

# Applied Learnings for Adversarial Distributed Systems

Methods, Explorations for everything  
blockchains

BY  
Supragya Raj

PUBLISHED IN THE WILD



# Contents

<b>1</b>	<b>ZKVMs: Zero Knowledge Virtual Machines</b>	<b>5</b>
1.1	RISC-V as a common target . . . . .	5
1.1.1	Why RISC-V though? . . . . .	5
1.2	The general design of ZKVMs . . . . .	6
1.3	Writing constraints for these instructions . . . . .	6
1.4	Self-modifying code and dealing with it . . . . .	6
1.5	Identifying the program . . . . .	6
1.6	Proof generation without pre-compiles . . . . .	6
1.7	Proof generation with pre-compiles . . . . .	7
<b>2</b>	<b>Version Control using Git</b>	<b>9</b>
2.1	Using Git via SSH . . . . .	9
<b>3</b>	<b>Ethereum</b>	<b>11</b>
3.1	Legacy Ethereum (Ethereum 1.0) . . . . .	11
3.1.1	Accessing Historical Block Hashes: EIP: 2935 . . . . .	11
3.2	Ethereum after Merge (Ethereum 2.0) . . . . .	11
3.2.1	The Proof-of-Stake Block . . . . .	11
3.2.2	Finality Metalayer: Casper FFG . . . . .	12
<b>4</b>	<b>Solana</b>	<b>13</b>
4.1	The Transaction Processing Unit . . . . .	13
4.2	Proof of History: VDF based transaction ordering . . . . .	13
4.3	Entries, Slots, Bank . . . . .	13
<b>5</b>	<b>General Concepts in Adversarial Distributed Systems</b>	<b>15</b>
5.1	FLP Impossibility Theorem . . . . .	15
5.1.1	The FLP Impossibility Claim . . . . .	15
5.1.2	Key concepts . . . . .	16
5.1.3	Consensus, System Configurations, Events and Schedules . . . . .	17
5.2	Proof of Stake: The Nothing at Stake problem . . . . .	17
5.2.1	No penalties = No canonicalization . . . . .	18
5.2.2	Example: The 1% attacker . . . . .	18
5.2.3	The Tragedy of the Commons . . . . .	18
5.2.4	Simulating the "wrong choice is costly" . . . . .	18
5.3	Wealth Inequality and the Gini Coefficient . . . . .	19
5.4	Miner's Extractable Value (MEV) . . . . .	19
5.4.1	Basic on-chain DeX based arbitrage . . . . .	19
5.5	The ever growing state, state expiry . . . . .	19
5.6	Rebuilding the chain / Chain Derivation . . . . .	20
5.7	Supporting a different virtual machine . . . . .	20

<b>6</b>	<b>Layer 2 Scaling Solutions</b>	<b>21</b>
6.1	General classification . . . . .	21
6.2	Where sits the enourmous data? . . . . .	21
6.3	Plasma: Each one save one . . . . .	22
6.3.1	Operator misbehaviors and exits . . . . .	22
6.3.2	Plasma Cash . . . . .	22
6.4	Bottlenecked by DA . . . . .	22
6.5	Rollups and levels of confirmation . . . . .	23
6.6	Faster Layer 2 Interoperability . . . . .	23
<b>7</b>	<b>Golang</b>	<b>25</b>
<b>8</b>	<b>Rust</b>	<b>27</b>
8.1	Iterators . . . . .	27
<b>9</b>	<b>Binary Search</b>	<b>29</b>
9.1	Basic: Minimize $k$ such that $f(k)$ is true . . . . .	29
9.2	Advanced: Search spaces with <code>feasible(x) -&gt; bool</code> . . . . .	29
9.2.1	Problem [SeqShip]: Sequential objects shipping within $D$ days . . . . .	29
9.2.2	Problem [KokoBanana]: Monkey eating piles of bananas . . . . .	30
9.2.3	Problem [SplitM]: Split array into $m$ subarrays . . . . .	30
9.3	Advanced: Search spaces with <code>enough(x) -&gt; bool</code> . . . . .	31
9.3.1	Problem [KSmallMul]: K-th smallest number in multiplication table . . . . .	31
<b>10</b>	<b>Dynamic Programming</b>	<b>33</b>
10.1	Memoization . . . . .	33
10.2	Most important subproblems . . . . .	34

# Chapter 1

## ZKVMs: Zero Knowledge Virtual Machines

This chapter intends to give a brief overview of the zero-knowledge virtual machines and their current state of usability.

### 1.1 RISC-V as a common target

As soon as we find ourselves exploring ZK-VMs, we quickly find ourselves with a bunch of projects such as RISC-Zero, Succinct's SP1, Mozak, Polygon Miden, Jolt etc, all of which for some reason are RISC-V compatible. What that means is that they expect input instructions to be composed of RISC-V instruction set.

#### 1.1.1 Why RISC-V though?

Modern compiler infrastructure such as that of LLVM makes the code generation process modular. A lot of programming languages can be supported if we target RISC-V, for example C++ through Clang, Rust through RustC, Golang through TinyGo can all convert into LLVM bytecode. This can at the end be compiled down to a well known small instruction set target such as RISC-V. RISC-V essentially is the 5th reduced-instruction-set-architecture that came out of UC Berkeley; hence the name. The main reason however that RISC-V becomes a compilation target of interest is because of two simple reasons apart from language-support: **minimalism** and **modularity**.

**Minimalism in RISC-V:** RISC-V is relatively simple ISA. It has 32 registers that can hold 32 bits each (`x0-x31`) alongside a special register `pc` that holds the address of currently executing instruction's address. The base 32-bit version of RISC-V dealing in integers is `riscvi` and has well-known simple instructions like `ADD`, `JAL`, `BNE` etc. All of these are relatively simple to describe.

**Modularity in RISC-V:** RISC-V provides a base, but also provides a bunch of useful extensions that one can opt-in to. For example, `+m` provides functionality such as multiplication (and division), `+f` provides floating point numbers, `+a` gives access to atomics among others. This modularity allows us to choose the tightest subset of features to implement for a ZKVM.

RISC-V compatible ZK-VMs are mostly targeted towards `riscvi+m`.

