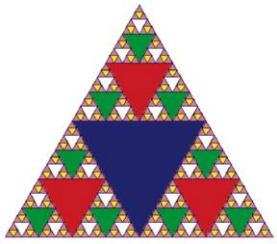
Visualizing Recursion

- Using Turtle Graphics to visualize recursion
- Base Case?
- Recursive Case?



Visualizing Recursion

Sierpinski Triangle - illustrates a three-way recursive algorithm

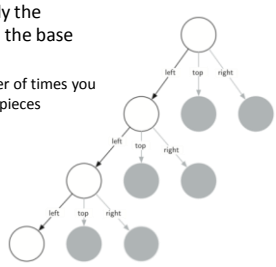


Algorithm for Drawing by Hand

1. Start with a single large triangle.
2. Divide this large triangle into four new triangles by connecting the midpoint of each side.
3. Ignoring the middle triangle that you just created, apply the same procedure to each of the three corner triangles.
4. Each time you create a new set of triangles, you recursively apply this procedure to the three smaller corner triangles.
5. You can continue to apply this procedure indefinitely if you have a sharp enough pencil.

Sierpinski Triangles

- Since we can continue to apply the algorithm indefinitely, what is the base case?
  - Base it on the degree, or number of times you want to divide the triangle into pieces
  - Now when do we stop?



Building a Sierpinski Triangle

Finding the Maximum

- Iterative Version
- Have a list called A. We want to find the maximum element (max() doesn't exist)

```
def findMax(A):
    biggest = A[0]
    for item in A[1:]:
        if item > biggest:
            biggest = item

    return biggest
```

Recursive Maximum

- Base Case: What is the smallest size list I would ever want to find the max element in?
- Recursive Case:
  - Suppose you have a list A (with >1 elements)
  - How can I describe finding the maximum element as involving ***finding the maximum element of a smaller sized list?***
  - Hint: Suppose my list has 5 elements. My best friend knows how to find the largest value in a list, but only for 4 elements. How can I use him to solve my problem?

Recursive Maximum

- findMax(A)
- Base Case: if len(a) == 1: return A[0]
- Recursive Case: if len(a) > 1:
  - Find the max element in A[1:] (whole list except A[0])
    - Call it M
  - if M > A[0]: return M
  - else: return A[0]

Recursive Maximum

- findMax(A)
- Base Case: if len(a) == 1: return A[0]
- Recursive Case: if len(a) > 1:
  - Find the max element in A[1:] (whole list except A[0])
    - Call it M
  - if M > A[0]: return M
  - else: return A[0]

A = [7, 9, 8]

Call findMax([7, 9, 8])
A = [7, 9, 8]
M = (recursive call)

Call findMax([9, 8])
A = [9, 8]
M = (recursive call)

Call findMax([8])
A = [8]
Base case!

Recursive Maximum

- findMax(A)
- Base Case: if len(a) == 1: return A[0]
- Recursive Case: if len(a) > 1:
  - Find the max element in A[1:] (whole list except A[0])
    - Call it M
  - if M > A[0]: return M
  - else: return A[0]

A = [7, 9, 8]

Call findMax([7, 9, 8])
A = [7, 9, 8]
M = (recursive call)

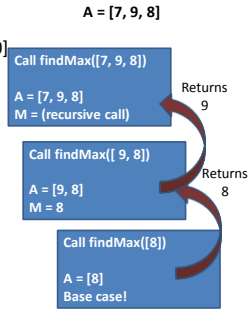
Call findMax([9, 8])
A = [9, 8]
M = (recursive call)

Returns 8

Call findMax([8])
A = [8]
Base case!

