Blockchain Tutorial Exercises (Individual)

-

Maxime Chauveau

ID: 25292257

10/10/2025

-

GitHub repository: *GitHub*

Course Leader: Duaa Zuhair Al-Hamid

Table of Contents

Report Overview

This report covers all the questions from the assignment, including topics such as Proof-of-Work simulation, Solidity smart contracts, and NFT minting using Alchemy. Each section provides explanations, code snippets, and examples to demonstrate the concepts in practice.

To view the source code, visit the GitHub repository: GitHub

Exercise 1

Question 1:

The script for this exercise implements a simplified proof-of-work where the goal is to find a nonce such that the hash of the block is below a target value. I used quite the same code as in the Lab5 and just modified the condition, so I still have the part of random. We are now in Hexadecimal, so the targets are also different:

target = int("0x000fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff", 16) (Easy : 3 zeros)

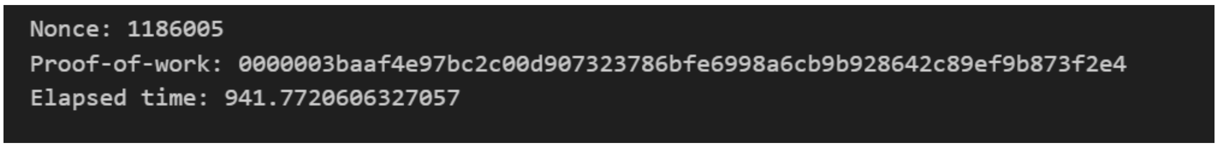
target = int("0x0000ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff", 16) (Harder : 4 zeros)

The new condition says if pickled block is under the target, then we found the block and we print the Nonce, Proof of Work and the Elapsed time for find it.

We can find blocks easily with both, but it is interesting to compare outputs. Outputs figuring bellows are example, every time I run the code, because of the random part, it will give different output.

Easy	Harder
• Nonce: 2330 • Proof-of-work: 000b866d4f5db87d443f664b1c8810571c7fcd38ce6c228a6ea15474617b44c0 • Elapsed time: 0.011459112167358398s	• Nonce: 67220 • Proof-of-work: 00003927c8d89164725943089548e9f743623463c2e4fca55e32ba202b46ea45 • Elapsed time: 2.214970350265503s

As we can see, this is more difficult to find a solution for four zeros than for three. For three zeros, the solution was found instantly and for four zeros it was more than two seconds low to find solution. Moreover, we can see the huge difference of nonce between the two version: more than 28 time more trials for the four zeros version. I also wanted to try the five zero versions, and I needed to wait for more than 15 minutes to find a Nonce:



```
Nonce: 1186005
Proof-of-work: 0000003baaf4e97bc2c00d907323786bfe6998a6cb9b928642c89ef9b873f2e4
Elapsed time: 941.7720606327057
```

