Distributed Recursion

March 2011

Sergio Rajsbaum
Instituto de Matemáticas, UNAM

joint work with Eli Gafni, UCLA

Introduction

- Though it is a new field, computer science already touches virtually every aspect of human endeavor

From "Foundations of Computer Science" by Al Aho and Jeff Ullman

But...

- fundamentally, computer science is a science of abstraction
- creating the right model for thinking about a problem and devising the appropriate mechanizable techniques to solve it.

From "Foundations of Computer Science" by Al Aho and Jeff Ullman

Algorithms

- the techniques used to obtain solutions by manipulating data as represented by the abstractions of a data model

From “Foundations of Computer Science” by Al Aho and Jeff Ullman

Recursion

- a very useful technique for defining concepts and solving problems
- Whenever we need to define an object precisely or whenever we need to solve a problem, we should always ask, "What does the recursive solution look like?"

From "Foundations of Computer Science" by Al Aho and Jeff Ullman

Recursion

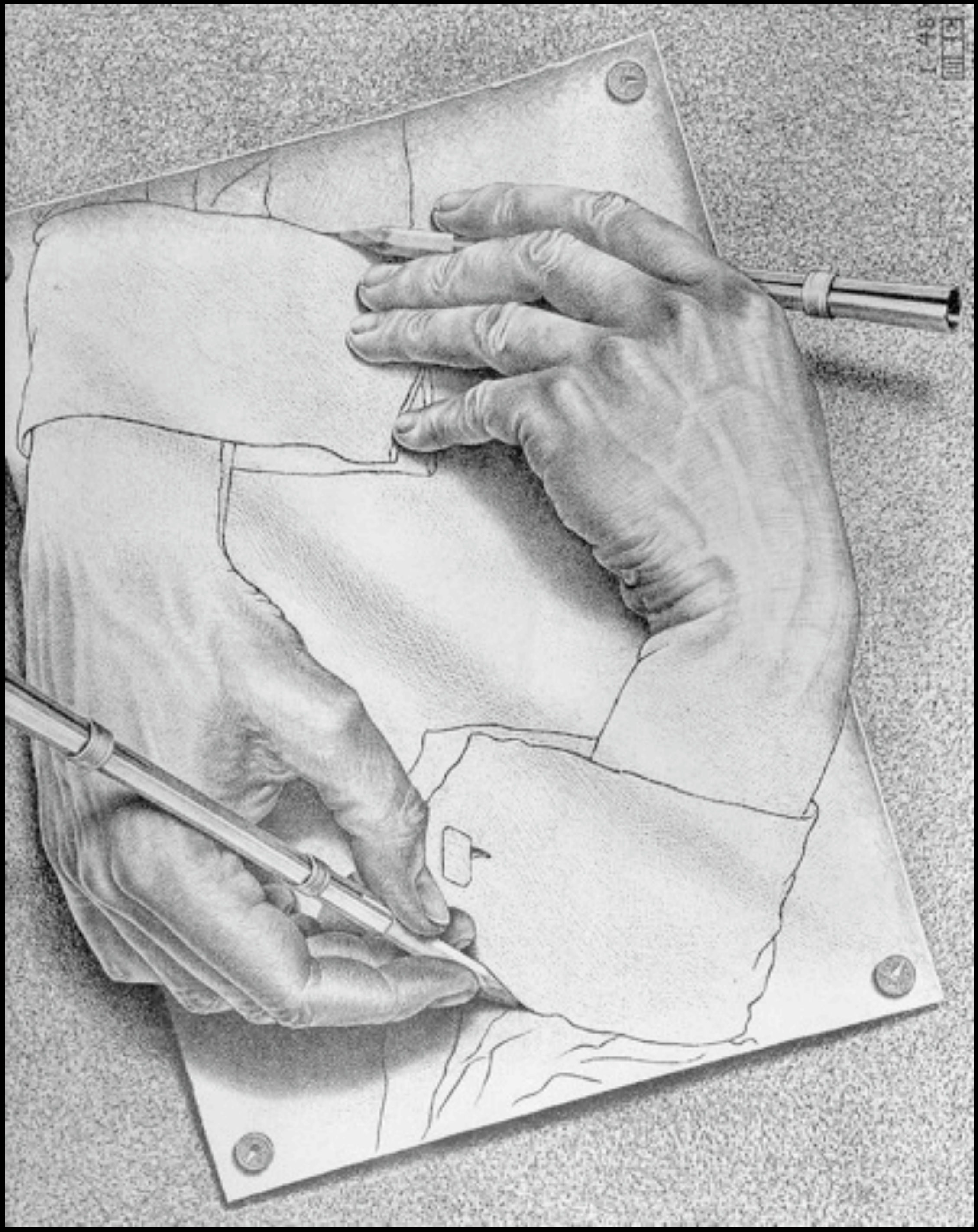
- The power of computers comes from their ability to execute the same task, or different versions of the same task, repeatedly.
- in recursion a concept is defined, directly or indirectly, in terms of itself.

From “Foundations of Computer Science” by Al Aho and Jeff Ullman

Recursive definitions

define a class of objects in terms of the objects themselves.

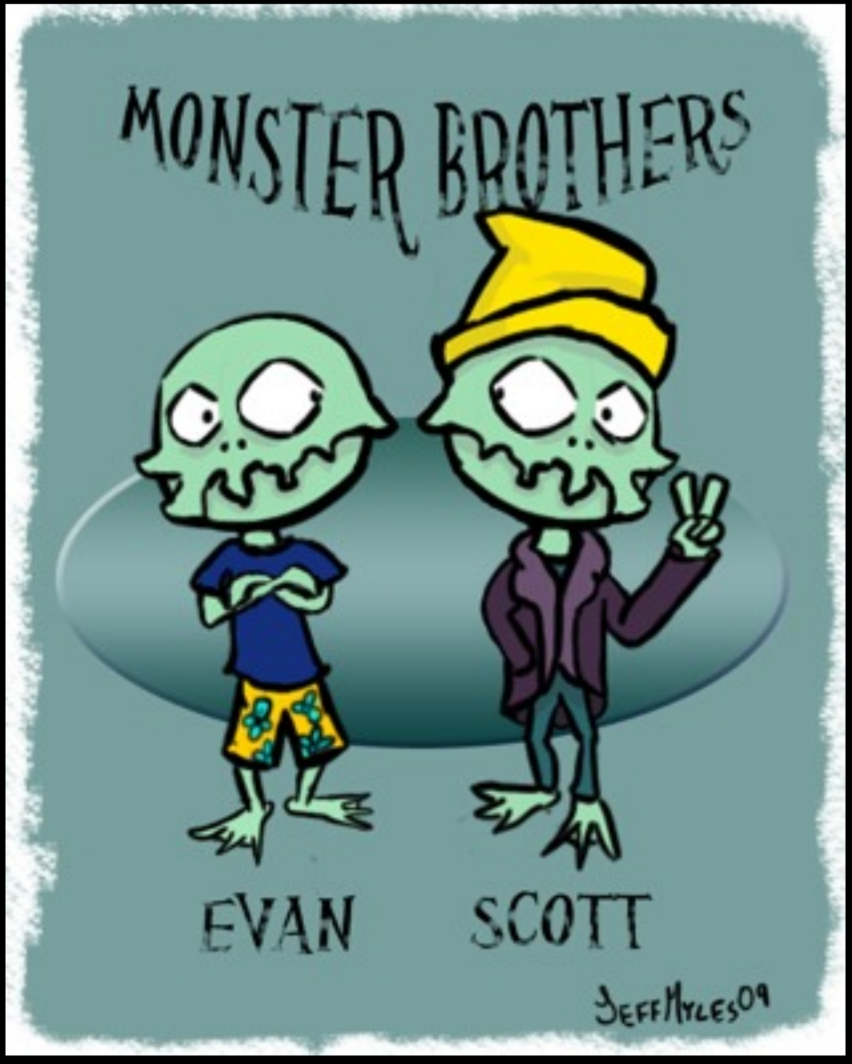
To be meaningful...



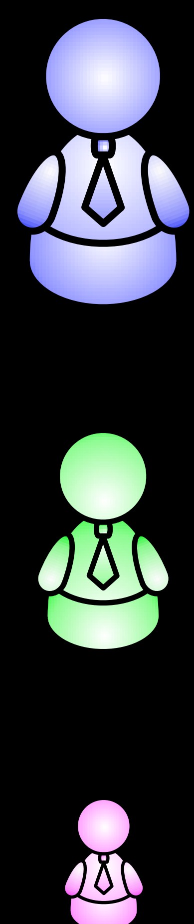
To be meaningful

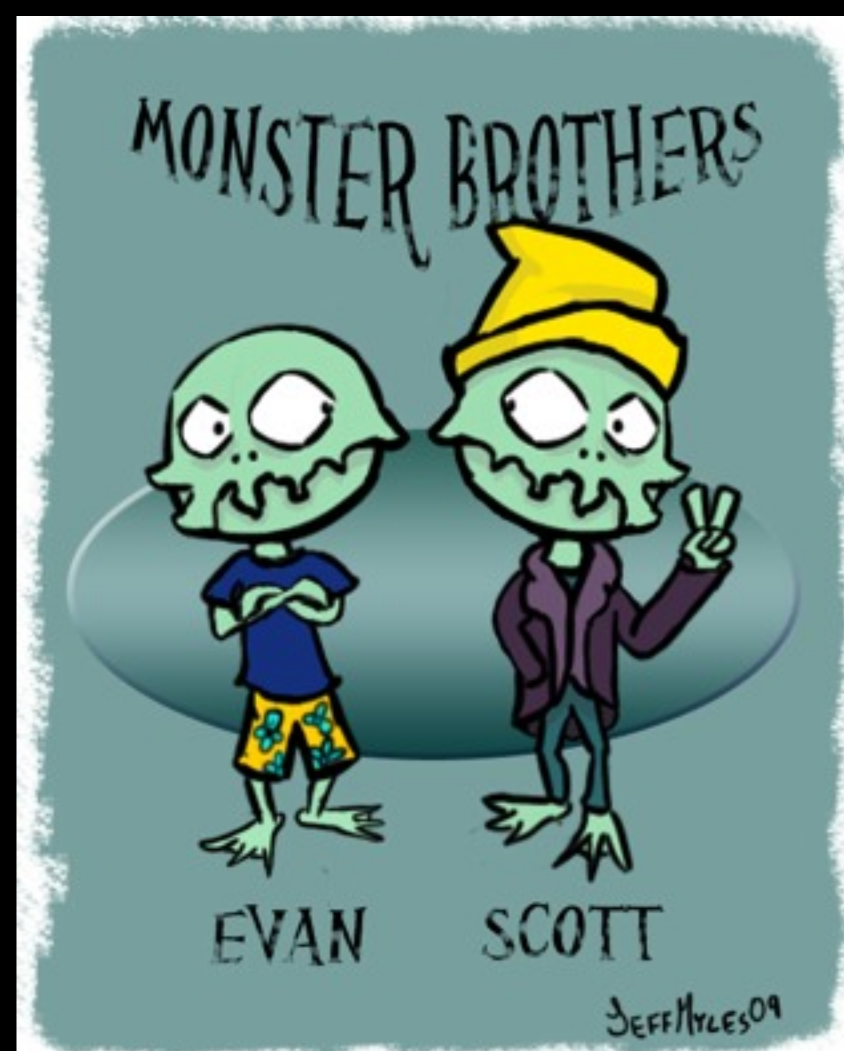
1. One or more basis rules, in which some simple objects are defined, and
2. Inductive rules, whereby larger objects are defined in terms of smaller ones in the collection.