## 1.2 The general design of ZKVMs

## 1.3 Writing constraints for these instructions

## 1.4 Self-modifying code and dealing with it

## 1.5 Identifying the program

## 1.6 Proof generation without pre-compiles

While the numbers described below present the situation as seen in December 2024, they still give good information on what we can expect in the signature verification regime. ED25519 is one of the signature schemes used quite commonly in blockchains like Solana, Tendermint-based chains like Cosmos as well as Polkadot.

Test Description	Execution Trace Generation Time	RISC-0 Segments Count	Total Cycles	User Cycles	Proving Time
ED25519 1 signature no-precompile 3-byte msg	91ms	4	3.67M	3.12M	130s
ED25519 1 signature no-precompile 3000-byte msg	91ms	4	4.19M	3.48M	147s
ED25519 2 signatures no-precompile 3-byte msg	171ms	7	6.81M	6.06M	242s
ED25519 2 signatures no-precompile 3000-byte msg	191ms	8	7.60M	6.78M	270s

Analysis of the above gives us the following key points:

- The proof generation procedure is undertaken in two stages: a simulation phase where the execution "trace" is generated and the proof generation phase where the execution trace is proven.
- Compared to the actual proving time in each which includes both the stages, the first stage (execution "trace" generation) takes miniscule amount of time. For example in the test running 1 ed25519 signature verification in zkvm, 91ms were taken in execution "trace" generation step while around 130,000ms were taken for proof generation.
- On average, on a very beefy machine without GPU, Risc-0 can clock in around **28,000 instruction / seconds** of proving.
- In large calculations, the verification is broken down into multiple smaller proving segments, limiting on average, proving time **30s / segment, 8GB RAM usage / segment**. Maybe we can use parallelization of these segments for faster proving, but we need to see who "proves things". Also, segmentation of proof provides larger overhead in proof accumulation and continuation checks not accounted for in these benchmarks.

Another important test we run is for a hash function: SHA256. Alongside ED25519, SHA256 becomes the primary underlying algorithm used extensively in chains like Solana

Test Description	Execution Trace Generation Time	RISC-0 Segments Count	Total Cycles	User Cycles	Proving Time
SHA256 1 byte	3.21ms	1	131K	8.9K	4.5s
SHA256 10 byte	3.25ms	1	131K	8.9K	4.5s
SHA256 100 byte	3.39ms	1	131K	15.4K	4.62s
SHA256 1,000 byte	4.9ms	1	262K	100K	9.26s
SHA256 10,000 byte	20.5ms	1	1.05M	0.96M	37s
SHA256 100,000 byte	175.43ms	10	10.48M	9.5M	368s

## 1.7 Proof generation with pre-compiles

A few other things we learn from the above experiments:

- We find that zkVM needs to run at least  $2^{17}$  **cycles** of instructions regardless of what our code does. If instructions are less, the rest of the rows are padded, which amounts to extra zkvm cycles.
- The growth of total cycles required for execution grows linearly with amount of bytes in question.





## Chapter 2

# Version Control using Git

### 2.1 Using Git via SSH

We can use GIT via a specific SSH command override, useful for when we want to use a specific key for git clone or git push. This can be done via:

```
GIT_SSH_COMMAND='ssh -i ~/.ssh/your_private_key' git push
```



# Chapter 3

## Ethereum

### 3.1 Legacy Ethereum (Ethereum 1.0)

#### 3.1.1 Accessing Historical Block Hashes: EIP: 2935

For various applications, it is necessary to access the historical block hashes of the Ethereum blockchain. One such reason is that it provides a good source of randomness. Another good reason is to build an "introspection engine" where contracts on Ethereum can introspect its own history.

Block hashes of the current block can not be known during the context of execution of a transaction. One of the reasons for this is that during execution of a transaction, the block's receipt hash is not built yet, since that in turn depends on all the transactions in the block combined, each transaction cannot have the block-hash.

Ethereum however, has an opcode called `BLOCKHASH` which allows contracts to access the block hash of the 256 most recent blocks. This opcode is useful for applications that require randomness, but it is not enough for applications that require access to the entire history of block hashes.

**But what limits the access to historical block hashes more than 256 blocks old?** There have been two separate known attempts at solving this. One by Vitalik himself via [EIP-210](#) and another by Vitalik, Tomasz (Netermind) etc via [EIP-2935](#). As of writing, EIP-210 is marked as stagnant and EIP-2935 is in "Review" state since 4 years.

The primary idea of EIP-2935 was to create a new storage trie that stores the block hashes of all blocks. This would have been located at a special location: `0xffffffff_ffffffff_ffffffff_ffffffff_fffff`. This address would essentially be just a state without a bytecode, making it different from all other accounts. Hence, a new opcode `STOP` could

### 3.2 Ethereum after Merge (Ethereum 2.0)

#### 3.2.1 The Proof-of-Stake Block

In every slot (spaced twelve seconds apart) a validator is randomly selected to be the block proposer. They bundle transactions together, execute them and determine a new 'state'. They wrap this information into a **block** and pass it around to other validators.

**Other validators re-execute and validate:** Validators who hear about the new block re-execute the transactions to ensure they agree with the proposed change to the global state. Assuming the block is valid, they add it to their own database.

If a validator hears about two conflicting blocks for the same slot they use their fork-choice algorithm to pick the one supported by the most staked ETH.

Previously, the Ethereum's Proof-of-Work Block (PoW Block) only contained execution related information. Post-Merge, the new Proof-of-Stake Block (PoS Block) will contain ACE information:

- **Administration** (the metadata of the block)
- **Consensus** (Beacon Chain co-ordination)
- **Execution** (Block data)

This makes PoW Block more or less a subset of the new PoS Block. See [3.1](#).

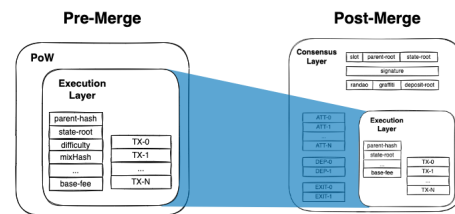


Figure 3.1: The Proof-of-Stake Block

### 3.2.2 Finality Metalayer: Casper FFG

Talk about:

- Plausible Liveness
- Accountable Safety

## Chapter 4

# Solana

Solana seems to be an interesting trade-off choice in the blockchain trilemma when compared to the likes of Ethereum. The blockchain trilemma is a trade-off between decentralization, security and scalability. Solana seems to be focusing more on the scalability, while security and decentralization are arguably worse as compared to Ethereum.