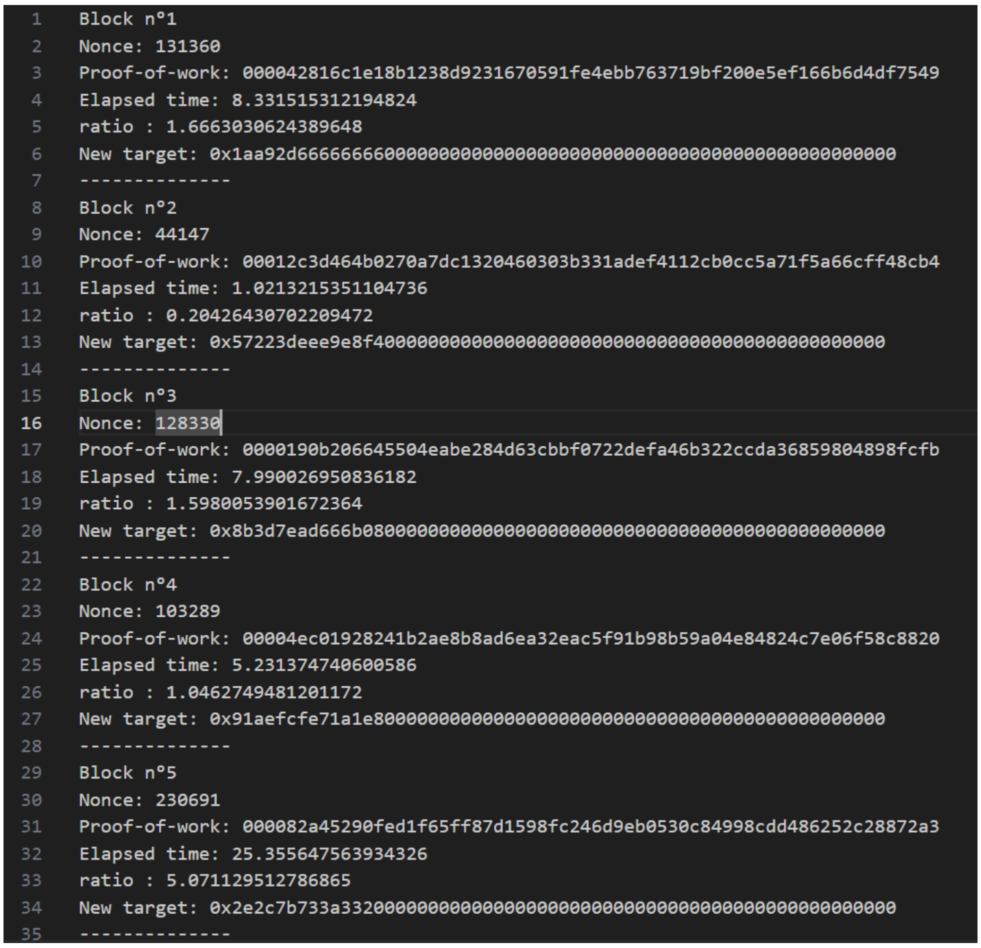
Figure 1: PoW for five zeros

Question 2:

To make it easier, I created a **Proof-of-Work** function and an **Adapt Difficulty** function. In the first function, I reused the code from the previous question and created three arguments: pickled block, target, and nonce. This function returns the elapsed time. For the second function, my arguments are pickled block, initial target (hex), target time (the time I expect to find a block), the number of tries, and nonce.

This function simply runs a for loop to perform the Proof-of-Work and record the elapsed time for each block. Using this value, I create a ratio with the target time and the time I got, which I can multiply with the previous target to adapt it. Using this ratio, if a block is found faster than expected, the target becomes smaller (harder), and if a block is found slower than expected, the target becomes larger (easier). On each iteration, I print the nonce, the PoW, the elapsed time, the ratio, and finally the new target.

I first tried to run the function by mining 5 blocks with an expected time of 5s:



```
Block n°1
Nonce: 131360
Proof-of-work: 000042816c1e18b1238d9231670591fe4ebb763719bf200e5ef166b6d4df7549
Elapsed time: 8.331515312194824
ratio : 1.6663030624389648
New target: 0x1aa92d6666666600000000000000000000000000000000000000000000000
--------------
Block n°2
Nonce: 44147
Proof-of-work: 00012c3d464b0270a7dc1320460303b331adef4112cb0cc5a71f5a66cff48cb4
Elapsed time: 1.0213215351104736
ratio : 0.20426430702209472
New target: 0x57223deee9e8f40000000000000000000000000000000000000000000000
--------------
Block n°3
Nonce: 128330
Proof-of-work: 0000190b206645504eabe284d63cbbf0722defa46b322ccda36859804898fcfb
Elapsed time: 7.990026950836182
ratio : 1.5980053901672364
New target: 0x8b3d7ead666b080000000000000000000000000000000000000000000000
--------------
Block n°4
Nonce: 103289
Proof-of-work: 00004ec01928241b2ae8b8ad6ea32eac5f91b98b59a04e84824c7e06f58c8820
Elapsed time: 5.231374740600586
ratio : 1.0462749481201172
New target: 0x91aefcfe71a1e80000000000000000000000000000000000000000000000
--------------
Block n°5
Nonce: 230691
Proof-of-work: 000082a45290fed1f65ff87d1598fc246d9eb0530c84998cdd486252c28872a3
Elapsed time: 25.355647563934326
ratio : 5.071129512786865
New target: 0x2e2c7b733a332000000000000000000000000000000000000000000000000
--------------
```

Figure 2: Output of Question 2 exercise 1

As we can see, when the elapsed time is less than 5s, meaning that the block is found too fast, the ratio is less than 1, so the new target becomes proportionally harder. When the elapsed time is more than 5s, the ratio is greater than 1, meaning that the new target becomes easier.