Understanding recursion using "friends"



Understanding recursion using “friends”

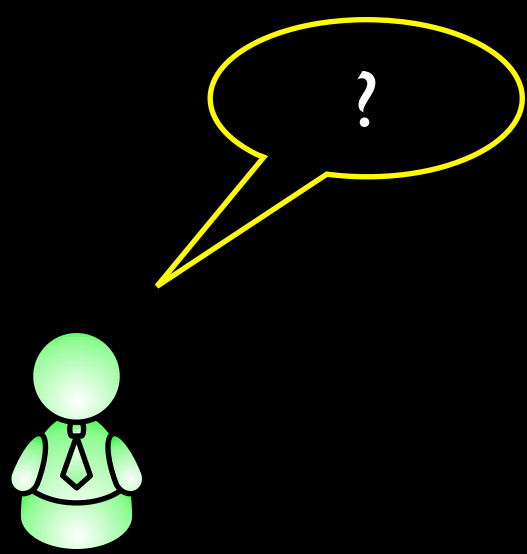


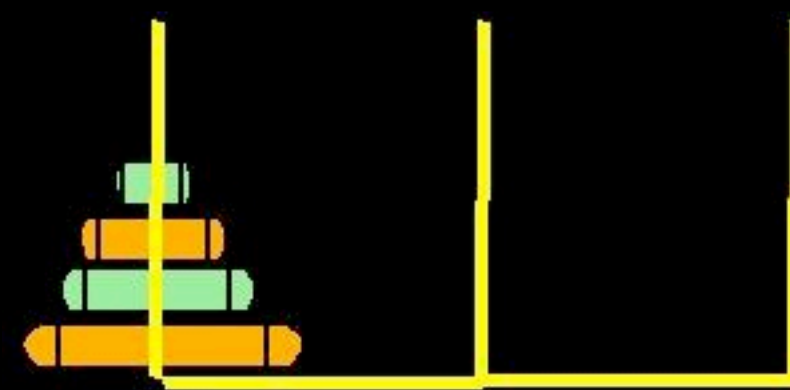


Towers of Hanoi

Towers of Hanoi using friends

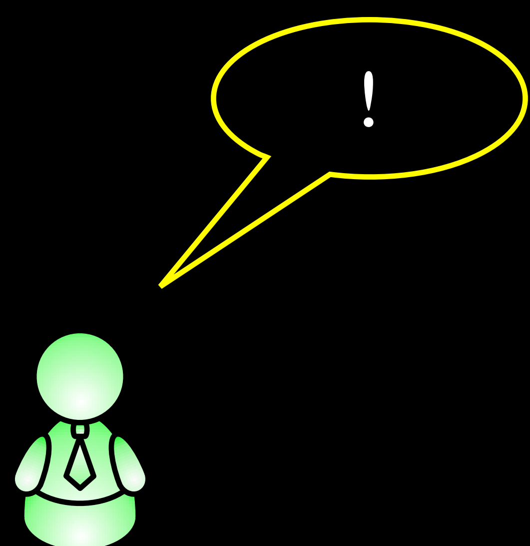
How do I solve
Towers of Hanoi?

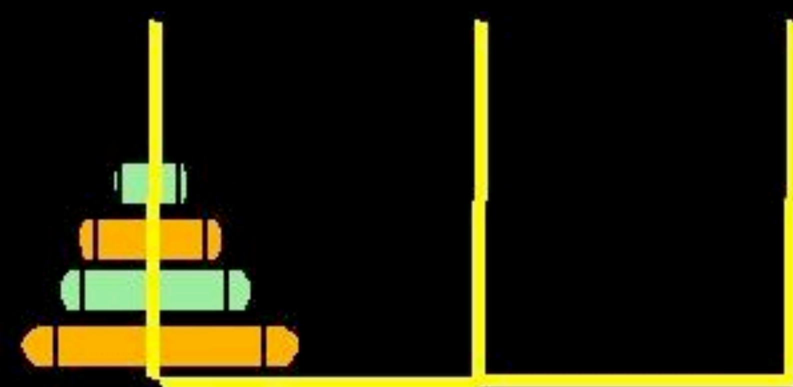




Towers of Hanoi using friends

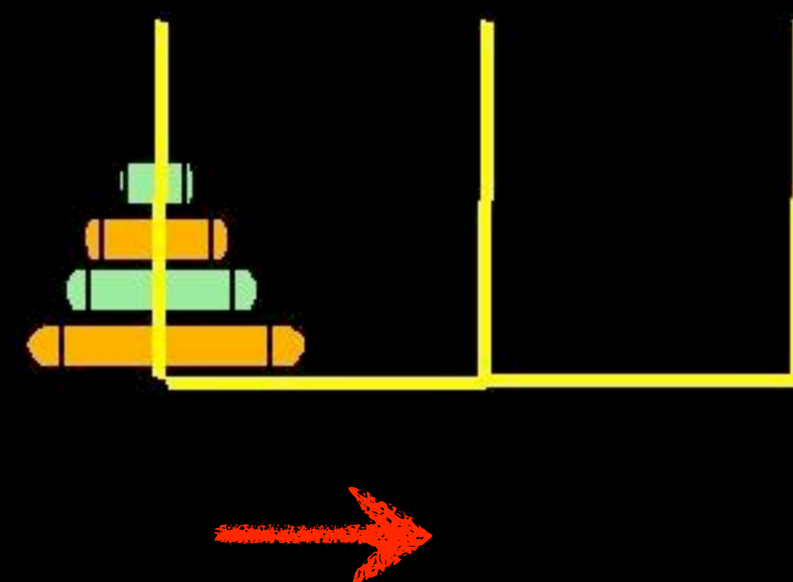
Ask a (younger) friend for help with a smaller problem





