PROOF OF STORAGE

SHOW ME YOUR MEMORY

Ruslan Shevchenko

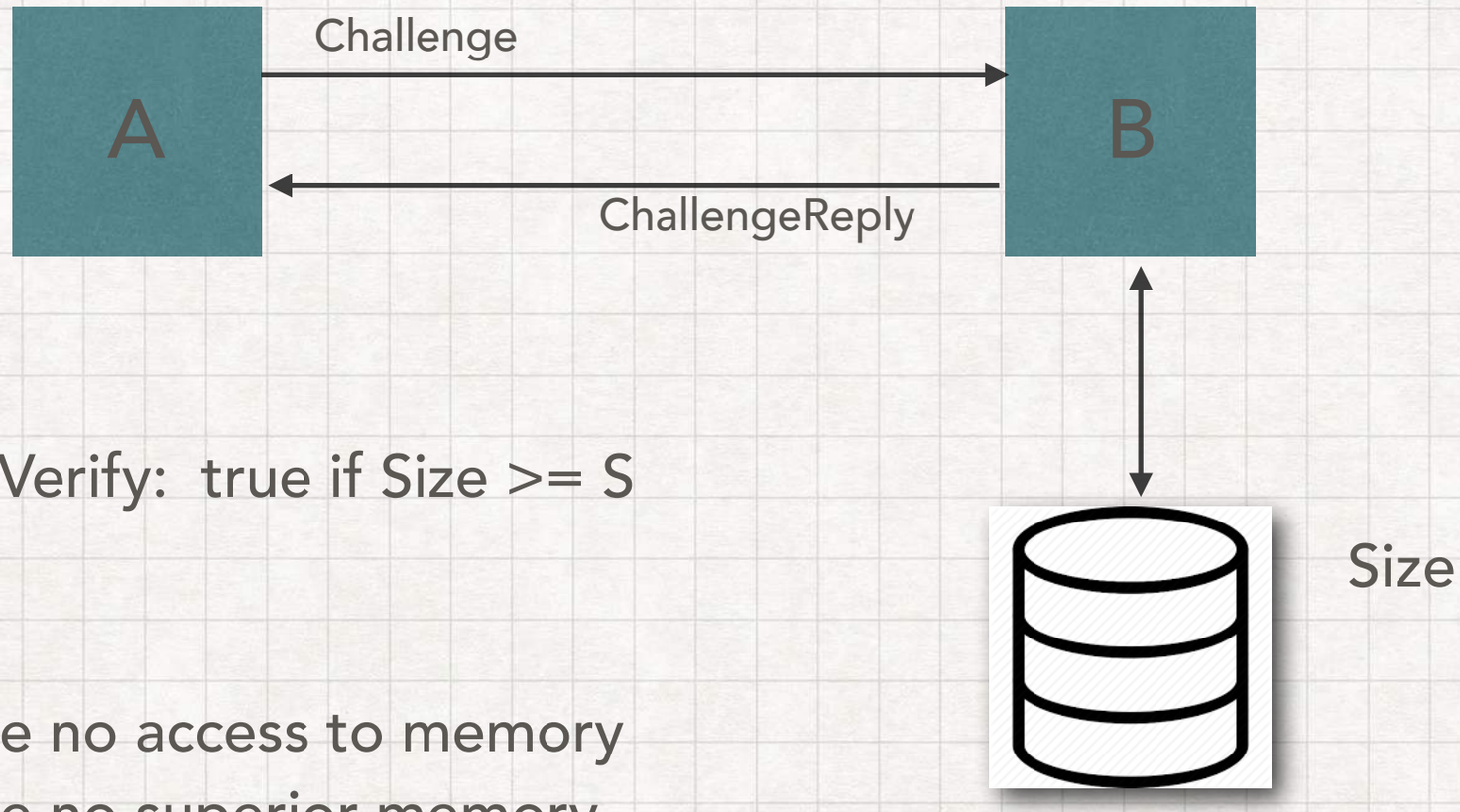
<ruslan@shevchenko.kiev.ua>
@rssh1
github/rssh

garuda.ai
http://garuda.ai

PROOF OF STORAGE

HOW TO PROVE THAT I HAVE MEMORY ?

Interactive protocol



Verify: true if Size >= S

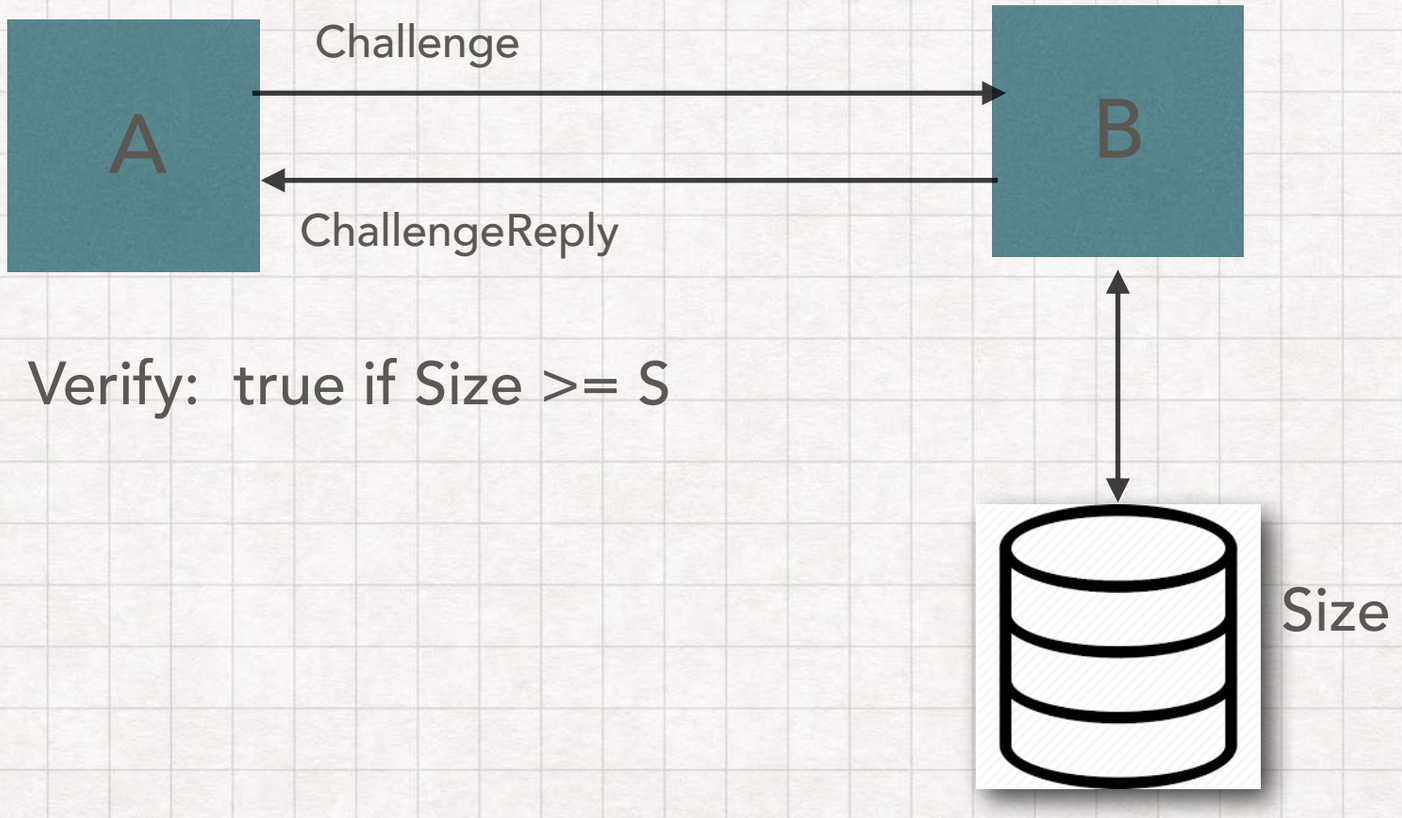
A

- have no access to memory
- have no superior memory

PROOF OF STORAGE

HOW TO PROVE THAT I HAVE MEMORY ?

? - *Ideas*



PROOF OF STORAGE

HOW TO PROVE THAT I HAVE MEMORY ?

? - *Ideas*

Strictly Incorrect, but useful:

- Inversion of one-way function.
- Meet in the middle collision attack

Base solution:

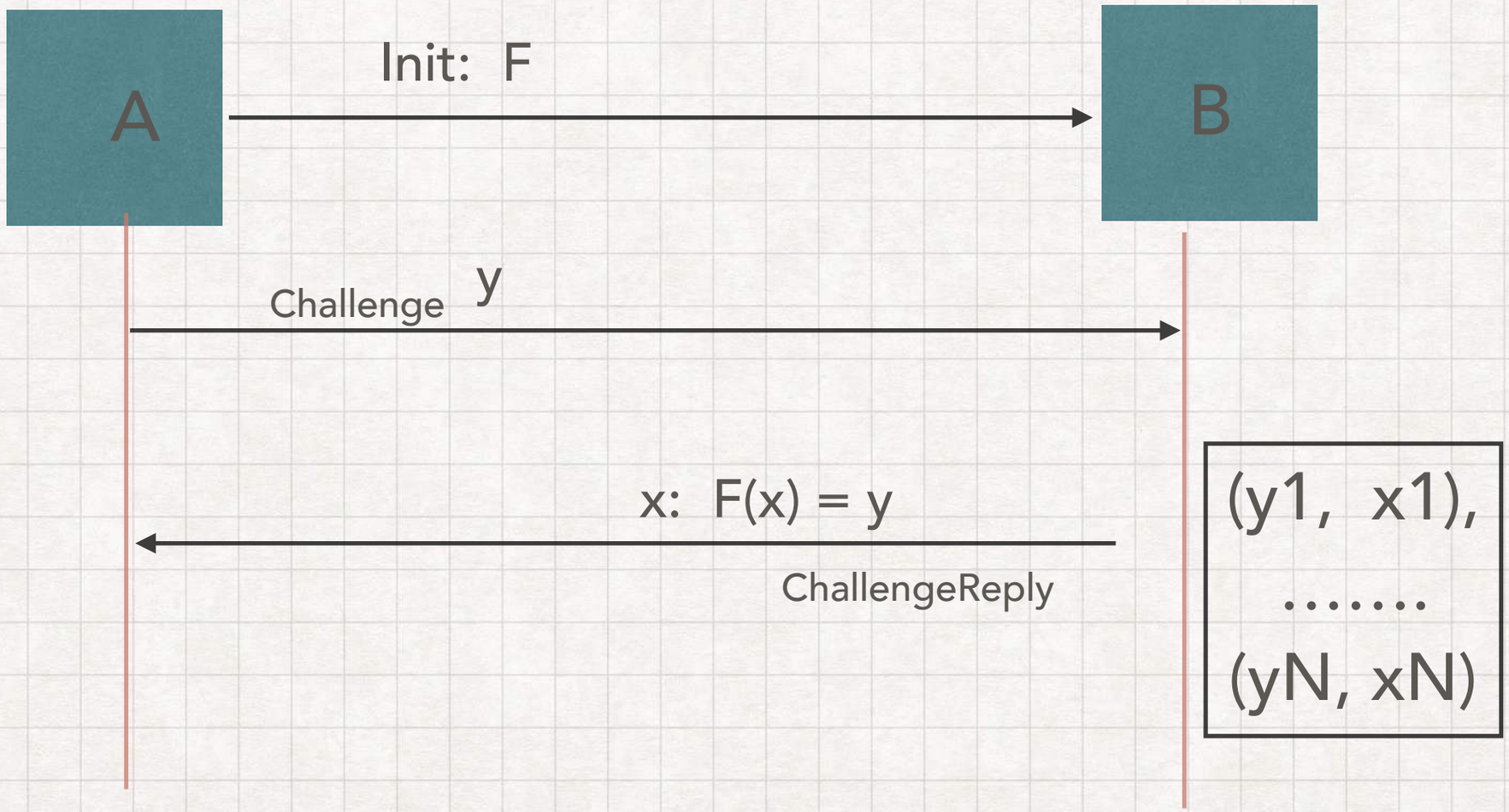
- Pebbling game.

MEMORY HARD ALGORITHM (?)

INVERSION OF ONE-WAY FUNCTION

- Init: one-way function F
- Challenge: y
- Reply: x: F(x) = y

Incorrect



MEMORY HARD ALGORITHM (?)

