Problem Set 3

Justin Ely

615.202.81.FA15 Data Structures

22 September, 2015

preface

My pseudo-code for most of these problems is very pythonic, and is often actual working python code. I don't mean to ignore the directive of producing pseudo code, but i assume the intent is to produce something abstract and easy-to-read. For many of these trivial examples, a straight python implementation seems to me to be sufficiently abstract and readable (often verbatim from the algorithm definition) while also giving me the ability to run and verify my thought-process.

1)

The definition of this recursive add algorithm is:

- $sum(a, b) = 1 + sum(a, b - 1)$ if $b > 0$
- $sum(a, b) = a$ if $b == 0$

In this case, when the second operand reaches 0, the trivial stopping case is reached and the recursion is halted. The implementation:

```
def recursive_add(a, b):
    if b == 0:
        return a
    else:
        return 1 + recursive_add(a, b-1)
```

2)

The definition of this recursive average algorithm is:

- avg(a) = $a[0]$ if $n_{elements} == 1$
- avg(a) = $\frac{a[0]+a[1]}{2}$ if $n_{elements} == 2$

- avg(a) $= \frac{a[n-1]+(n-1)*avg(a[0{:}n-1])}{n}$ if $n_{elements} > 2$

In this case, when the array is shrunk to just 2 elements (or a single element, if only 1 single element is initially input), the trivial stopping case is reached and the average of the two elements is returned. This implementation:

```
def recursive_avg(a):
    n = length(a)
    if n == 1:
        return a[0]
    elif n == 2:
        return (a[0] + a[1]) / 2.0
    else:
        return (a[n-1] + (n-1) * recursive_avg(a[0:n-1])) / n
```

3)

The sequence of calculations done to calculate a(2, 2) is shown below. From the last call, we can see that a(2, 2) eventually evaluates to 7, as the problem specifies. *I don't know of a better way to display the work in PDF form, as I can't render the tree structure here, but I can provide a scan of my scratch paper if necessary.*

```
a(2, 2) = a(1, a(2, 1))
a(2, 1) = a(1, a(2, 0))
a(2, 0) = a(1, 1)
a(1, 1) = a(0, a(1, 0))
a(1, 0) = a(0, 1)
a(0, 1) = 2
a(0, 2) = 3
a(1, 3) = a(0, a(1, 2))
a(1, 2) = a(0, a(1, 1))
a(1, 1) = a(0, a(1, 0))
a(1, 0) = a(0, 1)
a(0, 1) = 2
a(0, 2) = 3
a(0, 3) = 4
a(0, 4) = 5
a(1, 5) = a(0, a(1, 4))
a(1, 4) = a(0, a(1, 3))
a(1, 3) = a(0, a(1, 2))
a(1, 2) = a(0, a(1, 1))
a(1, 1) = a(0, a(1, 0))
a(1, 0) = a(0, 1)
```

```
a(0, 1) = 2
a(0, 2) = 3
a(0, 3) = 4
a(0, 4) = 5
a(0, 5) = 6
a(0, 6) = 7
```

4)

The binary search algorithm has log(n) performance, but since the question is asking for how many recursive calls, the first call will not be recursive. This means the worst performance will be log(n-1), and since an even number of calls need to be made this will be the rounded down to the integer value.

5)

The GCD definition has one stopping case and two recursive branches, where the stopping case is reached when y $<$ x and y divides evenly into x.

```
def gcd(x, y):
    if (y <= x) and (x%y == 0):
        return y
    elif (x < y):
        return gcd(y, x)
    else:
        return gcd(y, x%y)
```

6)

Two stopping cases are present in this definition, if n == 0 and if n == 1.

```
def gfib(f0, f1, n):
    if n == 0:
        return f0
    elif n == 1:
        return f1
    else:
        return gfib(f0, f1, n-1) + gfib(f0, f1, n-2)
```

7)

The stopping case for the recursive version of this algorithm is when the f() function returns False. At this point in the iterative procedure, the while loop will terminate and leave i unchanged. If f() still returns true, then the recursive call to g() is performed. This can translate to a recursive function as:

```
def recur(n)
    if f(n) == False:
        return n
    else:
        return recur(g(n))
```

8)

Using the previous problem as a template, a while loop looks to be a good iterative solution to this recursive call. The termination (of both the recursion and the loop) is if f() evaluates to True. If it evaluates to True, then the recursion is performed or the loop progresses. An iterative implimenation is given by:

```
def iter(n)
    i = n        # so as not to change n itself
    while f(i) == False:
        i = g(i)
    return i
```