Quiz Section 8

5/17/2018

Recursion

- Doug mentioned that decision trees can be constructed using a *recursive* algorithm

- What does that mean?

An example

How might you sort a large number of items?

I have 1000 index cards with numbers on them, and all of you, what's the easiest way to sort them?

An example

How might you sort a large number of items?

I have 1000 index cards with numbers on them, and all of you, what’s the easiest way to sort them?

Ok, I have an algorithm for you:

The merge sort algorithm

1. *Split your list into two halves*
2. *Sort the first half*
3. *Sort the second half*
4. *Merge the two sorted halves, maintaining a sorted order*

The merge sort algorithm

1. *Split your list into two halves*
2. *Sort the first half*
3. *Sort the second half*
4. *Merge the two sorted halves, maintaining a sorted order*

But now there are 500 cards in each pile…
If I knew how to sort quickly, I wouldn't be here in the first place?!?

The merge sort algorithm

1. *Split your list into two halves*
2. *Sort the first half*
3. *Sort the second half*
4. *Merge the two sorted halves, maintaining a sorted order*

But now there are 500 cards in each pile…
If I knew how to sort quickly, I wouldn't be here in the first place?!?

Here's a crazy idea: let's use merge sort to do this

Let’s take a step back …

Factorial

- A simple function that calculates n!

```
# This function calculated n!
def factorial(n):
    f = 1
    for i in range(1,n+1):
        f *= i
    return f
```

```
>>> print factorial(5)
120
>>> print factorial(12)
479001600
```

Factorial

- But … there is an alternative **recursive** definition:

$$n! = \begin{cases} 1 & if \quad n = 0 \\ (n-1)! \times n & if \quad n > 0 \end{cases}$$

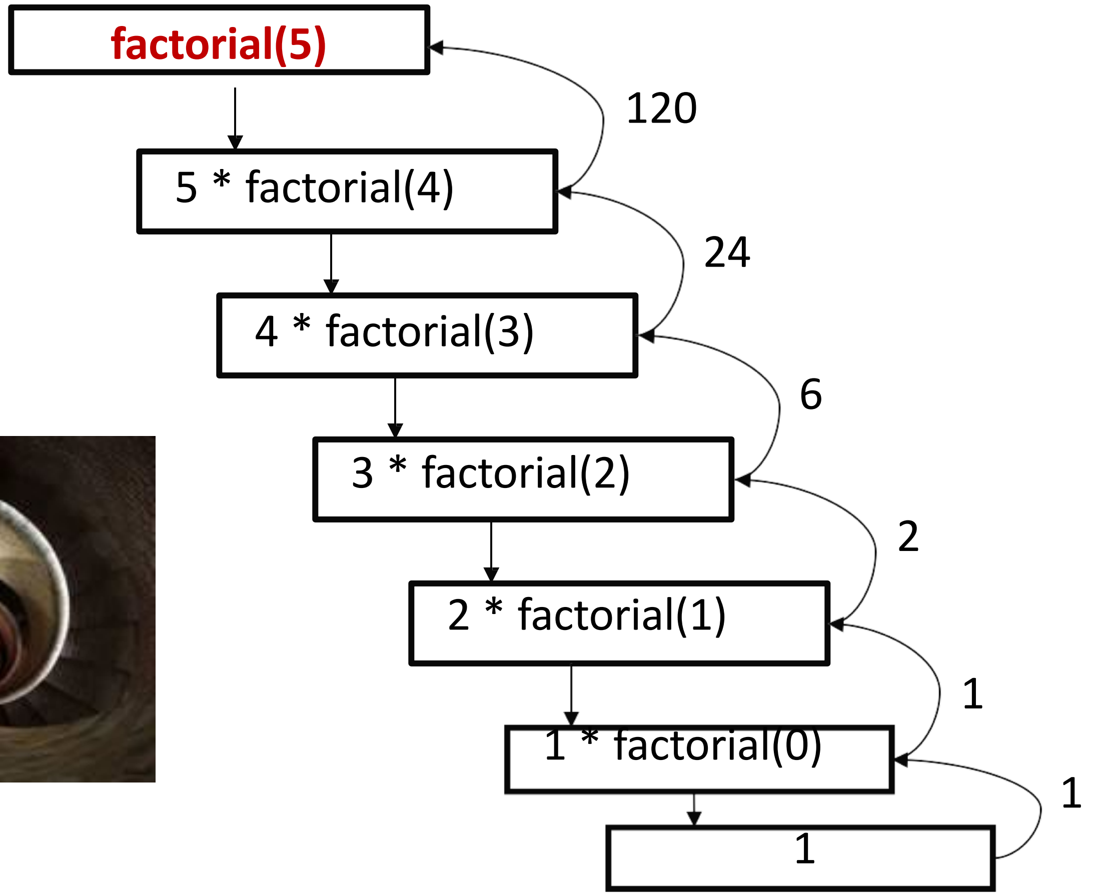
- So … can we write a function that calculates n! using this approach?

```
# This function calculated n!
def factorial(n):
    if n==0:
        return 1
    else:
        return n * factorial(n-1)
```

- **Well …**
  We can! It works! And it is called a *recursive* function!

Why is it working?

```
# This function calculated n!
def factorial(n):
    if n==0:
        return 1
    else:
        return n * factorial(n-1)
```



Recursion and recursive functions

- **A function that calls itself**, is said to be a **recursive** function (and more generally, an algorithm that is defined in terms of itself is said to use recursion or be recursive)

(A call to the function "recurs" within the function; hence the term "recursion")

- In may real-life problems, recursion provides an intuitive and natural way of thinking about a solution and can often lead to very elegant algorithms.

mmm…

- If a recursive function calls itself in order to solve the problem, isn't it circular?
  (in other words, why doesn't this result in an infinite loop?)

