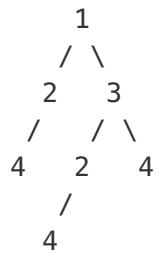
Find Duplicate Subtrees

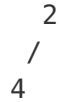
Given a binary tree, return all duplicate subtrees. For each kind of duplicate subtrees, you only need to return the root node of any **one** of them.

Two trees are duplicate if they have the same structure with same node values.

Example 1:



The following are two duplicate subtrees:



and

Therefore, you need to return above trees' root in the form of a list.

Solution 1

```
public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    List<TreeNode> res = new LinkedList<>();
    postorder(root, new HashMap<>(), res);
    return res;
}

public String postorder(TreeNode cur, Map<String, Integer> map, List<TreeNode> res)
{
    if (cur == null) return "#";
    String serial = cur.val + "," + postorder(cur.left, map, res) + "," + postorder
(cur.right, map, res);
    if (map.getOrDefault(serial, 0) == 1) res.add(cur);
    map.put(serial, map.getOrDefault(serial, 0) + 1);
    return serial;
}
```

written by compton_scatter original link here

Solution 2

First the basic version, which is O(n²) time and gets accepted in about 150 ms:

```
def findDuplicateSubtrees(self, root):
    def tuplify(root):
        if root:
            tuple = root.val, tuplify(root.left), tuplify(root.right)
            trees[tuple].append(root)
            return tuple
    trees = collections.defaultdict(list)
    tuplify(root)
    return [roots[0] for roots in trees.values() if roots[1:]]
```

I convert the entire tree of nested `TreeNode`s to a tree of nested `tuple`s. Those have the advantage that they already support hashing and deep comparison (for the very unlikely cases of hash collisions). So then I can just use each subtree's `tuple` version as a key in my dictionary. And equal subtrees have the same key and thus get collected in the same list.

Overall this costs only O(n) memory (where n is the number of nodes in the given tree). The string serialization I've seen in other posted solutions costs O(n^2) memory (and thus also at least that much time).

So far only O(n²) time

Unfortunately, tuples don't cache their own hash value (see this for a reason). So if I use a tuple as key and thus it gets asked for its hash value, it will compute it again. Which entails asking its content elements for *their* hashes. And if they're tuples, then they'll do the same and ask *their* elements for *their* hashes. And so on. So asking a tuple tree root for its hash traverses the entire tree. Which makes the above solution only O(n^2) time, as the following test demonstrates. It tests linear trees, and doubling the height quadruples the run time, exactly what's expected from a quadratic time algorithm.

The code:

```
from timeit import timeit
import sys
sys.setrecursionlimit(5000)

def tree(height):
    if height:
        root = TreeNode(0)
        root.right = tree(height - 1)
        return root

solution = Solution().findDuplicateSubtrees
for e in range(5, 12):
    root = tree(2**e)
    print(timeit(lambda: solution(root), number=1000))
```

The printed times:

```
0.0661657641567657
0.08246562780375502
0.23728608832718473
0.779312441896731
2.909226393471882
10.919695348072757
43.52919811329259
```

Caching hashes

There's an easy way to add caching, though. Simply wrap each tuple in a `frozenset`, which *does* cache its hash value:

```
def findDuplicateSubtrees(self, root):
    def convert(root):
        if root:
            result = frozenset([(root.val, convert(root.left), convert(root.right))
])
            trees[result].append(root)
            return result
    trees = collections.defaultdict(list)
    convert(root)
    return [roots[0] for roots in trees.values() if roots[1:]]
```

Running the above test again now shows O(n) behaviour as expected, doubling of size causing doubling of run time:

```
0.06577755770063994
0.056785410167764075
0.14042076531958228
0.22786156533737006
0.4496169916643781
0.932876339417438
1.8611131309331435
```

And it's much faster than the original version using only tuples. That said, both solutions get accepted by LeetCode in about 150 ms, since the test suite's trees are sadly pretty small and simple.

That of course doesn't mean that the solution is now O(n) time. Only that it's apparently O(n) time for such linear trees. But there's a catch. If two subtrees have the same hash value, then they'll still get fully compared. There are two cases:

1. **Different trees having the same hash value**, i.e., hash collisions. LeetCode uses 64-bit Python and thus hash collisions are very unlikely unless you have a huge number of subtrees. The largest tree in the test suite has 5841 subtrees. The probability of having no collisions for 5841 different values is about 99.9999999999078%:

```
>>> reduce(lambda p, i: p * (2**64 - i) / 2**64, range(5841), 1.0)
0.999999999999078
```

Even if all 5841 subtrees were different and all of the test suite's 167 test cases were like that, then we'd still have about a 99.99999998460193% chance to not have any hash collisions anywhere:

```
>>> 0.999999999999078**167
0.9999999998460193
```

Also, I could of course use a stronger hashing algorithm, like SHA256.

