Homework 1: Merkle Trees

Deadline: November 4th

1. Administration

Each homework in this course will contribute 20 % to the final grade. There will be 4 homework assignments and one final project. If needed, we might have a bonus assignment, which can replace your worst homework, but not the final project.

In this assignment we will need to modify an existing Python program in order to implement some functionalities of Merkle Trees.

You should extend the library BitcoinMerkle.py, which you receive together with this assignment.

The library BitcoinMerkle.py has no implementation for certain methods, and your assignment asks to implement these, and send the modified .py file with your solution. We will design test data to check whether your implementation works as desired.

Your solution should be sent to the following email:

- domagojvrgoc@gmail.com

There is no restriction or penalty for using materials you found online. It is a good practice to mention those if you use them. This will not result in any penalty to your homework.

2. The homework

In this assignment, we will implement three new methods in the library BitcoinMerkle.py, and also a new class. The methods are the following:

1. `def generate_proof(self, hashesOfInterest):`

   This is a function of the class MerkleTree, and the function returs the proof needed to construct the Merkle root we used in `populate_tree(self, flag_bits, hashes)` from the class PartialMerkleTree. That is, the function returns the sequence of bits `flag_bits`, and the sequence of hashes `hashes`, needed to verify if the hashes in the list `hashesOfInterest` belong to our Merkle tree. We can assume that all `hashesOfInterest` are leaves in the Merkle tree.

   To be concrete, the object that the function `generate_proof` returns is of the class:

```
class MerkleProof:
    def __init__(self, hashesOfInterest, nrLeaves=None,\\
                       flags=None, hashes=None):
        self.hashesOfInterest = hashesOfInterest
        self.nrLeaves = nrNodes
        self.flags = flags
        self.hashes = hashes
```

This class has no class method/function and is only used to store the proof of inclusion in a Merkle tree. As you can see, this proof contains all the information necessary to validate whether the hashes in the list `hashesOfInteres` belong to the Merkle tree with some root `Mroot` (which is not a member of the class).Recall that for this we also need to know the number of leaves, `nrLeaves`.

Remark: It is possible that you will have to augment the class `MerkleTree` so that it includes all the levels of the tree upon receiving the data of the leaves. For this, you can modify the code of the class `PartialMerkleTree`, or come up with a new implementation.

2. `class SortedTree:`

```
.
.
.
```

```
def proof_of_non_inclusion(self,hash):
```

The class `SortedTree` will be used to generate a proof of non-inclusion of a certain hash in our Merkle tree. For this, the class should construct an ordered Merkle tree. The leaves of the tree should be ordered by the hexadecimal representation of their hashes (note that the tree itself needs these data in bytes, since we will use it compute the higher levels of the tree). It is up to you to implement any method that you will need to use further on.

The function `proof_of_non_inclusion(self,hash)` receives as its input a single `hash`, and should return the proof that `hash` received as input does not belong to the tree. The proof should be an object of the class `MerkleProof`. If you find it necessary, you can define a new class to store the proof, but `MerkleProof` is already good enough.

3. `def verify_non_inclusion(hash, merkleRoot, proof):`

This method receives as input a `hash`, and a proof `proof` of non-inclusion of `hash` in the Merkle tree with the root `merkleRoot` (also an input parameter). The function should verify whether the proof is correct or not (it returns true if `hash` *does not* belong to the Merkle tree with the root `merkleRoot`, and false otherwise; including when the proof is incomplete or incorrect).

Points. The total number of points is 20, and it is distributed as follows:

- `generate_proof` – 14 points
- `SortedTree` y `proof_of_non_inclusion` – 3 points
- `verify_non_inclusion` – 3 points.