Question 3:

It is possible to have a Proof of Work blockchain with a real-time difficulty adjustment For instance, Bitcoin adjusts the difficulty approximately every 2016 blocks to maintain a target block time.

The advantage of creating an adjustment function is that you can control the number of blocks found in each time interval. This keeps a blockchain stable and accessible for miners Even if the mining power increases significantly, the blockchain will adapt itself to make solutions harder and keep the blockchain stable.

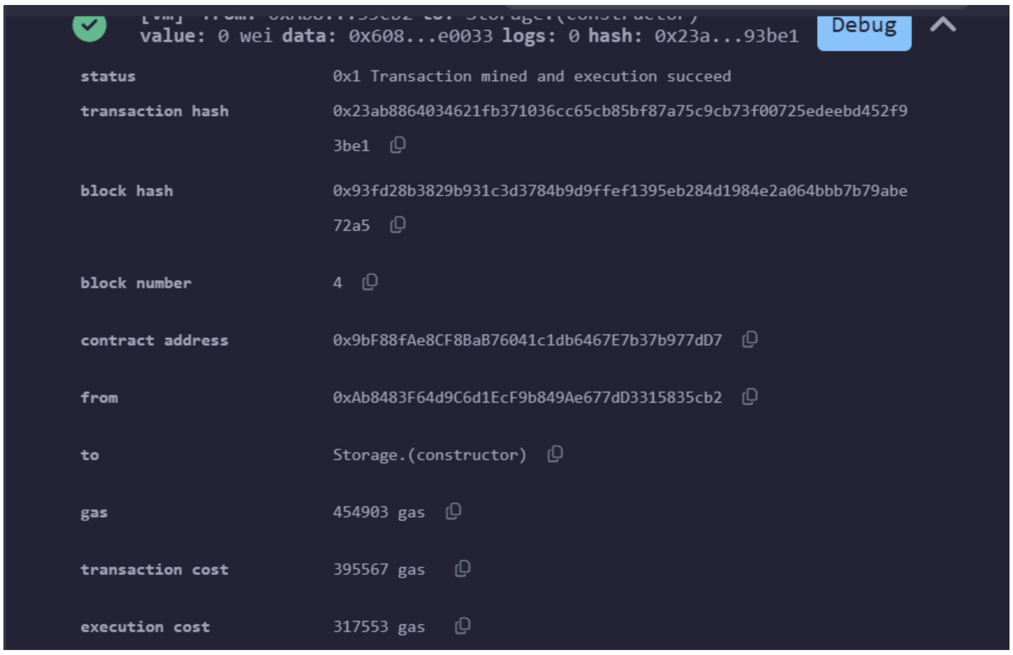
It is not easy to maintain a stable interval between two blocks. In the previous question, we saw that I was calculating the ratio after every block But because of the randomness of blocks, if one block is found much faster than expected, the ratio will be significantly smaller than expected, making the target extremely difficult to reach for the next blocks. The risk is that this can cause sudden large increases or decreases in difficulty, creating an unstable blockchain.

In Bitcoin, the way to solve this problem is to wait for multiple blocks and calculate the ratio after 2016 blocks, for example. Using this strategy decreases the risk that a randomly too easy or too difficult block distorts the ratio for subsequent blocks.

Exercise 2

Question 1:

By modifying Storage.sol in Remix, I can now store and retrieve a name. By looking at the debug information after deploying the contract, I can get details about it:

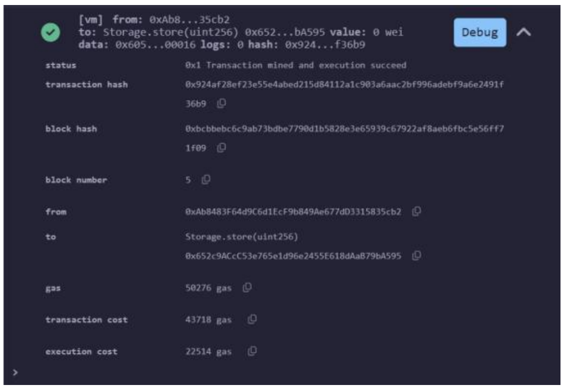


```
value: 0 wei data: 0x608...e0033 logs: 0 hash: 0x23a...93be1
status              0x1 Transaction mined and execution succeed
transaction hash    0x23ab8864034621fb371036cc65cb85bf87a75c9cb73f00725edeebd452f9
                    3be1
block hash          0x93fd28b3829b931c3d3784b9d9ffef1395eb284d1984e2a064bbb7b79abe
                    72a5
block number        4
contract address    0x9bF88fAe8CF8BaB76041c1db6467E7b37b977dD7
from                0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to                  Storage.(constructor)
gas                 454903 gas
transaction cost    395567 gas
execution cost      317553 gas
```