- Factorial, for example, is not circular because we eventually get to 0!, whose definition **does not rely** on the definition of another factorial and is simply 1.
  - This is called a **<u>base case</u>** for the recursion.
  - When the base case is encountered, we get a closed expression that can be directly computed.

Defining a recursion

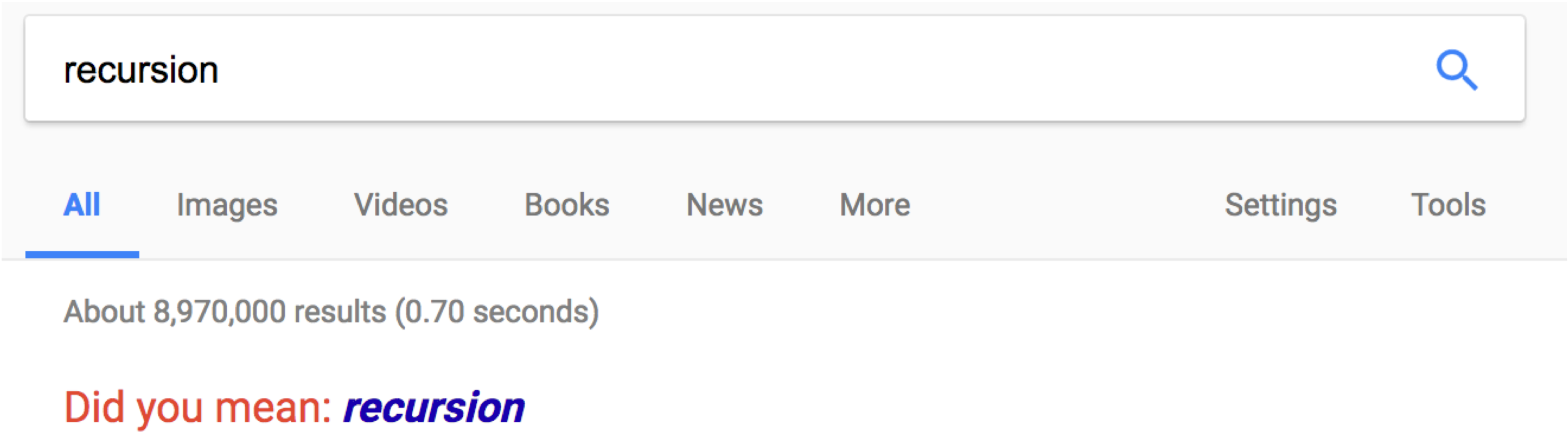
- Every recursive algorithm must have two key features:

  1. There are one or more ***base cases*** for which no recursion is applied.

  2. All recursion chains eventually end up at one of the base cases.

The simplest way for these two conditions to occur is for each recursion to act on a ***smaller*** *version of the original problem.*
A very small version of the original problem that can be solved without recursion then becomes the base case.

A bad computer scientist joke



What's wrong with this recursive "algorithm"?

Finally,
let’s get back to our merge sort

The merge sort algorithm

1. *Split your list into two halves*
2. *Sort the first half (**using merge sort**)*
3. *Sort the second half (**using merge sort**)*
4. *Merge the two sorted halves, maintaining a sorted order*

The merge sort algorithm

1. *Split your list into two halves*
2. *Sort the first half (**using merge sort**)*
3. *Sort the second half (**using merge sort**)*
4. *Merge the two sorted halves, maintaining a sorted order*

4 helper function

```
# Merge two sorted lists
def merge(list1, list2):
    merged_list = []
    i1 = 0
    i2 = 0

    # Merge
    while i1 < len(list1) and i2 < len(list2):
        if list1[i1] <= list2[i2]:
            merged_list.append(list1[ii])
            i1 += 1
        else:
            merged_list.append(list2[i2])
            i2 += 1

    # One list is done, move what's left
    while i1 < len(list1):
        merged_list.append(list1[i1])
        i1 += 1
    while i2 < len(list2):
        merged_list.append(list2[i2])
        i2 += 1

    return merged_list
```

```
# merge sort recursive
def sort_r(list):
    if len(list) > 1: # Still need to sort
        half_point = len(list)/2                      # 1
        first_half = list[:half_point]
        second_half = list[half_point:]

        first_half_sorted = sort_r(first_half)        # 2
        second_half_sorted = sort_r(second_half)      # 3

        sorted_list = merge \                         # 4
            (first_half_sorted, second_half_sorted)
        return sorted_list
    else:
        return list
```

List of size 1. Base case

Here's a puzzle: how to calculate the sum of a list (of any length) without for and while loops?

```
def sumList(list1):
    #List sum calculation here
```

```
my_list = [0,5,3,4,8]
```

Hint: One way to show this mathematically

sum = (0 + (5 + (3 + (4 + (8)))))

Recursion vs. Iteration

- There are usually similarities between an iterative solutions (e.g., looping) and a recursive solution.
  - In fact, anything that can be done with a loop can be done with a simple recursive function!
  - In many cases, a recursive solution can be easily converted into an iterative solution using a loop (but not always).

- Recursion can be very costly!
  - Calling a function entails overhead
  - Overhead can be high when function calls are numerous (stack overflow)

Recursion - the take home message

- **Recursion is a great tool to have in your problem-solving toolbox.**
- In many cases, recursion provides a natural and elegant solution to complex problems.
- If the recursive version and the loop version are similar, prefer the loop version to avoid overhead.
- Yet, even in these cases, recursion offers a creative way to **think** about how a problem could be solved.