DATA 120 - Homework 5 - Spring 2024

Trimble

Due: Wednesday 2024-05-08 11:59pm

Preparing for Homework 5

Create an empty repostitory on github; put your function definitions into `pa5.py` and submit one of the branches of your repository to github.

0.0.1 Problem 1: GCD

Euclid devised an algorithm for finding the greatest common divisor of two natural numbers:

$$gcd(a, 0) = a$$

$$gcd(a, b) = gcd(b, a \bmod b)$$

where $a \geq b$ in each case; if the values are in the wrong order, swap them.

Implement a function, `gcd`, which takes in two integers and performs this formula. For credit, use recursion, not a loop, and follow the above approach.

0.0.2 Problem 2: Directions

Imagine we have a maze, like the kind sometimes found in puzzle books. The goal of the maze is for you to find a path from a starting point to the exit that does not require you to walk through any walls. We might think of the maze as being oriented with north being up, east being to the right, and so on.

Imagine someone has come up with a path through the maze, hopefully a path that solves the maze (gets from the starting point to the exit). This path could be recorded as a string that indicates a series of directions to walk, each of which for one unit of distance. For instance, the path "EESWS" says to start at a starting point, walk east for two units of distance, then turn south and move in that direction for one unit, then turn to the west and travel for one unit, finally turning south again and traveling for one unit.

Such a path might contain "U-turns". For example, the following direction string includes a venture eastward into a room, only to be followed by an immediate westward step, which undoes its previous step: "EEWSES".

There might be extended U-turns longer than two steps as well, as in this direction string: "EEEWWSES".

Note that both of these direction strings are equivalent to the following optimized solution: "ESES", in the sense that this shorter string will get to the same destination, but without some or all of the wasted U-turns. (Note that this function is not necessarily intended to provide the fully-optimized solution. For instance, per the rule mentioned below, "EEEWWSES" will only be improved to "EEWSES".)

Write the function `remove_pairs` to take in a direction string path, and return a direction string such that all "turnaround pairs" (emphasis on the word pairs) have been removed. For example, `remove_pairs("EEWN")` returns `"EN"`, `remove_pairs("SSNS")` returns `"SS"`, and `remove_pairs("ESNW")` returns `"EW"`. In other words, you should remove adjacent, redundant pairs, but not further remove new pairs that become adjacent because of the removal of other pairs in between them.

For credit, write a recursive solution that indexes and/or slices into the string and performs string concatenation as needed, but does not use string methods such as `replace`, and does not use any loops. There are various ways to approach this problem, but we want you to get practice with the recursive one in particular.

This definition does not uniquely define the behavior of `remove_pairs`: `"WEWE"` could be collapsed to `"WE"` or to `""`. We endeavor not to include even-odd-abmiguous cases among the test fixtures.

0.0.3 Problem 3: Bisection Method

The *Bisection method* is one simple way of finding a root (zero or x-intercept) of a function. It proceeds as follows:

1. We begin with a pair of x-values that "bracket" the root (in other words, they are on either side of the root, in the sense that the y-value of the function is positive for one of them an negative for the other). By the Intermediate Value Theorem, a continuous function must have a root somewhere in between these x-values.

2. If the y-value for either of these x-values is very close to zero (within 0.001 of it), we declare it (close enough to) a root and return it (the x-value) as the answer.

3. Otherwise, calculate a new x-value halfway between the two, splitting the interval in half. Compute the y-value for this new x-value. It will have the opposite sign from one of the two prior x-values. Choose this new x-value and the old one with the opposite sign as your new pair of x-values. Then repeat this process (go back to step 2).

4. Should the y-values for both of your x-values ever have the same sign, then this procedure has somehow failed; we cannot expect to find a root between two points on the same side of the x-axis. Raise a `ValueError` in this case.

Implement the function `bisection_root()` that takes in, in order, a function (which should have a single input, a number, and a single output, a number), and the two initial x-values, and returns the x-value of a root somewhere in between them. For instance, `bisection_root(math.sin, 2, 4)` should produce, approximately, pi.

This function will only work as well as the initial guesses, the behavior of the function being analyzed, and the method itself allow. You are not responsible for heroics, such as finding a root even when given bad initial guesses. Various nuances exist, such as the fact that there are indeed roots between two successive "peaks" of the sin() function, but these are bad guesses for this algorithm nonetheless since the y-values are both 1 for these guesses (and therefore it doesn't seem like there would be a root between them).