Proof of Work

Recordemos que Proof of Work, a grosso modo, son varios nodos utilizando su poder computacional para ganar una carrera. La carrera se gana resolviendo un "acertijo matemático" encontrando un número llamado "nonce". Los bloques pasan a ser parte de la red cuando el nonce es encontrado y los demás nodos llegan a un acuerdo.

Para este capítulo no haremos una clase. Nos vamos a centrar en acertijo que hay que resolver y el como se halla la solución.

El acertijo

Vamos a desarrollar un pequeño ejemplo para visualizar cómo se resuelve un acertijo en Proof of Work.

In [15]:
```
from Crypto.Hash import SHA256
contenido_a_cifrar = "ultra secreto"

# De manera normal, si hacemos un hash de lo de arriba se ve asi.
hash_normal = SHA256.new(contenido_a_cifrar.encode())
hash_normal.hexdigest()
```

Out[15]:
```
'ebd510e521801dc7ab91d89dcbbcd7aa30fd31ff8bb591401b2f8c87f1c2dd4a'
```

El acertijo en Blockchain suele ser encontrar un hash que cumpla ciertas condiciones:

1. Que los primeros "n" caracteres del hash sea 0. "n" podrían ser 3, 5 o 7 ceros.
2. Que el valor Hash en decimal, sea mayor a un target.

Veámoslo:

Si solo hacemos un hash no vamos a resolver nada. Necesitamos iterar muchas veces para encontrar un HASH que cumpla las condiciones. Para ello, existe el nonce.

Lo primero sería cifrar lo que sea que queramos cifrar.

In [1]:
```
from Crypto.Hash import SHA256

contenido_a_cifrar = {
    "contenido": "ultra secreto",
    "nonce": 0} # Incluimos el nonce en el contenido a cifrar
```

Definimos una dificultad "target" que nuestro hash debe de igualar o superar para considerar nuestro hash válido.

In [3]:
```
difficulty_hash = 0x0000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
difficulty_decimal = 176684706477838432958329750074291851582748389687561895812160
```

difficulty_hash y difficulty_decimal equivalen a lo mismo.

En este ejemplo, nuestro hash tiene que tener en sus primeros 4 digitos el número 0.

Obtenemos el primer hash con el nonce en 0.

In [4]:
```
hash_resultante = SHA256.new(str(contenido_a_cifrar).encode())
contenido_a_cifrar["nonce"], hash_resultante.hexdigest()
```

Out[4]:
```
(0, '63c03dc83f41c06e31d8daf87718110fe4b4dc8a6b47f2c9db8b1d2b52e7046f')
```

Para nada está cerca. Por lo que vamos a incluir un bucle While que vaya aumentando el nonce y revisando que el hash haya alcanzado el target.

In [5]:
```
# Solo Si el hash resultante es mayor o igual a el target de dificultad, puedes sal
# Si no cumple la condicion, se aumenta el valor del nonce, y se vuelve a cifrar el
while int(hash_resultante.hexdigest(), 16) >= difficulty_hash: # En vez de difficul
    contenido_a_cifrar["nonce"] += 1
    hash_resultante = SHA256.new(str(contenido_a_cifrar).encode())
print(hash_resultante.hexdigest(), ", nonce: ", contenido_a_cifrar["nonce"]) # El r
```

```
0000882567fe350c0c382ee3d5dd3bf34478069b8508a0e84f6124096949abae , nonce:  159043
```

Con el acertijo resuelto, un bloque está listo para ser anexado a la red.

El nonce se encuentra después de varias iteraciones. Todos los nodos que están compitiendo para ser los primeros en ganar la carrera buscan este valor, puede que algunos lo intenten con algún algoritmo distinto, pero en esencia es lo mismo.

En cadenas de bloques que contienen a muchos mineros en su red (Bitcoin), cuentan con un sistema en el que cada que se mina un bloque se ajusta la dificultad en relación a los mineros trabajando. Por ejemplo, Bitcoin ajusta su dificultad para que cada bloque se mine en aproximados 10 minutos.

La principal desventaja del consenso Proof of Work es el consumo excesivo de energía. Para resolver este problema, se creó un consenso más amigable para el ambiente llamado Proof of Stake. Veámoslo en el siguiente capítulo.