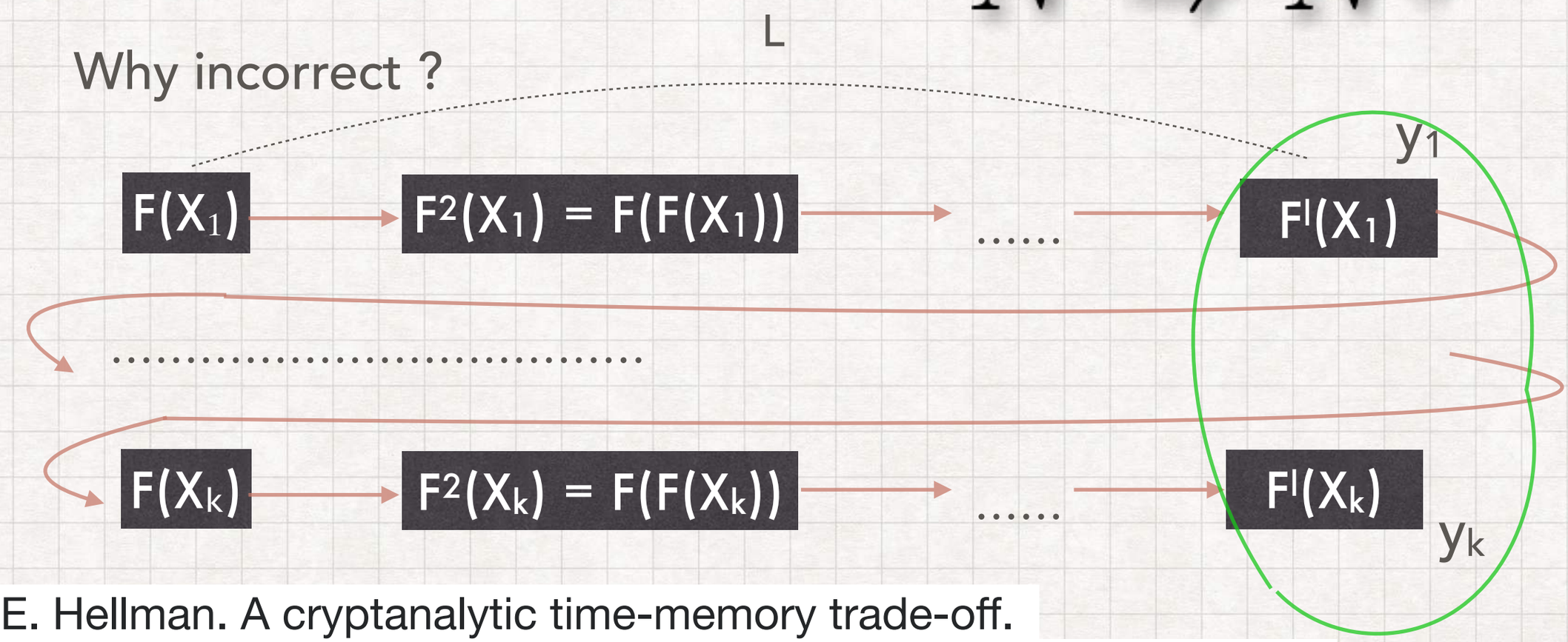
INVERSION OF ONE-WAY FUNCTION

- Init: one-way function F
- Challenge: y
- Reply: x: F(x) = y

Hellman Construction

$$N \Rightarrow N^{\frac{2}{3}}$$

Why incorrect ?



Martin E. Hellman. A cryptanalytic time-memory trade-off.
IEEE Transactions on Information Theory, 26(4):401–406, 1980

MEMORY-HARD ALGORITHMS

MEET IN THE MIDDLE ATTACK

Incorrect

$$k = (k1, k2) \qquad k \in 2^n, k_1 \in 2^{n/2}, k_2 \in 2^{n/2}$$

$$E(k, x) = E(k_1, E(k_2, x))$$

Many chippers have such structure. (DES)

$$y = E(x)$$ we know x & y for some case, let find k

$$E^{-1}(k_1, y) = E(k_2, x)$$

enumeration k1		enumeration k2	$2^{\frac{n}{2}+1}$ *Memory*
$E^{-1}(k_1, y)$	sort =	$E(k_2, x)$	$2^{\frac{n}{2}+1}$ *Time*

MEMORY-HARD ALGORITHM (?)

MEET IN THE MIDDLE ATTACK

Why incorrect ?

Pollard memoryless collision search.

Calculate E(E(….(x)…) and search collision at the same time.

$$Time \times Memory = C$$

enumeration k1

$$E^{-1}(k_1, y)$$

sort

=

enumeration k2

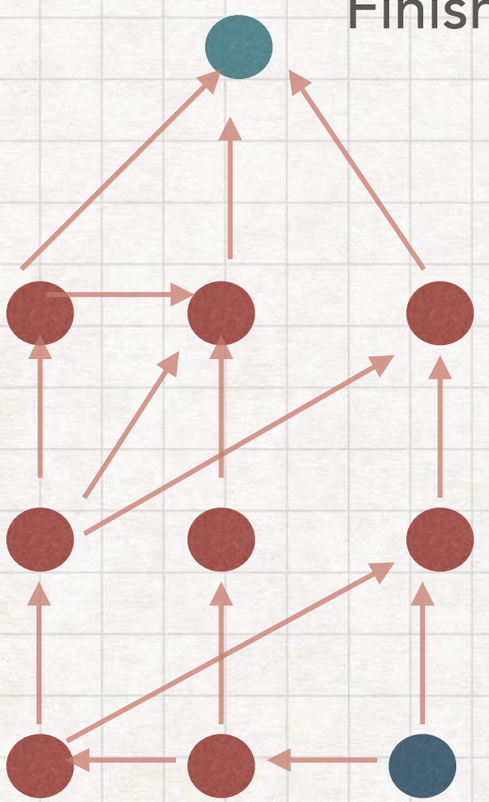
$$E(k_2, x)$$

$2^{\frac{n}{2}+1}$ $Memory$

$2^{\frac{n}{2}+1}$ $Time$

PEBBLE GAME

MEMORY HARD ALGORITHM



We can pebble vertex, if

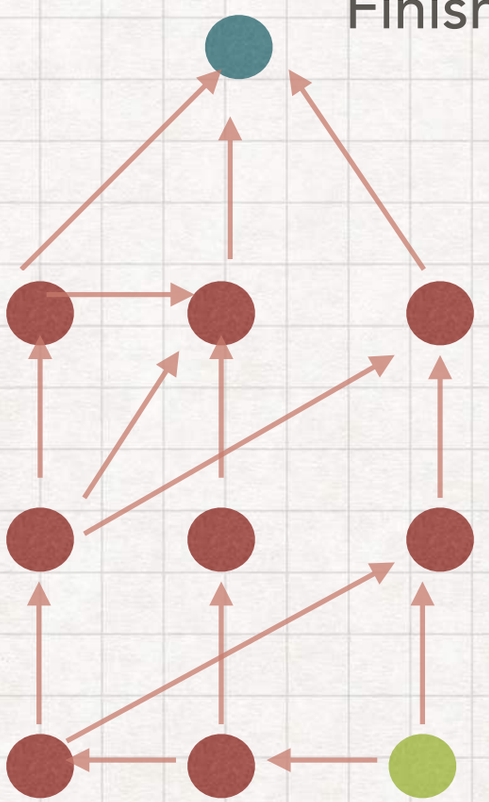
- this is start vertex
- all input vertex are pebbled.

Goal - pebble finish vertex

Use minimal number of pebbles

PEBBLE GAME

MEMORY HARD ALGORITHM



1,1

We can pebble vertex, if

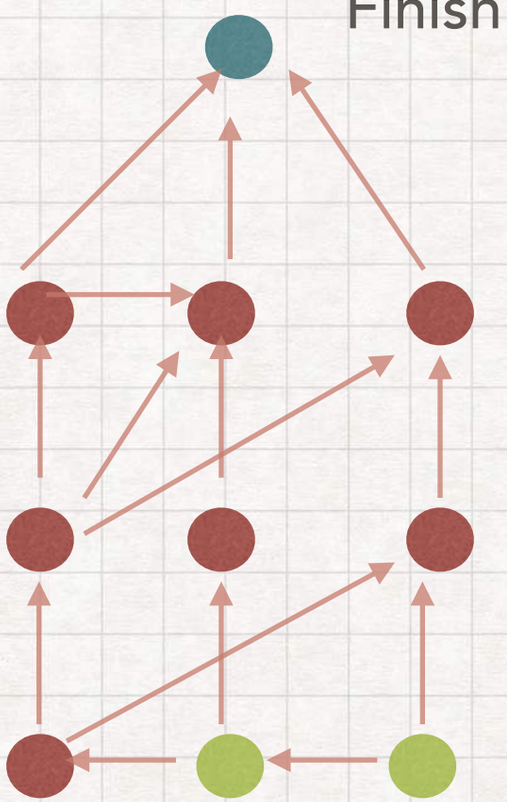
- this is start vertex
- all input vertex are pebbled.

Goal - pebble finish vertex

Use minimal number of pebbles

PEBBLE GAME

MEMORY HARD ALGORITHM



2,2

We can pebble vertex, if

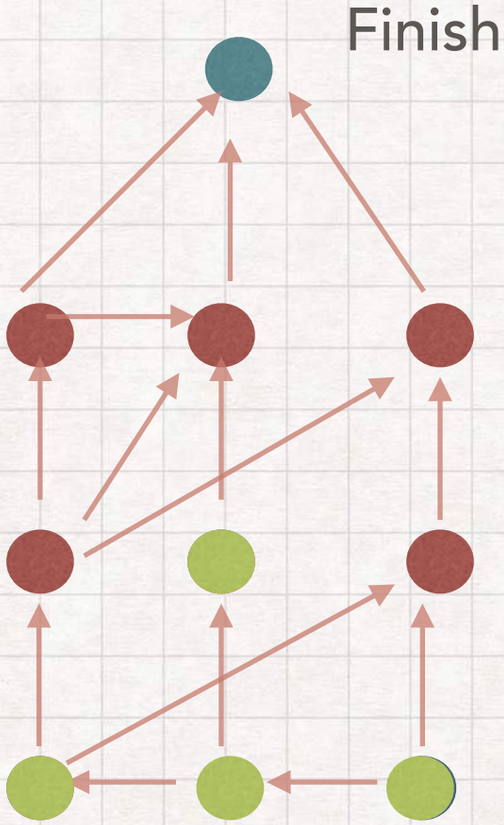
- this is start vertex
- all input vertex are pebbled.

Goal - pebble finish vertex

Use minimal number of pebbles

PEBBLE GAME

MEMORY HARD ALGORITHM



We can pebble vertex, if

- this is start vertex
- all input vertex are pebbled.

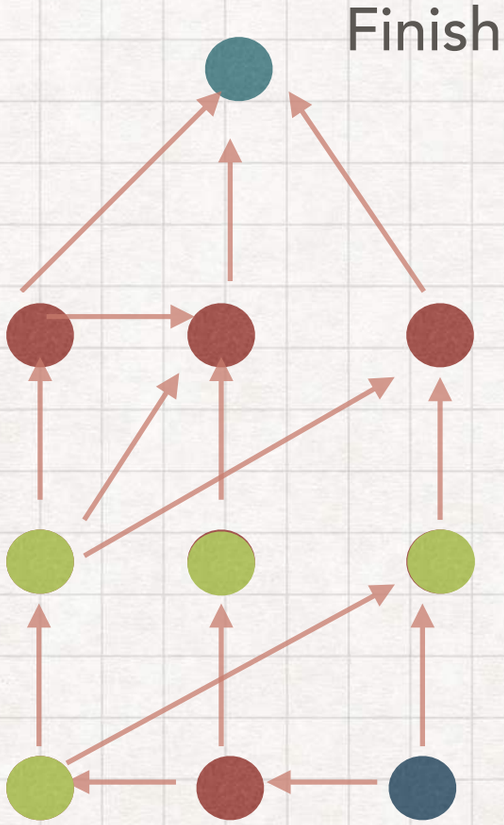
Goal - pebble finish vertex

Use minimal number of pebbles

3,4

PEBBLE GAME

MEMORY HARD ALGORITHM



4,4

We can pebble vertex, if

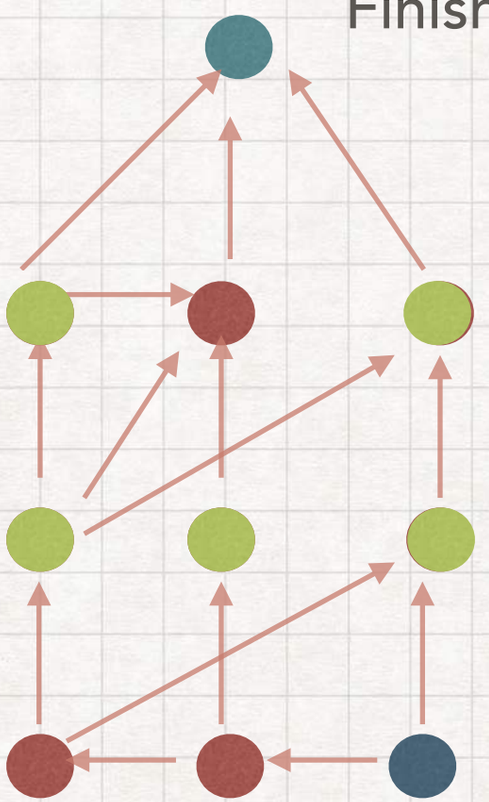
- this is start vertex
- all input vertex are pebbled.

