LAB 2 EXAM QUESTIONS

December 2020

Q1 - BALANCED BRACKETS

Write a function that will return all possible strings composed of n balanced square bracket pairs in a list, assuming that the string can only contain the characters `[` and `]`. The output list of possible strings should contain no duplicates. The order of the output strings does not matter for grading.

Hint: Think about how you can generate the string starting from scratch. At each step, you can choose to open a parenthesis and/or close a parenthesis depending on the number of parentheses you've opened/closed so far. Remember that you can use a helper with extra function parameters to keep track of the problem state with (opening parentheses and closing parentheses) counters.

Tip: You can use `x = list(set(x))` to remove duplicates from a list of strings `x` if you need to.

Example runs:

```
>>> balanced_brackets(0)
[]
>>> balanced_brackets(1)
['[]']
>>> balanced_brackets(2)
['[][]', '[[]]']
>>> balanced_brackets(3)
['[][][]', '[][[]]', '[[]][]', '[[[]]]', '[[]]][]']
>>> balanced_brackets(4)
['[[[[]]]]', '[[[]]]]', '[[[]]][]', '[[[]]]][]', '[[]][[]]', '[[]][]]', '[[]]][]',
'[[]][[]]', '[[]][]][]', '[][[[]]]', '[][[]]]', '[][[]][]', '[][]][]', '[][][][]']
```

Specifications:

- Use RECURSION. Iteration is not forbidden but will make solving the problem difficult. It can still be useful for some basic helpers.
- Feel free to define helper functions.
- Do not import any modules.
- Do not use the input() function.
- PLEASE REMOVE print() function calls before submitting your solution.

Solution:

```
def balanced_brackets(n):
    """
    n:  A non-negative integer, the number of balanced bracket pairs.
    """
    if n == 0:
        return []
    return balanced_brackets_helper(n, 0, 0, '')

def balanced_brackets_helper(n, op, cl, pars):
    if op == n and cl == n:
        return [pars]
    result = []
    if op < n:
        result += balanced_brackets_helper(n, op + 1, cl, pars + '[')
    if op > cl:
        result += balanced_brackets_helper(n, op, cl + 1, pars + ']')
    return result
```

Q2 - SMARTIO

Mario is once again ready to go on an adventure to save Princess Peach! However, he's gotten old now and is tired of getting injured every time... Instead, he scouted the vicinity of the castle he wants to reach and wants you to write a program to figure out how many different ways there are to reach the castle safely, just so he can feel safe.

The path scouted by Mario is provided as a list of strings, with each string denoting a tile along the path having a certain property:

- The first tile in the list will always be "start", the starting point of Mario's adventure.
- Somewhere in the list is a single "castle", which Mario wants to reach. Note that the path may contain other tiles after the castle.
- There might be some of Mario's beloved "mushroom"s on the path.
- There are also dangers along the path that Mario wants to avoid. These are "turtle shell"s, "spike"s, "hole"s and "bush"es. Why bushes? Well, he's just scared of vegetation after having faced so many carnivorous plants!
- Any other tile such as "pavement", "bridge", "r" or "q123x" is safe for Mario.

Now here's the deal: Mario is initially at the first tile, "start". At each step, he can either move one tile forward, or jump over one tile and move two tiles forward. Exceptionally, when Mario is on a "mushroom"tile, he gets hyped up and can also move three tiles forward with a long jump (jumping over two tiles), but can still choose to move only one or two tiles as usual. His goal is to reach the castle without stepping on any dangerous tiles, as specified above.

Your task is to write a function that takes the scouted path as an argument, and returns how many ways there are for Mario to reach the castle safely. If it is impossible for Mario to reach the castle without going through dangerous tiles, then the result should be zero.

Example runs:

Here are some example runs (with possible jump sequences shown as comments):

```
>>> num_safe_paths_to_castle(['start', 'castle'])
1 # start->castle = (1)
>>> num_safe_paths_to_castle(['start', 'road', 'castle'])
2 # start->road->castle or start->castle = (1) or (2)
>>> num_safe_paths_to_castle(['start', 'road', 'bush', 'castle'])
1 # start->road->castle = (1, 2)
>>> num_safe_paths_to_castle(['start', 'road', 'bush', 'road', 'road', 'castle'])
2 # (1, 2, 2) or (1, 2, 1, 1)
>>> num_safe_paths_to_castle(['start', 'road', 'bush', 'road', 'road', 'road', 'castle'])
3 # (1, 2, 1, 1, 1), (1, 2, 1, 2) or (1, 2, 2, 1)
>>> num_safe_paths_to_castle(['start', 'road', 'road', 'road', 'road', 'castle'])
8 # (1, 1, 1, 1, 1), (1, 1, 1, 2), (1, 1, 2, 1), (1, 2, 1, 1), (2, 1, 1, 1), (1, 2, 2), (2, 1, 2) or (2, 2
>>> num_safe_paths_to_castle(['start', 'bush', 'spike', 'castle'])
0 # None!
>>> num_safe_paths_to_castle(['start', 'mushroom', 'bush', 'spike', 'castle'])
1 # (1, 3)
```

Specifications:

- Use RECURSION. Iteration is not forbidden but will make solving the problem more difficult.
- Feel free to define helper functions.
- Do not import any modules.
- Do not use the input() function.
- PLEASE REMOVE print() function calls before submitting your solution.

Solution:

```
def num_safe_paths_to_castle(road):
    """
    Returns the number of different ways Mario can reach the castle safely. The
    answer should be zero if the castle cannot be reached safely.

    path:   A list of strings. Represents the tiles leading to and around the
            castle. The first element is always "start".
    """
    if road:
        tile = road[0]
        if tile == 'castle':
            return 1
        elif tile in ('bush', 'hole', 'turtle shell', 'spike'):
            return 0
        paths = num_safe_paths_to_castle(road[1:]) + num_safe_paths_to_castle(road[2:])
        if tile == 'mushroom':
            paths += num_safe_paths_to_castle(road[3:])
        return paths
    return 0
```