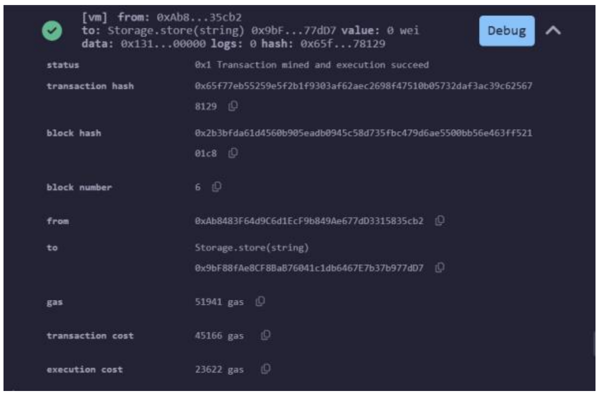
Figure 3: output of contract string storage deployment

After testing the two versions, I compared the transaction cost of storing an integer and a string First, I stored the number 22 (my age) using the integer contract. This execution had a transaction cost of 43,718 gas. For the second try, I used the second contract to store my name (Maxime). The transaction cost for this action was 45,166 gas.

Although the difference is small, the transaction cost for a string is usually higher than for an integer. This is expected because storing six characters in a string requires more gas than storing a simple number like 22. We can see in the screenshots below the debug for the two executions I performed:



```
[vm] from: 0xAb8...35cb2
to: Storage.store(uint256) 0x652...bA595 value: 0 wei
data: 0x605...00016 logs: 0 hash: 0x924...f36b9
status              0x1 Transaction mined and execution succeed
transaction hash    0x924af28ef23e55e4abed215d84112a1c903a6aac2bf996adebf9a6e2491f
                    36b9
block hash          0xbcbbebc6c9ab73bdbe7790d1b5828e3e65939c67922af8aeb6fbc5e56ff7
                    1f09
block number        5
from                0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to                  Storage.store(uint256)
                    0x652c9ACcC53e765e1d96e2455E618dAa879bA595
gas                 50276 gas
transaction cost    43718 gas
execution cost      22514 gas
```



```
[vm] from: 0xAb8...35cb2
to: Storage.store(string) 0x9bF...77dD7 value: 0 wei
data: 0x131...00000 logs: 0 hash: 0x65f...78129
status              0x1 Transaction mined and execution succeed
transaction hash    0x65f77eb55259e5f2b1f9303af62aec2698f47510b05732daf3ac39c62567
                    8129
block hash          0x2b3bfda61d4560b905eadb0945c58d735fbc479d6ae5500bb56e463ff521
                    01c8
block number        6
from                0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to                  Storage.store(string)
                    0x9bF88fAe8CF8BaB76041c1db6467E7b37b977dD7
gas                 51941 gas
transaction cost    45166 gas
execution cost      23622 gas
```

Figure 4: Output for store function for both contracts

Question 2:

When a contract is deployed in Ethereum, it is stored directly on-chain. This means that each node can see and use the contract if they know the address where it is stored.

It is important to understand the difference between the different cost outputs provided by Remix. Gas refers to the maximum amount of gas that can be consumed by the transaction. The transaction cost represents the total amount actually paid. The execution cost corresponds only to the gas consumed for executing the code itself, excluding the intrinsic transaction overhead.

We can now compare the transaction costs of deploying, storing, or reading information on-chain using the previous contract (myname) and the string "Maxime":

Deployment:	Transaction Cost: around 400,000 gas
Store Data:	Transaction Cost: around 25,000 gas
Read Data:	Transaction Cost: around 0 gas

As we can see, deploying a contract on-chain is the most expensive operation because all the code must be stored on the blockchain. Storing data is less expensive, as we only need to store a single variable (string, int, array, etc.). Finally, reading data does not cost anything because we are only accessing public data without modifying the chain.