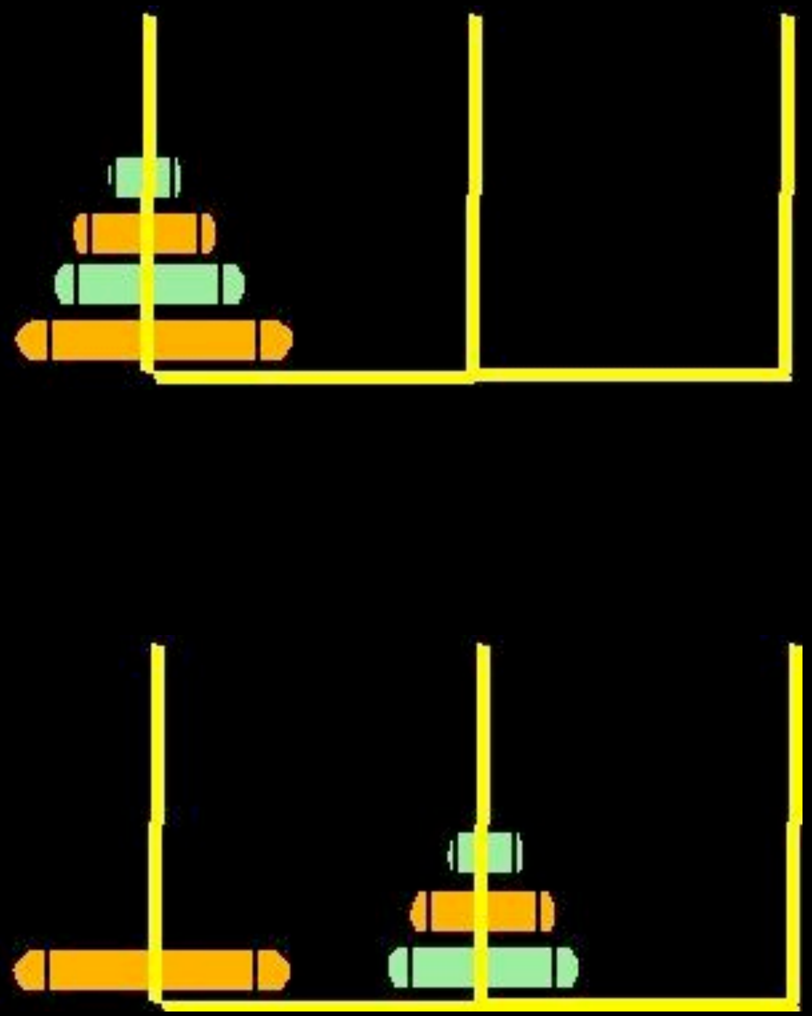
Towers of Hanoi using friends

Ask a (younger) friend for help with a smaller problem



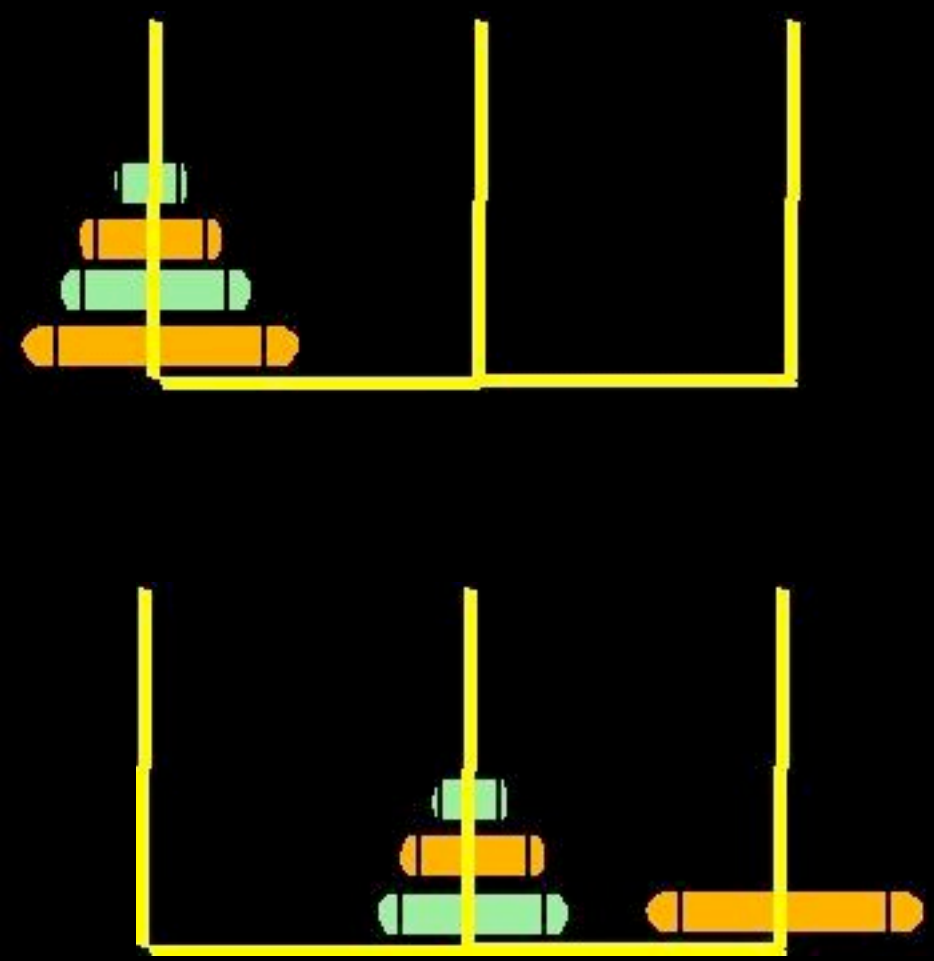
Towers of Hanoi using friends

Ask a (younger) friend for help with a smaller problem



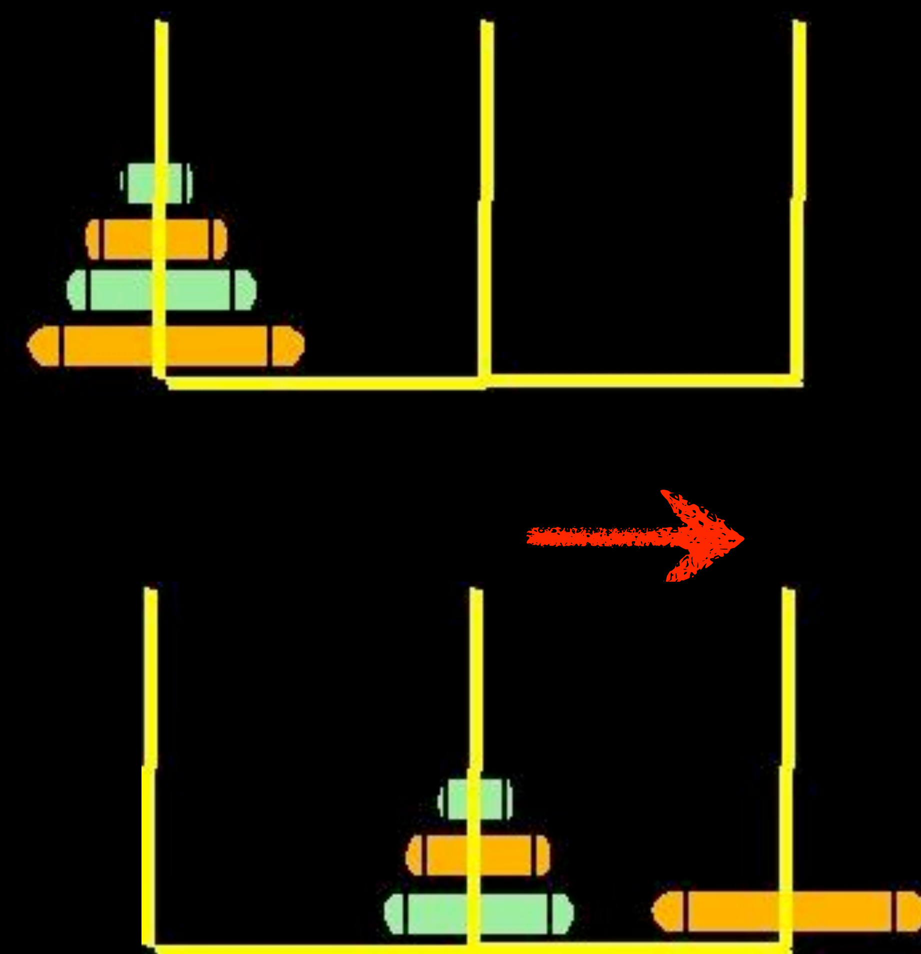
Towers of Hanoi using friends

Ask a (younger) friend for help with a smaller problem



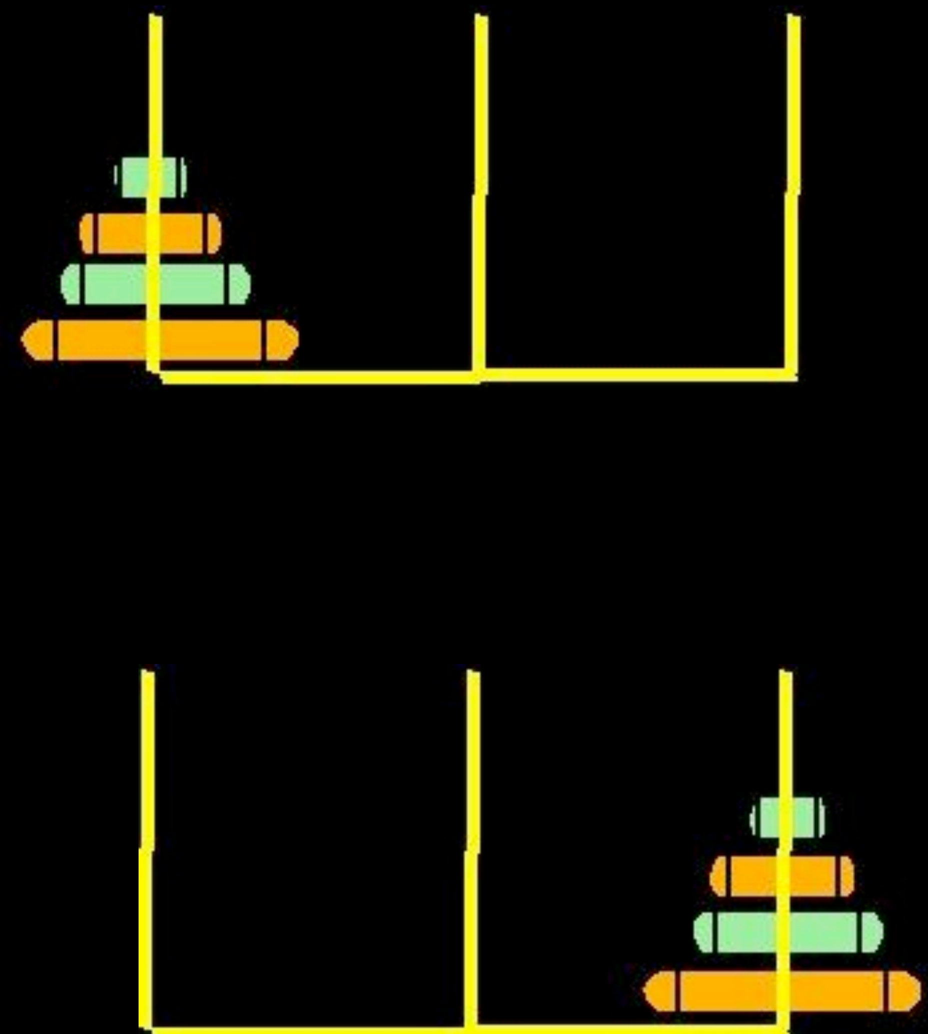
Towers of Hanoi using friends

Ask a (younger) friend for help with a smaller problem



Towers of Hanoi using friends

Ask a (younger) friend for help with a smaller problem



Towers of Hanoi using friends

Towers of Hanoi using friends

Basic elements in a recursive function *f* :

Towers of Hanoi using friends

Basic elements in a recursive function *f* :

- “Split” into smaller problems

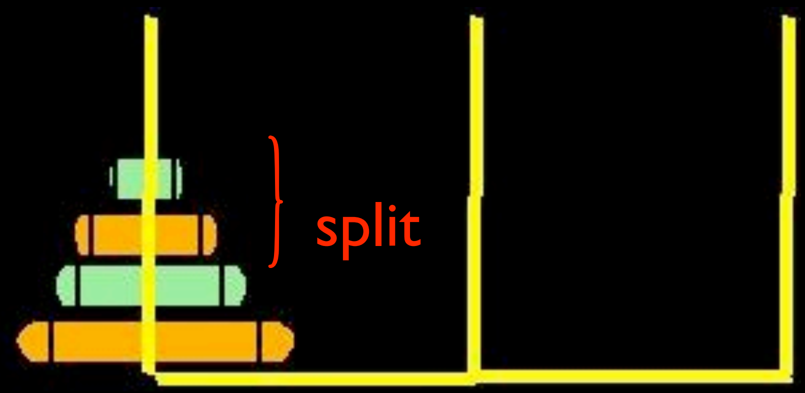
Towers of Hanoi using friends

Basic elements in a recursive function *f* :

- "Split" into smaller problems
- invoke *f* on them

Towers of Hanoi using friends

Basic elements in a recursive function *f* :

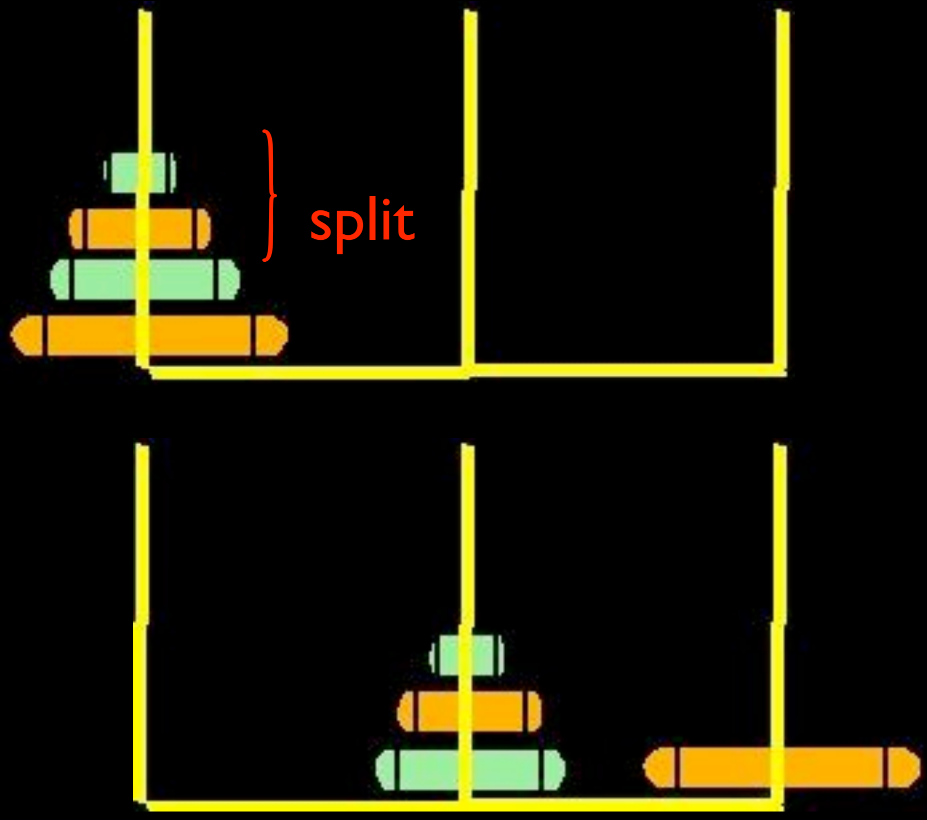


- "Split" into smaller problems
- invoke *f* on them
- "merge" the results

Towers of Hanoi using friends

Basic elements in a recursive function *f* :

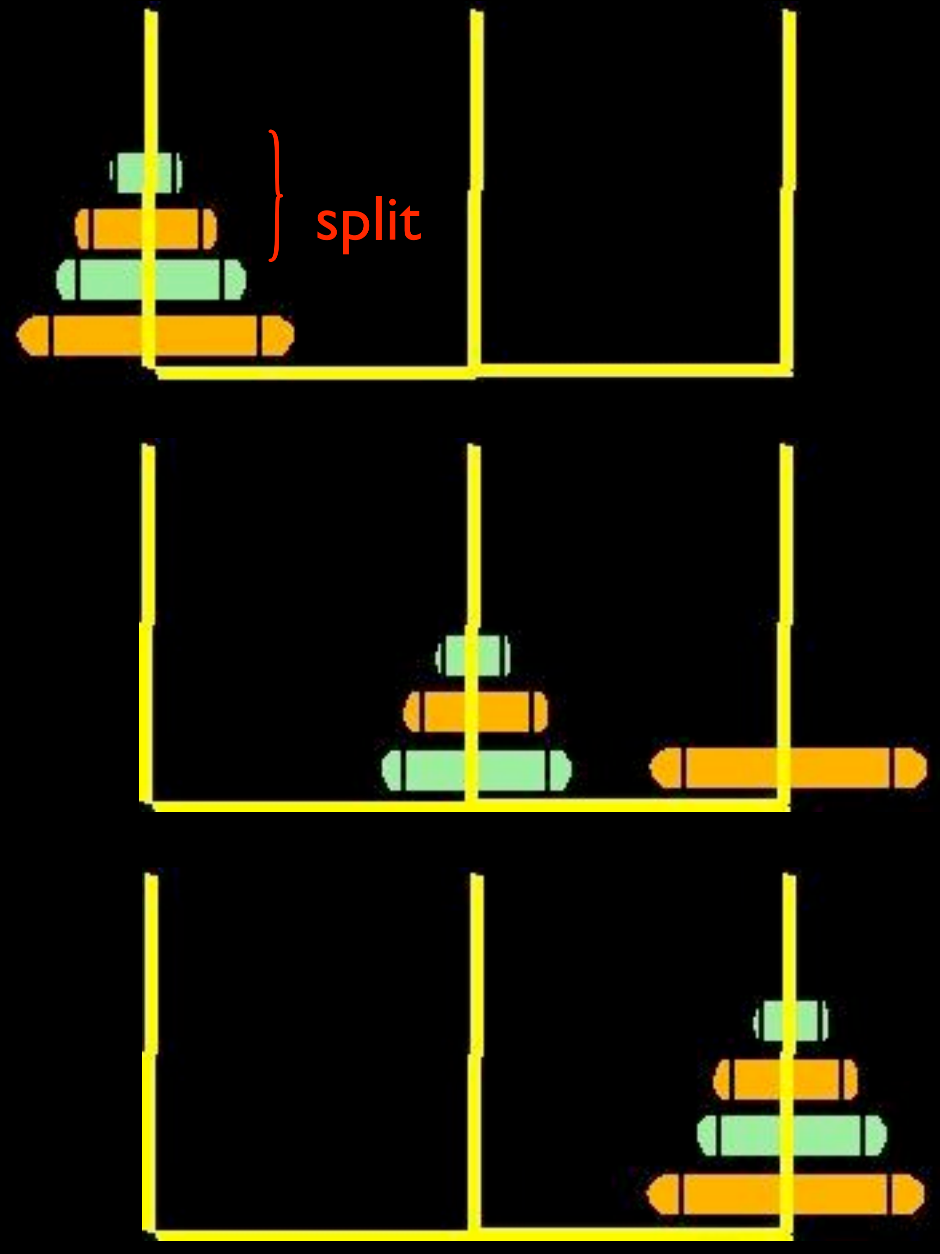
- "Split" into smaller problems
- invoke *f* on them
- "merge" the results



Towers of Hanoi using friends

Basic elements in a recursive function *f* :

- "Split" into smaller problems
- invoke *f* on them
- "merge" the results



Towers of Hanoi

Challenge: find a non-recursive algorithm

Recursive programs are often more succinct or easier to understand than their iterative counterparts.