Goal - pebble finish vertex

Use minimal number of pebbles

PEBBLE GAME

MEMORY HARD ALGORITHM



5,5

We can pebble vertex, if

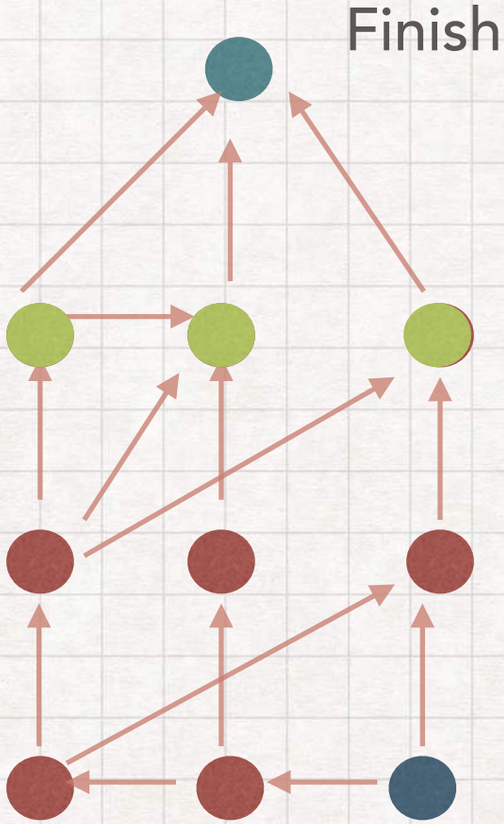
- this is start vertex
- all input vertex are pebbled.

Goal - pebble finish vertex

Use minimal number of pebbles

PEBBLE GAME

MEMORY HARD ALGORITHM



6,3,5

We can pebble vertex, if

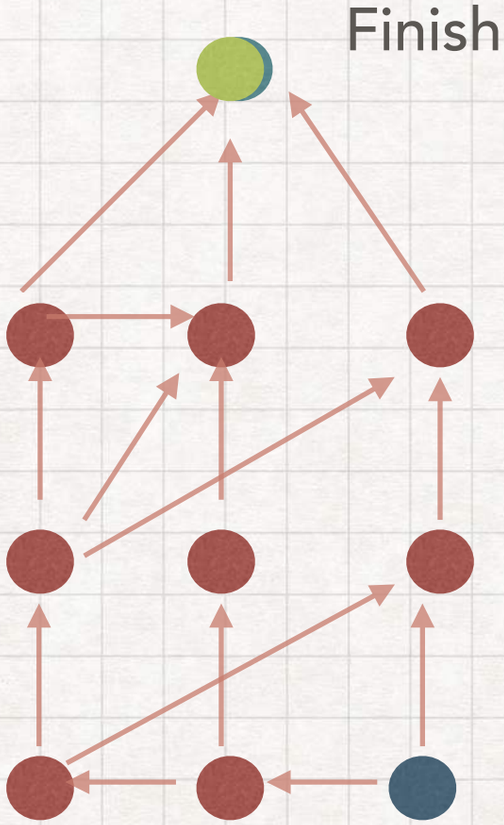
- this is start vertex
- all input vertex are pebbled.

Goal - pebble finish vertex

Use minimal number of pebbles

PEBBLE GAME

MEMORY HARD ALGORITHM



Finish = F(x[3,1],x[3,2],x[3,3])

x[3,1] = F(x[2,1]); x[3,2]=F(x[3,1],x[2,1],x[2,2])
x[3,3] = F(x[2,1],x[2,3])

x[2,1] = F(x[1,1]); x[2,2]=F(x[1,2])
x[2,3] = F(x[1,1],x[1,3])

x[1,1] = F(x[1,2]); x[1,2]=F(x[1,1])
x[1,1] = Start

7,1,5

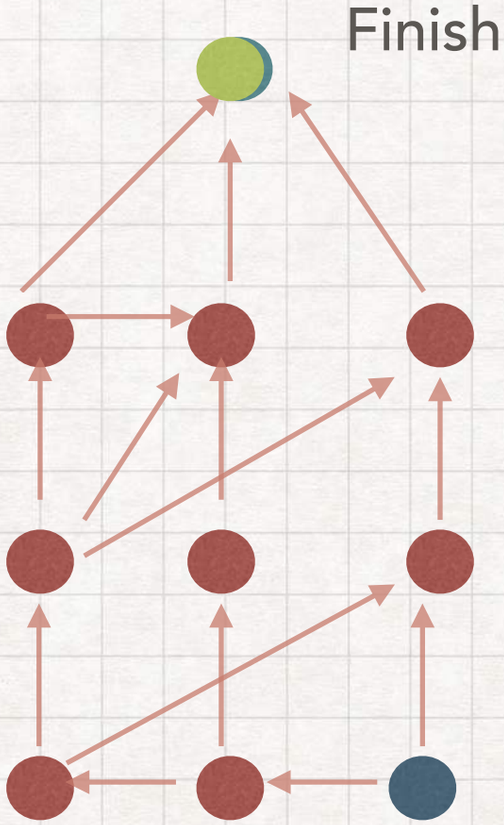
Game = Computation Schema

Moves = Time

Pebbles = Memory

PEBBLE GAME

MEMORY HARD ALGORITHM



Finish = F(x[3,1],x[3,2],x[3,3])

x[3,1] = F(x[2,1]); x[3,2]=F(x[3,1],x[2,1],x[2,2])
x[3,3] = F(x[2,1],x[2,3])

x[2,1] = F(x[1,1]); x[2,2]=F(x[1,2])
x[2,3] = F(x[1,1],x[1,3])

