# Pythian Developer Challenge

## Summary

Pythian takes great pride in being not only at the forefront of cutting edge technologies, but also in engineering these technologies in the most novel and efficient manner. Our engineers strive for good design and architectures that scale, both in development and at runtime.

In this challenge, you will be given a series of questions that judge your ability to read, understand, debug and write solutions to programming problems. Your answers will be evaluated based on clarity in design, novelty, and efficiency.

You can expect the entire process to take 1-3 hours. The final product can be of any type of document, but we prefer a link to a Google document, a PDF or a Word document.

If you have any questions, please let us know and we will do our best to answer them.

## What We Evaluate

When we review the submissions for this challenge, we use the following criteria to determine the quality of the work.

|  |  |
| --- | --- |
| Skill | Description |
| Writing Skill | Applicant’s document is easy to read and free of spelling and grammatical errors. |
| Comprehension of Problem | Does the applicant show an understanding of the stated problem? |
| Critical Thinking | Did the applicant carefully analyze and evaluate a problem before arriving at a solution? |
| Ability to Breakdown Problem | Did the applicant successfully break their solution into smaller or simpler parts, then used that to devise a holistic solution? |
| Ability to Test Meaningfully | Did the applicant successfully write meaningful and thorough unit tests that would discover future problems? |

## Background

React is a web development framework which allows the developer to write code for how the HTML DOM should look like given some input of application state. Whenever the input changes, React constructs a virtual DOM using the new input, compares it with the existing DOM that is currently being rendered, and merges only those DOM tree nodes that are applicable.

Under the covers, React is just a smart tree diff-and-merge algorithm. For this exercise, you will assume the role of a developer on the React team.

A colleague of yours is testing out a performance improvement to the React code: before doing the merge of the virtual DOM tree with the actual DOM tree, we want to know if a virtual DOM exists in the current DOM.

The problem boils down to this: given a tree called *dom*, and another tree called *vdom*, is *vdom* a subtree of *dom*?

To simplify the problem, your colleague is only dealing with binary trees right now. Your colleague has come up with a smart way of determining whether a tree is a subtree of another: simply create a string from a preorder traversal of both trees, then check if one string is a subset of another. He has the following code that he asked you to code review. He also has a fairly complex test case that verifies that this algorithm works.

(Note that you can paste the code into Typescript playground at <https://www.typescriptlang.org/play/>.)

**interface** DomNode {  
 value: **string**;  
 left?: **DomNode**;  
 right?: **DomNode**;  
}  
  
**function** isSubtree(dom: **DomNode**, vdom: **DomNode**): **boolean** {  
 **return** stringFromPreOrder(dom).indexOf(stringFromPreOrder(vdom)) > -**1**;  
}  
  
**function** stringFromPreOrder(tree: **DomNode**): **string** {  
 **if** (!tree) {  
 **return** "";  
 }  
  
 **return** tree.value + stringFromPreOrder(tree.left) + stringFromPreOrder(tree.right);  
}

// complex test case... W00t! It works!

**const** dom: **DomNode** = {  
 value: "root",  
 left: {  
 value: "a",  
 left: {  
 value: "c",  
 left: {  
 value: "g"  
 },  
 right: {  
 value: "h"  
 }  
 },  
 right: {  
 value: "d",  
 left: {  
 value: "i"  
 }  
 }  
 },  
 right: {  
 value: "b",  
 left: {  
 value: "e",  
 right: {  
 value: "j",  
 left: {  
 value: "k"  
 },  
 right: {  
 value: "l"  
 }  
 }  
 },  
 right: {  
 value: "f"  
 }  
 }  
}  
  
**const** vdom: **DomNode** = {  
 value: "a",  
 left: {  
 value: "c",  
 left: {  
 value: "g"  
 },  
 right: {  
 value: "h"  
 }  
 },  
 right: {  
 value: "d",  
 left: {  
 value: "i"  
 }  
 }  
}  
  
console.log(isSubtree(dom, vdom));

## Questions

1. This code will not work in certain scenarios. Identify the scenarios it will not work for.
2. What would you add or modify to this algorithm to make it work?
3. Write a few test cases that show that your new code covers different scenarios.
4. Now, in the real world, the DOM is a tree with an arbitrary number children on each node. How would you modify this code, which is currently for a binary tree, to make it work for a k-ary tree, where k is the maximum number of children a node can have?
5. Constructing strings and doing string comparisons is immensely expensive, both memory-wise and computation-wise. Devise an algorithm other than comparing generated strings to determine if a k-ary tree is a subtree of another k-ary tree.