CS545-01/CS245-02 Project 2: subtasks

How you approach this project is up to you. What I recommend below is just one of many possible ways to work on the project, you do not have to follow these suggestions.
Note: you may *not* modify signatures of methods in the starter code, but you may add additional / helper methods in class `CompactPrefixTree`.

Suggested steps:

1) Go over the project description and your notes from the class.

2) Look at the compact prefix tree example given in the description (that has the following words: ape, apple, cart, cat, cats, demon, demons, dog) and think about the following:

- How would you print all words stored in this tree?
- How you can check whether a particular word is in the tree? For instance, how would you check if "apple" is in the tree? How would you check whether "app" is in the tree? (it is not)
- How would you check to see if a given prefix is a prefix of some word in the dictionary? For instance, how would you check if "c" is a prefix of some word? How about "do" or "al"?

3) Click on the given link to get the starter code, and carefully look at the starter code. Don't hesitate to go the instructor or TAs for help.

4) Start coding by writing several helper methods in this class (without them, writing and/or testing other methods will be difficult).

- Write a method (let's say, we call it `getIndexOfCharacter`) that takes a character and returns its index in the array of node's children. Array has 26 cells, each corresponds to one character: index 0 to 'a', index 1 to 'b', index 2 to 'c' and so on. See the project description on how to compute this index. Test this method separately; you will need it for many methods.

- Write a helper method `findLongestCommonPrefix` that computes the longest common prefix of two strings. Test it thoroughly.

- Add a `temporary` method in class CompactPrefixTree where you construct a very simple prefix tree ***manually*** (so that you could write and test methods print, printTree, check, checkPrefix before you get the add method to work).
  To give you an idea of how this method should look like, here is the code that adds two children "ap" and "ca" to the root of the tree.

```
Node child1 = new Node();
child1.prefix = "ap";
root.children[getIndexOfCharacter('a')] = child1;

Node child2 = new Node();
child2.prefix = "ca";
root.children[getIndexOfCharacter('c')] = child2;
```

You can add children to child1 and child2 using the same approach. I recommend manually building the example from the description this way, so that you can debug your methods on it.
Note: this method is for debugging purposes only until you write (and test) the `add` method. You can remove it from your code once you write the add method.

5) Fill in code in the print method: `private void print(String s, Node node)` that prints all words stored in the tree. It should be recursive:

- if this node's valid bit is `true`, you should print `s+node.prefix` (since parameter `s` is the string obtained by concatenating all the prefixes on the path from the root to this node)
- iterate over all the children of this node, and call this method recursively on each child (think of what string to pass as a parameter!)

6) Fill in code in the `printTree` method that prints all nodes in the compact prefix tree, using indentations to show the hierarchy (see the subsection of the project description called "Printing the structure of the tree". Also see my lectures slides where I talked about printing a human readable tree). The method should be recursive.

7) Create a CompactPrefixTree in the Driver and add data to it by calling the method you wrote earlier that adds nodes manually. Call methods `print` and `printTree` on this tree and see if the result looks correct. Commit/push this version of the code to github.

8) Now write methods `check` and `checkPrefix`. Examples you did in 2) should help with that. Thoroughly test them on the tree you constructed manually. Commit/push this version of the code to github. Note: these methods should be resursive.

9) Do more examples on paper on how to add a new word into the tree. Think of it recursively, and identify all the different cases you need to consider when you insert a new word.

10) Start writing the code of "add". The method should be recursive. This method is the most interesting one of all the methods in this project, and might take you the longest to write (the code is not long, but the algorithm is interesting).

If you are confused about why this method returns a Node, look at the lecture on binary search trees + look at the code of insert (for binary search trees).

- ✓ Write code for simple cases:
  - When a new word is inserted into a null tree;
  - When a new word equals the prefix stored at the node
  - When a new word starts with the prefix stored at the root

- ✓ Test this method on these simple cases. Commit/push this version of the code to github.

- ✓ Move onto a more complex case, when you need to find the longest common prefix. Example: when the new word is "armor" and the prefix of the node is "ap". In this case, you need to find the longest common prefix between "armor" and "ap" and update the tree accordingly (see project description for details).

- ✓ Test the `add` method thoroughly on many examples. Only when it works, move onto suggest. Commit/push this version of the code to github.

11) Work on `suggest` and test it. Commit/push this version of the code to github.

12) Write other methods as needed (the constructor that takes a filename, the print method that prints to a file etc.). Commit/push the code to github.

13) Run the basic tests provided by the instructor; I recommend adding your own tests. Commit/push the code to github.

14) Clean up your code, test everything again, write a readme that explains how your suggest method works, and submit the code to github. Go to your private repo on github and make sure your most recent java code is there.

15) If there is time, try to make your `suggest` method better. Add the description of this project to you resume!