Question 3:

Every contract needs to be stored on-chain. Indeed, if we want to be able to use it, we need to know where it is stored. That is why we see "STORAGE AT 0x...". 0x... is the hexadecimal address that points to the location of the contract code and its storage on-chain. That means that every node that knows this address can use this contract.

There is an important difference between a user address and a contract address. A user address is used to send, receive, or hold ETH and tokens. On the other hand, a contract address points to the contract code. Users interact with contracts by calling their functions rather than transferring cryptocurrencies directly.

Exercise 3:

Question 1:

It is important to make the difference between a fungible token and a Non-Fungible Token (NFT).

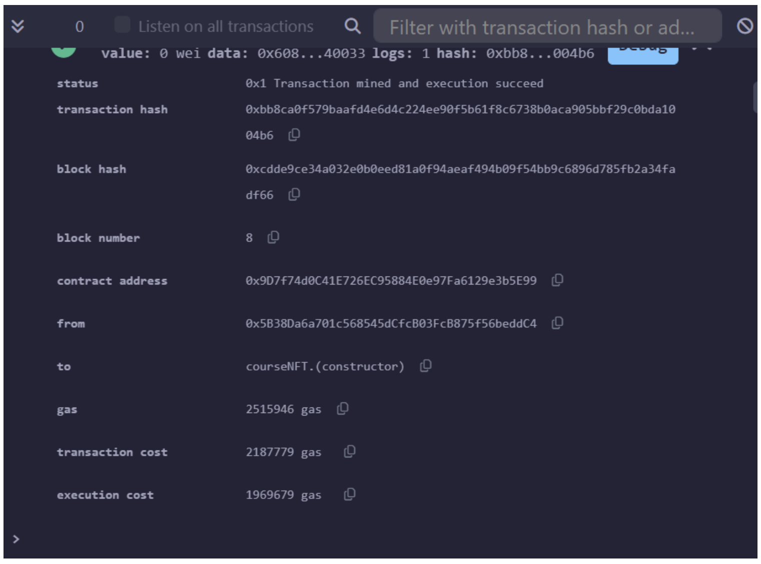
A fungible token is something that one person can own, exchange, and that has the same value and properties as another identical token. For instance, a $10 bill owned by one person has the same value as another $10 bill owned by someone else. A bus ticket can also be considered fungible, because every ticket has the same value and use.

On the other hand, a non-fungible token is unique, meaning that each token has its own value and specific properties. It cannot be swapped on a one-to-one basis like fungible tokens. For example, collectible artwork has its own value depending on its attributes, and trading cards like Pokémon cards have different values depending on their rarity or condition (often graded out of 10). Each card is unique and has its own worth.

Question 2:

To determine how much gas the deployment of my contract cost, I used Remix. As we can see in the picture below, the transaction cost of this contract is 2,187,779 gas. Comparing it to the string storage contract from Exercise 2 shows a large difference. The deployment of this contract cost me 400,000 gas.

It is important to note the difference between these two contracts to explain the large difference in transaction cost. Indeed, the string storage contract only stores one string and has two functions: one to store and one to retrieve. On the other hand, the NFT contract uses standards such as ERC721. The contract needs to import and deploy multiple external libraries, and each library has many features. When deployed, everything must be stored on the blockchain, which is why this contract costs much more than the first one we deployed in Exercise 2.


0
Listen on all transactions
Filter with transaction hash or ad...

```
value: 0 wei data: 0x608...40033 logs: 1 hash: 0xbb8...004b6
status              0x1 Transaction mined and execution succeed
transaction hash    0xbb8ca0f579baafd4e6d4c224ee90f5b61f8c6738b0aca905bbf29c0bda10
                    04b6
block hash          0xcdde9ce34a032e0b0eed81a0f94aeaf494b09f54bb9c6896d785fb2a34fa
                    df66
block number        8
contract address    0x9D7f74d0C41E726EC95884E0e97Fa6129e3b5E99
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  courseNFT.(constructor)
gas                 2515946 gas
transaction cost    2187779 gas
execution cost      1969679 gas
```

Figure 5: Output contract Exercise 3 deployment

To better understand how gas fees work, the picture below shows the relationship between gas limit, current gas price (in gwei — gigawei, with wei being the smallest unit of Ether), and the maximum transaction fee (in Ether). As we can see, the total fee paid depends on the gas limit, estimated by the system, and the gas price, which is the amount a user is willing to pay per unit of gas. Indeed, users can offer higher gas prices to get their transactions processed faster, creating a competitive market.