More importantly, some problems are more easily attacked by recursive programs than by iterative programs.

From "Foundations of Computer Science" by Al Aho and Jeff Ullman

Recursion in distributed algorithms
(need real friends)

Garfield and Friends

Motivation

The benefits of designing and analyzing *sequential* algorithms using recursion are well known.

Motivation

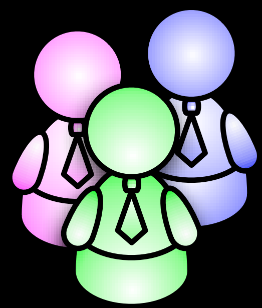
The benefits of designing and analyzing *sequential* algorithms using recursion are well known.

However, little use of recursion has been done in *distributed* algorithms

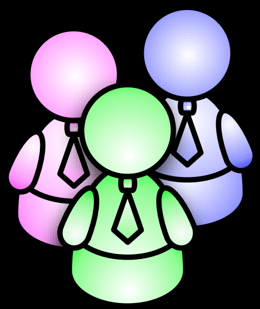
Recursion in distributed algorithms

Recursion in distributed algorithms

- Instead of just one process, many

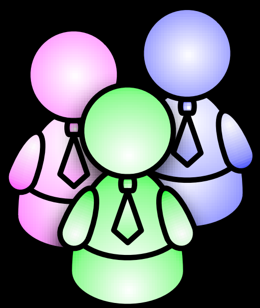


Recursion in distributed algorithms



- Instead of just one process, many
- Split the problem now means: subproblems for fewer processes

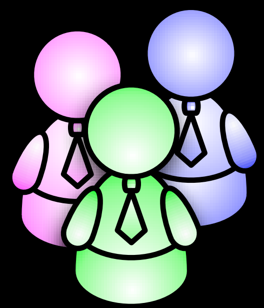
Recursion in distributed algorithms



- Instead of just one process, many
- Split the problem now means: subproblems for fewer processes
- Get help from smaller friend groups to solve the subproblems

Recursion in distributed algorithms

- Instead of just one process, many

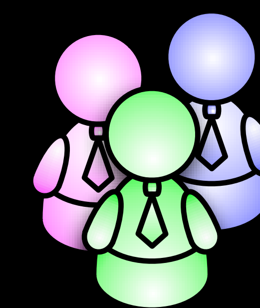


- Split the problem now means: subproblems for fewer processes

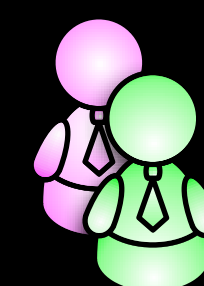
- Get help from smaller friend groups to solve the subproblems
- Until a problem for one friend is reached

Recursion in distributed algorithms

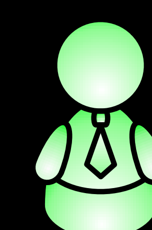
- Instead of just one process, many



- Split the problem now means: subproblems for fewer processes
- Get help from smaller friend groups to solve the subproblems



- Until a problem for one friend is reached



Problems

Problems

- In seq., functions: one input, one output

Problems

- In seq., functions: one input, one output
- In dist., we consider *tasks*: distributed inputs/outputs, represented as vectors

Problems

- In seq., functions: one input, one output
- In dist., we consider *tasks*: distributed inputs/outputs, represented as vectors
- Interested mainly in coordination, local computation power disregarded

Problems

- In seq., functions: one input, one output
- In dist., we consider *tasks*: distributed inputs/outputs, represented as vectors
- Interested mainly in coordination, local computation power disregarded
- see some examples...

Agreement tasks

Agreement tasks

- consensus: agree on *1* value

Agreement tasks

- consensus: agree on 1 value
- k-set agreement: on at most k values

Agreement tasks

- consensus: agree on *1* value
- *k*-set agreement: on at most *k* values
- snapshots: on possible views of a run, subsets ordered by containment

Disagreement tasks

Disagreement tasks

- Leader election: one of the participants

Disagreement tasks

- Leader election: one of the participants
- Symmetry breaking: not all decide the same value

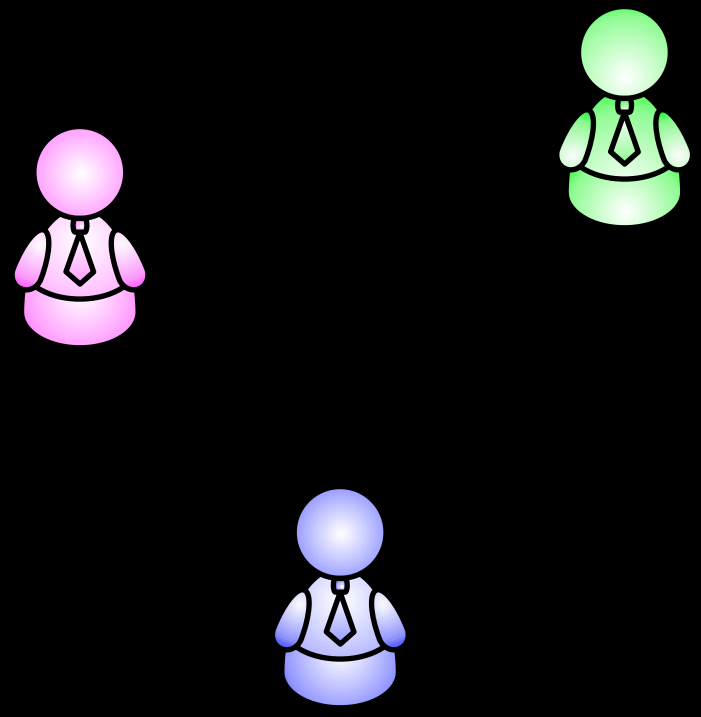
Disagreement tasks

- Leader election: one of the participants
- Symmetry breaking: not all decide the same value
- Renaming: all decide different values, names on a small name space

Wait-free read/write shared memory model

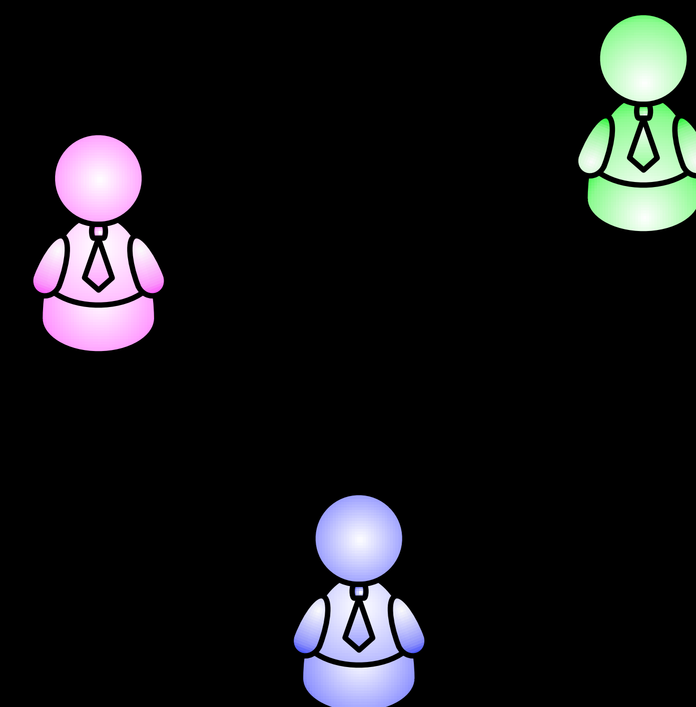
Wait-free read/write shared memory model

- n Processes



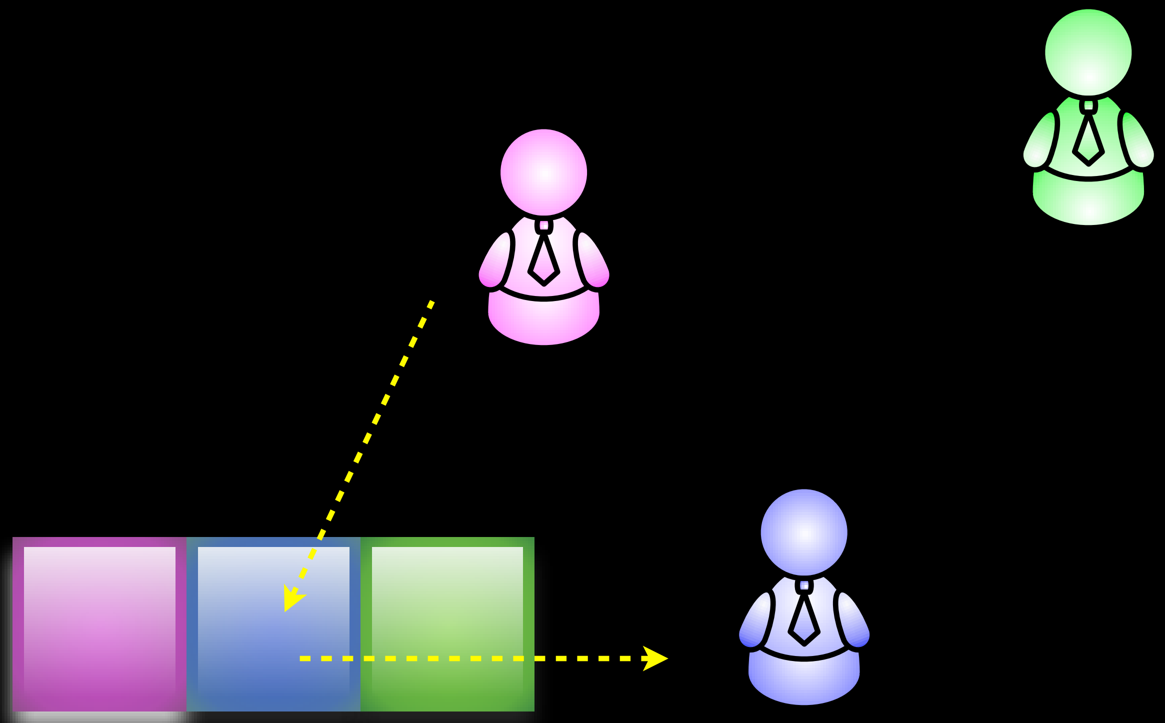
Wait-free read/write shared memory model

- n Processes
- Communication



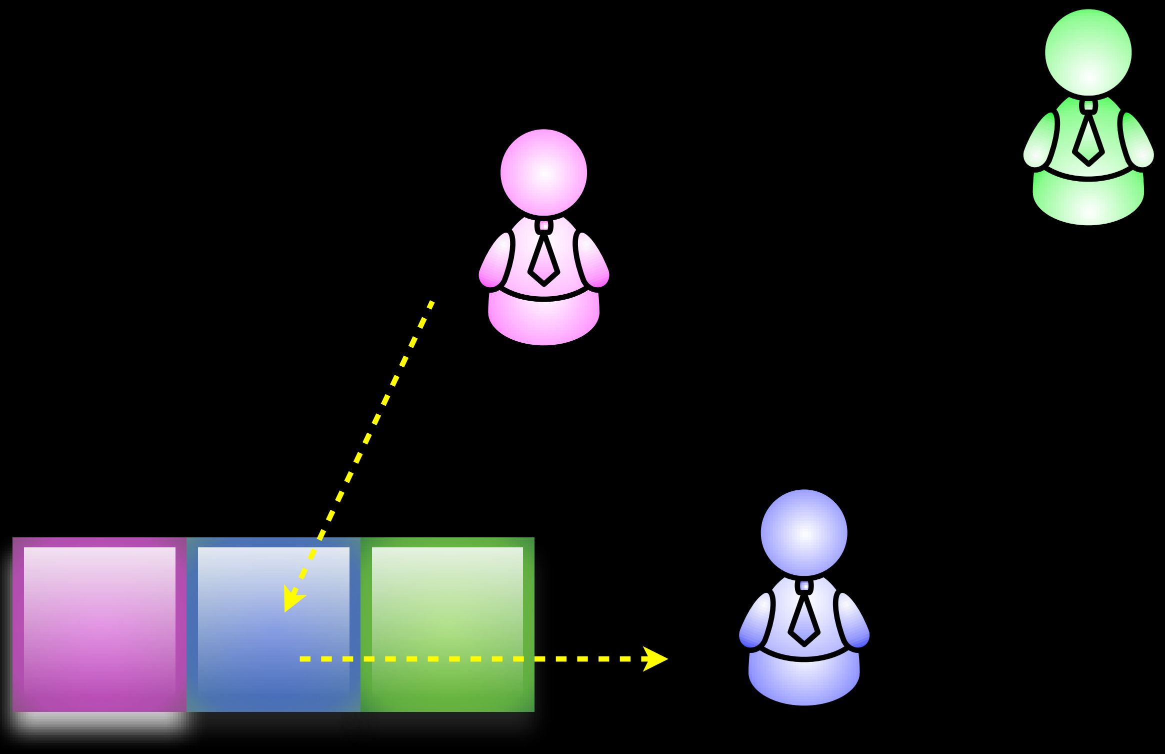
Wait-free read/write shared memory model

