CS61A Notes – Week 8: Recursion and tree recursion

We say a procedure is "recursive" if it calls itself in its body. Below is an example of a recursive procedure to find the factorial of a positive integer n:

```
def factorial(n):
    if n <= 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Upon first glance, it seems like this shouldn't work, since it doesn't make intuitive sense to define something in terms of itself. However, note that we do have one case that is simple: when n is 0. This case, called our **base case**, gives us a starting point for our definition. Now we can compute the factorial of 1 in terms of the factorial of 0, and the factorial of 2 in terms of the factorial of 1, and the factorial of 3… well, you get the idea.

Two simple steps to success:

1. **Figure out your base case** – Ask yourself, "what is the simplest argument someone could possibly give me?" The answer should be simple, and is often corresponds to the "smallest" input possible (i.e. n = 0 for factorial)
2. **Make a recursive call with a simplified argument** – Simplify your problem somehow, and assume that a recursive call for this new problem will just work. This takes a bit of faith, but as you get used to it, it'll be easier and easier. (For factorial we make the recursive call `factorial(n-1)`, and assume that we know how to solve it.) Now use the result of this recursive call to solve your problem.

Finally, here's one last bit of advice:

Trust the recursion.

EXERCISES

1. Write a procedure `expt(base, power)`, which implements the exponent function. For example, `expt(3, 2)` returns `9`, and `expt(2, 3)` returns `8`. Use recursion!

```
def expt(base, power):
    if power == 0:
        return 1
    elif power < 0:
        return 1 / expt(base, -power)
    else:
        return base * expt(base, power - 1)
```

2. Write a procedure `merge(s1, s2)` which takes two sorted (smallest value first) lists and returns a single list with all of the elements of the two lists, in ascending order. (*Hint*: if you can figure out which list has the smallest element out of both, then we know that the resulting merged list will be that smallest element, followed by the merge of the two lists with the smallest item removed. Don't forget to handle the case where one list is empty!).

Use recursion.

```
def merge(s1, s2):
    if len(s1) == 0:
        return s2
    elif len(s2) == 0:
        return s1
    elif s1[0] < s2[0]:
        return s1[0] + merge(s1[1:], s2)
    else:
        return s2[0] + merge(s1, s2[1:])
```

Multiple recursive calls at a time: Tree recursion

Say we had a procedure that requires more than one possibility to be computed in order to get an answer. A simple example is this definition for a function that computes Fibonacci numbers:

```
def fib(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

This is where recursion really begins to shine: it allows us to explore two different calculations at the same time. In this case we're exploring two different possibilities (or paths): the n – 1 case and the n – 2 case. With the power of recursion, exploring all possibilities like this is very straightforward. You simply try everything using recursive calls for each case, then combine the answers you get back.

We often call this type of recursion, where we use more than one recursive call to find an answer, **tree recursion**. We call it tree recursion because the expression tree that results from using a procedure like this looks like a tree! Here's a simplified expression tree for the expression `fib(4)`:

```
          fib(4)
          /   \
         /     \
      fib(3)  fib(2)
      /   \
     /     \
  fib(2)  fib(1)
```

Like before, we could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally fairly difficult to write iteratively, and require the use of additional data structures to hold information. As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion!

1. I want to go up a flight of stairs that has `n` steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a procedure `count_stair_ways` that solves this problem for me.

   ```
   def count_stair_ways(n):
       if n < 0:
           return 0
       elif n == 0:
           return 1 # don't move and that's a way to get up 0 stairs!
       else:
           return count_stair_ways(n – 1) + count_stair_ways(n – 2)
   ```

2. Consider the `subset_sum` problem: you are given a list of integers and a number k. Is there a subset of the list that adds up to k? For example:

   ```
       >>> subset_sum([2, 4, 7, 3], 5)  # 2 + 3 = 5
       True
       >>> subset_sum([1, 9, 5, 7, 3], 2)
       False
   ```

   ```
   def subset_sum(lst, k):
       if len(lst) == 0:
           return False
       elif k in lst:
           return True
       else:
           return subset_sum(lst[1:], k - lst[0]) +\
                  subset_sum(lst[1:], k)
   ```

3. We will now write one of the faster sorting algorithms commonly used, named Mergesort. Merge sort works like this:
   a. If there's only one (or zero) item(s) in the sequence, it's already sorted!
   b. If there's more than one item, then we can split the sequence in half, sort each half recursively, then `merge` the results (using the `merge` procedure from earlier in the notes). The result will be a sorted sequence.

   Using the described algorithm, write a function `mergesort(s)` that takes an unsorted sequence `s` and sorts it.

   ```
   def mergesort(lst):
        if len(lst) <= 1:
            return lst
        else:
            return merge(mergesort(lst[:len(lst)/2]),
                         mergesort(lst[len(lst)/2:]))
   ```

4. Pascal's triangle is a useful recursive definition that tells us the coefficients in the

expansion of the polynomial *(x + a)*^{n}. Each element in the triangle has a coordinate, given by the row it is on and its position in the row (which you could call a column). Every number in Pascal's triangle is defined as the sum of the item above it and the item above it and to the left (its position in the row, minus one). If there is a position that does not have an entry, we treat it as if we had a 0 there. Below are the first few rows of this triangle:

```
Item:    0   1   2   3   4   5  ...
Row 0:   1
Row 1:   1   1
Row 2:   1   2   1
Row 3:   1   3   3   1
Row 4:   1   4   6   4   1
Row 5:   1   5   10  10  5   1
...
```

Define the procedure `pascal(row, column)` which takes a row and a column, and finds the value at that position in the triangle. Don't use the closed-form solution, even if you know it!

```
def pascal(row, column):
    if column > row or row < 0 or column < 0:
        return 0
    elif column == 0:
        return 1
    else:
        return pascal(row - 1, column - 1) + pascal(row - 1, column)
```