CISC-235

Assignment 2

Pseudo-code Solutions

Here are tips and pseudo-code solutions for the four problems. Please use these to help students figure out what to do – Problem 2 in particular seems to be challenging them. It's ok to guide them strongly on this one.

There are lots of equivalent stack-based solutions to these problems.

Function 1:

This recursive function prints the famous Collatz sequence, starting at any given positive integer n. (Actually it prints the sequence in reverse order.)

```
F(n):                                # n is an integer
    if n > 1:
        if  n % 2 == 0:
            F(n/2)                   # integer division
        else:
            F(3*n + 1)
    print n
```

Show the results of running the original F(n) and your non-recursive function on all values in the set $\{7, 18, 19, 22, 105\}$

Hint: In each of these problems we use the stack to hold a set of tasks that need to be completed in the future. We push them on the stack in the correct order so that the very last thing we have to do is at the bottom of the stack.

Hint: In this particular problem we make one of two possible recursive calls, and then print n. Since the print step has to be deferred, that is what we put on the stack. Since all of the deferred operations are "print", we can simply print each value as it comes off the stack.

Hint: some students use floating point division instead of integer division. This is probably ok in this problem but will cause trouble in Problem 2

Pseudo-code:

```
F1(n):
     S = new Stack()
     S.push(n)

     while n > 1 :
          if  n % 2 == 0:
               n = n/2
          else:
               n = 3*n + 1
          S.push(n)

     while not S.isEmpty():
          print S.pop()
```

Function 2:

```
F(n):                                    # n is an integer
     if n >= 6:
          F(n/3)                         # integer division
          F(2*n/3)                       # integer division
     print n
```

Show the results of running the original F(n) and your non-recursive function on all values in the set $\{7, 18, 19, 22, 43\}$

*Hint: in this problem we either just print the value, or we make two recursive calls and then print the value. We need the stack in the second situation: we need to execute F(n/3), then F(2*n/3), then print n. We use the stack to store the "F(2*n/3)" task and the "print n" task. With those saved for later, we can go ahead with the F(n/3) task. And executing the F(n/3) task involves either printing n/3 (if it is < 6) or executing F(n/3/3), F(2n/3/3), print n/3. We do this by pushing two of these onto the stack and going ahead with F(n/3/3). Clearly this pattern of pushing two tasks onto the stack and going ahead with the first task continues until the value is small enough to just print it. At that point we go back to the stack to retrieve the most recent task that we deferred.*

Hint: We can think about what happens when we pop an instruction off the stack. If it is a print command, we just do that. But if it is an "F(x)" command we need to work a bit harder: if x is < 6, we just print it. But if x >= 6, we have to do the same as described above: push two subtasks onto the stack, and go ahead with the first subtask.

Hint: We will talk about how to store instructions on the stack after we get the algorithm logic sorted out.

First draft of pseudo-code

```
F(n):
     S = new Stack()
     S.push("F(n)")      # assuming we can push instructions onto
                         # the stack somehow
     while not S.isEmpty():
          command = S.pop()
          if command == "F(x)":
               while x >= 6:
                    S.push("print x")
                    S.push("F(2*x/3)")
                    x = x/3
               print x
          else:
               # we know command == "print x"
               print x
```

or

```
F(n):
     if n < 6:
          print n
     else:
          S = new Stack()
          S.push("print n")
          S.push("F(2*n/3)")
          current = n/3
          while True:
               while current >= 6:
                    S.push("print current")
                    S.push("F(2*current/3)")
                    current = current/3
               print current
               if S.isEmpty():
                    return
               command = S.pop()
               while command == "print(x)":
                    print x
                    if S.isEmpty():
                         return
                    else:
                         command = S.pop()
               # now command must be of the form "F(x)"
               current = x
```

This solution is less elegant but still valid

Some students may want to push all three subtasks onto the stack. That's perfectly valid, and actually reduces the size of the code (although it actually does more work!) This method would look something like this:

```
F(n):
     S = new Stack()
     S.push("F(n)")
     while not S.isEmpty():
          command = S.pop()
          if command == "F(x)":
               if x < 6:
                    print x
               else:
                    S.push("print x")
                    S.push("F(2*n/3)")
                    S.push("F(n/3)")
          else:
               # it's a print command  "print x"
               print x
```

Hint: Now let's think about this idea of pushing instructions onto the stack. How can we do this when our stack class is designed to hold integers?

