6.009 Quiz 1, Practice 2 -- Spring 2017

This quiz is **open-book and closed-internet**. Feel free to use any physical materials you have brought, but you may not access resources online (except web.mit.edu/6.009). Proctors will be available to answer administrative questions and clarify specifications of coding problems, but they should not be relied on for coding help.

Each problem is worth 5 points, except for the last problem which is worth 8 points, for a total of 28 points. Your raw test score will be scaled by 20/28 to compute the number of final points.

You must submit your modified `quiz.py` via web.mit.edu/6.009 before the deadline in order to receive credit. This quiz assumes you have Python 3.5 (or a later version) installed on your machine.

The `resources` directory contains the **Python Language Reference** and the **Python Library Reference** in PDF form. Please **do not import any Python modules** to use as part of your solutions -- quizzes with `import` statements (or their equivalent!) will be given a grade of 0.

As in the labs, `test.py` can be used to test your code in `quiz.py`. Remember that you can run specific tests by listing the test numbers, like so

```
python3 test.py 1 3 5      # run tests #1, #3, #5
```

Your score will be computed from the number of tests you pass. If you pass all the tests for a problem, you'll receive full credit. If you pass 2 of the 5 tests, you'll receive 40% of the credit for the problem. Note that to pass some of the tests in the time allotted, your solution may have to be reasonably efficient.

This practice quiz is on the long side -- the actual quiz will have one fewer question.

Problem 1: `median` (tests 1--5)

Given a list of numbers, return the median. If the list has an odd number of elements, this is the middle element when the list is written in increasing order. If the list has an even number of elements, this is the mean (average) of the two middle elements.

Examples:

`median([1, 2, 3])` should return `2`.
`median([1, 2, 3, 4])` should return `2.5`.

Problem 2: `is_quasidrome` (tests 6--10)

A palindrome is a sequence of elements that is identical when read forwards or backwards. A quasidrome is a sequence that is either a palindrome or can become a palindrome by removing one element.

Given a string, return a Boolean indicating whether it is a quasidrome. Spaces are treated as any other character.

Examples:

`is_quasidrome("aa")` should return `True`
`is_quasidrome("aab")` should return `True`
`is_quasidrome("aa b")` should return `False`
`is_quasidrome("cab")` should return `False`

Problem 3: `is_permutation` (tests 11-15)

Given two strings, return a Boolean indicating whether one is a permutation of the other.

Examples:

`isPermutation("aabc", "abca")` should return `True`
`isPermutation("aabc", "abc")` should return `False`
`isPermutation("aabc", "abac")` should return `True`
`isPermutation("aabc", "abcc")` should return `False`

Problem 4: `count_triangles` (tests 16--20)

Three vertices `A`, `B`, `C` form a triangle if the edges `AB`, `BC`, and `AC` all exist. Note that `(A, B, C)` and `(B, C, A)` are the same triangle, and `AB` and `BA` are the same edge.

Implement `count_triangles(edges)`, which takes a list of edges, and returns the number of triangles that can be made with `edges`. Each edge is a 2-element lists `[string1, string2]`, where `string1` and `string2` are are names of two vertices connected by the edge.

You may assume that at most one edge exists between any two vertices.

Examples:

`count_triangles([["1","2"], ["2","3"], ["3","1"]])` should return `1`

`count_triangles([["1","2"], ["2","3"]])` should return `0`

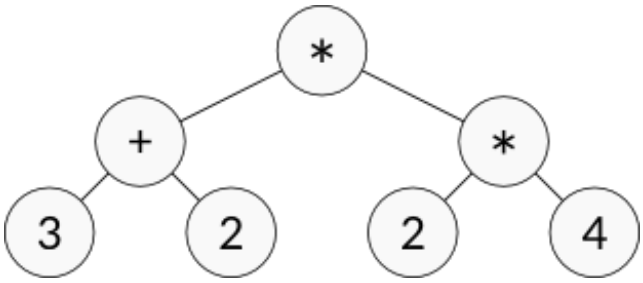
`count_triangles([["1","2"], ["2","3"], ["3","4"], ["1","3"], ["2","4"]])` should return `2`

Problem 5: `eval_ast` (tests 21--28)

Consider a data structure `AST`: a binary tree with children `"left"` and `"right"`, and a value given by `"node"`. The `node` is either an integer or an operation: add (`"+"`) or multiply (`"*"`). If the node encodes an operation, the operands are given by the children of this node: `"left"` and `"right"`. If `node` is neither `"+"` nor `"*"`, it must be an integer and the node has no children.

The `AST` encodes an algebraic expression. For example, `(3 + 2) * (2 * 4)` is encoded by the following `AST`:

```
AST_1 = { "node":   "*",
          "left":   { "node": "+",
                      "left": { "node": 3 },
                      "right": { "node": 2 } },
          "right":   { "node": "*",
                      "left": { "node": 2 },
                      "right": { "node": 4 } }
        }
```



INPUTS:

- `ast`, a dictionary, as defined above.

OUTPUT:

An integer, the result of evaluating the algebraic expression the AST encodes.

EXAMPLES:

- `eval_ast({"node": 1}) = 1`, trivially
- `eval_ast({"node": "+", "left": {"node": 1}, "right": {"node": 4}}) = 5` because `1 + 4 = 5`.

- `eval_ast(AST_1) = 40` because `(3 + 2)*(2 * 4) = 5 * 8 = 40`.