Steven Sawtelle Honors Contract

CSE294 – Sp 17

1.

Version 1

- Program shall take in a number of test cases, and then for that many number of following lines get a string and a number. The string will contain any number of two characters – "+" for happy pancakes and "-" for sad pancakes. Using a figurative griddle that can flip the given number of pancakes at a time, it will work its way down the line and flip that many pancakes when seeing a sad pancake. If ever all pancakes are happy, it will print the current number of flips in this style: "Case 1: 3" where, and if it reaches the end of the pancake line without a fully happy lineup, it will change the number of flips to IMPOSSIBLE

Version 2

- main: this function will take in first the size of the inputs for this iteration. Then, that many times it will take in a line, split the line with the first element as a string and the second as an int, and then send those to flip(). Depending on the result of flip it will either print the success case or impossible case as shown in Version 1. It shall wrap the calls to casting and calling in try except blocks for general flexibility, as the casts are very specific and likely to crash when not properly formatted
- flip(string, int): this function will take in a string representing pancakes status and an int representing how many pancakes can be flipped simultaneously. if the string is all + then it will return 0, otherwise call doAPass() with a running string and the same int. if this call is executed as many times as the length of the string then we know it is impossible to make the pancakes all happy, so we return -1 to signify impossible. this is the sentinel the main class uses to signify impossibility
- doAPass(string, int): function will do one pass through a given string with a griddle that can flip the integer parameter number of pancakes at a time. at each character until the character at length of string – the integer parameter, it checks if it is a -. if it is the first - found then it will change the next integer parameter characters – if they are + to – and vice versa. if it is not - or not the first - then just add the character at that location from the string parameter. at this point the resulting string could be a number of different lengths, so fill in the rest with the contents at that location from the string parameter. finally return this many-times concatenated monstrosity

Version 3

1. main():
    a. get int <- inputsize

    b. for numbers in range [1, inputsize+1)
        i. read line <- line
        ii. split up line by spaces <- splitline
        iii. first element of splitline <- inputstring
        iv. second element of splitline <- inputnum
        v. flip(inputstring, inputnum) <- resultnum
        vi. if resultnum is not -1
            1. print success case from version 1
        vii. otherwise print impossible case
2. flip(inputstring, inputnum):
    a. 1 <- x
    b. inputstring <- result
    c. create length of inputstring '+' string <- target
    d. is result equals target return 0
    e. while x < length of inputstring + 1
        i. doAPass(result, inputnum) <- result
        ii. if result equal target return x
        iii. x++
    f. return -1
3. doAPass(inputstring, inputnum):
    a. "" <- newstring
    b. 0 <- x
    c. False <- flipped
    d. while x < length of inputstring -inputnum+1
        i. if current x at inputstring is - and flipped not True
            1. for each y from [x to x+inputnum)
                a. flip + to - and - to + and add to newstring
            2. x += inputnum
            3. True <- flipped
        ii. otherwise
            1. add current x at inputstring to newstring
    e. length of newstring <- leng
    f. length of inputstring - leng <- diff
    g. if diff not 0:
        i. for numbers in range [0, diff)
            1. add char at inputstring[leng+y) to newstring
    h. return newstring

2. see attached pancake.py

3.

Version 1

- Program shall take in a number of test cases, and then for that many number of following lines get a line contatining one integer. It will print out the first “tidy number” <= this number starting from this number and subtracting by 1, where “tidy” is a property where each digit is (not strictly) increasing from left to right. The output shall be in the forma “Case # /case number/: /closest tidy number/”

Version 2

- main(): shall be the driver class for this file. it will take in an integer representing number of test cases, then for that many lines read in an integer, pass it to makeTidy(), and print the result of that call.

- makeTidy(n): function to drive the tidiness operations. it will get all digits of n as a list for easy access, then call continuousLoop() with that list. after that it will reconstruct the list into an integer and return that integer as the final tidy number

- continuousLoop(splitvals): function to continually make numbers tidy until they are. what this means is that the easiest way to make a number tidy is to find the first spot where the digit is not less than or equal to the digit on its right, then decrement that digit and set all the other digits to 9. there is no guarantee that this will make it tidy though, so this for loop will be nested inside of a while loop whose condition is a call to checkTidy(splitvals)

- checkTidy(lst): function to check if a given list, when read from left to right, is tidy. executes a for loop over the list and checks if each element is > than that to its right. if it ever is, it returns false. otherwise it gets to the end and knows the list represents a tidy number, and returns True

Version 3

1. main():
    a. read in line as int <- inputSize
    b. for numbers in range [1, inputSize+1)
        i. line as int <- currentNum
        ii. makeTidy(currentNum) <- tidynum
        iii. print output of case as described in version 1
2. makeTidy(n):
    a. [] <- splitVals
    b. while n > 0
        i. n % 10 <- currentVal
        ii. add currentVal to splitVals
        iii. n / 10 <- n
    c. reverse splitVals
    d. 1 <- base
    e. 0 <- tidyNum
    f. for numbers in range [0, length of splitVals) <- x
        i. tidynum + (base * splitVals at x) <- tidynum
        ii. base * 10 <- base

        g. return tidynum
    3. continuousLoop(splitvals):
        a. while checkTidy(splitvals) returns False
            i. for x in range [0, length of splitvals -1)
                1. if splitvals at x > splitvals at (x + 1)
                    a. decrement splitvals at x
                    b. for numbers in range [x+1, length of splitvals) <- y
                        i. 9 <- splitvals at y
    4. checkTidy(lst):
        a. for numbers in range [0, length of lst -1)
            i. if lst at x > lst at (x+1)
                1. return False
        b. return True

4. see attached tidy.py