Very briefly, the blockchain trilemma goes as follows:

- **Decentralization:** The more decentralized a blockchain is, the more secure it is. This is because it is harder for a single entity to take over the network. This is often measured using the *nakamoto coefficient*<sup>1</sup>.
- **Security:** The more secure a blockchain is, the more guarantees it can provide, the harder it becomes for an adversary to compromise the system. Guarantees come in the form of economically-backed guarantees (which are subject to elements such as total tokens staked in a network etc) or mathematically-backed guarantees (such as zk proofs, hash non-collision etc.).
- **Scalability:** The more scalable a blockchain is, the more user requests, often measured in form of "transactions per second" or TPS it can perform.

Solana focuses a whole lot more on the "transactions per second" part of the blockchain, a.k.a. the "scalability" aspect.

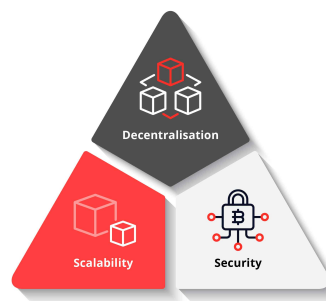


Figure 4.1: The blockchain trilemma

### 4.1 The Transaction Processing Unit

### 4.2 Proof of History: VDF based transaction ordering

### 4.3 Entries, Slots, Bank

---

<sup>1</sup>**Nakamoto Coefficient:** First proposed by Balaji Srinivasan and Leland Lee and named in honor of Satoshi Nakamoto, the pseudonymous creator of Bitcoin, the Nakamoto Coefficient is a measure of the smallest number of independent entities that can act collectively to shut down a blockchain. It is often closely related to the "Gini Coefficient" of the economical system under question. For current nakamoto number of public blockchains using Proof-of-Stake, see <https://nakaflo.io/>



## Chapter 5

# General Concepts in Adversarial Distributed Systems

### 5.1 FLP Impossibility Theorem

The domain of distributed systems saw a major breakthrough in 1978 when Leslie Lamport in his paper "[Time, Clocks, and the Ordering of Events in a Distributed System](#)" introduced the concept of logical clocks. Effectively, this paper presented an affirmative answer to the following question:

**1978 Leslie Lamport.** Can we impose total ordering to events received by different nodes in a distributed system?

As a primer, a partial ordering is an order on a set of elements which satisfies the following properties:

- **Reflexivity:**  $a \leq a$  i.e. an element is always ordered with itself
- **Antisymmetry:** If  $a \leq b$  and  $b \leq a$ , then  $a = b$ , i.e. no two distinct elements precede each other.
- **Transitivity:** If  $a \leq b$  and  $b \leq c$ , then  $a \leq c$ , i.e. if  $a$  precedes  $b$  and  $b$  precedes  $c$ , then  $a$  precedes  $c$ .

A total ordering is a partial ordering which satisfies the following additional property that each pair of elements is comparable, i.e. for any two elements  $a$  and  $b$ , either  $a \leq b$  or  $b \leq a$ .

This seemed at the time to be promising for the distributed systems community, as it led to seeming guarantees of consensus over a strict order in distributed systems.

However, a paper that came out in 1985 by Fischer, Lynch and Paterson, titled "[Impossibility of Distributed Consensus with One Faulty Process](#)" showed the following claim is not always true:

**1985 Fischer, Lynch, Paterson.** Can we always come up with a way of getting computers to agree on a common value in a distributed system, even if some of the computers are faulty?

This paper is often referred to as the **FLP Impossibility Theorem**. Let's understand this further.

#### 5.1.1 The FLP Impossibility Claim

We take the FLP paper's abstract verbatim that reads the following:

**FLP Impossibility Theorem abstract.** The consensus problem involves an **asynchronous system** of **processes**, some of which may be **unreliable**. The problem is for the **reliable processes to agree** on a binary value. In this paper, it is shown that every protocol for this problem has **the possibility of**

**nontermination**, even with only **one faulty process**.

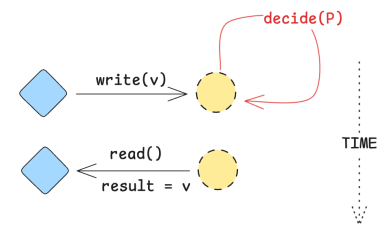
The key takeaway from this abstract, without key terms being defined is: If we exist in a distributed system where each "process" / node is willing to participate in a consensus protocol, which may take infinite amount of time to reach a consensus, with message delays being unbounded, with at-most one process being faulty, i.e. dies without notice to others, then it is not always possible to reach to a common global value.

Before we establish this however, we need to define some key concepts:

### 5.1.2 Key concepts

Let's begin with a setting which is not distributed in nature. Consider a client process CL and a server node  $S$ . The server  $S$  saves a single bit value with itself and presents two distinct operations to the client CL:

- `write(v: 0/1)`: This operation writes the value of a variable given by client at the server  $S$ .
- `read() → 0/1`: This operation returns the value of a variable stored at the server  $S$ .



A crucial component in this system is existence of a function `decide()`, which can be anything that operates on the internal state of the server. A simplest form of `decide()` could be a function that returns the latest value stored in the server. Another variant could be a function that returns the XOR value of the last 5 writes.

Figure 5.1: The single node case of read and write

In our case, we can extend the same construction but to a distributed system outwardly presenting itself as a unit but internally being composed of multiple servers called **nodes**. In essence, we desire that the same `write()`, `read()` and `decide()` functions that were present in the single node case, be present in the distributed system as well and **they should act exactly the same**.

The FLP paper imagines the distributed setting to be describable as follows:

There are  $n$  nodes in the distributed system. The nodes are connected to each other in connected but not necessarily a complete way in a graph-theoretic sense (a.k.a between any two nodes a path exist but not necessarily of length 1). Each process  $p_i$  is defined by  $p_i = (I_i, O_i, f_i)$  where  $I_i$  is input in the set  $\{0, 1\}$ ,  $O_i$  is a one-time write only output in the set  $\{\phi, 0, 1\}$ ,  $\phi$  being the value it exercise if it has not yet been written to, and  $f_i$  is a deterministic function that defines the internal state transition algorithm of  $p_i$ .

