Final CSE 294
Steven Sawtelle

Exercises:

1.

Version 1:
take in a bunch of numbers, split up by newline characters. the first float will represent an expected max value, and then an indeterminate amount of following floats will represent the price of silver on that many days. find the largest profit one could make by buying on one day and then selling on any subsequent day, and print this out such that the user sees the test case #, the expected max profit as inputted, and the actual max profit as calculated, with a final line declaring if the case passed or not, based on whether expected is equal to actual. do this until no more lines are read from stdin

Version 2:

main:
- main will be the primary driving class for this file
- init count as 0 to track test case count
- read in a line, and set the first value as the expected max
- parse the rest into a list, and send it to process(lis)
- use the value of this call and the first value from before to determine output for this test case. this will be count, then the expected max profit, then the actual max profit as calculated in process(lis), then a Boolean comparison on expected and max, with passed being true if the values are equal

process(lis):
- this function takes in a list and will find the max profit of the elements, with the assumption that they are chronologically ordered and the “buy” number will have to precede the “sell” number in order of indices
- iterate through the list and keep track of the minimum element so far, as well as the max profit based off of that minimum, which is only calculated when the current element is not the new minimum
- once the list has been fully processed, the max profit var will be returned

Version 3:
1. main:
    a. read in from stdin a line
    b. convert that line to a list of integers <- prices
    c. first int of prices <- expectedCost
    d. remove first element of prices
    e. process(prices) <- actualCost, actualMin, actualMax

    f. if the values are the same print out the appropriate prompt – example
       "Running test case 1
       Expected: Max pro t is 0.04
       Actual: Buy at 10.01, Sell at 10.05, Max pro t is 0.04 Test case passed"
        i. change "passed" to "FAILED" if they are not equal

2. process(lst):
    a. first element of lst <- minSoFar
    b. first element of list <- currentMax
    c. 0 <- maxProfit
    d. iterate through whole list <- x
        i. if x is less that minSofar set minSoFar to x
        ii. otherwise do x-minSofar and if that result is greater than maxProfit, replace maxProfit with it and replace currentMax with x
    e. return (maxprofit, minSofar, currentMax)

2. see attached maxprofit.py

3.
Version 1:
keep reading in input until none is left. first read in a line "begin". then for each line it reads in it will add each element of that line to the next place in a binary tree, with – added for null values. when end is read it will invert the binary tree and print out the elements, one row at a time. then go back to the "begin" check

Version 2:

main:
- main will be the primary driving class for this file
- operate through all lines of stdin
- read in a begin, then continue reading lines until end
- for each line, take all integers or - and add them in order to a binary tree
- pass the head node of this tree to proceed(lst)

proceed(lst):
- create a binary tree out of given list lst
- accomplish this by setting first value of list as head node and add it to a queue
- then set subsequent pairs of values as children of the pop of the queue and then add them to the queue as well
- do this until no values left
- call flip(root) to vertically invert the tree
- call printout(root) to print out the new values of the flipped tree

flip(root):

- recursive function
- base case is null root, return nothing here
- otherwise swap root.left and root.right by assigning them each others values
- then call flip on both of these children nodes

printout(root):

- set a list to the root
- then for each element in list print it and then add its children if they are valid to a new list
- set the original list to this new list, and keep iterating while this first list is not Null

Version 3*:

*one key understanding is that a TreeNode class will need to be created that will be able to track val, left, and right. This will be included in the final submission but would be superfluous to add here as it is a common class creation in Python

```
1. main:
    a. read in line from stdin until it reads begin
    b. [] <- fullList
    c. for every line the user puts in:
        i. if not going
            1. if the line is “begin”:
                a. set going to true
                b. clear fullList
        ii. otherwise
            1. if the line is “end”:
                a. set going to false
                b. procees(fullList)
            2. otherwise
                a. get all values as list and add each to fullList
2. proceed(lst):
    a. 0 <- count //basically a Boolean checker, swaps from 0 to 1
    b. new queue <- q
    c. new TreeNode of first element <- tempNode
    d. tempNode <- root
    e. add tempNode to q
    f. None <- tempNode
    g. for elements indexed 1…length of lst:
        i. new TreeNode of that element <- newTemp
        ii. add newTemp to q
        iii. if count==0:
            1. pop q <- tempNode
            2. 1 <- count
```

```
                3. newTemp <- left node of tempNode
            iv. otherwise
                1. 0 <- count
                2. newTemp <- right node of tempNode
        h. flip(root)
        i. printOut(root)
3. flip(root):
        a. it root is None return immediately
        b. left node of root <- temp
        c. right node of root <- left node of root
        d. temp <- right node of root
        e. flip(left node of root)
        f. flip(right node of root)
        g. return root
4. printout(root):
        a. list containing root <- thislevel
        b. while thislevel is not empty:
             i. [] <- nextlevel
            ii. for each element in thislevel <- n
                1. print value of element
                2. if element has left or right children nodes, append them to
                   nextlevel
           iii. nextlevel <- thislevel
```

4. see attached bintree-flip.py