x[1,1] = F(x[1,2]); x[1,2]=F(x[1,1])
x[1,1] = Start

7,1,5

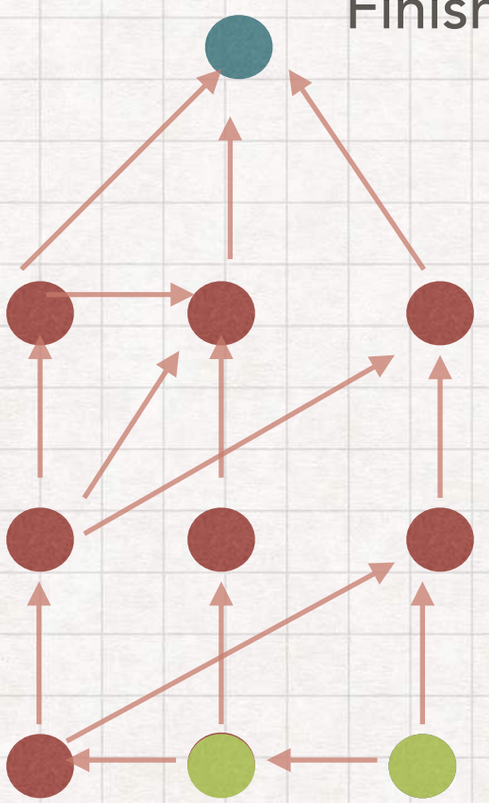
Game = Computation Schema

Moves = Time

Pebbles = Memory

PEBBLE GAME

MEMORY HARD ALGORITHM



Let’s play with 3 pebbles

Game = Computation Schema

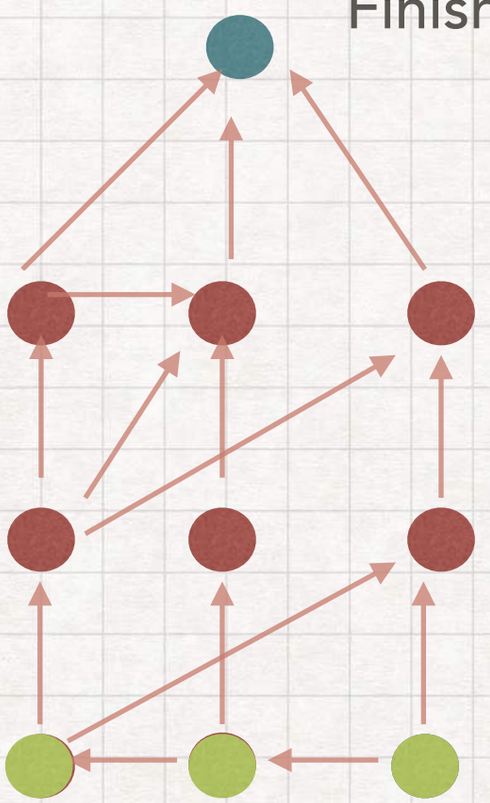
Moves = Time

Pebbles = Memory

2,2

PEBBLE GAME

MEMORY HARD ALGORITHM



3,3

Let’s play with 3 pebbles

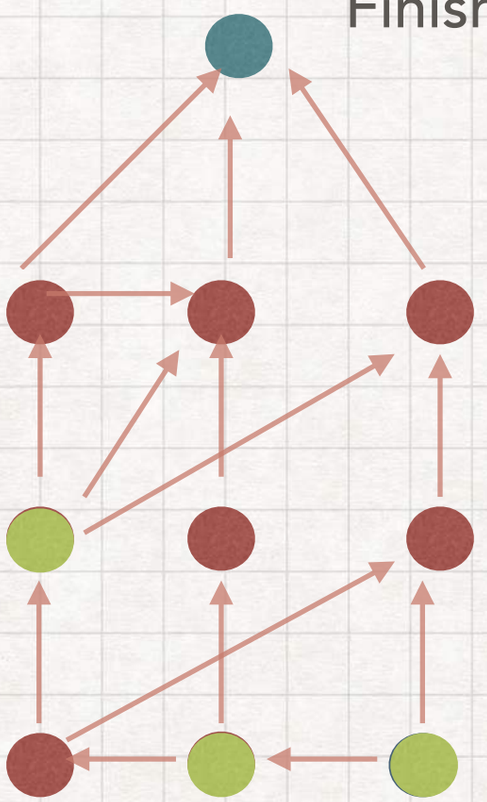
Game = Computation Schema

Moves = Time

Pebbles = Memory

PEBBLE GAME

MEMORY HARD ALGORITHM



Let’s play with 3 pebbles

Game = Computation Schema

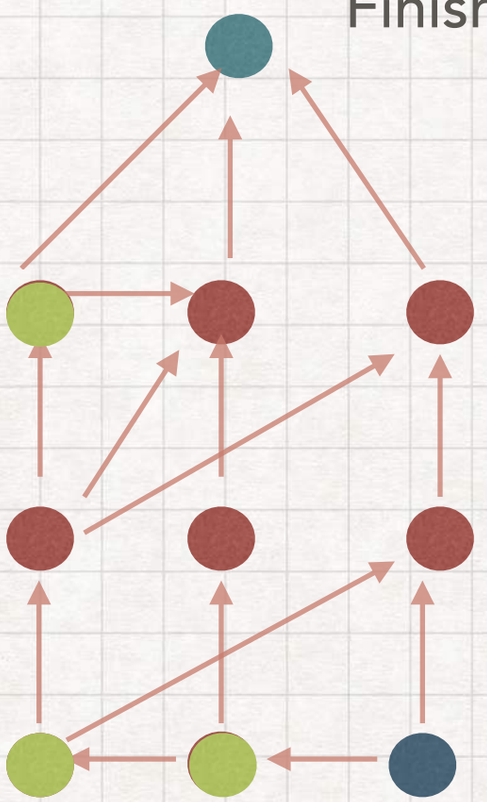
Moves = Time

Pebbles = Memory

4,3

PEBBLE GAME

MEMORY HARD ALGORITHM



Let’s play with 3 pebbles

Game = Computation Schema

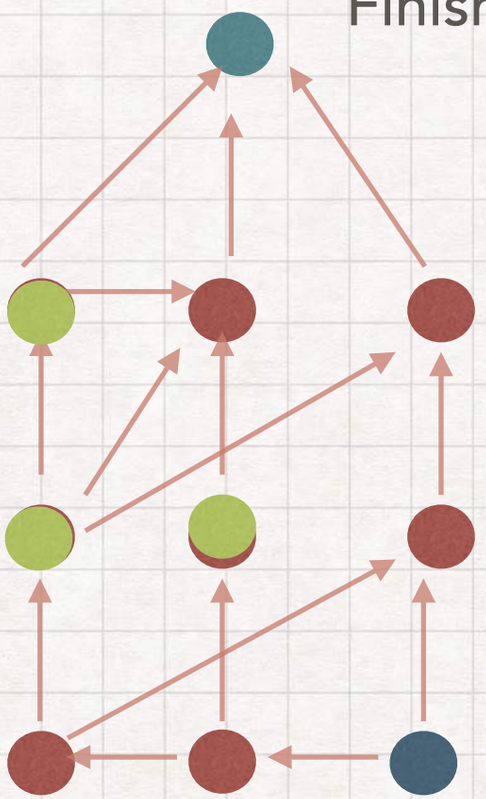
Moves = Time

Pebbles = Memory

5,3

PEBBLE GAME

MEMORY HARD ALGORITHM



6,3

Let’s play with 3 pebbles

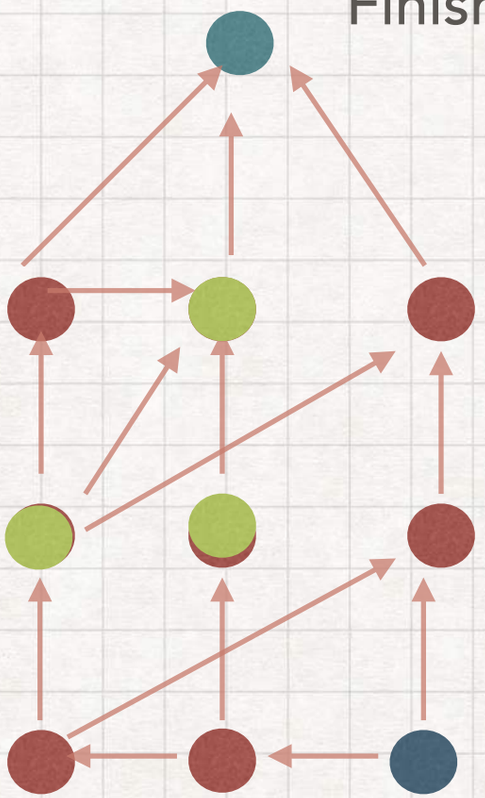
Game = Computation Schema

Moves = Time

Pebbles = Memory

PEBBLE GAME

MEMORY HARD ALGORITHM



Let’s play with 3 pebbles

Game = Computation Schema

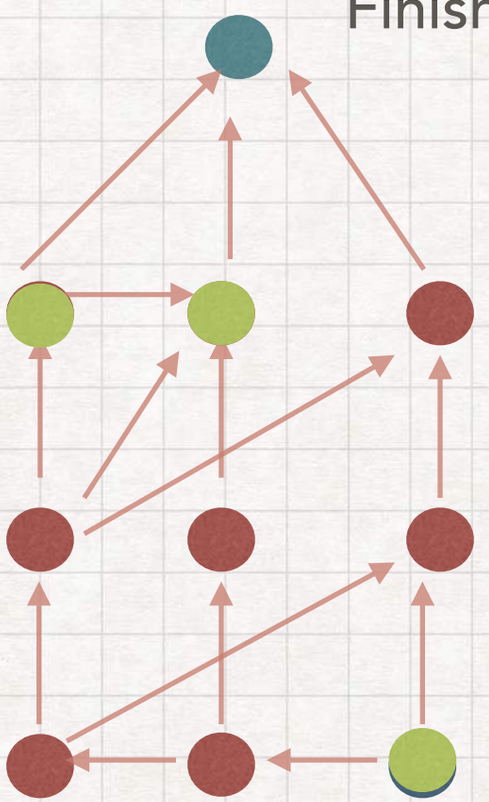
Moves = Time

Pebbles = Memory

7,3

PEBBLE GAME

MEMORY HARD ALGORITHM



Let's play with 3 pebbles

Game = Computation Schema

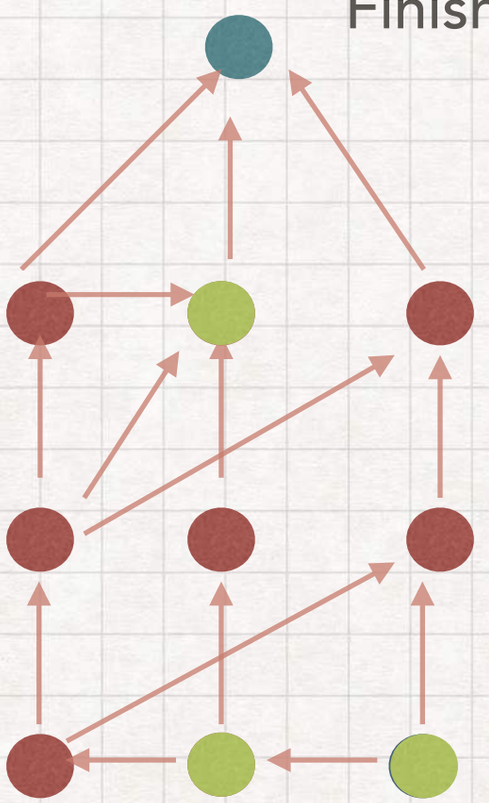
Moves = Time

Pebbles = Memory

8,3

PEBBLE GAME

MEMORY HARD ALGORITHM



Let’s play with 3 pebbles

Game = Computation Schema

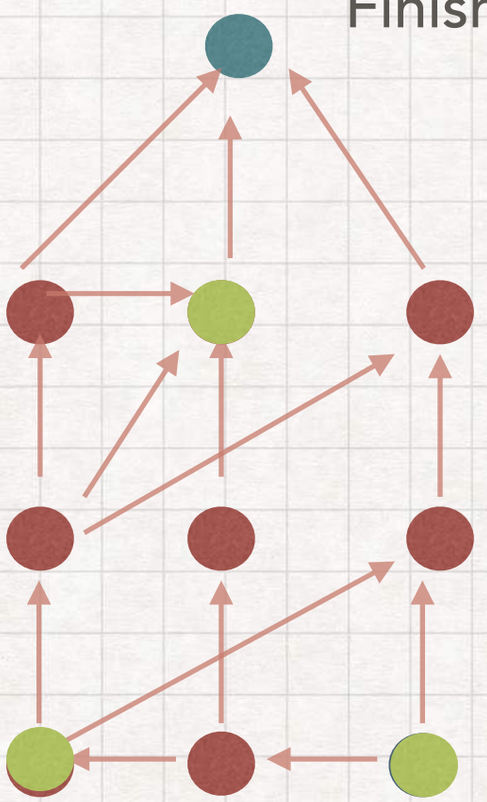
Moves = Time

Pebbles = Memory

9,3

PEBBLE GAME

MEMORY HARD ALGORITHM



Let's play with 3 pebbles

Game = Computation Schema

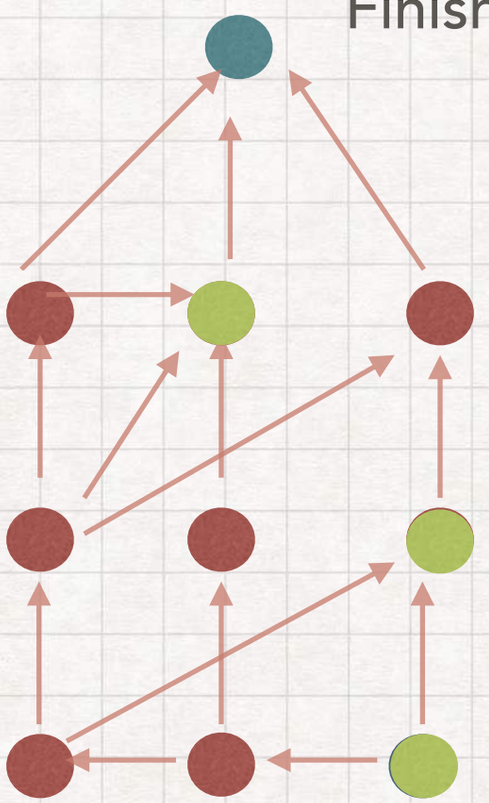
Moves = Time