Each process  $p_i$  can send a message  $m \in M$  ( $M$  being the "message universe") to some other process  $p_x$  at any time as a tuple  $(p_x, m)$  submitted to a globally singleton multi-set **message buffer** (note, source  $p_i$  is not described in the tuple).

The **message buffer** is a special non-deterministic (the only component in the system that is not deterministic) entity that presents each process with a function `receive(i) → {ϕ, m}`. This function when called with index  $i$ , either returns  $\phi$ , a null message or  $m$ , one of the messages the buffer has received for delivery to process  $p_i$ . In the latter case, upon delivering the message, a tuple  $(p_i, m)$  is removed from the message buffer. Note how message buffer sufficiently describes an asynchronous message passing system. It can choose to return the null message any number of times, in effect simulating unbounded delay.

In a **synchronous** setting however, we have some upper bound on the time it takes for a message to be delivered. In an **asynchronous** setting, we have no such bound. As such, it is impossible to detect if the counterparty process has failed or is just taking a long time to respond.

One should also discuss a bit about what it means to "fail" in a distributed system. Typically we categorize failures in 4 different categories:

- **Fail-stop**: The process halts, notifies everyone of its death and does not respond to any further messages.



- **Crash:** The process halts, does not notify anyone of its death and does not respond to any further messages.
- **Byzantine:** The process can send arbitrary messages to other processes, including lying about its own state.
- **Permissionless Byzantine:** The process can send arbitrary messages to other processes, including but not limited to impersonating a third-party.

In context of the FLP paper, we are concerned with the second type of failure, the "crash" without notification.

We define a **System Configuration**  $C$  as a unique combination of the internal states of all processes and the messages in the message buffer.

### 5.1.3 Consensus, System Configurations, Events and Schedules

Consensus in distributed systems is defined as any algorithm that satisfies the following properties:

- **Agreement:** All non-faulty processes decide on the same value.
- **Termination:** All non-faulty processes eventually decide on a value.
- **Validity:** All non-faulty processes decide on a value that was some process's internal state at the start.  
*Corollary:* If all non-faulty processes start with the same input value, then the decision value must be the same as the input value.

The system configuration  $C$  advances to a new configuration  $C'$  when it encounters **events**.

**Events** are messages  $(p, m)$  from the message buffer. Since there are many different messages that can be extracted from  $C$  due to message buffer being non-deterministic, many different configurations can stem out from single configuration  $C$ . A list of events extracted from  $C$  is called a **schedule**.

If at any system configuration  $C$ , all schedules lead to a final terminating consensus decision of value  $v$ , then we say that the system configuration  $C$  is **decisive / univalent** for value  $v$ . This can also be written as  $v$ -valent. Since we are in binary domain, decisive configurations are either 0-valent or 1-valent.

If at any system configuration  $C$ , different schedules lead to different consensus decisions, then we say that system configuration  $C$  is **indecisive / multivalent**. Since we in binary domain, it also is called *bi*-valent.

## 5.2 Proof of Stake: The Nothing at Stake problem

At a first glance, Proof of Stake seems to be a great idea. It is more energy efficient than Proof of Work, and it is more democratic in the sense that the more tokens you have, the more say you have in the network.

However, it is not quite comparable to Proof of Work. To understand why, we need to notice the following two properties of PoW:

- **Finite prowess:** In proof of work systems, the prowess (often calculated as the number of puzzles a miner can try) of a miner is limited by the amount of computational power they have. Whether they use COTS, ASICs, FPGAs, all are limited inherently. Whenever a miner sees a fork in the chain, they have a choice to make: which chain to mine on. Should they dedicate their prowess to the fork #1 or should they dedicate their prowess to fork #2? Or should they mine on both equally? Regardless of their choice of prowess distribution, their total prowess is limited.
- **Wrong choice is costly:** If a miner chooses to mine on the wrong fork, they lose out on the rewards they could have gotten had they mined on the right fork. This is because the right fork will have more blocks, and hence other miners due to longest chain fork-choice rule will focus their effort onto it, making it "the canonical chain" in due time, leaving the wrong fork blocks to be wasted effort / orphaned.

The above two properties in Proof-of-Work chains, coupled with the assumption that each miner will make their own choice of fork-selection independently, make them inherently a good canonicalization machine, i.e. given enough time all forks will converge to a single chain.

However, in Proof of Stake, the above two properties are not true. The prowess of a validator is not limited by computational power, but by the amount of tokens they have staked. Any validator can freely attest on both the forks. Secondly, since they are not making a choice here, there is no concept of "wrong choice" being costly. This leads to a problem called "Nothing at Stake".

### 5.2.1 No penalties = No canonicalization

In many early (all chain-based) proof of stake algorithms, including [Peercoin](#), there are only rewards for producing blocks, and **no penalties**. This has the unfortunate consequence that, in the case that there are multiple competing forks, it is in a validator's incentive to try to make blocks on top of every fork at once, just to be sure.

The result is that if all actors are narrowly economically rational, then even if there are no attackers, a blockchain may never reach consensus on a "canonical state", and all validators will keep building on both forks. If there is an attacker, then the attacker need to only overpower the altruistic nodes (who would exclusively stake on the original chain), and not rational nodes (who would stake on both the original chain and the attacker's chain), in contrast to proof of work, where the attacker must overpower both altruists and rational nodes.

### 5.2.2 Example: The 1% attacker

TODO: Show how a person controlling >0% stake can canonicalize the wrong fork and thus you cannot trust any fork until its made canonical

### 5.2.3 The Tragedy of the Commons

The Tragedy of the Commons is a concept which states that if many people enjoy unfettered access to a finite, valuable resource, such as a pasture, they will tend to overuse it and may end up destroying its value altogether. Even if some users exercised voluntary restraint, the other users would merely replace them, the predictable result being a "tragedy" for all.

