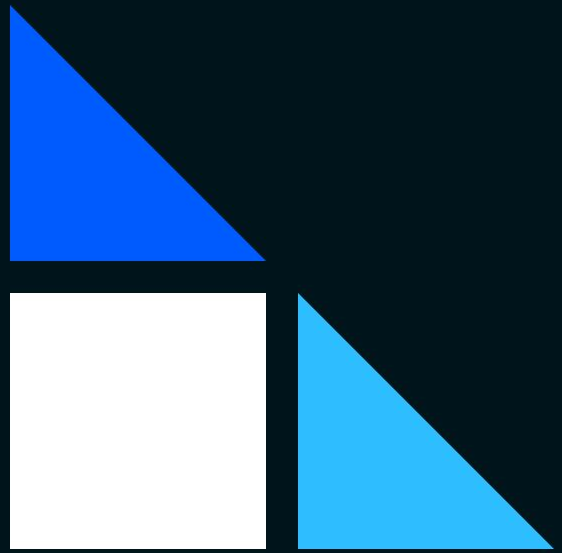
Blockchain In Rust

#02: Mining



geeklaunch

not a geek? start today!

Before we start

Download and install Rust if you want to code along: https://www.rust-lang.org/

If you're on Windows, you'll also need Microsoft Visual C++ Build Tools 2017: https://visualstudio.microsoft.com/downloads/#build-tools-for-visual-studio-2017. Alternatively, it's *very easy* to install Rust on the Windows Subsystem for Linux (WSL).

If you're on Linux, you'll need some form of GCC, which probably came preinstalled on your system.

Optionally, you may also want to install Git: https://git-scm.com/

The code written in this series can be found at: https://github.com/GeekLaunch/blockchain-rust

Concept: Mining

Block Hashing: Review

The current version of Bitcoin uses SHA-256^2 (hash of hash of data) for its proof-of-work algorithm "hashcash"

1. Concatenate together all the bytes composing the block's fields (aside from the hash field)
2. Generate unique data fingerprint: the hash

```
SHA-256("GeekLaunch") =
a17d5669f2148e2982baab7c0b4c7d81100c7cf52c45a8d7deb429aeba156ea6

SHA-256("Geek Launch") =
1d8e72d84c2b72777008ec1db496712096f6c8ab8e943e4fabdb796c40fdd472

SHA-256([196, 201, 152, 0]) =
1d8e7257e1778f4a2cde1374987eafd6f279366512389acfe7a07e56f93da5b5
```

The first 3 bytes match. This took my laptop 17 minutes 44 seconds to calculate (brute force). Imagine how long it would take to find a 100% matching collision! **Remember this—it's important!**

Difficulty

Bitcoin stores the difficulty value more compactly than this, but this is simpler and we don't have to worry about space efficiency

SHA-256 generates a 32-byte hash. **Difficulty** (in our case) specifies the unsigned 128-bit integer value that the *most significant* 16 bytes of the hash of a block must be less than before it is considered "valid" (if those bytes are interpreted as a single number instead of a series of bytes). Difficulty will be stored as a field of the Block struct.

Difficulty could also be expressed as:

- The first *n* bytes of the hash that must be zero
- The number of bits or bytes at the beginning of the hash that must be zero

These options are essentially different ways of expressing the same thing.

See: https://en.bitcoin.it/wiki/Difficulty#How_is_difficulty_stored_in_blocks.3F

Little vs. Big Endian

Endianness: Order of bytes stored in memory.

Example: 42u32

Hex representation	`0x0000002a`
Stored in big-endian order	`00 00 00 2a`
Stored in little-endian order (most common)	`2a 00 00 00`

If we treat it like a little endian representation of a number, the *most significant* 16 bytes of our hash will appear at the *end* of our hash's byte vector [16, 32).

See: https://en.wikipedia.org/wiki/Endianness and https://crates.io/crates/byteorder

Implement: Check Difficulty

Nonce

A hash is a unique, reproducible fingerprint for some data. Therefore, to make a “valid” hash (per difficulty), we must somehow change the bytes we send to the function (the pre-image). Remember that even *one small change* to the input changes the resultant hash *drastically*. This effect is commonly called *avalanching*.

Of course, we can’t actually change the information stored in a block willy-nilly. Therefore, we introduce an additional piece of data called a **nonce**: an *arbitrary* (but not necessarily random) value added as a field to each block, and hashed along with the data. Since it has been declared arbitrary, we can change it as we please.

You can think of it like this: generating the correct hash for a block is like the puzzle, and the nonce is the key to that puzzle. The process of finding that key is called **mining**.

Mining Strategy (Algorithm)

1. Generate new nonce
2. Hash bytes (this is the computationally heavy step)
3. Check hash against difficulty
    a. Insufficient? Go back to step 1
    b. Sufficient? Continue to step 4
4. Add block to chain
5. Submit to peers, etc. Since this is out-of-scope for this video and we have no networking capabilities implemented (yet!), we'll just skip this step.

Implement: Mining Algorithm