Notas de Aula #04: Princípios da Mineração de Bitcoin

1. O QUE É A MINERAÇÃO DE BITCOIN?

A mineração de Bitcoin é um processo computacional competitivo no qual diversos participantes, conhecidos como mineradores, utilizam seu poder de processamento para resolver um problema matemático complexo. O primeiro minerador a encontrar a solução para este problema é recompensado com uma nova quantidade de Bitcoin. A tarefa central desse processo consiste em encontrar um valor numérico específico, chamado "***nonce***", que, quando combinado com outros dados, gera um resultado que atende a um critério pré-definido pela rede.

1.1. Passo a Passo do Processo de Mineração

O processo de mineração pode ser resumido nas seguintes etapas:

- **Receber Transações..... :** O minerador agrupa um conjunto de transações pendentes na rede, que são representadas como uma sequência de bytes;
- **Adicionar o Nonce** ..... : Um valor numérico inicial chamado "nonce" é adicionado ao início do conjunto de transações;
- **Calcular o Hash**..... : A combinação do nonce com os dados das transações é processada por meio do algoritmo de hash SHA-256;
- **Verificar a Dificuldade** ... : O resultado do hash é verificado para determinar se ele começa com um número específico de zeros. Esse número de zeros é conhecido como "dificuldade" da rede;
- **Encontrar a Solução**..... : Se o hash não atender ao critrio de dificuldade, o nonce é incrementado e o processo se repete até que um nonce válido seja encontrado.

1.2. O que é o SHA-256?

O **SHA-256** (***Secure Hash Algorithm 256-bit***) é um algoritmo de *hash* criptográfico fundamental para a segurança do Bitcoin. Suas principais características são:

- **Saída de Tamanho Fixo**..... : Ele recebe uma entrada de qualquer tamanho e gera uma saída de tamanho fixo com 256 bits, comumente representada por 64 caracteres hexadecimais;
- **Efeito Avalanche**..... : Mesmo uma pequena alteração nos dados de entrada resulta em uma mudança drástica e imprevisível no hash de saída.

A Biblioteca hashlib em Python:

Para trabalhar com algoritmos de *hash* como o SHA-256 em Python, utilizamos a biblioteca padrão `hashlib`. Ela fornece uma interface simples e segura para criar hashes de dados:

- `hashlib.sha256(data)`..... : Esta função cria um objeto hash que utiliza o algoritmo SHA-256. É importante notar que ela espera que os dados de entrada (`data`) estejam no formato de bytes. É por isso que strings em Python precisam ser codificadas (por exemplo, usando `b"meus dados"` ou `"meus dados".encode()`) antes de serem passadas para a função;
- `obj_hash.hexdigest()`..... : Este é um método que se aplica ao objeto *hash* (`obj_hash`) criado (por `hashlib.sha256()`). Ele retorna o *hash* calculado como uma *string* de caracteres hexadecimais, que é o formato de 64 caracteres que comumente vemos e utilizamos.

Exemplo Prático em Python:

O código a seguir demonstra o "efeito avalanche". Note que a única diferença entre `strEntrada1` e `strEntrada2` é a capitalização da letra "O", mas os hashes gerados são completamente distintos.

```python
import hashlib

# ------------------------------------------------
# Entrada 1: 'Ola Mundo' com 'O' maiúsculo
strEntrada1 = b'Ola Mundo'
objHash1    = hashlib.sha256(strEntrada1).hexdigest()
print(f'Entrada 1 ..: \'{strEntrada1.decode()}\'')
print(f'Hash 1 .....:  {objHash2}\n')


# ------------------------------------------------
# Entrada 2: 'ola Mundo' com 'o' minúsculo
strEntrada2 = b'ola Mundo'
objHash2    = hashlib.sha256(strEntrada2).hexdigest()
print(f'Entrada 2 ..: \'{strEntrada2.decode()}\'')
print(f'Hash 2 .....:  {objHash2}\n')
```

1.3. O que é o Nonce?

O termo "**nonce**" é uma abreviação de "***number used once***" (número usado uma única vez). No contexto da mineração de Bitcoin, o nonce é a peça-chave que os mineradores ajustam para resolver o quebra-cabeça computacional. Trata-se de um número, geralmente começando em zero, que é sistematicamente alterado a cada tentativa de encontrar um *hash* válido.