- n Processes
- Communication



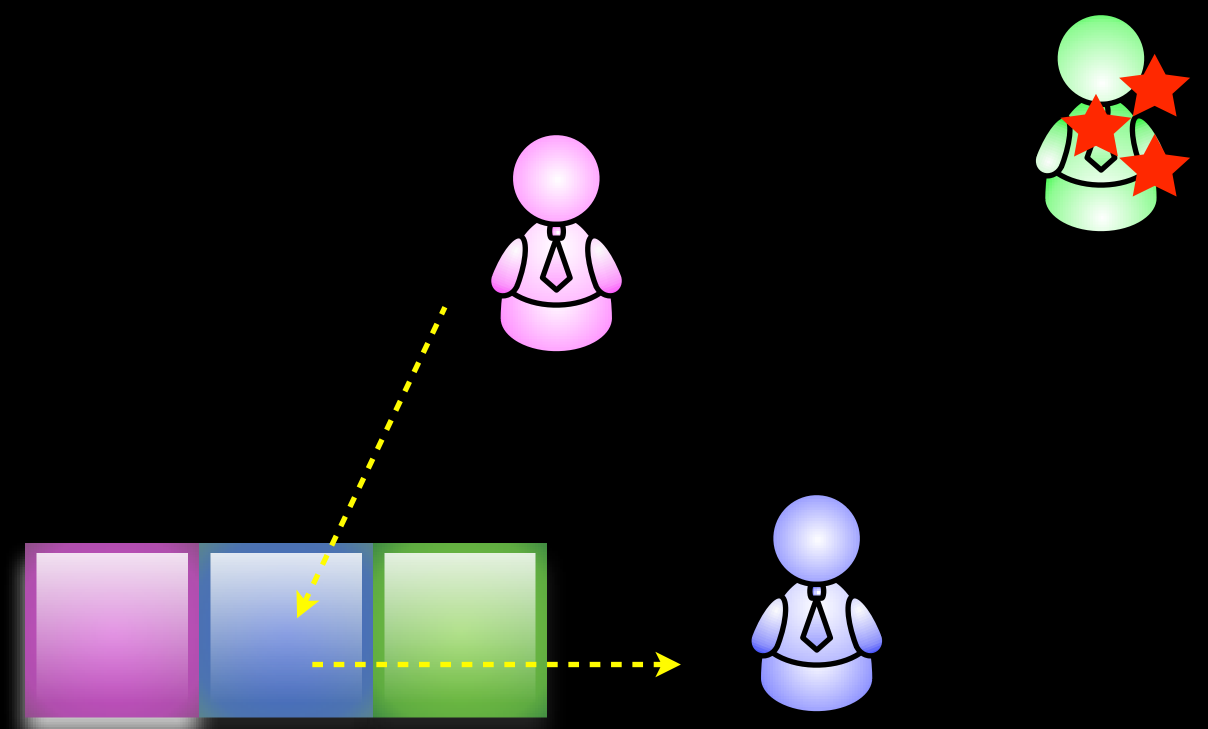
Wait-free read/write shared memory model

- n Processes
- Communication
- Asynchronous



Wait-free read/write shared memory model

- n Processes
- Communication
- Asynchronous
- Any number may crash



Distributed splitters

asking help from smaller groups of friends

Distributed splitters

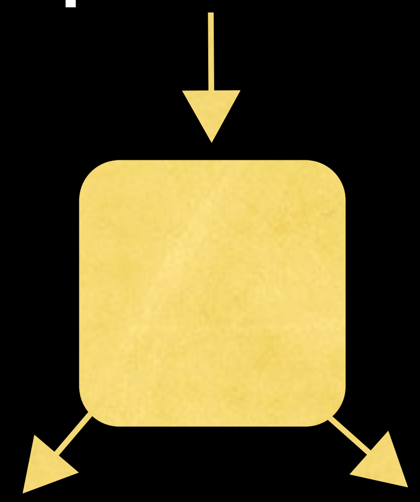
Distributed splitters

- Is there a wait-free algorithm to split in two?

Distributed splitters

- Is there a wait-free algorithm to split in two?
- Perfect splitting No!

n processes

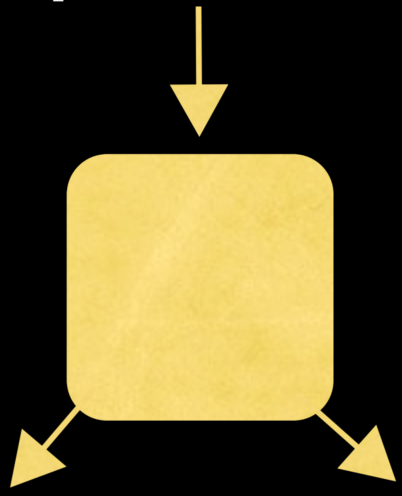


left: *n/2*

right: *n/2*

Strong splitter

n processes

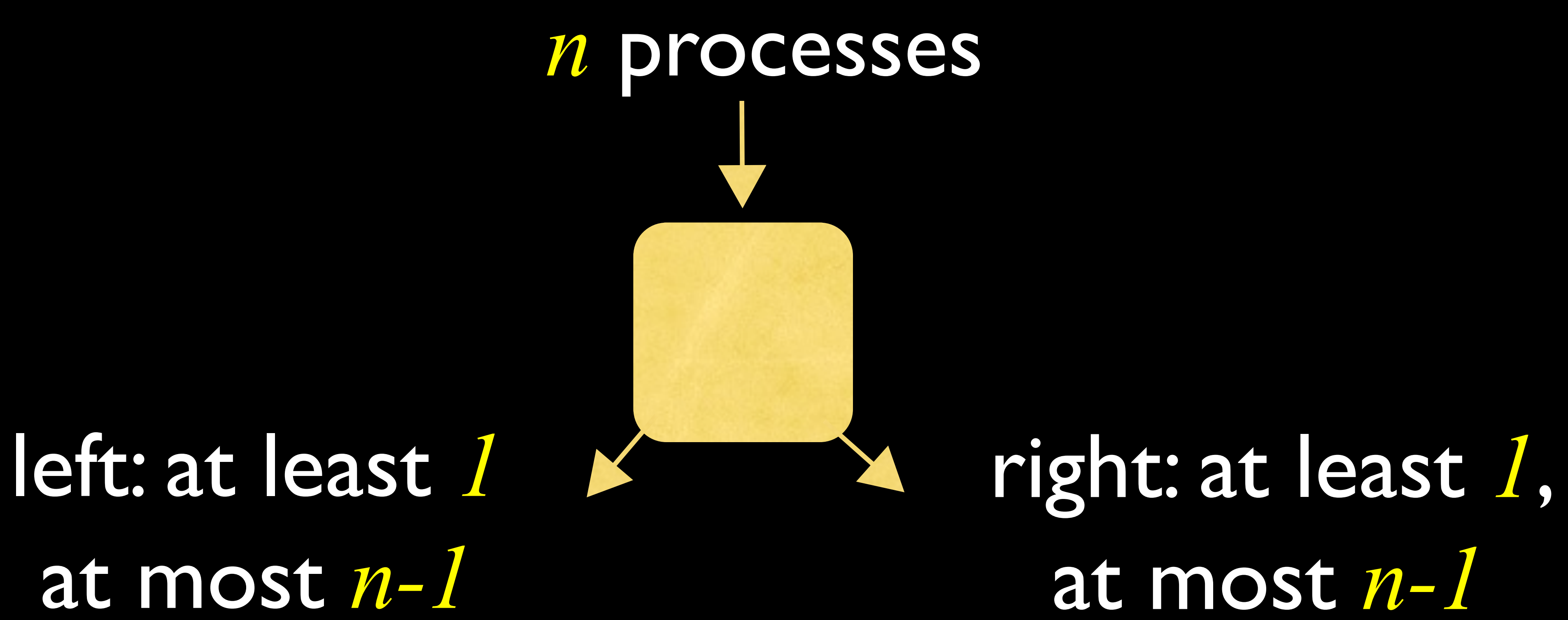


left: at least 1
at most n-1

right: at least 1,
at most n-1

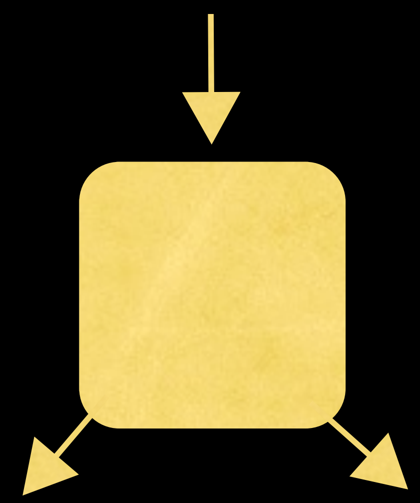
Strong splitter

- No! Need objects stronger than read/write
  (except for some values of *n*: WSB problem [Castañeda,Rajsbaum podc08])