Pebbles = Memory

10,3

PEBBLE GAME

MEMORY HARD ALGORITHM



11,3

Let’s play with 3 pebbles

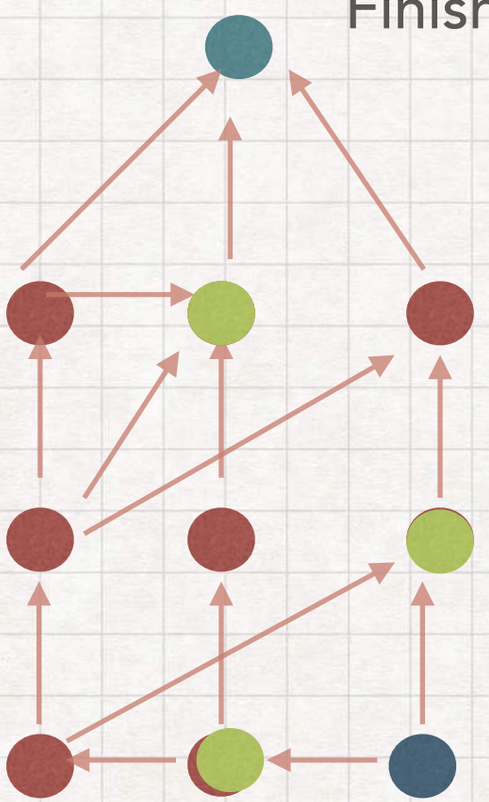
Game = Computation Schema

Moves = Time

Pebbles = Memory

PEBBLE GAME

MEMORY HARD ALGORITHM



Let’s play with 3 pebbles

Game = Computation Schema

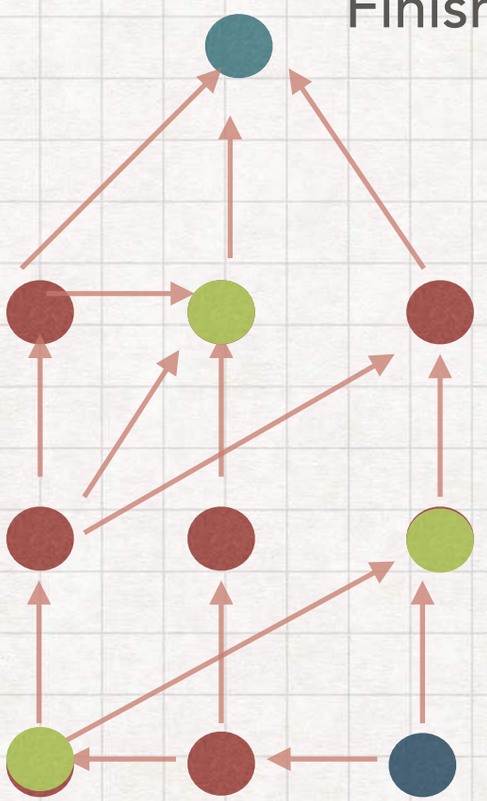
Moves = Time

Pebbles = Memory

12,3

PEBBLE GAME

MEMORY HARD ALGORITHM



Let’s play with 3 pebbles

Game = Computation Schema

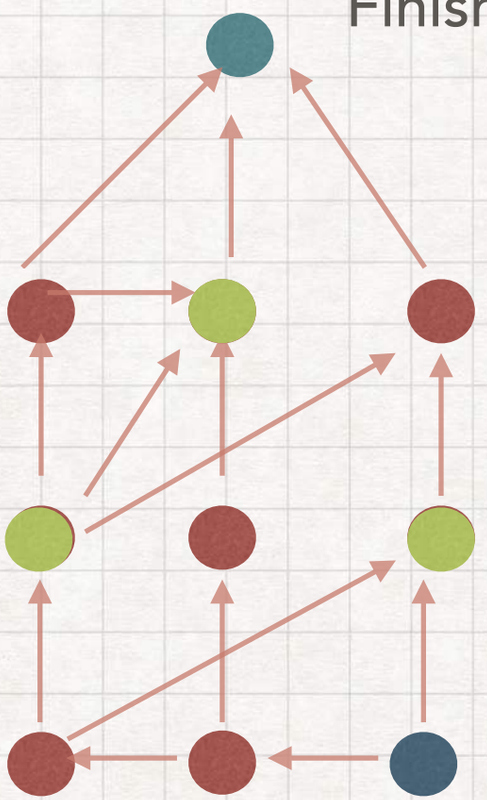
Moves = Time

Pebbles = Memory

12,3

PEBBLE GAME

MEMORY HARD ALGORITHM



13,3

Let’s play with 3 pebbles

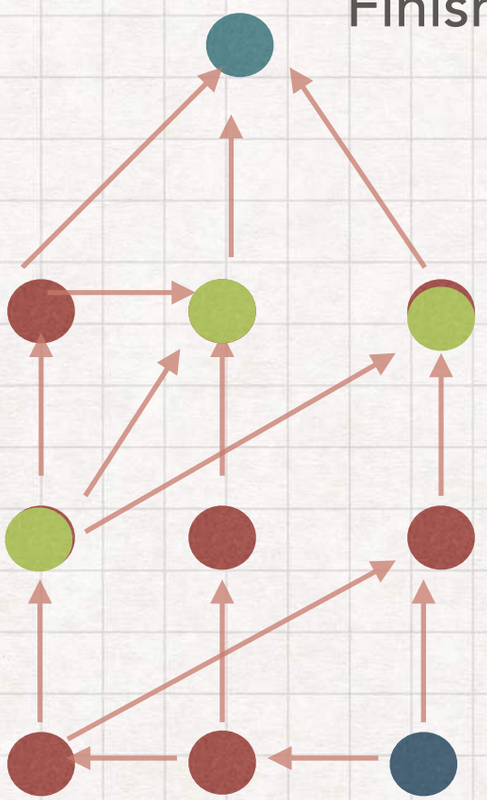
Game = Computation Schema

Moves = Time

Pebbles = Memory

PEBBLE GAME

MEMORY HARD ALGORITHM



14,3

Let’s play with 3 pebbles

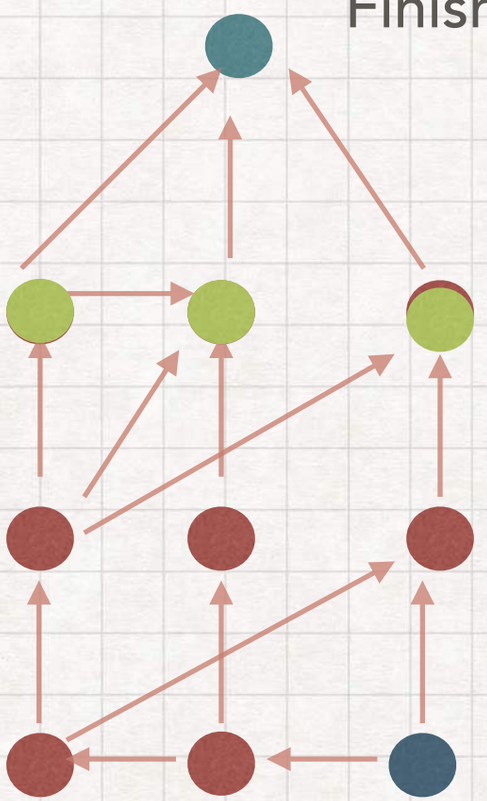
Game = Computation Schema

Moves = Time

Pebbles = Memory

PEBBLE GAME

MEMORY HARD ALGORITHM



15,3

Let’s play with 3 pebbles

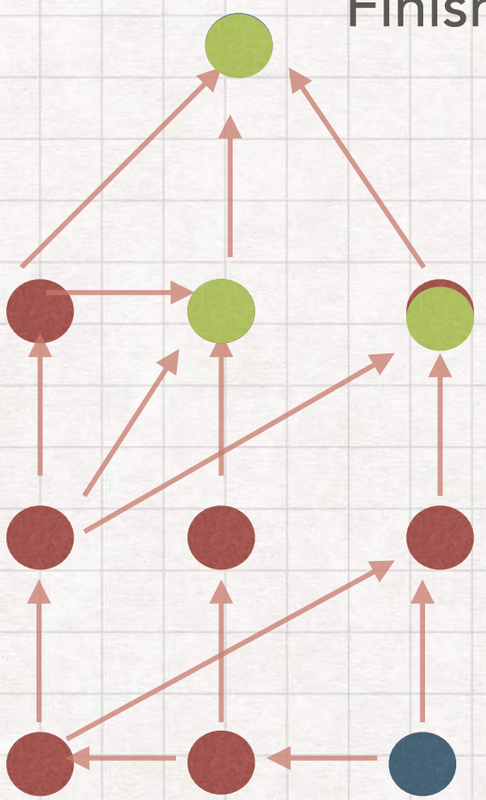
Game = Computation Schema

Moves = Time

Pebbles = Memory

PEBBLE GAME

MEMORY HARD ALGORITHM



16,3

Let’s play with 3 pebbles

Game = Computation Schema

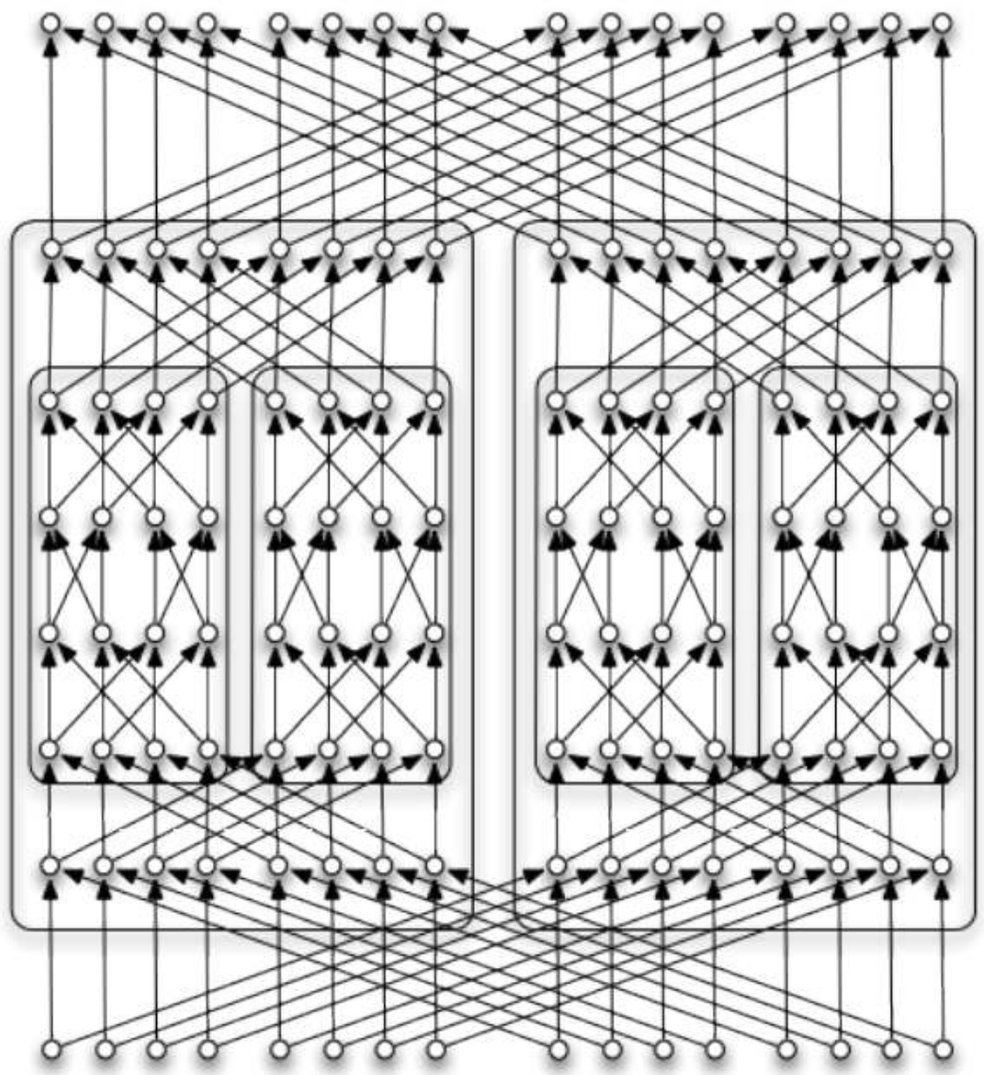
Moves = Time

Pebbles = Memory

PROOF OF STORAGE

HARD TO PEBBLE GRAPH

Butterfly graph

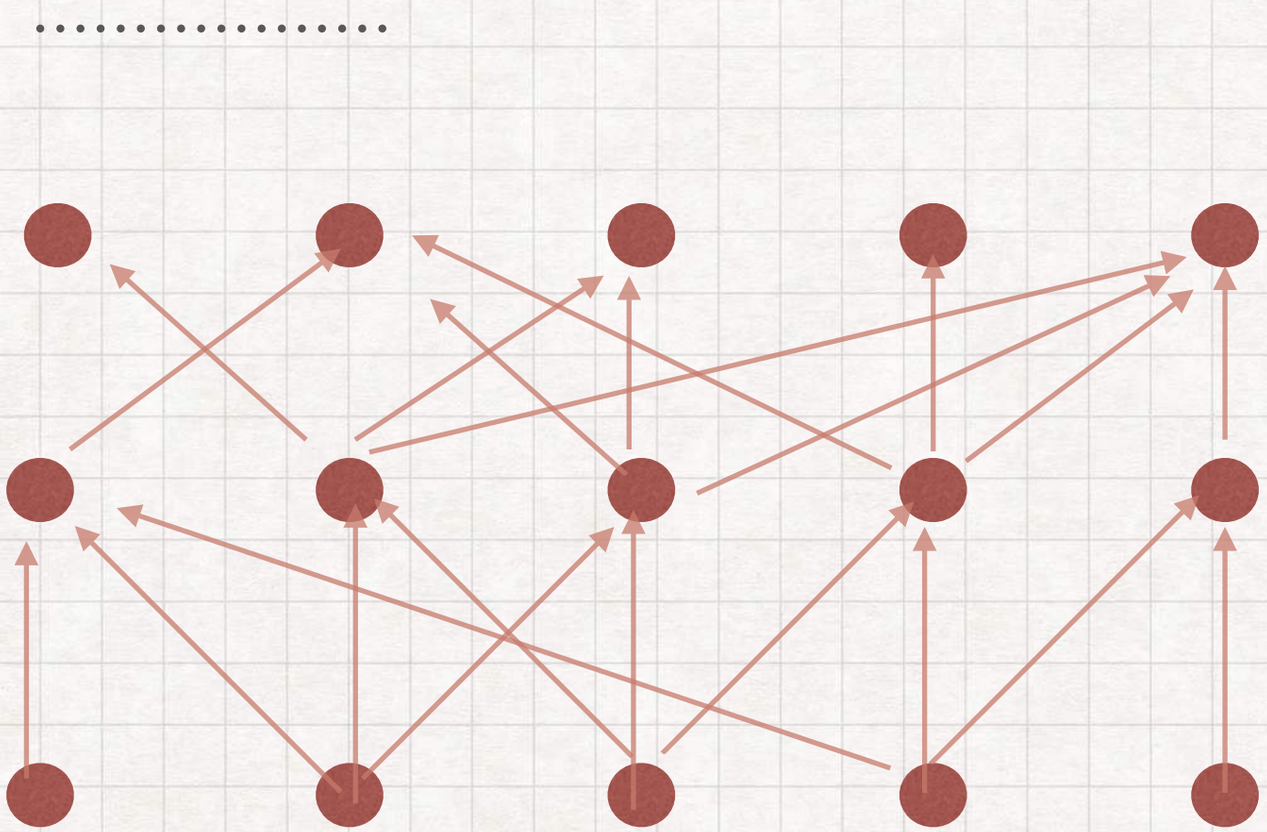


Idea:
— maximize time penalty
for missing pebble.

$$\Delta T \approx e^{\Delta M}$$

PROOF OF STORAGE

PROBABILISTIC HARD TO PEBBLE GRAPH



from each node in Layer
D egdes to the next
layer

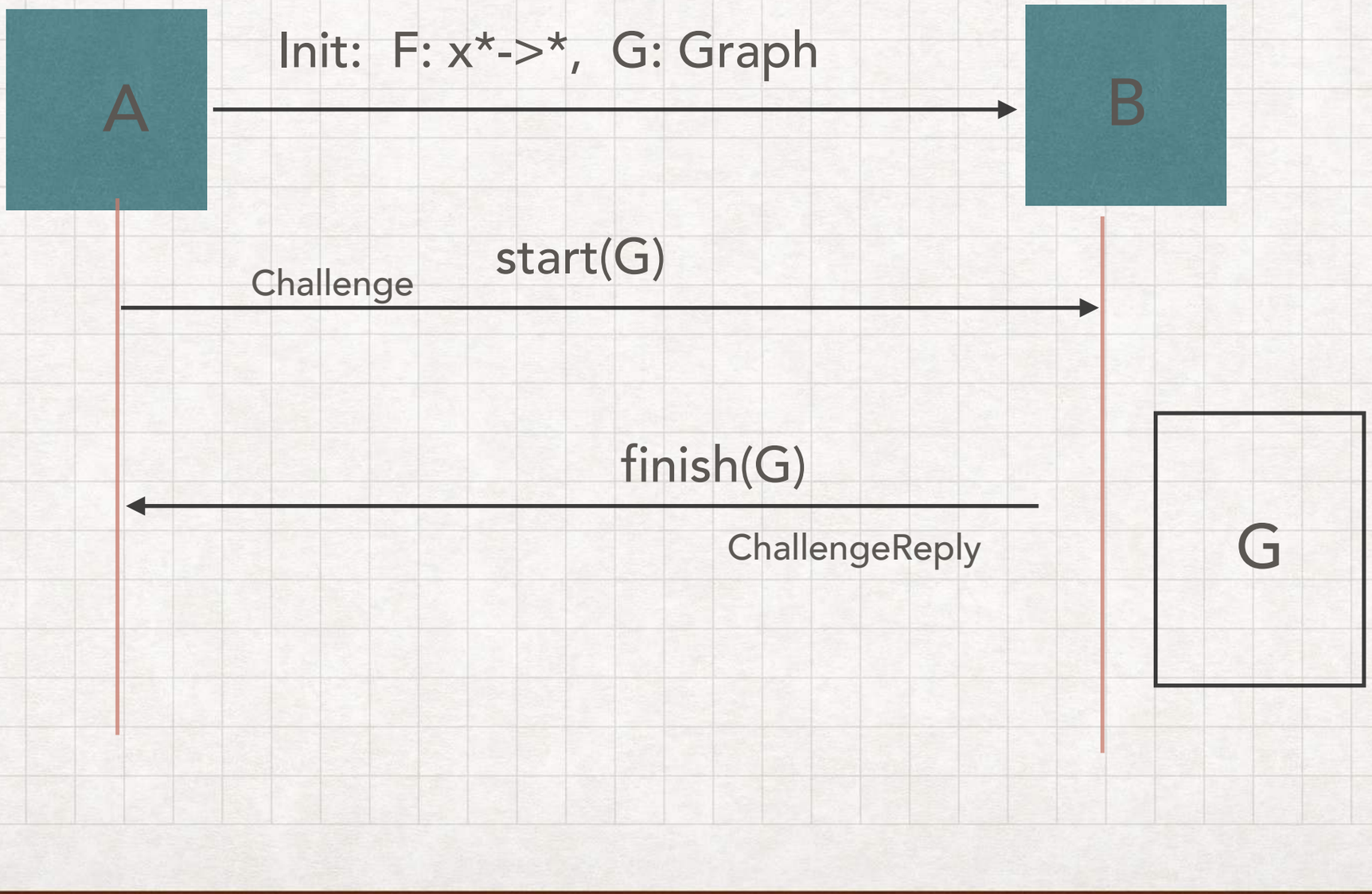
Randomly generated Chung expanders are hard to pebble

Pinsker (D predescors for each sink)

MEMORY HARD ALGORITHM (?)