In the context of Proof of Stake, each individual stakeholder might only have a 1% chance of being "pivotal" (i.e. being in a situation where if they participate in an attack then it succeeds and if they do not participate it fails), and so the bribe needed to convince them personally to join an attack would be only 1% of the size of their deposit; hence, the required combined bribe would be only 0.5-1% of the total sum of all deposits. Additionally, this argument implies that any zero-chance-of-failure situation is not a stable equilibrium, as if the chance of failure is zero then everyone has a 0% chance of being pivotal.

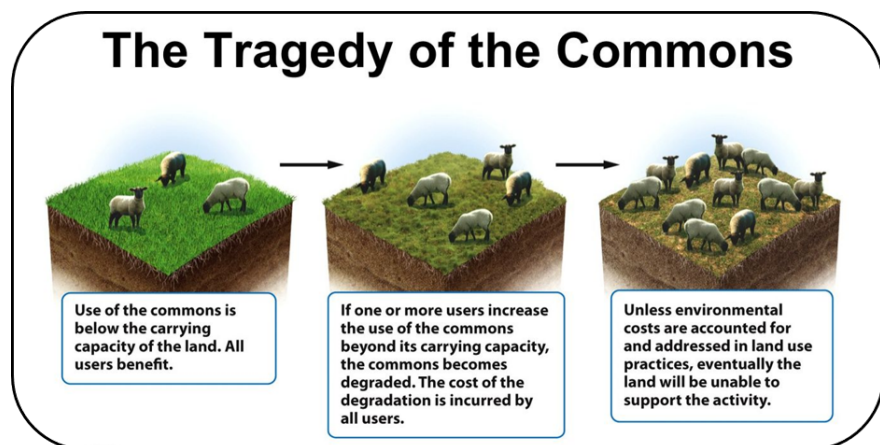


Figure 5.2: Tragedy of the commons

### 5.2.4 Simulating the "wrong choice is costly"

There are two ways in which one can try solving the "Nothing at Stake" problem, both of which involve making the "wrong choice" costly:

- **Slasher:** In the Slasher algorithm, if a validator is caught signing two conflicting blocks on two different forks, they lose their part / entire stake. **TODO: requires elaboration, see why they call it too harsh**
- **Casper FFG:** Every validator is forced to choose a fork to validate on. If there are two competing chains, A and B, then if a validator creates a block on B, they get a reward of +R on B, but the block header can be included into A (in Casper this is called a "dunkle" or *proof of malfeasance*) and on A the validator suffers a penalty of -F (possibly  $F = R$ ). **TODO: show in Casper design how this is done**

## 5.3 Wealth Inequality and the Gini Coefficient

Many popular blockchains follow a Proof-of-Stake model of consensus. In PoS, the more tokens you have staked, the more faith you present to the network. In return, the network rewards you with more tokens for protocol validation services you provide to the network.

This mainly means that a certain actor with more tokens can stake more tokens, and assuming does follow protocol right, earn more tokens as a result. This is a positive feedback loop, and is often referred to as the "rich get richer" problem. This is a problem because it leads to wealth inequality in the network.

**TODO: talk about lorenz curve and gini coefficient**

## 5.4 Miner's Extractable Value (MEV)

Miner's Extractable Value (MEV) is a term coined by Phil Daian et al. in their [Flash Boys 2.0](#) paper. It refers to the value that miners can extract from the transactions they include in a block. MEV is a result of the fact that miners have the ability to order transactions in a block and can choose which transactions to include or exclude. This gives them the power to front-run transactions, re-order transactions, and censor transactions.

Sometimes MEV is also described as "maximal extractable value".

General purpose blockchains provide an almost turing-complete environment<sup>1</sup> for executing smart contracts / programs. Smart contract / programs are sequence of steps or rules that can be deployed at will. Some of these major programs are Decentralized Exchanges (DEX), Automated Market Makers (AMM), and lending protocols. These programs are often used to trade assets, provide liquidity, and borrow assets.

### 5.4.1 Basic on-chain DeX based arbitrage

If an arbitrage opportunity exists of the following format:

- **Transaction 1:** Decentralized exchange  $D1$  is selling asset  $A$  in exchange of  $B$  at rate  $x : 1$
- **Transaction 2:** Decentralized exchange  $D2$  is buying asset  $A$  and giving  $B$  at rate  $y : 1$
- **Arb Condition:**  $x > y$ , i.e.  $D1$  gives more  $A$  per unit of  $B$  than  $D2$  needs to exchange for a unit of  $B$

## 5.5 The ever growing state, state expiry

State expiry refers to removing state from individual nodes if it hasn't been accessed recently. There are several ways this could be implemented, including:

**Expire by rent:** charging "rent" to accounts and expiring them when their rent reaches zero. This design is implemented in Solana.

**Expire by time:** making accounts inactive if there are no reading/writing to that account for some amount of time

<sup>1</sup>Almost in the sense that execution is still bounded to some maximum number of steps, called gas limit in Ethereum and compute limit in Solana

## 5.6 Rebuilding the chain / Chain Derivation

## 5.7 Supporting a different virtual machine

Execution of code in any blockchain assumes an "execution environment". This execution environment is often called the "virtual machine" that blockchain is running. For example, Ethereum runs the EVM, the **E**thereum **V**irtual **M**achine. Solana runs the SVM similarly.

A curious question arises: **Can a virtual machine non-native to a blockchain be introduced as a co-execution environment?**. The answer to this is a resounding yes. However, one needs to question the intent of doing this. Almost all execution environments are [turing complete](#), however the cost of executions may be prohibitive at times.

One such use-case is extending the EVM with capabilities to run WASM or RISC-V code. A couple interesting projects doing this would be [Arbitrum Stylus](#) and [R55](#).

### Arbitrum Stylus

Stylus extends the execution environment to run WASM-targeted smart contracts. Instead of writing in solidity, one writes the code in languages like Rust, Zig, Go, Sway, Swift. Rust is supported as primary language and [stylus-sdk](#) is used to build such programs.