Very Weak splitter

n processes



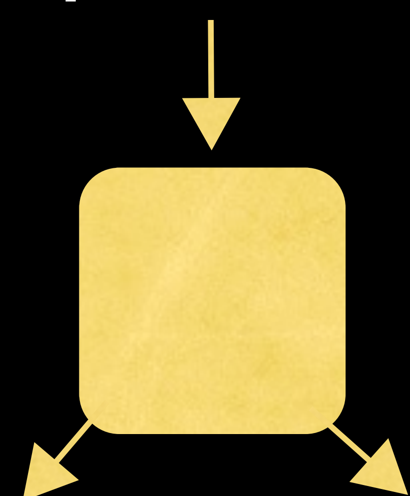
left: at least *0*, at most *n-1*

right: at least *1*, at most *n*

Very Weak splitter

- there is a wait-free algorithm

n processes



left: at least 0, at most $n-1$

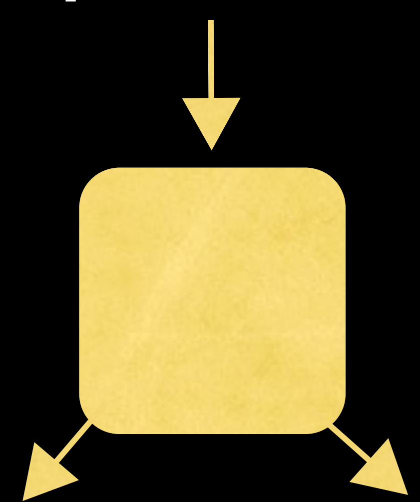
right: at least 1, at most n

Very Weak splitter

- there is a wait-free algorithm

when less than n arrive, they go left

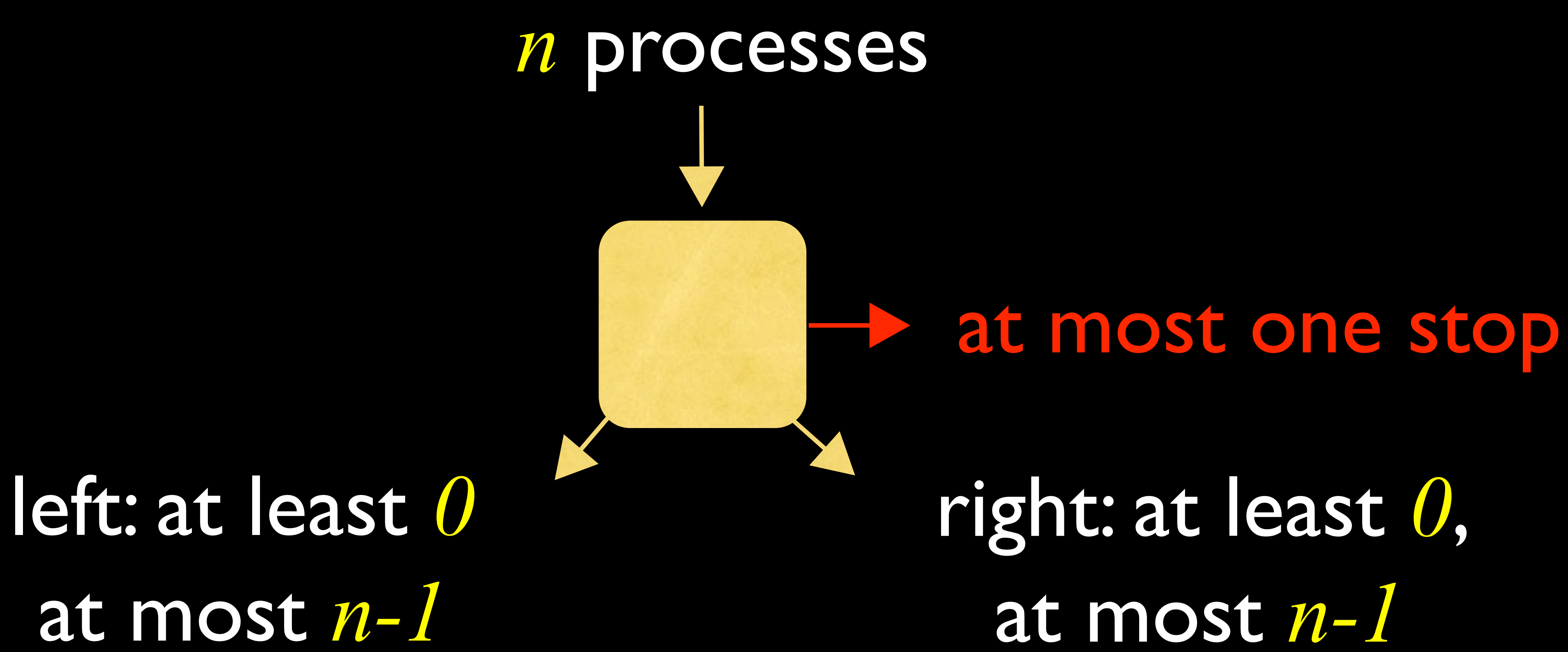
n processes



left: at least 0, at most $n-1$

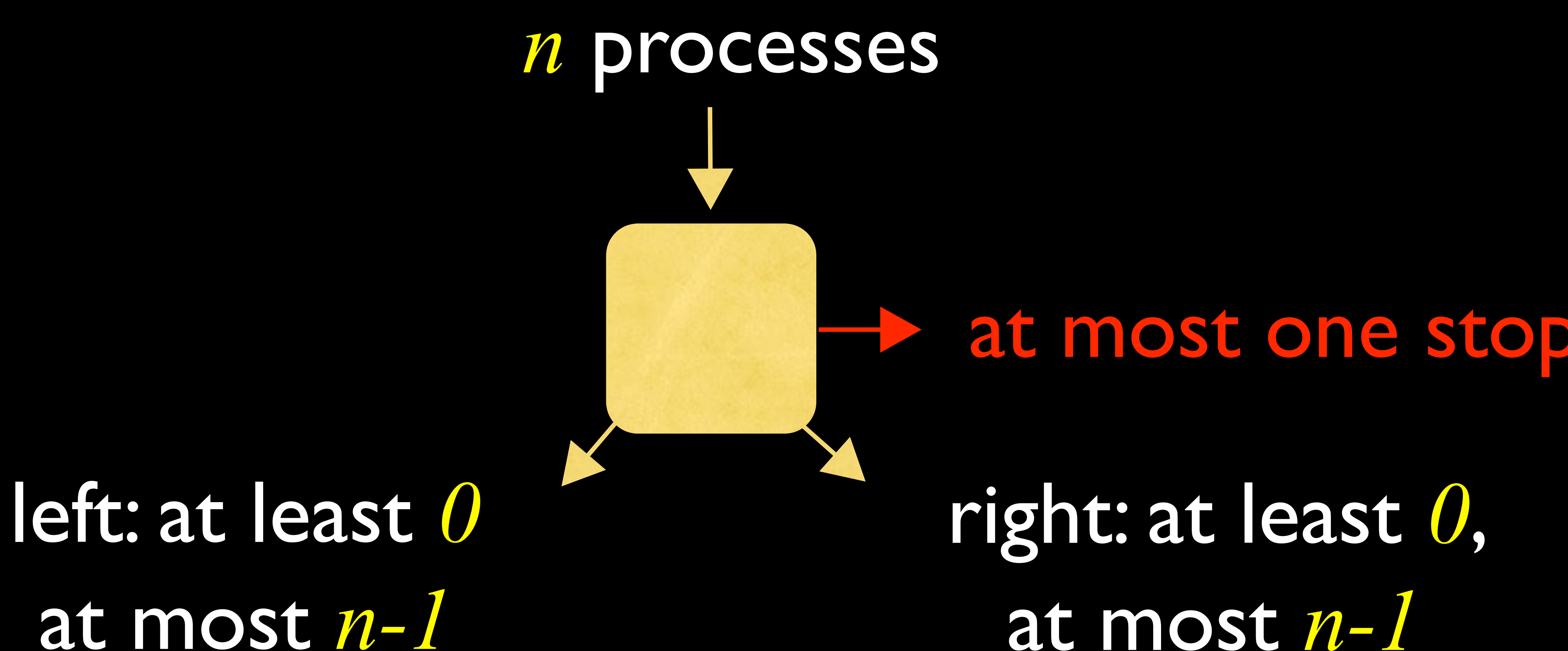
right: at least 1, at most n

Weak splitter



Weak splitter

- Hence there is a wait-free algorithm



Very weak splitter

- Algorithm <u>VWsplitter $_{id}$</u>(n):
  - write id, read all registers
  - if |read-set| = n, then <u>return</u> right
- else
  - <u>return</u> left

Very weak splitter

- Algorithm VWsplitter $_{id}$(n):
  - write id, read all registers

    at least one sees all

  - if |read-set| = n, then return right
- else
  - return left

Very weak splitter

- Algorithm VWsplitter $_{id}$(n):
  - write id, read all registers
  - if |read-set| = n, then return right
- else
  - return left

at least one sees all

at most n-1 call this

Weak splitter

- Algorithm Wsplitter $_{id}$ (n):
  - write id, read all registers
  - if |read-set| = n, then
    - if id= max{read-set} return stop
    - else return right
- else
  - return left

Weak splitter

- Algorithm Wsplitter $_{id}$ (n):
  - write id, read all registers
  - if |read-set| = n, then
    - if id= max{read-set} return stop
    - else return right
- else
  - return left

at least one sees all

Weak splitter

- Algorithm Wsplitter $_{id}$(n):
  - write id, read all registers
  - if |read-set| = n, then
    - if id= max{read-set} return stop
    - else return right
- else
  - return left

at least one sees all

at most n-1 call this

Weak splitter

- Algorithm Wsplitter$_{id}(n)$:
  - write id, read all registers
  - if |read-set| = n, then
    - if id= max{read-set} return stop
    - else return right
- else
  - return left

