CS 142
Recursion III

Rhodes College

Announcements

Reminder:

- Program 3 due Sun, Feb 22nd by 11:55pm
- Midterm 1 on Wednesday
- Review on Monday

Analyzing Recursive Functions

- We know that binary search is O(log n), but how do we determine that?
  - For iterative algorithms, we can simply count the number of iterations it takes to find the solution
    - Feed in the worst-case input, and count iterations to determine if it is equal to n, less than n or more than n
  - For recursive algorithms, the number of iterations is determined by the number of recursions.
    - We have to figure out how deep the stack of nested function calls will get.

Recursive Factorial

```
def factrec(n):
    if n <= 1:
        return 1
    else:
        return n * factrec(n - 1)
```

When n = 5, we call fact(5), fact(4), fact(3), fact(2), fact(1)

When n = 10, we call fact(10), fact(9), fact(8), fact(7), fact(6), fact(5), fact(4), fact(3), fact(2), fact(1)

What's the big-O run-time of recursive factorial?

Recursive Palindrome

```
def rec_isPalindrome(phrase):
    if len(phrase) <= 1:
        return True
    else:
        first = phrase[0]
        last = phrase[-1]
        rest = phrase[1:-1]
        return first == last and rec_isPalindrome(rest)
```

Let n = len(original phrase)
If n = 5, we call rec_isPalindrome with words of length 5, 3, 1
If n = 10, we call rec_isPalindrome with words of length 10, 8, 6, 4, 2, 0
If n = 20, we call rec_isPalindrome with words of length 20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0

What's the run-time of rec_isPalindrome?

Fibonacci Numbers

- As you'll see in your practice today, sometimes you can speed up the run-time of a function by using recursion.
- Calculating the n^{th} Fibonacci number is a classic example of a very inefficient recursive algorithm.

- Fibonacci sequence is the sequence of numbers 1, 1, 2, 3, 5, 8, ...
- Starts with two 1s, and successive numbers are the sum of the previous tow.
- We can write this iteratively using a loop, and always keep track of the previous 2 Fib numbers

Iterative Fibonacci Numbers

```
def loopFib(n):
       #pre: n > 0
       #returns the nth Fibonacci number
       curr = 1
       prev = 1
       for i in range(n-2):
              temp = curr
              curr = curr + prev
              prev = temp
       return curr
```

What is the big-O run-time of loopFib?

Recursive Fibonacci

Fibonacci sequence has an elegant recursive definition

$$fib(n) = \begin{cases} 1, & if\ n < 3 \\ fib(n-1) + fib(n-2), & otherwise \end{cases}$$

We can turn this recursive definition directly into a recursive function.

```
def recFib(n):
    if n < 3:
        return 1
    else:
        return recFib(n-1) + recFib(n-2)
```

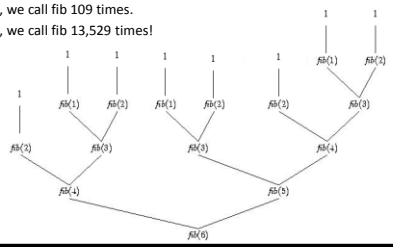
Analyzing Recursive Fibonacci

Demo – compare run-times

Exponential Runtime!!!

So why is recursive Fibonacci so much slower?

When n = 3, we call fib 3 times
When n = 6, we call fib 15 times
When n = 10, we call fib 109 times.
When n = 20, we call fib 13,529 times!



Conclusions

- Recursion is just one more tool in your problem-solving arsenal
- Sometimes a recursive solution is a good one
  - If it is more elegant
  - If it is more efficient
- If the iterative (loop) and recursive functions have similar run-times, use whichever you're more comfortable with
- If the recursive version is like Fibonacci, avoid it!
  - You'll learn later in your computer science career, ways to use recursion and avoid recalculating the same values over and over again.