### R55-Eth

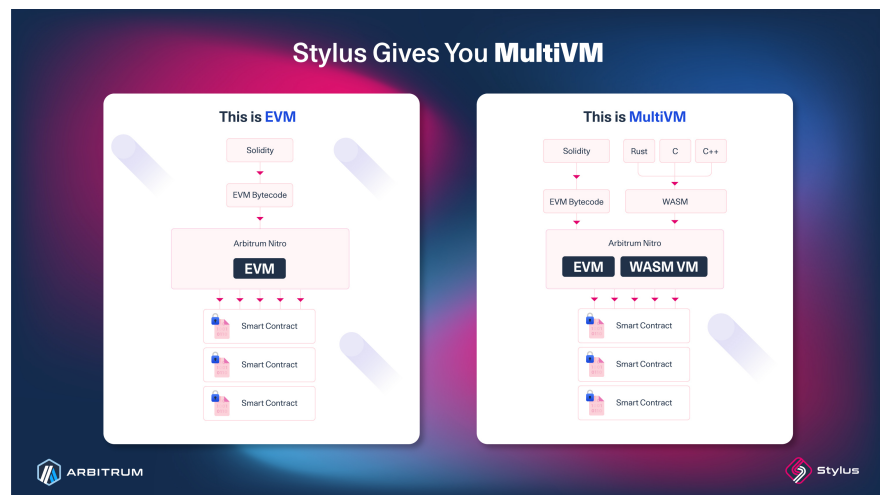


Figure 5.3: Arbitrum Nitro

## Chapter 6

# Layer 2 Scaling Solutions

### 6.1 General classification

We classify Layer 2 scaling solution according to their setup:

- **Plasma:** A tree of sidechains (chains of chains) where each user is asked to ensure the part of state it owns is available. Here, significant interaction with L1 is only required in case of dispute resolution. This came much before other solutions (proposed by Joseph Poon and Vitalik Buterin in 2017) but it was too complex to build a useful plasma.
- **Rollup:** An ecosystem running in parallel to a Layer 1 solution that has its own state transition function (STF). After a bunch of transactions and blocks are created, the rollup posts transaction data / state-diff on chain and proves the combined state transition via optimistic or succinct zk proof.
- **Validium:** Similar to a rollup, just that the data-availability of the transaction is kept not on L1, but a separate *data-availability committee* or the validium operator themselves. Validium saves on recurring cost to make data available on L1, but at the cost of security.

### 6.2 Where sits the enormous data?

Whenever a layer 2 scaling solution is built, the layer 1 contract is provided with the root commitment of the updated state. But a commitment is useless unless it can be opened. A *witness* links a leaf to the commitment and it requires a view of the state / transaction data for things like disputes and withdrawals.

Where does this data exist for generating a witness to do anything useful? A lot of answers can come up:

- Let the data sit on the Layer 1 itself, the rollup route. This is the simplest of the solutions but seldom the most practical. Layer 1s are expensive to post data to.
- Let the data sit on some other special data-availability layer which has some presence on the Layer 1 concerned. For example, posting on celestia. This is the validium route. Here you still pay to post on Celestia albeit much less.
- Let the data sit on a data-availability layer even cheaper than celestia, maybe with a smaller quorum. This is an even cheaper validium route. But the trade-off is security, as the data-availability layer may not be as secure as for example celestia.
- Let the data sit with the users themselves. Let them maintain the data and post it when required. This is the plasma route. This is the most decentralized of the solutions but also the most complex to build. Also, it requires the users to be online all the time. A pretty bad UX.

## 6.3 Plasma: Each one save one

The setup of Plasma entails that if each part of the state has a well-defined owner, then the owner can be made responsible for keeping their part of the system available.

Essentially, all Plasma chains have an **Operator** who keeps the chain moving. The system is setup optimistically with an intrinsic level of trust on this operator. If the operator misbehaves, it remains the responsibility of the users to exit the chain.

Plasma chains are multi-layered chains where every "Parent Chain" settles for its child chains. The topmost "Parent Chain", the root chain settles on a well-known settlement platform and calls it the Layer 1. Each parent chain is responsible for three basic events through its contract:

- **New Deposit:** Whenever anyone submits a new token deposit of the form (depositAddress, amount, blockHeight).
- **Block Progression:** Only submit the root-hash of the block, *not the data contained within it*. The data within hash to be kept in a good condition (and available) by the user. This takes the form (blockHeight, rootHash).
- **New Withdrawal:** If the operator misbehaves, the users are supposed to exit the plasma chain. This request is posted directly on Layer 1, as it may be assumed that the operator is not available / misbehaving. This takes the form (withdrawaladdress, amount, blockheight, txIndex, outputIndex).

### 6.3.1 Operator misbehaviors and exits

### 6.3.2 Plasma Cash

In **Plasma Cash**, since users are only keeping track of their own tokens, they don't know who owns any of the other tokens. When a user wants to send their token to another user, they need to prove that they actually own that token! This creates a problem where each user is expected to be *perennially online* to ensure they can generate a witness against the current state-root of plasma chain.

In such a setup however, other inefficiencies also arise, because of the non-fungible nature of the tokens involved. In a ledger model of accounting, spending 0.1 Token or spending 20 Token out of a 50 Token balance are the same complexity-wise. However, the same is much harder with NFTs given we have to establish the denominations beforehand for all possible transfers in the future, or deal with splitting and merging of tokens.

## 6.4 Bottlenecked by DA

Layer 2 rollups have been around for a while. The first Layer 2 scaling solution was Plasma, proposed by Joseph Poon and Vitalik Buterin in 2017. The idea was to create a tree of sidechains, each of which would be able to process transactions in parallel. The root chain would be responsible for the security of the system, while the sidechains would be responsible for processing transactions.

Rollups however needs to guarantee that the state of rollup is **data-available**. Otherwise, if state is not available and the rollup-operator stops working, users cannot withdraw. As, even though the root commitment to the entire state is available, the opening are not. This makes rollups constrained by data-availability capacity of the Layer 1 chain.

Ethereum provides 375 KB blobs every 12 seconds.

The following question hence often comes up: *Can we eliminate need for state availability in L2 scalability solutions?*

## 6.5 Rollups and levels of confirmation

## 6.6 Faster Layer 2 Interoperability

In late 2020, Ethereum community started focusing on [rollup-centric roadmap](#). The idea of this was to enable various Layer 2 scaling solutions that may potentially provide a whole lot more of execution capacity than the Ethereum (henceforth to be referred as Layer1 or L1) will be capable of.