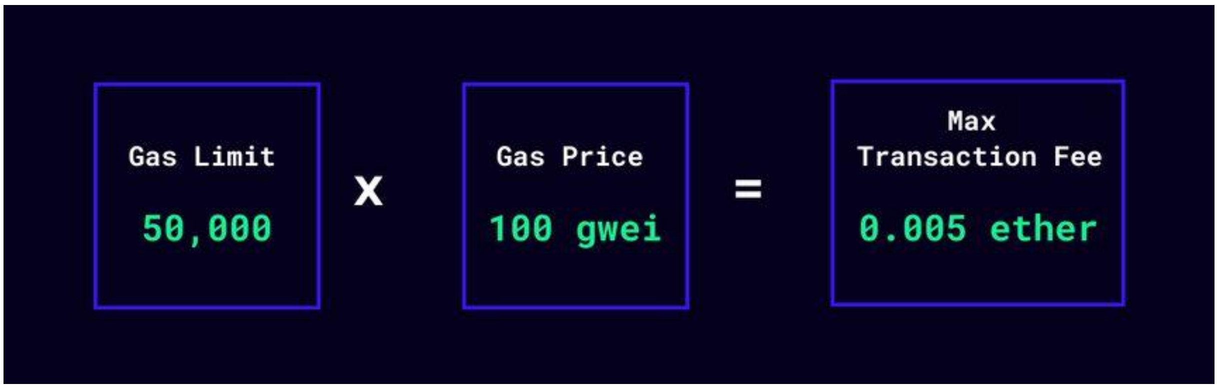


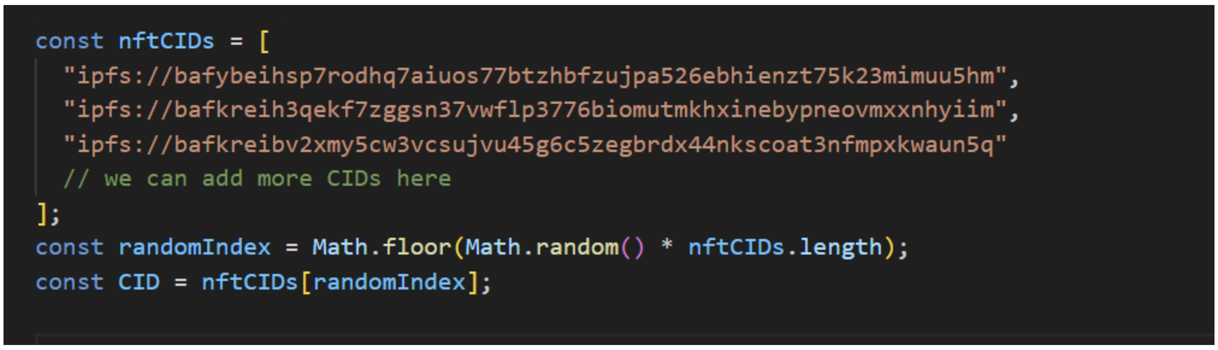
Figure 6: How to calculate the maximum Transaction Fee

Question 3:

To mint another NFT, I uploaded a new image (cutedog.jpg) to Pinata. I then updated the metadata JSON to point to the new CID (bafkreih3qekf7zggsn37vwflp3776biomutmkhxin-ebypneo-vmxxnhyiim), which corresponds to that image. Finally, I ran my minting script again, which created a new NFT using the updated metadata.

Question 4:

In order to generate a new NFT every time the mint function is called, we can upload multiple JPG images to Pinata to obtain their CIDs. Then, we can create an array of CIDs and randomly select one each time the mint function is called.



```
const nftCIDs = [
  "ipfs://bafybeihsp7rodhq7aiuos77btzhbfzujpa526ebhienzt75k23mimuu5hm",
  "ipfs://bafkreih3qekf7zggsn37vwflp3776biomutmkhxinebypneovmxxnhyiim",
  "ipfs://bafkreibv2xmy5cw3vcsujvu45g6c5zegbrdx44nkscoat3nfmpxkwaun5q"
  // we can add more CIDs here
];
const randomIndex = Math.floor(Math.random() * nftCIDs.length);
const CID = nftCIDs[randomIndex];
```

Figure 7: array of CIDs with random choosing

This allows us to generate a new NFT each time by using a different CID.