IE 172 Laboratory 4: The Nature of Recursion

Dr. T.K. Ralphs

Due March 3, 2016

1 Laboratory Description and Procedures

1.1 Learning Objectives

1. Understand each of the key terms listed below.
2. Understand how to implement and use the queue and priority ADT.
3. Understand how to write a client application using the interface to a data type.
4. Understand how to implement a basic simulation using the `random` package.

1.2 Key Words

1. recursion
2. stack
3. seed
4. randint

1.3 Scenario

Many natural processes can be effectively modeled with recursion. We saw in the very beginning of the course that the Fibonacci sequence, which represents the way a population grows over several generation, has a recursive definition and can be computed with a recursive algorithm. Similarly, the process of cell growth can be modeled with a branching recursion where the "branching" arises from the splitting a single cell into two cells. In this laboratory, you will analyze how recursion can be used to draw a realistic looking tree. Here, branching has a literal interpretation in that the "branching" of the tree will directly correspond to the branching in the recursion.

The book shows a very simple recursive method for drawing a tree in Figure 4.6, reproduced here:

```
def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-10,t)
```

```
t.right(20)
t.backward(branchLen)
```

The idea is to start by drawing a straight line representing the trunk of the tree. At the end of the line, we attach two shorter lines (branches) that emerge at different angles so as to form a "Y" shape with the original line. We then recursively apply the same procedure to each of these new branches. By giving the branches shorter lengths each time we recurse one level deeper and stopping once they get below a certain length, we end up with a tree that looks reasonably realistic.

2 Program Specification

The `trees.py` file you will have been provided with contains an object-oriented implementation of the basic tree-drawing algorithm from the book. Your job will be to make the tree more realistic through the use of randomization and some color changes. You will need to familiarize yourself with the `randint` command, as well as the commands for using Python's turtle graphics package.

You will also be asked to make similar modifications to a non-recursive version of the drawing function. The implementation uses the stack implementation of Lab 3 again to keep track of the drawing jobs currently waiting to be done. Instead of the recursive calls, we instead put some information on the stack to "remember" what has to be done. There is then a while loop that continuously pops tasks off the stack and performs them, adding new ones as needed. Once there are no more tasks on the stack, the function exits.

3 Laboratory Assignments

3.1 Programming (30 points)

1. (10 points) Change the program so that as the branches get shorter, their thickness is also reduced, and they are finally colored green like leaves (hint: use the `pensize` and `color` methods).
2. (10 points) Change the program so that branch length is reduced by a random amount instead a fixed amount (hint: use the `randint` method).
3. (10 points): Modify the `draw_nr` method so that it draws the tree in reverse order.

3.2 Analysis (10 points)

1. (10 points) Explain the non-recursive implementation line by line and explain why it draws the same tree as the recursive implementation.

3.3 Follow-up Questions (40 points)

1. (10 points) Solve each of the following recurrences, assuming that $T(n)$ is constant for sufficiently small values of n. Explain how you got your answer.

   (a) (5 points) $T(n) = 2T(n/2) + n^3$
   (b) (5 points) $T(n) = 2T(n/4) + \sqrt{n}$
   (c) (5 points) $T(n) = T(\sqrt{n}) + 1$ (This one is tough! Use telescoping)
   (d) (5 points) $T(n) = 4T(n/2) + n^2\sqrt{n}$

2. (10 points) Consider the following recursive sorting algorithm:

```
def sort(list, i, j):
    if list[i] > list[j]:
        list[j], list[i] = list[i], list[j]
    if i + 1 >= j:
        return
    k = (j - i + 1)/3
    sort(list, i, j - k)
    sort(list, i + k, j)
    sort(list, i, j - k)
```

   What is the running time of the above algorithm (hint: use the Master Theorem)? Explain your logic.

3. (Extra Credit 10 points) Suppose you are given a list of numbers and told it is *unimodal*, i.e., the values increase until some "peak" value and then decrease again after that. Describe a recursive algorithm for finding the "peak value" and analyze its running time. Your algorithm should be as efficient as possible.