at least one sees all

at most n-1 call this

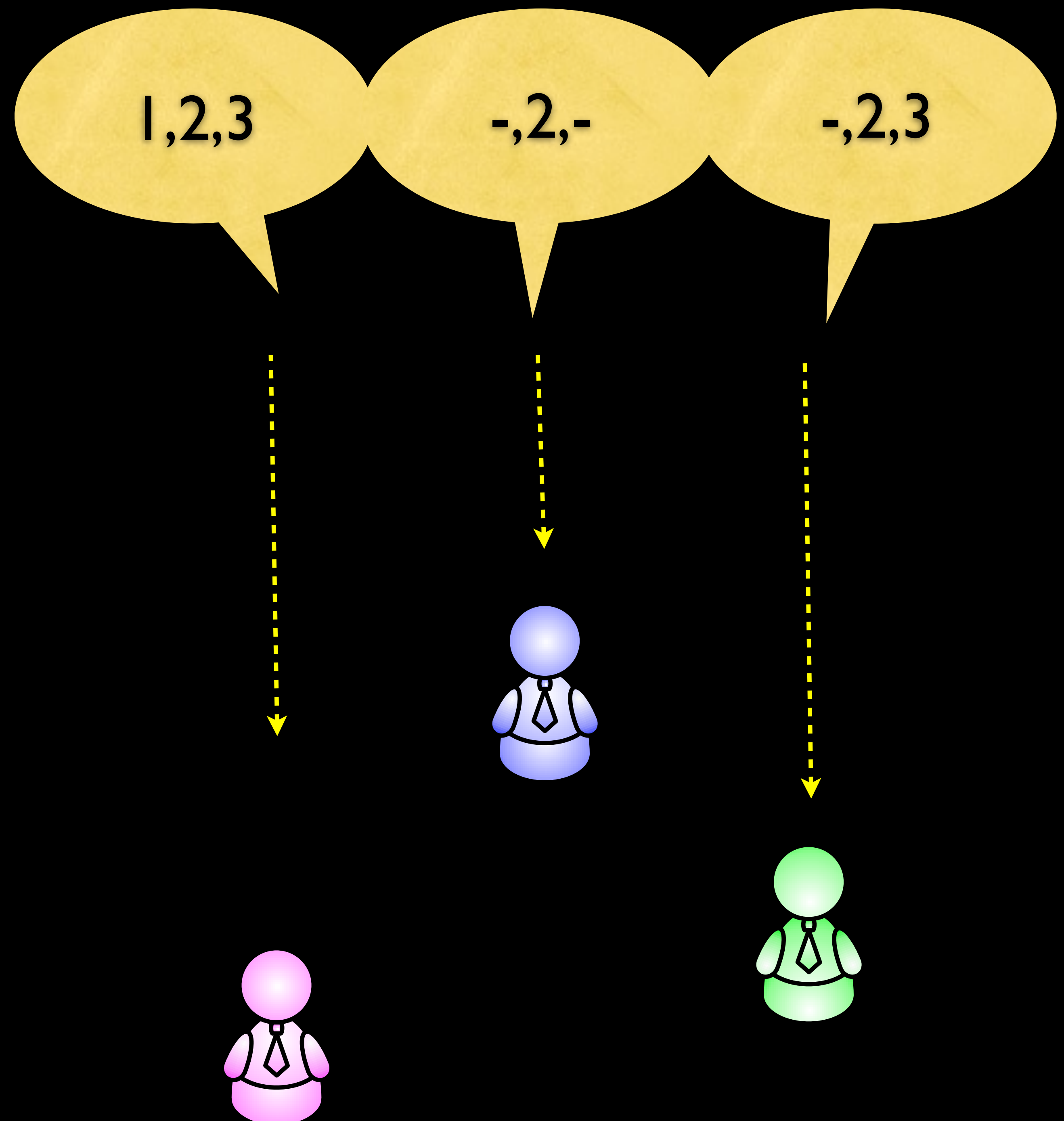
at most n-1 call this

Recursive distributed programming

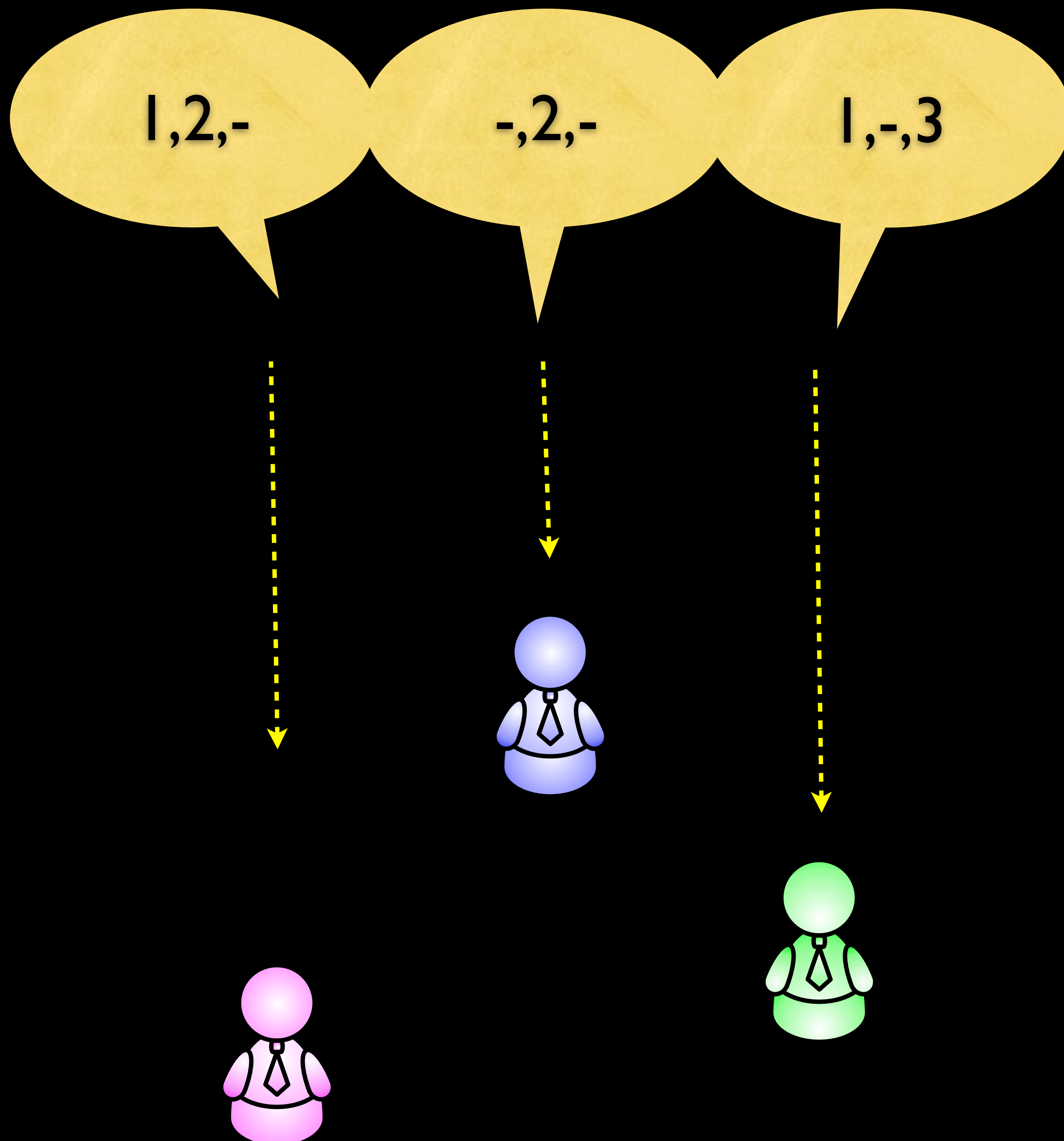
snapshots task

- The goal:
  - Each process obtains a set of ids of participating processes
  - the sets can be ordered by containment
- Used to obtain consistent views of an execution: ids in the same set are concurrent

ok
views



NOT ok
views



VWsplitter snapshots

- Algorithm Snapshot $_{id}$ (n):
  - write id, read all registers
  - if |read-set| = n, then return read-set
- else
  - Snapshot $_{id}$ (n-1)

VWsplitter snapshots

- Algorithm Snapshot $_{id}$ (n):
  - write id, read all registers
  - if |read-set| = n, then return read-set
- else
  - Snapshot $_{id}$ (n-1)

at least one
sees all

VWsplitter snapshots

- Algorithm Snapshot $_{id}$ (n):
  - write id, read all registers
  - if |read-set| = n, then return read-set
- else
  - Snapshot $_{id}$ (n-1)

at least one sees all

at most n-1 call this

VWsplitter snapshots

- Algorithm Snapshot *id* (n):
  - write id, read all registers
  - if |read-set| = n, then return read-set
- else
  - Snapshot *id* (n-1)

at least one sees all

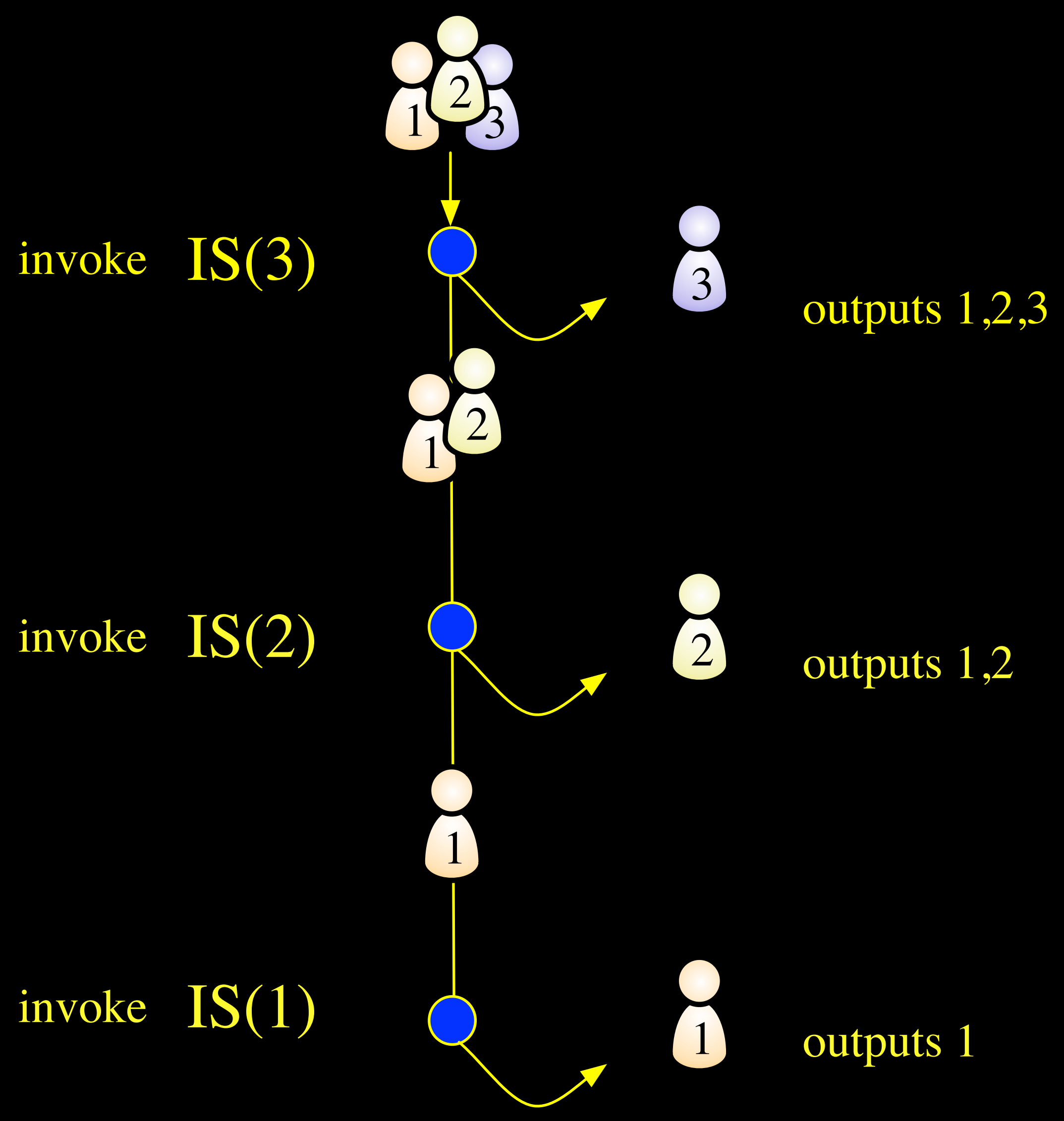
contained in the previous sets

Immediate snapshots

- Algorithm <u>Snapshot $_{id}$</u>(n) computes more than snapshots:
- the snapshot of a process happens immediately after its write
- i in read-set of j then

  read-set of i subset of read-set of j

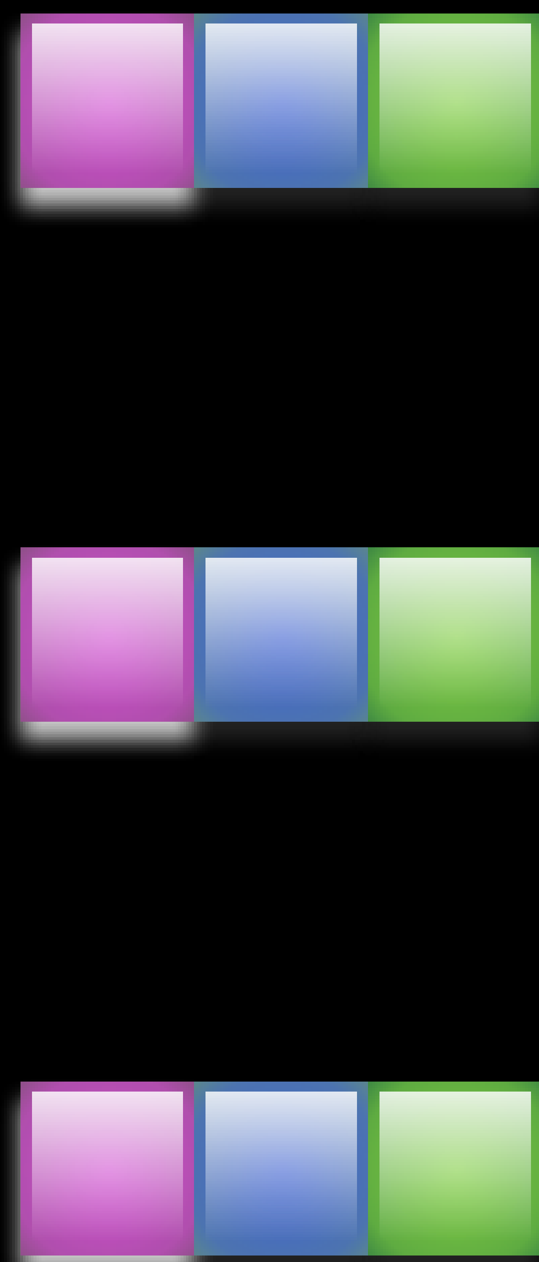
Linear recursion



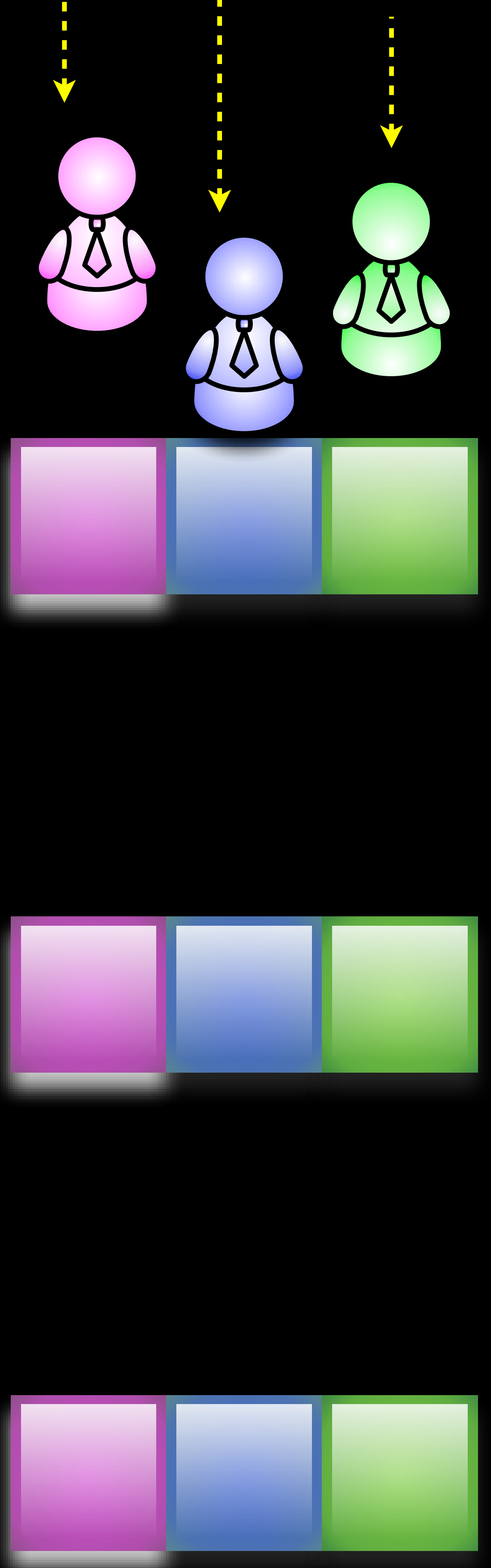
Recursive -> iterated

- when we unfold the recursion, we get a run on a sequence of read/write memories
- because each recursive call works with a fresh memory