2. **Equal trees having the same hash value** . Well duh, obviously equal trees have the same hash value. So how big of a problem is that? How many equal trees can we have, and of what sizes? In the above case of the whole tree being linear, there are *no* equal subtrees at all (except for the trivial empty subtrees, but they don't matter). So let's go to the other extreme end, a perfect tree. Then there are *lots* of equal trees. There are n subtrees but only about $\log^2(n)$ *different* subtrees, one for each height. Subtrees of the same height are all equal. So pretty much *every* subtree is a duplicate and thus shares the hash with a previous subtree and thus will get completely compared to the previous one. So how costly is it to traverse all subtrees completely (in separate traversals)? The whole tree has n nodes. Its two subtrees have about n/2 nodes each, so about n nodes together (actually n-1 because they don't contain the root). The next level has 4 trees with about n/4 nodes each, again adding n nodes overall. And so on. We have $\log_2(n)$ levels, so overall this takes O(n log n) time. We can again test this, doubling the size of a tree makes the run time a bit worse than double:

```
def tree(height):
    if height:
        root = TreeNode(0)
        root.left = tree(height - 1)
        root.right = tree(height - 1)
        return root

solution = Solution().findDuplicateSubtrees
for e in range(10, 17):
    root = tree(e)
    print(timeit(lambda: solution(root), number=100))
```

The printed times:

```
0.10957473965829807
0.22831256388730117
0.48625792412907487
1.010916041039311
2.131317089557299
4.5137782403671025
9.616743290368206
```

The last two trees have 15 and 16 levels, respectively. If the solution does take

Θ(n log n) time for perfect trees, then we expect it to take 2 * 16/15 as long for the tree with 16 levels as for the tree with 15 levels. And it did: 4.51 * 2 * 16/15 is about 9.62. (If you find that suspiciously accurate, you're right: I actually ran the whole test 50 times and used the averages).

So is the `frozenset` solution O(n log n) time? Nope. Sadly not. It's still only $O(n^2)$. Remember how the original tuple solution took $O(n^2)$ time for linear trees? And how caching the hashes improved that to O(n), because there were no duplicate subtrees whose equal hashes would cause expensive deep comparison? Well, all we have to do to create the same nightmare scenario again is to have *two* such linear subtrees under a common root node. Then while the left subtree only takes O(n), the right subtree takes $O(n^2)$. Demonstration again:

```
def tree(height):
    if height:
        root = TreeNode(0)
        root.right = tree(height - 1)
        return root

solution = Solution().findDuplicateSubtrees
for e in range(5, 12):
    root = TreeNode(0)
    root.left = tree(2**e)
    root.right = tree(2**e)
    print(timeit(lambda: solution(root), number=1000))
```

The printed times:

```
0.1138048859928981
0.19950686963173872
0.5518468952197122
1.8595846431294971
6.7689327267056605
26.291197508748162
106.77212851917264
```

O(n) time and space

Fortunately, @Danile showed an O(n) time solution and it's pretty simple. Here's my implementation using their idea:

```
def findDuplicateSubtrees(self, root, heights=[]):
    def getid(root):
        if root:
            id = treeid[root.val, getid(root.left), getid(root.right)]
            trees[id].append(root)
            return id
    trees = collections.defaultdict(list)
    treeid = collections.defaultdict()
    treeid.default_factory = treeid.__len__
    getid(root)
    return [roots[0] for roots in trees.values() if roots[1:]]
```

The idea is the same as Danile's: Identify trees by numbering them. The first unique subtree gets id 0, the next unique subtree gets id 1, the next gets 2, etc. Now the dictionary keys aren't deep nested structures anymore but just ints and triples of ints.

Running the "perfect trees" test again:

```
0.05069159040252735
0.09899985757750773
0.19695348072759256
0.39157652084085726
0.7962228593508778
1.5419999122629369
3.160187444826308
```

And the "linear trees" test again:

```
0.034597848759331834
0.05386034062412301
0.12324202869723078
0.22538750035305155
0.46485619835306713
0.8654176554613617
1.7437530910788834
```

And the "two parallel linear trees" test again:

```
0.05274439729745809
0.0894428275852537
0.18871220620853896
0.3264557892339413
0.7091321061762685
1.3789991725072908
2.7934804751983546
```

All three tests now show nice doubling of the times, as expected from O(n).

This solution gets accepted in about 100 ms, and the best time I got from a handful of submissions was 92 ms.

Revisiting the probability calculation for hash collisions

Above I calculated the probability 99.9999999999078% using *floats*. Which of course have rounding errors. So can we trust that calculation? Especially since we're multiplying so many special factors, all being close to 1? Turns out that yes, that's pretty accurate.

We can do the calculation using integers, which only round down. Just scale the "probability" by 10^{30} or so:

```
>>> p = 10**30
>>> for i in range(5841):
        p = p * (2**64 - i) // 2**64

>>> p
999999999999075407566135201054
```

Since those divisions round down, this gives us a lower bound for the true value. And if we instead work with *negative* numbers, then the divisions effectively round *up* and we get an *upper* bound for the true value:

```
>>> p = -10**30
>>> for i in range(5841):
        p = p * (2**64 - i) // 2**64

>>> -p
999999999999075407566135206894
```

Comparing those two bounds we can see that the true probability is 99.99999999907540756613520...%. So of the previous result 99.9999999999078% using floats, only that last digit was wrong.

Using this value in the subsequent calculation again, we get a 99.99999998455928% chance for not having any collisions anywhere even if all 167 test cases had 5841 subtrees without duplicates:

```
>>> 0.99999999999075407566135206**167
0.9999999998455928
```

written by StefanPochmann original link here

Solution 3

```
class Solution {
public:
    vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
        unordered_map<string, vector<TreeNode*>> map;
        vector<TreeNode*> dups;
        serialize(root, map);
        for (auto it = map.begin(); it != map.end(); it++) {
            if (it->second.size() > 1) {
                dups.push_back(it->second[0]);
            }
        }

        return dups;
    }

private:
    string serialize(TreeNode* node, unordered_map<string, vector<TreeNode*>>& map)
{
        if (!node) return "";
        string s = "(" + serialize(node->left, map) + to_string(node->val) + serial
ize(node->right, map) + ")";
        map[s].push_back(node);
        return s;
    }
};
```

written by alexander original link here