CS 120 (Fall 21): Introduction to Computer Programming II

Long Project #5

due at 5pm, Tue 28 Sep 2021

REMEMBER: The `itertools` and `copy` libraries in Python are **banned.**

1 Overview

In the Short Project, you started working with Linked Lists. The Long Project will be the same - but slightly more complex problems.

And yes, every "list" in this project refers to a linked list, **not** an array.

2 Basic Requirements

You will need to submit one Python file, named `linked_list_long.py` . It will contain five functions: `is_sorted()`, `list_sum()`, `partition_list()`, `accordion_4()`, and `pair()`.

2.1 Rule: No Creating or Changing `ListNode` Objects

Just like in the Short problem, you **must not** create any `ListNode` objects, or change their values, unless the problem **specifically** requires it.

(In this project, the only function that requires you to create new nodes is `pair()`.)

2.2 Rule: No Temporary Arrays

Sometimes, students are tempted to solve linked list problems by building a temporary array to hold their nodes (or the values). They then solve the problem using arrays, and finally convert things back to linked lists.

While there's nothing wrong with doing this in other situations, this practice is **banned** for this project. I want you to practice with **working with lists.** It's a critical skill!

2.3 Rule: List Class

All of the code in this project must import the file `list_node.py` , which I have provided. You **must not** modify this file. All of the list node objects that I provide you will be of the `ListNode` class, which is defined in that file.

Likewise, if you want to create any linked list objects, you must create them with ListNode. You can do so by calling the constructor, and passing it a value:

```
new_node = ListNode("abc123")
```

3 `is_sorted(head)`

This function takes a single parameter, which is a list, and returns `True` or `False`, indicating whether or not the values in the list are sorted (in ascending order). Note that duplicates may exist in the list; these count as being in order.

An empty list should return `True`, as should a list of length 1.

4 `list_sum(head)`

This function takes a single parameter, which is a list, where all of the values are numeric. It returns the sum of all of the values stored in the list.

An empty list should return zero.

5 `partition_list(head)`

This is a little like `split_list()` from the Short problem, except that, instead of splitting the list into two by cutting it into the middle, you will now build two lists to return, using alternate values.

The first value in the input list should be returned at the head of the first new list; the second value should be the head of the second list. Keep on alternating from there, putting one new value on the first list, and one on the second. (But remember that the length of the input list might be odd.)

Example

Suppose you have the following input list:

```
10 -> 13 -> -1 -> 1000 -> 0
```

It should return the following two lists:

```
10 -> -1 -> 0
13 -> 1000
```

6 `accordion_4(head, start_pos)`

This function takes a linked list, and removes three out of every four nodes; the first node that it returns is given by the `start_pos` parameter.

The `start_pos` gives the index of the first node to keep: if `start_pos` is zero, keep the head; if `start_pos` is one, then keep the node immediately after the head, and so on. If the list doesn't have enough nodes to satisfy the `start_pos` (including, of course, an empty list), then simply return an empty list. You may assume that `start_pos` is non-negative (meaning that zero is OK, and any positive number - but that negative numbers are not allowed).

After the `start_pos`, keep every fourth node, but discard the other two. For instance, if `start_pos` is 1, then keep the second, sixth, tenth nodes, and so on.

The surviving nodes should be linked together into a list; all of the other nodes are simply discarded (presumably to be garbage collected).

6.1 Example

If you have a linked list

```
10 -> 11 -> 17 -> 19 -> 23 -> 31 -> 37 -> 43 -> 53 -> 59
```

and `start_pos=2`, then you should return the following:

```
17 -> 37
```

7 `pair(list1, list2)`

This function takes **two** lists, which may be of different lengths (either or both of them might be empty). It builds new `ListNode` objects, where the value of each one is a tuple, consisting of one value from the first list, and one value from the second list, in the same order as they existed in those input lists.

If the lists are different lengths, then the length of the returned list will be equal to the length of the shorter one.

7.1 Special Requirement

This sort of problem generally requires you to add new nodes at the **end** of the list ("adding to the tail") instead of the simpler "adding at the head."

Some students decide to only keep the head pointer; they thus have to iterate through the entire list, every time that they want to add a new element. Can you explain why this sort of algorithm would run in $O(n^2)$ time?

A **far better** solution is to keep a second pointer, called a "tail pointer," which helps you keep track of where you're inserting new values. Then, the process of inserting many elements into the list takes $O(n)$ time. Can you explain why?

So here's the requirement: you **must not** iterate from the head every time you want to append to a list. Instead, you must use the tail pointer (or some equivalently-efficient mechanism). If you do the $O(n^2)$ insertion on this problem, then you will lose 10 points off your total score for this assignment.

8 Turning in Your Solution

You must turn in your code using GradeScope.

9 Acknowledgements

Thanks to Saumya Debray and Janalee O'Bagy for many resources that I used and adapted for this class.