Sua função principal é garantir que a entrada para o algoritmo SHA-256 seja diferente a cada cálculo, mesmo que os dados das transações permaneçam os mesmos. Como o SHA-256 produz uma saída completamente diferente para qualquer pequena mudança na entrada, a alteração do nonce permite que os mineradores gerem bilhões de hashes distintos por segundo.

A Função `nonce.to_bytes()`:

Para que o número do nonce possa ser combinado com os dados das transações (que já estão em bytes), ele também precisa ser convertido para o formato de bytes. Em Python, fazemos isso com o método `.to_bytes()`, que se aplica a qualquer variável inteira:

- `nonce.to_bytes(length, byteorder)` .....: Este método converte um número inteiro (`nonce`) em uma sequência de bytes, onde `length` define o número de bytes que a sequência resultante terá. No protocolo Bitcoin, o campo do nonce tem um tamanho fixo de 4 bytes e `byteorder` define a ordem dos bytes (endianess). `'big'` significa ***big-endian***, a ordem em que o byte mais significativo vem primeiro. Essa é a ordem padrão para protocolos de rede (***network byte order***) e é a exigida pelo Bitcoin para a montagem dos blocos.

Exemplo Prático em Python:

O código abaixo simula exatamente o processo de tentativa e erro. A cada iteração do loop, o *nonce* é combinado com os dados da transação para criar uma nova entrada para o hash.

```python
import hashlib

# Dados da transação (fixos)
bytesTransacoes = b'dados da transacao'

# Loop para simular a busca do minerador
for nonce in range(5):
    # Converte o nonce para bytes e o combina com as transações
    # O formato 'big' é o padrão de rede (network byte order)
    bytesEntrada = nonce.to_bytes(4, 'big') + bytesTransacoes

    # Calcula o hash da entrada combinada
    hashCalculado = hashlib.sha256(bytesEntrada).hexdigest()

    # Exibe o resultado de cada tentativa
    print(f'Tentativa com Nonce {nonce}: {hashCalculado}'')
```

1.4. A Questão do "Endianness"

Quando um computador armazena ou transmite dados maiores que 1 byte (como números inteiros), a ordem em que esses bytes são organizados é definida pelo "*endianness*". Isso é crucial em redes e programação de baixo nível.

- **Big-endian:**
  - Armazena o byte mais significativo na primeira posição de memória;
  - Essa ordem é intuitiva, pois corresponde à forma como lemos números da esquerda para a direita;
  - É o padrão utilizado em protocolos de rede (TCP/IP), sendo também chamado de network byte order;
  - Exemplo: O número `0x12345678` é armazenado como `12 34 56 78`.

- **Little-endian:**
  - Armazena o byte menos significativo na primeira posição de memória;
  - A ordem de leitura é da direita para a esquerda, o que pode ser contraintuitivo;
  - É o padrão utilizado por processadores de arquitetura x86 e x86_64, como os da Intel e AMD;
  - Exemplo: O número `0x12345678` é armazenado como `78 56 34 12`.

A incompatibilidade de *endianness* entre sistemas pode levar à interpretação incorreta de dados se a conversão adequada não for realizada.

1.5. Estudo de Caso: Simulando a Mineração com Python

Para exemplificar o processo, vamos analisar um código em Python que simula a busca por um *nonce* válido.

```python
# Importa a biblioteca padrão para funções de hash, incluindo SHA-256
import hashlib

# Define o conjunto de transações como bytes.
# Aqui usamos uma string simples representando três transações ficticias.
binTransacoes = b'transacao1;transacao2;transacao3'

# Define o nível de dificuldade, ou seja, o número de zeros que o hash deve começar.
# Quanto maior o número, mais difícil (e demorado) encontrar um nonce valido.
intDificuldade = 4

# Inicializa o nonce com zero. Esse número será ajustado até encontrar um hash valido.
nonce = 0

# Inicio de um loop infinito que tenta encontrar o hash com o número de zeros exigido.
while True:
    # Concatena o nonce (convertido para 4 bytes no formato big-endian)
    # com os dados da transação.
    bytesEntrada = nonce.to_bytes(4, 'big') + binTransacoes

    # Calcula o hash SHA-256 e converte o resultado para uma string hexadecimal.
    hashResultado = hashlib.sha256(bytesEntrada).hexdigest()

    # Verifica se o hash gerado começa com a quantidade de zeros definida
    # pela dificuldade. Se o o nonce foi encontrado o loop é encerrado.
    if hashResultado.startswith('0' * intDificuldade): break

    # Caso contrário, incrementa o nonce e tenta novamente.
    nonce += 1

print(f'Nonce encontrado ..: {nonce}')
print(f'Hash valido .......: {hashResultado}')
```