PEBBLING 'HARD TO PEBBLE' GRAPH

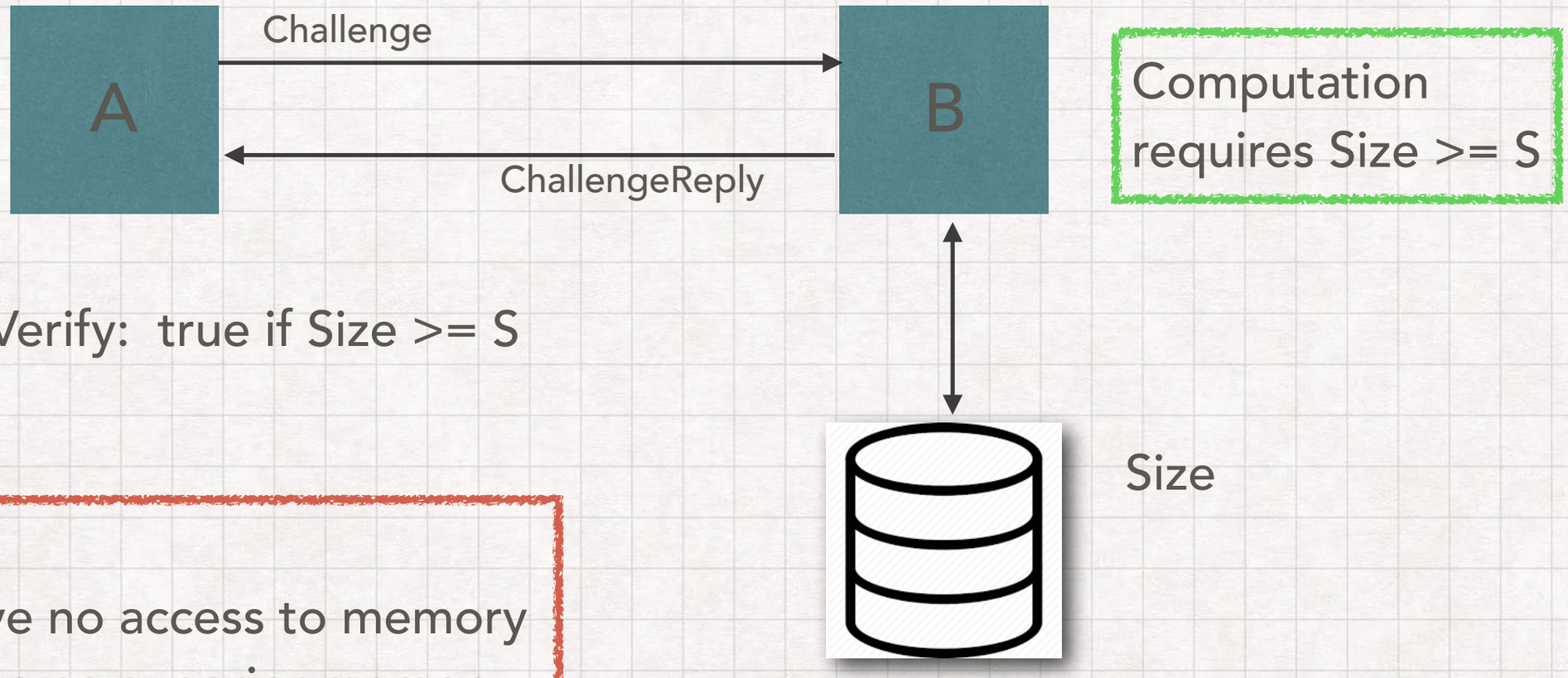
- Init: random function H and hard to pebble graph G
- Challenge: values of pebbles at start-vertex
- Reply: value of pebbles at finish versed



PROOF OF STORAGE

HOW TO PROVE THAT I HAVE MEMORY ?

Interactive protocol



Verify: true if Size >= S

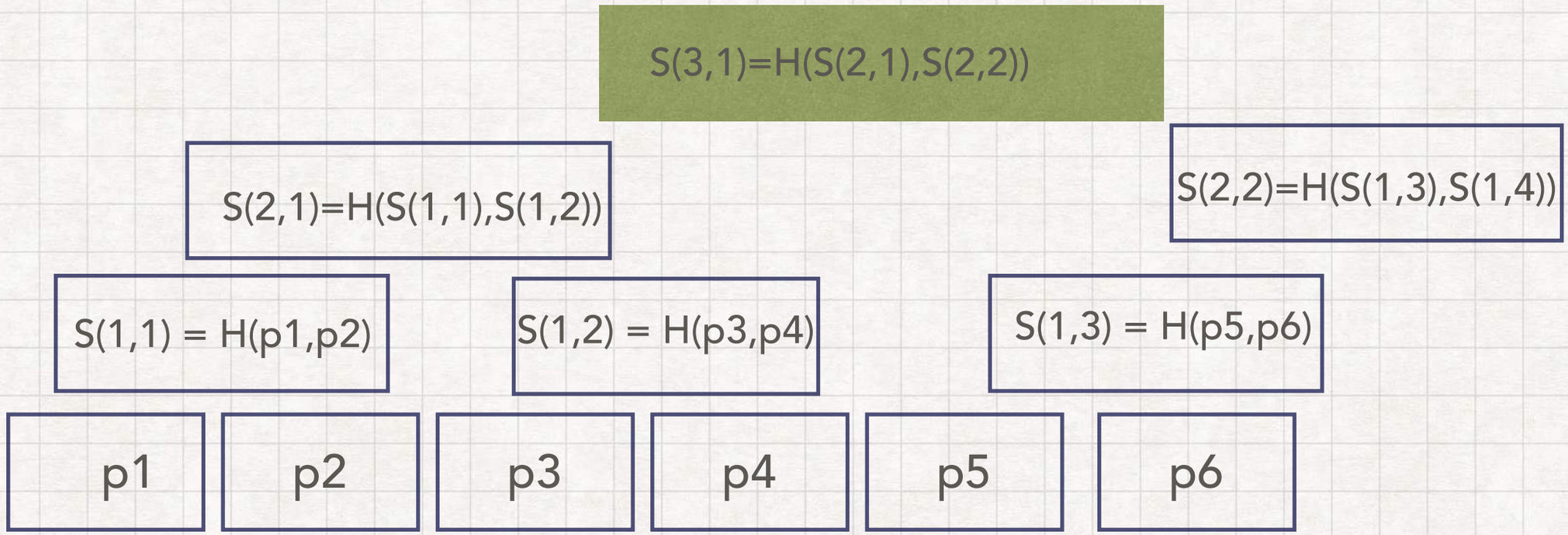
A
- have no access to memory
- have no superior memory

HOW TO PROVE COMPUTATION

AUTHENTICATED DATA STRUCTURES

1. Authenticate log.
2. Open part of this log for inspection.

Merkle Tree



If you will change something in log, root hash of merkle tree will be other

PROVE THAT I RUN GIVEN COMPUTATION SCHEME

1. A => B (send init values.)
2. B run scheme and calculate computation log Merkle tree root hash.
3. B => A (Merkle tree root hash)
4. A => B (Some parts of schema, i.e. coordinate in graph)
5. B rerun computation, recalculate Merkle tree
6. B => A
    1. : inputs and outputs of computation at specific vertexes.
    2. : hashes of this computation.
    3. : hashes, which allows A to recompute the Merkle tree root.
7. A :
    1. check that input and output at given vertexes are correct
    2. check, that Merkle tree root will be the same.

    // It will be hard for A to build Merkle Tree for other computations with the same hash.

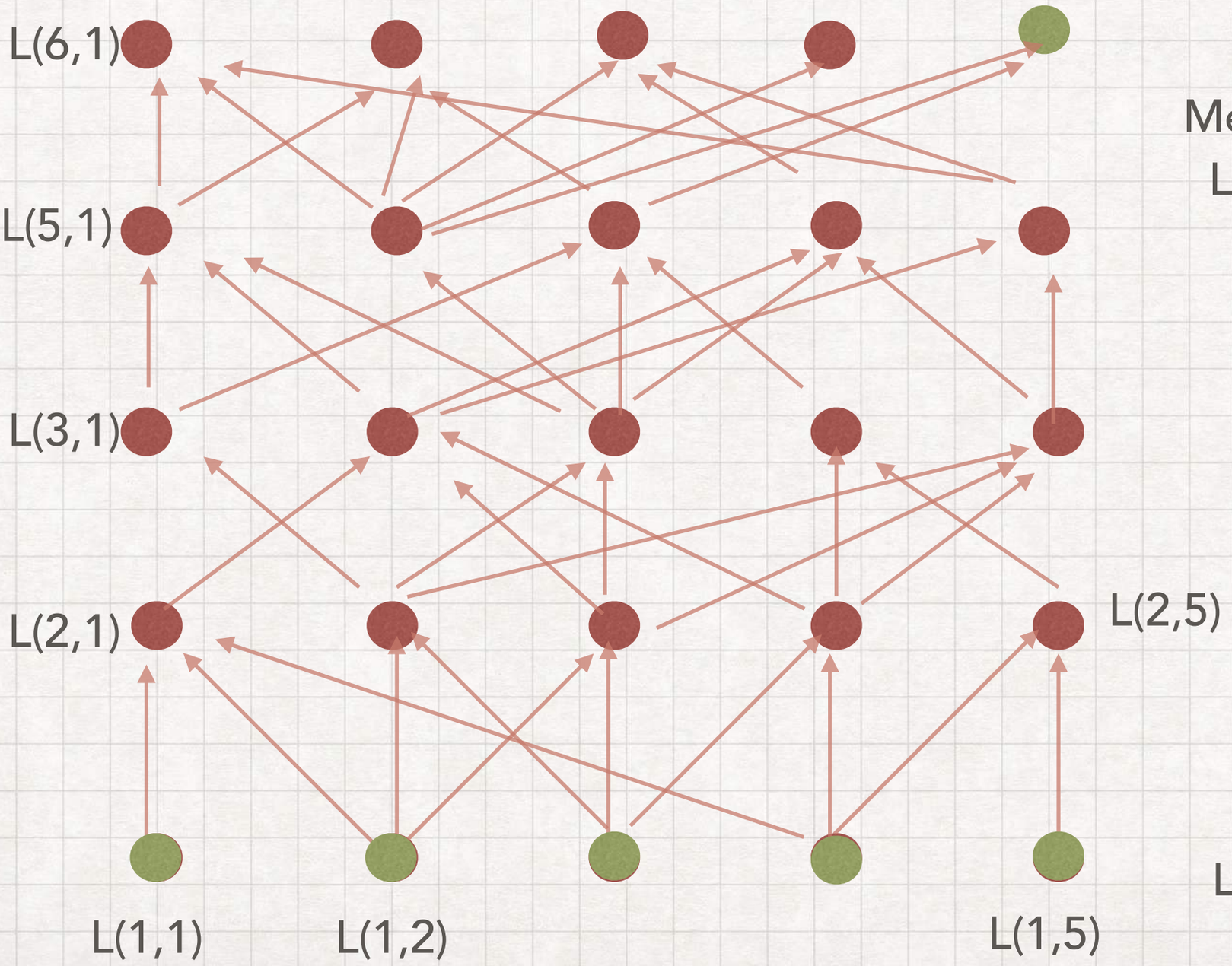
PROVE THAT I RUN GIVEN COMPUTATION SCHEME

1. A => B (send init values.)
2. B run scheme and calculate computation log Merkle tree root hash.
3. B => A (Merkle tree root hash)
4. A => B (Some parts of schema, i.e. coordinate in graph)
5. B rerun computation, recalculate Merkle tree
6. B => A
    1. : inputs and outputs of computation at specific vertexes.
    2. : hashes of this computation.
    3. : hashes, which allows A to recompute the Merkle tree root.
7. A :
    1. check that input and output at given vertexes are correct
    2. check, that Merkle tree root will be the same.

    // It will be hard for A to build Merkle Tree for other computations with the same hash.