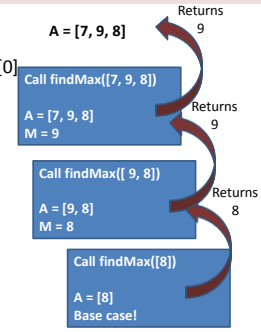
Recursive Maximum

- findMax(A)
- Base Case: if len(a) == 1: return A[0]
- Recursive Case: if len(a) > 1:
  - Find the max element in A[1:] (whole list except A[0])
    - Call it M
  - if M > A[0]: return M
  - else: return A[0]



Recursive Maximum

- findMax(A)
- Base Case: if len(a) == 1: return A[0]
- Recursive Case: if len(a) > 1:
  - Find the max element in A[1:] (whole list except A[0])
    - Call it M
  - if M > A[0]: return M
  - else: return A[0]



Writing the Recursive Version

- It's inefficient to takes slices of a list (creating multiple copies in memory), and can cause problems when returning indices
- Notice that our slices always involve chopping off the first element in the list A[0]
  - [7, 9, 8] -> [9,8] -> [8]
- How can we simulate an array slice without actually doing the slicing?
  - Hint: imagine reading a textbook. When you finish reading a page, you don't rip it out of the book, yet you want to be able to return to that place in the book later to study more. How do you solve this conundrum?

Recursive Version

- Use a "bookmark" to save your spot in the list
- When we make a recursive call, instead of passing an updated list, pass in an updated bookmark
- The function call will now be findMax(A, low)
  - low (the bookmark) represents the index of the bookmark in the list: everything before the bookmark is already read, everything afterwards is unread

Recursive Version

- findMax(A, low)
- Base Case: ?
- Recursive Case:
  - Find the maximum element in ???
    - Call it M
  - If ???: return M
  - else: return ??
- Where does the bookmark start?

Recursive Version

- findMax(A, low)
- Base Case: if low == len(A) -1
- Recursive Case:
  - Find the maximum element in everything after A[low]
    - M = findMax(A, low+1)
  - If M > A[low]: return M
  - else: return A[low]
- Initial call should be findMax(A, 0)

Binary Search

If you know that your list is ***sorted***, you can speed up the search from an O(n) (linear) to an O(log n) runtime using binary search.

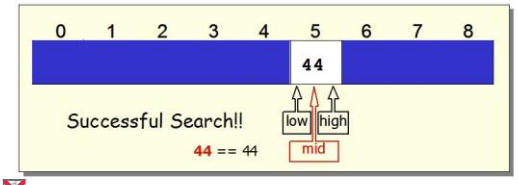
- Algorithm is similar to finding a name in the phone book
- Start in the middle
- Compare middle item with "thing"
- If "thing" is less, then search the first half; otherwise, search the second half.

Binary Search

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



Rhodes College

Iterative Binary Search

```
def binarySearch(lst, key):
    low = 0
    high = len(lst)-1
    while high >= low:
        mid = int((low + high) // 2)
        if key < lst[mid]:
            high = mid - 1
        elif key == lst[mid]:
            return mid
        else:
            low = mid+1
    return -1
```

Logarithmic-Time Algorithm since we continuously split our list in half.

Rhodes College

Binary Search

- Three variables that do most of the work:
  - low: the smallest index that could possibly contain the key
  - high: the largest index that could possibly contain the key
  - mid: the midpoint of low and high
- If low > high, we know the item is not found (stop).
- If A[mid] == key, item is found (stop)
- If A[mid] > key, repeat algorithm with only left half of the list
- If A[mid] < key, repeat algorithm with only right half of the list

Binary Search

- If low > high, we know the item is not found (stop).
- If A[mid] == key, item is found (stop)
- If A[mid] > key, repeat algorithm with only left half of the list
  - How do we change low & high?
- If A[mid] < key, repeat algorithm with only right half of the list
  - How do we change low & high?

Binary Search

- If low > high, we know the item is not found (stop).
- If A[mid] == key, item is found (stop)
- If A[mid] > key, repeat algorithm with only left half of the list
  - How do we change low & high?
  - high = mid - 1
- If A[mid] < key, repeat algorithm with only right half of the list
  - How do we change low & high?
  - low = mid + 1

Recursive Binary Search

Function: binarySearch(A, key, low, high)

- Base Cases:
  - Found key: return position found
  - low > high: return -1 (not found)
- Recursive Case:
- A[mid] > key: call function with updated high value
- A[mid] < key: call function with updated low value