Analisando o Código:

Vamos decompor o script para entender cada parte do processo de mineração simulado:

- **Linhas 1-6: Configuração Inicial**
  - `import hashlib`: Importa a biblioteca necessária para utilizar algoritmos de hash, como o SHA-256.
  - `binTransacoes = ...`: Define os dados das transações que farão parte do bloco. A letra b antes da *string* indica que ela deve ser tratada como uma sequência de bytes.
- **Linhas 8-13: Definição da Dificuldade e do Nonce**
  - `intDificuldade = 4`: Estabelece que um *hash* válido deve começar com quatro zeros. Aumentar este número eleva exponencialmente o tempo de processamento.
  - `nonce = 0`: Inicializa o *nonce*, que será o número variável em nossa busca.
- **Linha 16: O Loop de Mineração**
  - `while True:`: Inicia um loop infinito que só será interrompido quando um *nonce* válido for encontrado.
- **Linhas 17-19: Construção dos Dados de Entrada**
  - `entrada = nonce.to_bytes(4, 'big') + binTransacoes`: Esta é a etapa mais importante.
  - `nonce.to_bytes(4, 'big')`: Converte o número do *nonce* para uma sequência de 4 bytes no formato **big-endian**.
  - `+ binTransacoes`: Concatena o *nonce* convertido com os bytes das transações para formar o dado de entrada completo que será "hasheado".
- **Linhas 21-22: Cálculo do Hash**
  - `hash_resultado = hashlib.sha256(entrada).hexdigest()`: Calcula o *hash* SHA-256 da entrada e converte o resultado binário para uma *string* hexadecimal para facilitar a verificação.
- **Linhas 24-26: Verificação da Condição**
  - `if hash_resultado.startswith('0' * intDificuldade): break`: Verifica se a *string* do *hash* começa com o número de zeros definido pela `intDificuldade`. Se a condição for verdadeira, o loop é quebrado, pois a solução foi encontrada.
- **Linhas 28-29: Incremento do Nonce**
  - `nonce += 1`: Caso o *hash* não seja válido, o *nonce* é incrementado em 1, e o loop recomeça com o novo valor.
- **Linhas 31-32: Exibição do Resultado**
  - `print(...)`: Ao final, exibe o *nonce* que resolveu o problema e o *hash* válido correspondente.

1.6. Expansão e Aprofundamento

Este exemplo simplificado serve como base para explorar conceitos mais avançados, como:

- Medir o tempo de execução para diferentes níveis de dificuldade;
- Exibir o progresso da mineração a cada X tentativas;
- Testar com diferentes conjuntos de transações para observar a mudança no *hash* final;
- Implementar múltiplos threads para simular a competição entre mineradores.

1.7. Leitura Adicional Recomendada

Para os alunos que desejam aprofundar seus conhecimentos sobre os princípios fundadores do Bitcoin e entender diretamente a visão de seu criador, é altamente recomendável a leitura do artigo original de Satoshi Nakamoto. Este documento é a base de toda a tecnologia e detalha a solução para o problema do gasto duplo sem a necessidade de uma autoridade central.

- **Artigo Original (em Inglês)**.....: *Bitcoin: A Peer-to-Peer Electronic Cash System*
  - https://bitcoin.org/bitcoin.pdf
- **Tradução (em Português)** .....: *Bitcoin: Um sistema de dinheiro eletrônico peer-to-peer*
  - https://bitcoin.org/files/bitcoin-paper/bitcoin_pt_br.pdf