FIT1045 Algorithmic Problem Solving – Workshop 9.

Objectives

The **objectives of this workshop** are:

- To gain experience with stacks and queues.
- To devlop an in place partition algorithm

Task 1:

A bracket is considered to be any one of the following characters: (,), {, }, [, or]. Two brackets are considered to be a matched pair if the an opening bracket (i.e., (, [, or) occurs to the left of a closing bracket (i.e.,),], or) of the exact same type. There are three types of matched pairs of brackets: [], , and ().

A matching pair of brackets is not balanced if the set of brackets it encloses are not matched. For example, [(]) is not balanced because the contents in between { and } are not balanced. The pair of square brackets encloses a single, unbalanced opening bracket, (, and the pair of parentheses encloses a single, unbalanced closing square bracket,].

By this logic, we say a sequence of brackets is considered to be balanced if the following conditions are met:

- It contains no unmatched brackets.
- The subset of brackets enclosed within the confines of a matched pair of brackets is also a matched pair of brackets.

a) Write a function `basic_matching(str)` which takes as input a string containing only (and), and returns `True` if the string is a matching sequence and `False` otherwise.

b) Write a function `matching(str)` which takes as input a string containing any brackets, and returns `True` if the string is a matching sequence and `False` otherwise.

Note: You must use a stack for part b) of this task.

Examples:

- [()] - Matching
- [(]) - Not matching
- [[(())]] = Matching

Task 2:

Write a function `all_paths(M, u, v)` which takes as input an adjacency matrix of a graph, `M`, and two vertices `u, v` and returns a list of **all** paths from `u` to `v` in `M`. A path is a list of vertices. Note that a path cannot use the same vertex twice, but two different paths can use some of the same vertices. Use backtracking and recursion to solve this problem.
Example:

```
u=0
v=3
M = [[0, 1, 1, 0],
[1, 0, 1, 1],
[1, 1, 0, 1],
[0, 1, 1, 0]]
```

Paths: `[[0,1,3], [0,1,2,3], [0,2,3], [0,2,1,3]]`

Task 3 (Optional):

a) Write a function `partition(a_list)`. `partition` first selects an element from the input list to be the "pivot" element. In this exercise, we will always select the first element in the input to be the pivot. `partition` then reorders the list so that all elements less than the pivot come before the pivot in the list, and all elements greater than or equal to the pivot come after the pivot. The order of the elements before the pivot does not matter, and neither does the order of the elements after the pivot.

b) Rewrite the function from part 1 so that it does not create any additional lists. It simply moves the items around in the input list so that they end up in the correct order. If your function from part a) was already implemented this way, no need to do anything.

Example:
Input: [4, 7, 2, 3 ,8, 6, 1, 9]
Output: [{1, 2, 3} in any order, 4, {6, 7, 8, 9} in any order]