PROOF OF COMPUTATION OF GIVEN SCHEME

RN



MerkleTree:
L(1,1) …. L(6,5)

L1 - initial values

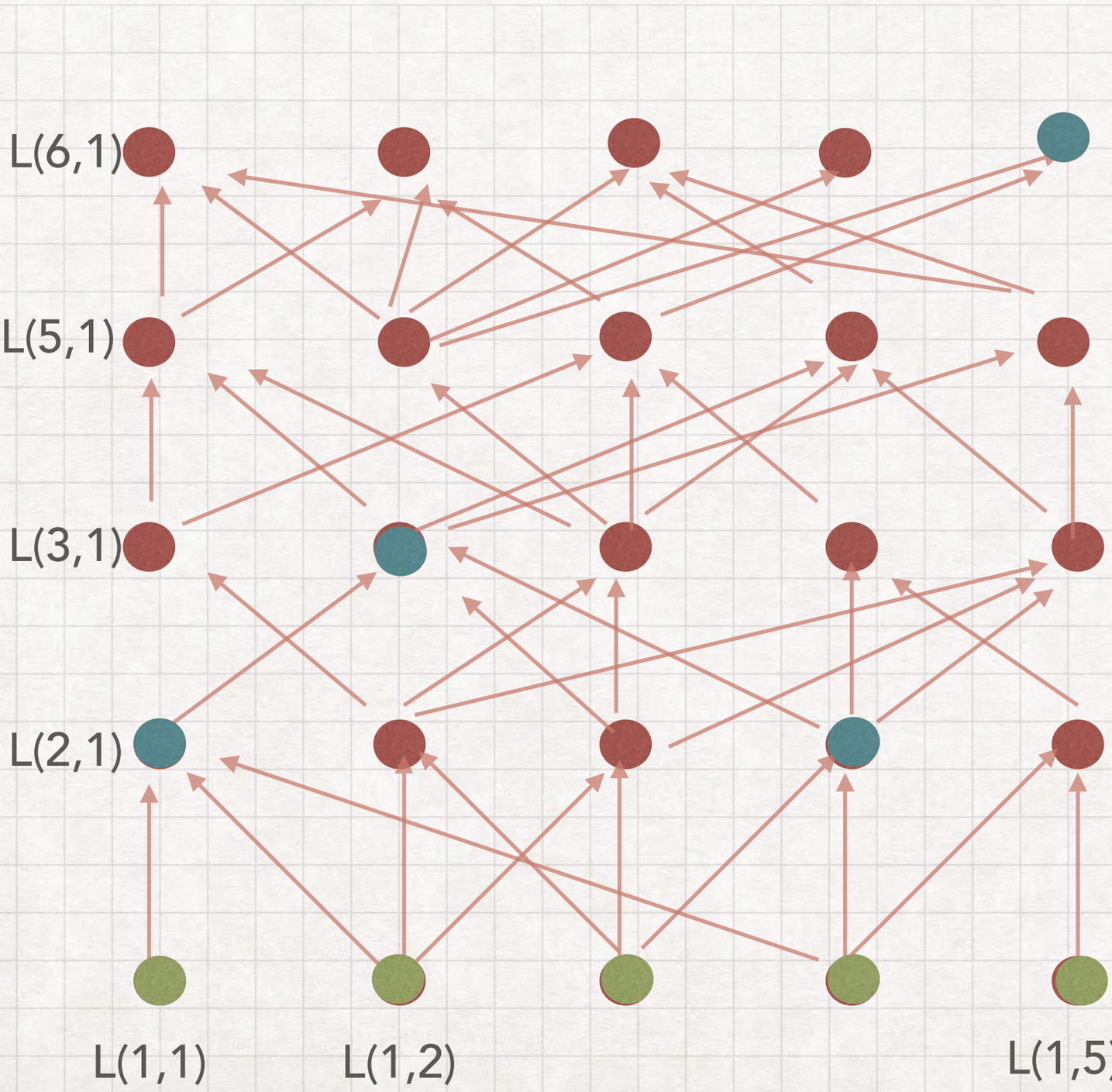
PROVE THAT I RUN GIVEN COMPUTATION SCHEME

1. A => B (send init values.)
2. B run scheme and calculate computation log Merkle tree root hash.
3. B => A (Merkle tree root hash)
4. A => B (Some parts of schema, i.e. coordinate in graph)
5. B rerun computation, recalculate Merkle tree
6. B => A
    1. : inputs and outputs of computation at specific vertexes.
    2. : hashes of this computation.
    3. : hashes, which allows A to recompute the Merkle tree root.
7. A :
    1. check that input and output at given vertexes are correct
    2. check, that Merkle tree root will be the same.

    // It will be hard for A to build Merkle Tree for other computations with the same hash.

PROOF OF COMPUTATION OF GIVEN SCHEME

RN



MerkleTree:

L(1,1) .... L(6,5)

L1 - initial values

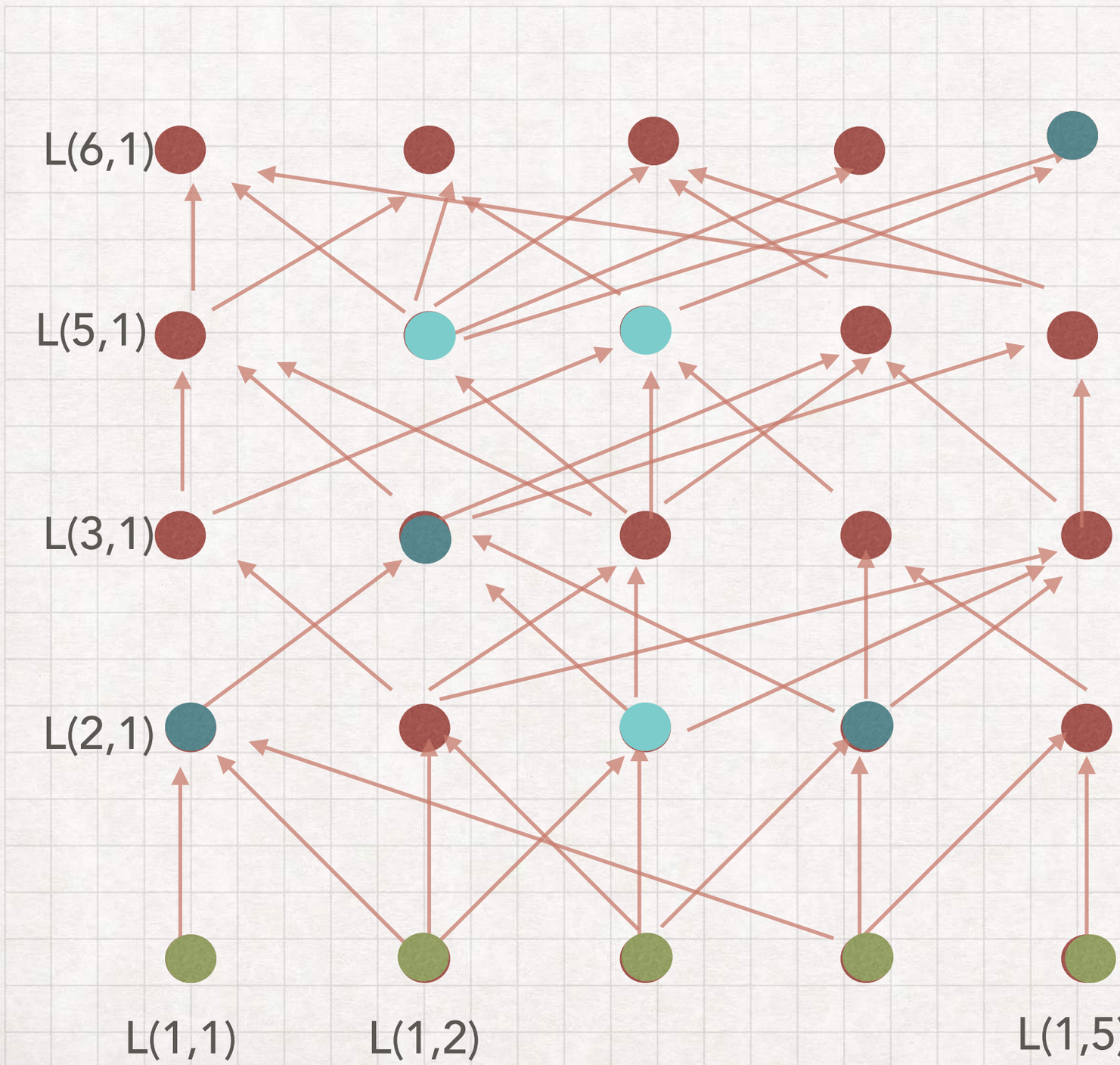
PROVE THAT I RUN GIVEN COMPUTATION SCHEME

1. A => B (send init values.)
2. B run scheme and calculate computation log Merkle tree root hash.
3. B => A (Merkle tree root hash)
4. A => B (Some parts of schema, i.e. coordinate in graph)
5. B rerun computation, recalculate Merkle tree
6. B => A
    1. : inputs and outputs of computation at specific vertexes.
    2. : hashes of this computation.
    3. : hashes, which allows A to recompute the Merkle tree root.
7. A :
    1. check that input and output at given vertexes are correct
    2. check, that Merkle tree root will be the same.

    // It will be hard for A to build Merkle Tree for other computations with the same hash.

PROOF OF COMPUTATION OF GIVEN SCHEME

RN



MerkleTree:
Path(root - L(2,1),
root - L(3,1)..)

A ask to open

Open depend from

L1 - initial values

PROVE THAT I RUN GIVEN COMPUTATION SCHEME

1. A => B (send init values.)
2. B run scheme and calculate computation log Merkle tree root hash.
3. B => A (Merkle tree root hash)
4. A => B (Some parts of schema, i.e. coordinate in graph)
5. B rerun computation, recalculate Merkle tree
6. B => A
    1. : inputs and outputs of computation at specific vertexes.
    2. : hashes of this computation.
    3. : hashes, which allows A to recompute the Merkle tree root.
7. A :
    1. check that input and output at given vertexes are correct
    2. check, that Merkle tree root will be the same.

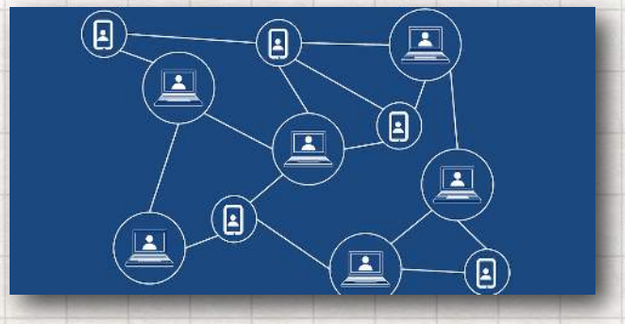
    // Profit.

PROOF OF STORAGE

PROFIT

ACIC resistant hashes

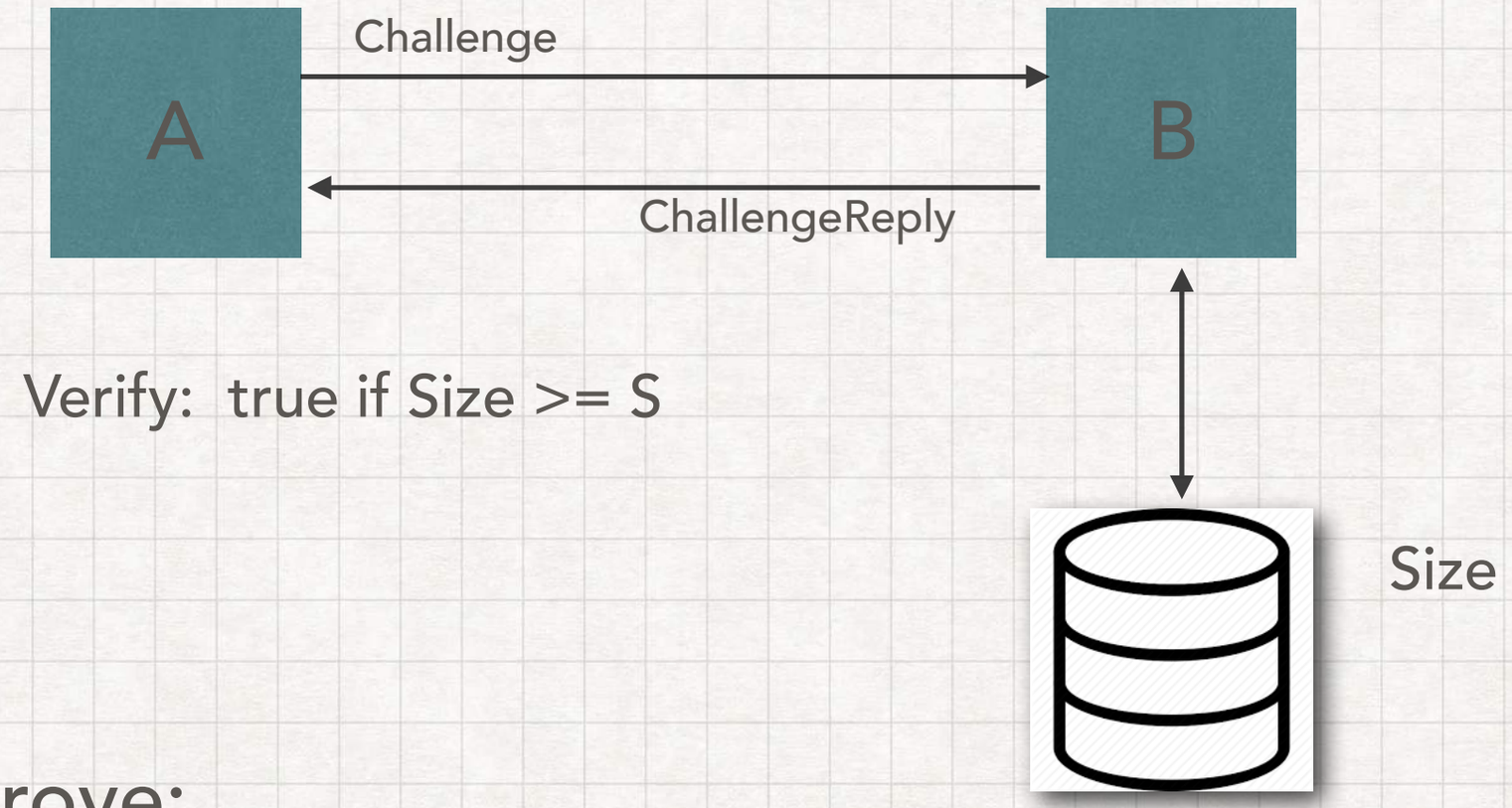
Proof of secure erasure



Decentralized storage.

