TRY: a nfT lotteRY

Federico Bernacca

1 Introduction

The following design choices were made to carry out the homework.

The collectibles assignable to NFTs can be found in the following repo: nft_lottery.
The basic idea is that the *tokenId* of the NFT is also the name of the collectible associated with it:

```
"_tokenId": "1",
"_image": "https://github.com/fedehsq/nft_lottery/master/collectibles/1.svg"
```

There are 3 contracts within the project: [1]

1. **ERC721**
2. **NFT**
3. **Lottery**

1.1 ERC721

The first definition is the abstract contract **ERC721.sol**, which defines the methods and events required to be compliant with the standard ERC721.

1.2 NFT

The second definition and implementation is of the **NFT.sol** contract, which inherits from the contract above.

1.3 Lottery

The third contract provides the implementation of the lottery, which uses the contract **NFT.sol** to mine tokens.

1.4 Functionality

The required functionality has been implemented correctly, plus the following assumption has been made: *If the number of NFTs available for each class during the draw is less than the number of winners, a token is mined on demand and sent to the winner.*

1.5 Log

The logs of operations required by the contract has been made with *Event* and *Emit*.

1.6 Operations

For proper contract operation, the order of deployment of contracts and operations should be as follows:

1. Deploy of **NFT.sol**.
2. Deploy of **Lottery.sol** with arguments *(nftAddress, roundDuration i.e 2)*. The first round is automatically opened after deployment.
3. Mint N tokens (i.e. 10) via the function *mintNtoken*.
4. Change sender address.
5. Purchase N tickets (i.e. 10) via the function *buyNRandomTicket*.
6. Repeat step 4 - 5.
7. Return to the address related to the *lotteryManager*.
8. Run *drawNumbers* to draw the winning numbers.
9. Run *givePrizes* to award the prizes.
10. You can now open a new round or close the lottery via the respective functions *openRound* and *closeLottery*.

The advice is to use **ganache** rather than **Js** because this crashes all the time.

[1] In the branch *openzeppelin*, the contract *ONFT.sol* extends the ERC721 implementation of *OpenZeppelin*

1.7 Gas estimation

For gas estimation, the idea was to find a function with high gas consumption and try to improve it. The function considered was *givePrizes*.
The initial implementation involved a loop in which all the numbers on a ticket were compared with the winning numbers drawn, for a complexity of $O(n^2)$.
The next idea involved sorting the numbers in ascending order of the tickets once they were bought, and then being able to use a binary search when comparing to obtain a total complexity of $n * log(n)$.
The differences in terms of gas spent are similar for the experiment conducted:

```
nTicket: 200
binarySearch gas cost: 21782135
normalSearch gas cost: 21561213
```

In addition, choosing to order the ticket numbers avoids having to compare the numbers one by one to check whether the user has entered the same number several times during purchase.
Thanks to the sorting, the complexity drops from $O(n^2)$ to $O(n)$ because the check will occur only between the ith number and the ith + 1.

1.8 Security

The random generator turns out to be secure because by doing the operation **blockhash** of a block $(X + K)$ that is not yet on the blockchain at the end of the round, i.e., at the end of the ticket purchase operation, an attacker cannot predict the numbers drawn by the generator because during the round, this block has not yet been mined, so it turns out to be impossible to predict its hash. (For simplicity, the parameter K is set to 0, because since the blockchain is local, additional transactions would have to be made before drawing the winning numbers.)