However, Layer 2 come with following new concerns:

- **Security:** The security of Layer 2's is the concern of how easy it is to compromise a L2 in comparison to an L1. Generally, in a proof-of-stake system like ethereum, a lot of security is borrowed from stake at play. As of writing, this is around 33.8 million ETH staked / around 136.8 billion USD securing the ecosystem. Can the L2 match this level of security?
- **Decentralization:** Control of the system especially with chains like Ethereum is fairly decentralized. As of writing, around 1 million validators run the nodes for the main chain<sup>1</sup>. Layer 2 solutions hardly have decentralized sequencers and hence sit almost on the far end of decentralization argument. Can the L2 match this level of decentralized controls?
- **Fragmentation:** Every L2 is almost like a separate "room", although situated within the same "house", the L1. This eventually leads to fragmentation of both users, and the liquidity. On a DEX, liquidity is directly tied to the quantity of tokens users have committed to the liquidity pools. If a crypto asset lacks sufficient liquidity, token holders may face difficulty selling their tokens when they wish. L2 brings in another dimension to this problem. Not only are tokens spread thin among various DeFi apps on the same chain, they are now spread among various DeFi apps on *various* chains.

The Interoperability problems is concerned with the *fragmentation* issue brought in by the advent of L2s.

In its most basic essence, Interoperability is concerned with a transaction that intends to affect state of two different L2 rollup chains. For example, a token transfer from *A*'s account on rollup *X* to *B*'s account on rollup *Y* is a transaction that affects the state of both *X* and *Y*.

---

<sup>1</sup>This number is a bit misleading given a lot is contributed by single institutions like [stake.fish](#) running multiple validator nodes given each validator can only stake 32 ETH per validator. [The Ethereum Pectra Upgrade \(EIP-7251\)](#) is supposed to fix it with increased MAX\_EFFECTIVE\_BALANCE the staking balance per validator, in hopes consolidating many such large validators into fewer entities





## Chapter 7

# Golang

Golang things



# Chapter 8

## Rust

RUSTY things

### 8.1 Iterators

IMPLEMENT RLE ITERATOR here: <https://leetcode.com/problems/rle-iterator/description/>



## Chapter 9

# Binary Search

Binary search is a simple technique of problem solving that splits the solution space in half at each step. But when it comes to implementation, it's rather difficult to write a bug-free code in just a few minutes.

A lot of issues arise when implementing binary search such as:

- Off-by-one errors
- Incorrect bounds
- Incorrect mid calculation
- Incorrect condition to update bounds

### 9.1 Basic: Minimize $k$ such that $f(k)$ is true

This is the most basic form. Basically because the search space is given directly. We would be given a vector  $[A_0, A_1, \dots, A_n]$  and we need to find the minimum  $k$  such that  $f(k)$  is true.

### 9.2 Advanced: Search spaces with `feasible(x) -> bool`

Finding the search space could be tricky sometimes<sup>1</sup>. However, one common element in these problems is the aspect of monotonicity. Here, if  $f(k)$  is true, then  $f(k + 1)$  is also true. Following are some good examples that establish the concept.

#### 9.2.1 Problem [SeqShip]: Sequential objects shipping within $D$ days

**Problem:** Assume a sequential line of objects  $O$  with their individual weights described in an array that are situated at port  $A$  and want to be shipped to port  $B$ . Every day, a ship of some maximum weight capacity  $m$  makes trip from  $A$  to  $B$  transporting some objects in  $O$ . Given a maximum days  $D$  that shipping company can tolerate to transport all objects in  $O$ , find lowest possible  $m$ . An example object line could look as follows:

$$O = [6, 7, 1, 2, 3, 4, 5, 8, 9, 10]$$

Given  $D = 5$ , the minimum  $m$  would be 15. Because we can ship items as:  $[6, 7]$ ,  $[1, 2, 3, 4, 5]$ ,  $[8]$ ,  $[9]$ ,  $[10]$ .

**Discussion:** This problem does not have a clear search space *prima facie*, but the thing we realize is it asks us to minimize  $m$ . Now let's run this idea: if some  $m$  works for shipping in  $D$  days, would  $m + 1$  work also? More

<sup>1</sup>A good description is available on Leetcode: <https://leetcode.com/discuss/study-guide/786126/Python-Powerful-Ultimate-Binary-Search-Template.-Solved-many-problems>

concretely, assume that a ship of capacity 10 is able to do the work at hand, can a ship of capacity 11 do it too? The answer is yes. We can call this function  $f(x)$  more aptly `feasible(x) -> bool`.

But what should be the search space? Well, the minimum we need is atleast  $\max(O)$  since we need to ship the heaviest object in some trip, even if it is the only item shipped that day. The maximum we need is probably shipping all objects in one day. So, the search space is  $[\max(O), \text{sum}(O)]$ .

**Solution:** The problem can now be reduced to: Find  $m$  such that  $f(m)$  is true where  $f(m)$  is defined as: *Can we simulate shipping  $O$  in  $D$  days with a ship of capacity  $m$ ?* in the search space  $[\max(O), \text{sum}(O)]$ .

Each simulation is an  $O(n)$  operation where  $n = |O|$ . Maximum simulations is  $\log_2(\text{sum}(O) - \max(O))$ . Hence runtime is  $O(|O| \cdot \log_2(\text{sum}(O) - \max(O)))$ .

### 9.2.2 Problem [KokoBanana]: Monkey eating piles of bananas

**Problem:** Assume a monkey is hungry and wants to eat some bananas. There are  $G$  piles of bananas, each pile  $i$  has  $b_i$  bananas. The monkey can eat  $k$  bananas per hour and has a maximum of  $D$  hours to eat all bananas. Each hour, the monkey can decide any one pile and eat  $k$  bananas from it, if there are less than  $k$  bananas, it eats all but doesn't eat from any other pile that hour. What is the minimum  $k$  for which the monkey can eat all bananas in  $D$  hours?

Example scenarios:

$$G = 4, \text{Piles} = [3, 6, 7, 11], D = 8 \quad k = 4$$

$$G = 5, \text{Piles} = [30, 11, 23, 4, 20], D = 5 \quad k = 30$$

$$G = 5, \text{Piles} = [30, 11, 23, 4, 20], D = 6 \quad k = 23$$

**Discussion:** Very similar to problem [SeqShip], this problem also has a monotonicity property. If  $k$  bananas per hour works, then  $k + 1$  bananas per hour would also work, albeit not the most optimal. This also uses `feasible(x) -> bool` function.