Here are two possible solutions. Both are based on using chosen integers as signals or flags to tell us what operations to carry out. One simple approach is to use 1 to signal "print" and 2 to signal "F()"

Proposal 1: Use two stacks, one to hold the operations (1 for "print", 2 for "F()") and the other to hold the values. We push and pop from the two stacks in sync.

The first pseudo-code solution becomes

```
F(n):
     SO = new Stack()
     SV = new Stack()
     SO.push(2)
     SV.push(n)
     while not SO.isEmpty():
          command = SO.pop()
          x = SV.pop()
          if command == 2:
               while x >= 6:
                    SO.push(1)
                    SV.push(x)
                    SO.push(2)
                    SV.push(2*x/3)
                    x = x/3
               print x
          else:
               # we know command == 1
               print x
```

Proposal 2: use a single stack, pushing each value and then its associated command. In this solution we always pop off the command, then the value.

```
F(n):
     S = new Stack()
     S.push(n)
     S.push(2)
     while not S.isEmpty():
          command = S.pop()
          x = S.pop()
          if command == 2:
               while x >= 6:
                    S.push(x)
                    S.push(1)
                    S.push(2*x/3)
                    S.push(2)
                    x = x/3
               print x
          else:
               # we know command == 1
               print x
```

Note: for Functions 3 and 4, you may want to redefine your Stack so that each element of the stack consists of a pair of integers.

Function 3:

```
F(a, b):                               # a and b are integers
      if  a <= b:
            m = (a+b)/2                # integer division
            F(a, m-1)
            print m
            F(m+1, b)
```

Show the results of running the original F(n) and your non-recursive function on these pairs of values $\{(0,7),(1,18),(4,19),(-1,22)\}$

Hint: now that we have seen how to solve Problem 2, this becomes a lot easier. We can adapt the same method: use two stacks, one for commands and one for values (pairs, in this case). That's completely satisfactory.

Hint: there is actually an easier way to manage this. By examining the given function, we can see that if a == b, the two recursive calls will do nothing because in each case the first argument is > the second argument. So when a == b, the only thing that happens is "print m" ... and m = (a+b)/2 = a in this case. This means that we can identify each subtask completely with just its (a,b) pair. If a < b then we simulate the deferred tasks by adding them to the stack. If a == b, we print a. If a > b, we do nothing. (We can also check for a > b before adding the pair to the stack.)

Hint: I'm going to assume our Stack class expects a pair of values for each push, and each pop returns an indexed object with a [0] and a [1] element

Pseudo-code:

```
F(a,b):

     if (a > b):
          return
     else:
          S = new Stack()
          S.push( (a,b) )
          while not S.isEmpty();
               pair = S.pop()
               x = pair[0]
               y = pair[1]
               if (x == y):
                    print x
               else:
                    m = (x+y)/2
                    if m+1 <= y :
                         S.push( (m+1, y) )
                    S.push( (m, m)   )
                    if x <= m-1 :
                         S.push( (x, m-1) )
```

The solution just shown does not push subtasks onto the stack if they will do nothing.

Function 4:

```
F(a, b):                              # a and b are integers
    if  a <= b:
        m = (a+b)/2                   # integer division
        F(a, m-1)
        F(m+1, b)
        print m
```

Show the results of running the original F(n) and your non-recursive function on these pairs of values $\{(0,7),(1,18),(4,19),(-1,22)\}$

Hint: this is a trivial modification of the solution to Problem 3

```
F(a,b):

    if (a > b):
        return
    else:
        S = new Stack()
        S.push( (a,b) )
        while not S.isEmpty();
            pair = S.pop()
            x = pair[0]
            y = pair[1]
            if (x == y):
                print x
            else:
                m = (x+y)/2
                S.push( (m, m)    )
                if m+1  <= y  :
                    S.push( (m+1 , y) )
                if x <= m-1  :
                    S.push( (x, m-1) )
```