PROOF OF STORAGE

DECENTRALIZED STORAGE



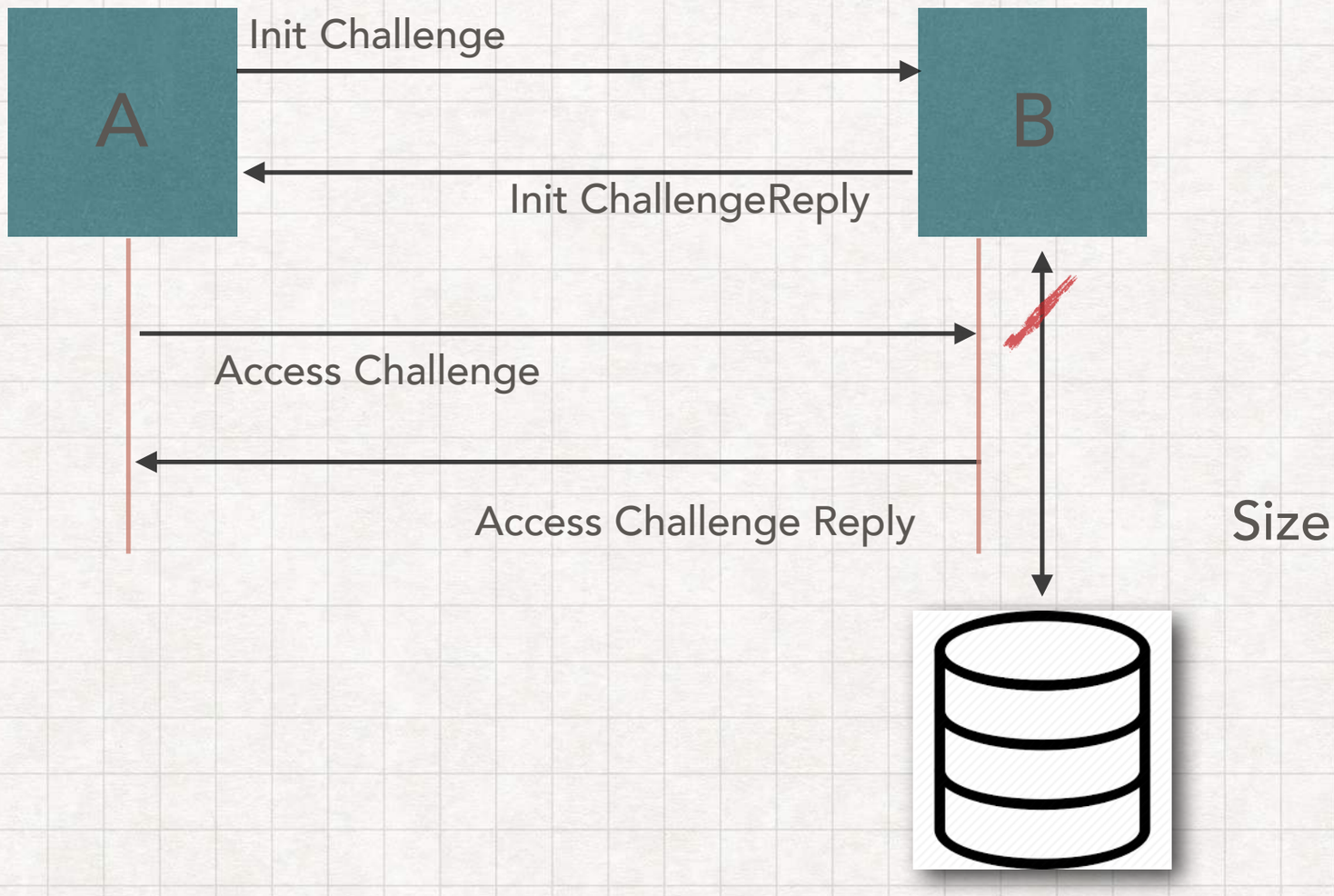
How to prove:

- B still have access to storage after init.
- B store something useful ....

PROOF OF STORAGE

B STILL HAVE ACCESS TO STORAGE AFTER 1-ST CHECK.

Verify: true if Size >= S

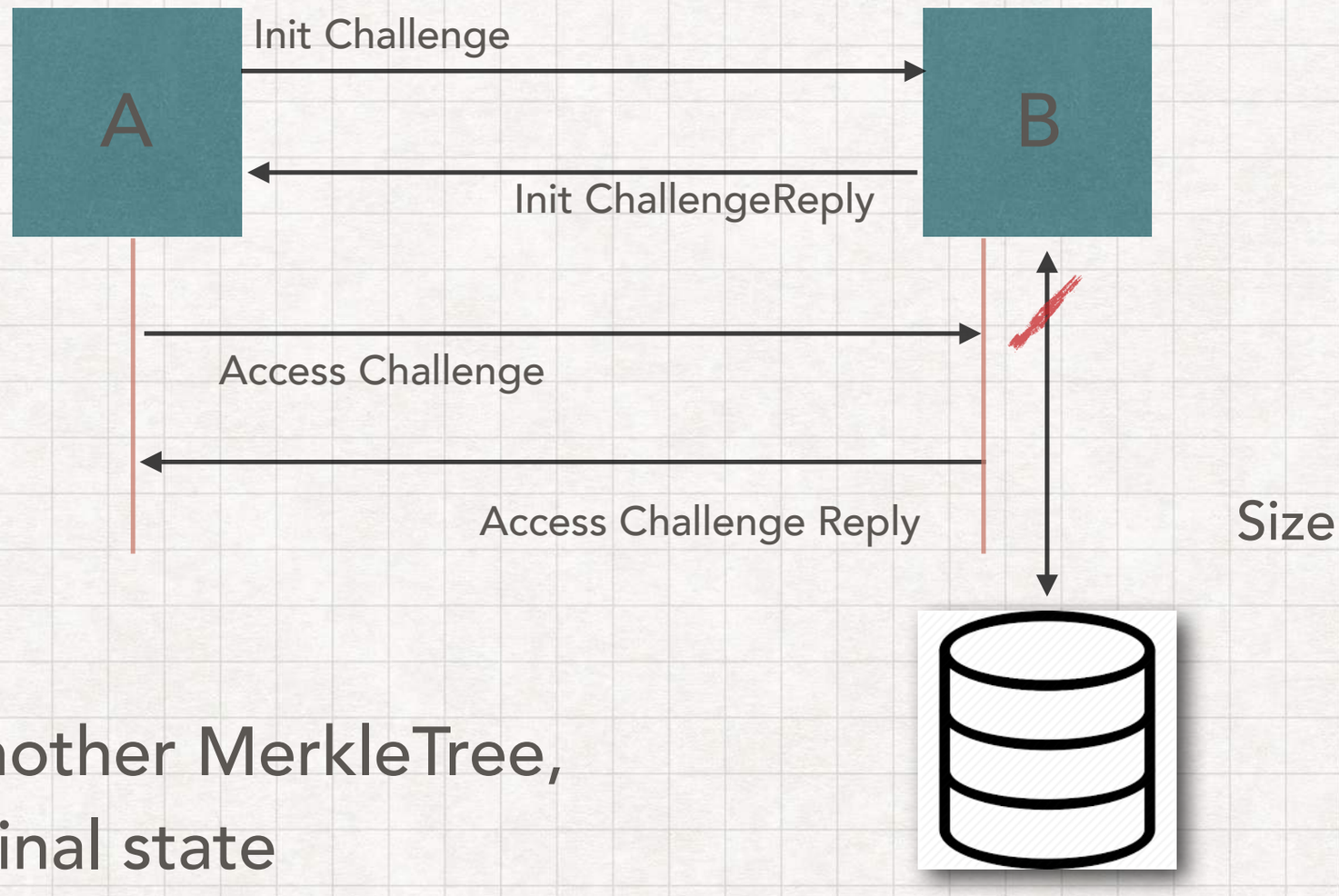


?

PROOF OF STORAGE

B STILL HAVE ACCESS TO STORAGE AFTER 1-ST CHECK.

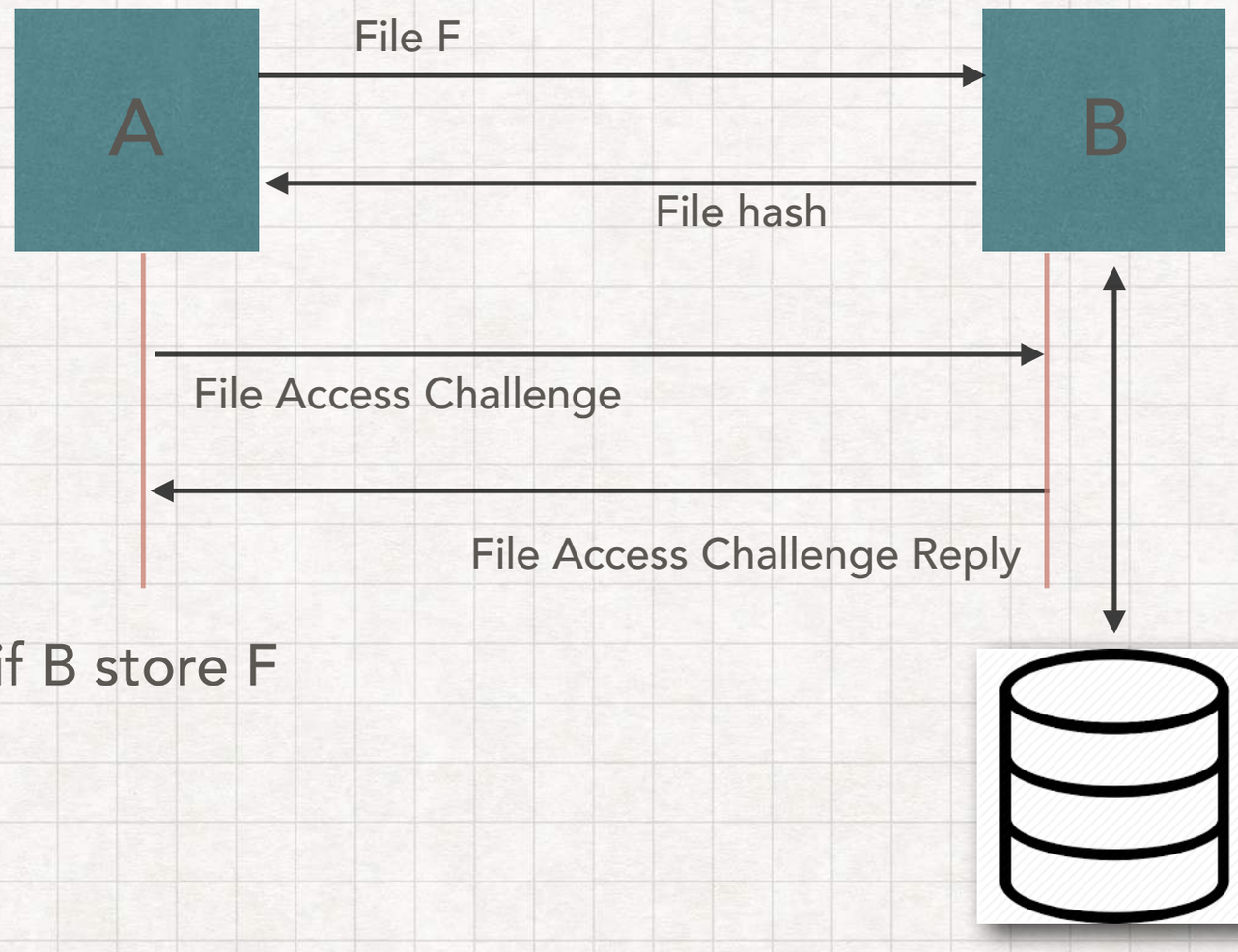
Verify: true if Size >= S



Idea — yet another MerkleTree,
for final state

PDP: PROVABLE DATA PROCESSION

STORE SOMETHING USEFUL



Verify: true if B store F

?

PDP, POR, POST .....

THIS WILL BE THE OTHER STORY

Thanks for attention.

[Q]

<ruslan@shevchenko.kiev.ua>
@rssh1
https://github.com/rssh

garuda.ai

Garuda.AI