But what is the search space? The minimum  $k$  would be 1 as that's the least a monkey has to eat per hour to make progress hour-on-hour. However, we can probably do better than that. The minimum should be no lesser than  $m = D / \text{sum}(\text{Piles})$  as that's the least number of bananas the monkey has to eat per hour to eat all bananas in  $D$  hours assuming no inefficiencies, i.e. every hour is spent eating exactly  $m$  bananas. This is unlikely since not all  $b_i$  may be exactly divisible by  $m$ .

The maximum  $k$  would be  $\max(\text{Piles})$  as that's the most bananas the monkey can eat in one hour given that in each hour it can only eat from a single pile.

The search space is hence  $[D / \text{sum}(\text{Piles}), \max(\text{Piles})]$ .

Also note an interesting property: a monkey cannot eat any faster than in  $G$  hours. Hence any problem where  $|\text{Piles}| > G$  would have no solution.

### 9.2.3 Problem [SplitM]: Split array into $m$ subarrays

**Problem:** Given an array  $A$  of non-negative integers, split it into  $m$  subarrays such that the maximum sum  $s$  of any subarray is minimized. Example:

$$A = [7, 2, 5, 10, 8], M = 2 \quad s = 18(\text{split}: [7, 2, 5], [10, 8])$$

**Discussion:** The monotonicity property still holds. If  $s$  is a valid maximum sum of any subarray, then  $s + 1$  is also a valid maximum sum of any subarray. This one still makes use of `feasible(x) -> bool` function.

## 9.3 Advanced: Search spaces with `enough(x) -> bool`

### 9.3.1 Problem [KSmallMul]: K-th smallest number in multiplication table

**Problem:** We define a multiplication table as a  $m \times n$  matrix where

$$\text{table}[i][j] = i \times j$$

Given  $m, n, k$ , find the  $k$ -th smallest number in the multiplication table. An example may look as follows:

$$m = 3, n = 3, k = 5 \quad \text{table} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix} \quad \text{Answer: } 3(1, 2, 2, 3, 3)$$

**Discussion:** For Kth-Smallest problems like this, what comes to our mind first is Heap. Usually we can maintain a Min-Heap and just pop the top of the Heap for  $k$  times. However, that doesn't work out in this problem. We don't have every single number in the entire multiplication table. Instead, we only have the height and the length of the table. If we are to apply Heap method, we need to explicitly calculate these  $m \times n$  values and save them to a heap. The time complexity and space complexity of this process are both  $O(mn)$ , which is quite inefficient.

The search space is not clear. But we can define a function `enough(x) -> bool` that tells us if there are at least  $k$  numbers in the multiplication table that are less than or equal to  $x$ . Ofcourse, if there  $x$  satisfies  $k$  numbers, then  $x + 1$  would also satisfy  $k$  numbers. We exploit this monotonicity property.

Now, we intend to beat time to be better than  $O(mn)$ . That means `enough(x) -> bool` should be considerably cheaper to compute. We exploit the following algorithm:

Listing 9.1: Rust implementation of enough function

---

```
fn enough(num: i32, m: i32, n: i32, k: i32) -> bool {
    let mut count = 0;
    // We run iteration for each row
    for row in 1..=m {
        // Each row has (num / row) elements less than or equal to num.
        // However, this has to be contained within n since at most there are n columns.
        let add = (num / row).min(n);
        if add == 0 {
            // Exit and do no further iterations since if there are no numbers
            // lower than num in this row, any further rows won't have any as well.
            break;
        }
        count += add;
    }
    count >= k
}
```

---

The above algorithm runs in  $O(m)$  time.

The search space of course is  $[1, m \times n]$ . For every  $x$  in this search space, we can calculate the number of elements in the multiplication table that are less than or equal to  $x$  in  $O(m)$  time. Hence, the runtime of this solution is  $O(m \cdot \log_2(m \times n))$ .





## Chapter 10

# Dynamic Programming

Dynamic Programming in it's crudest sense is nothing more than "clever bruteforce". What that means is while the solution eventually a large brute-force computation, the *cleverness* is in finding a sub-problem that helps solve for the larger problem instances. The key idea is "instances", not a single instance.

Take the following relation for example:

$$\begin{aligned} F(x) &= \text{Max}(F(x-1) + 1, 1) \\ F(0) &= 0 \end{aligned} \tag{10.1}$$

Here,  $F(2)$  will only be used for calculating  $F(3)$  and nowhere else.

On the other hand, the following relation:

$$\begin{aligned} F(x) &= \text{Max}(\text{Max}(nF(x-n) + 1) \forall n \in [0, x), 1) \\ F(0) &= 0 \end{aligned} \tag{10.2}$$

uses  $F(2)$  for all calculations where  $x > 2$  such as  $F(3)$ ,  $F(4)$ , etc.

The cleverness in the relation 10.2 that can potentially help it get to similar asymptotics as the 10.1 is **memoization**.

### 10.1 Memoization

Memoization is nothing but trade-off of space to gain in time. Naive implementation of 10.2 in code will yield results with almost constant space but exponential time. The time complexity leads to:

$$O(n) = \sum_{i=0}^{n-1} O(i) + 1 \tag{10.3}$$

If however, after every solution of an instance in range  $[0, k)$ , the solution is kept in some constant-time accessible storage  $M$ , the problem instance  $k$  dissolves into:

$$\begin{aligned}
 O(k) &= \sum_{i=0}^{k-1} O(i) + 1 \\
 &= \sum_{i=0}^{k-1} O(1) + 1 \\
 &= k + 1
 \end{aligned}
 \tag{10.4}$$

We are going to use this idea quite generously.

## 10.2 Most important subproblems

Most of the times while we find ourselves looking at different problem statements, many-a-times at their core, they stem from one of the following few recursive relations:

- **O/1 Knapsack:**
- **Unbounded Knapsack:**
- **Longest Common Subsequence (LCS):**
- **Longest Increasing Subsequence (LIS):**