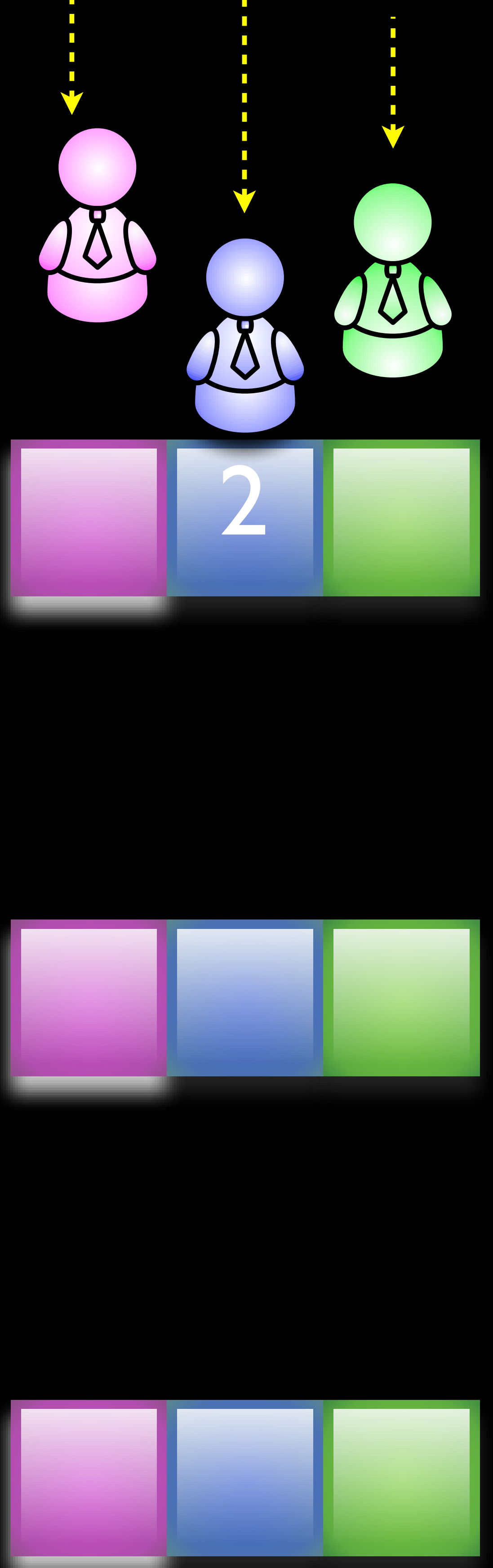
every copy is new



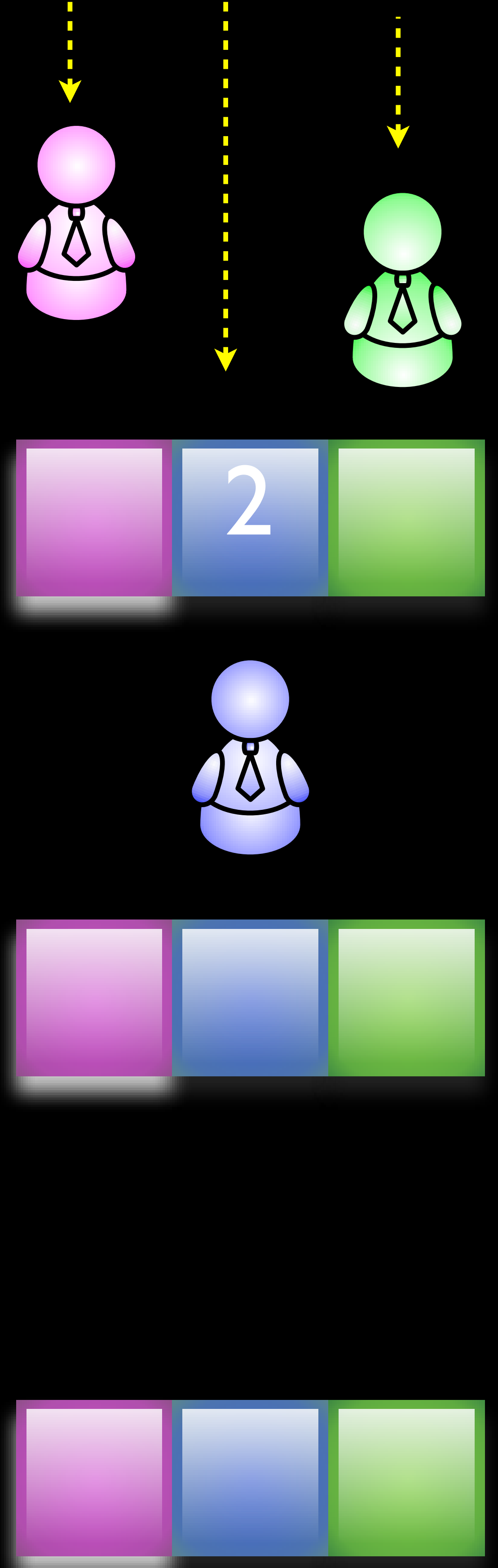
- arrive in arbitrary order
- last one sees all



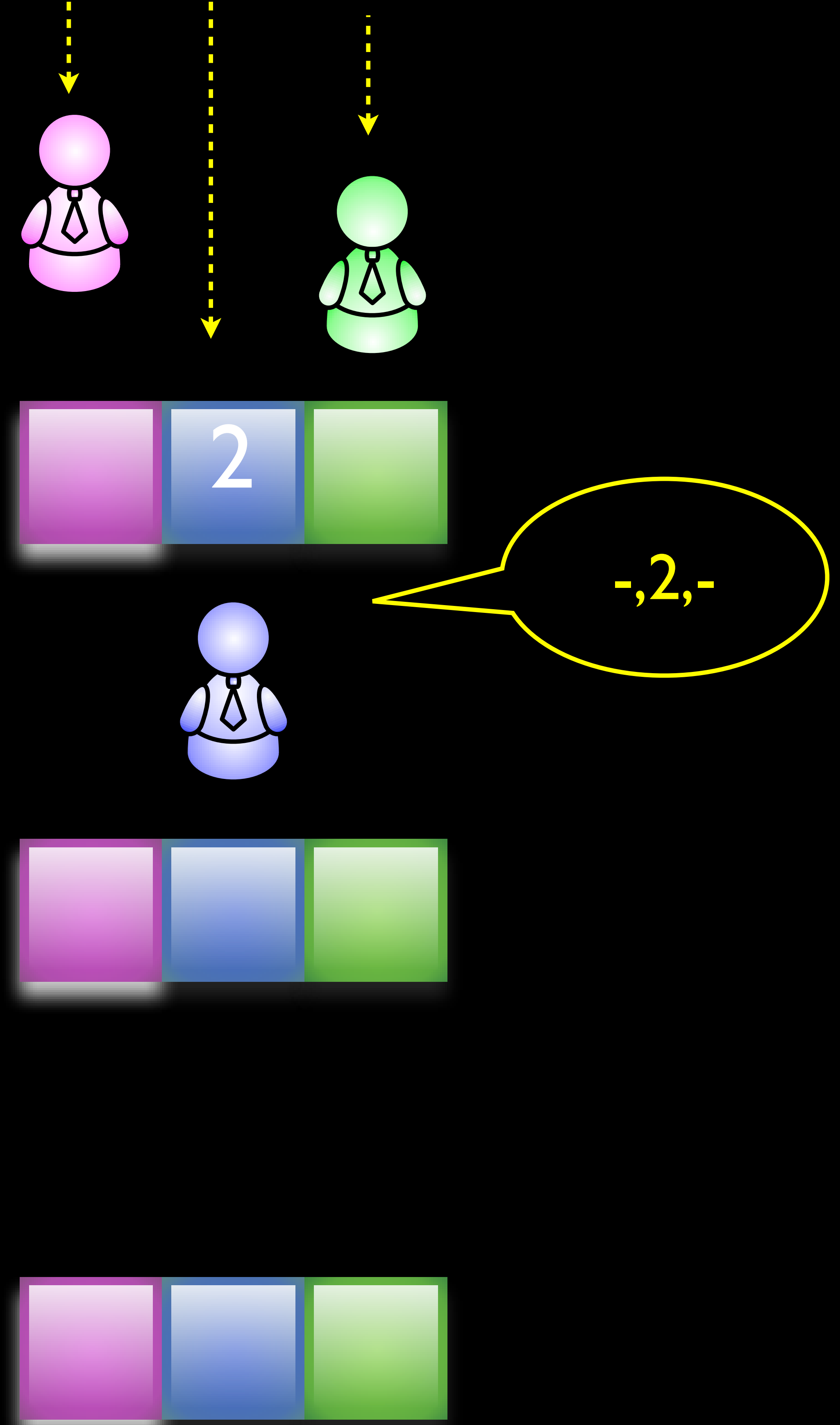
- arrive in arbitrary order
- last one sees all



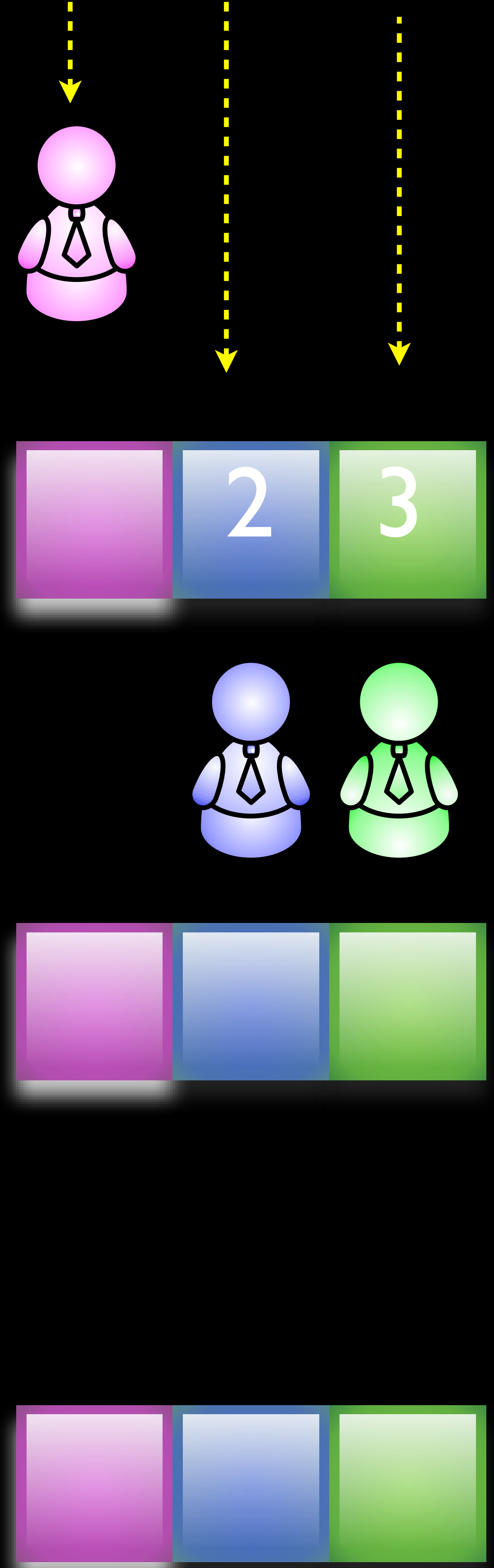
- arrive in arbitrary order
- last one sees all



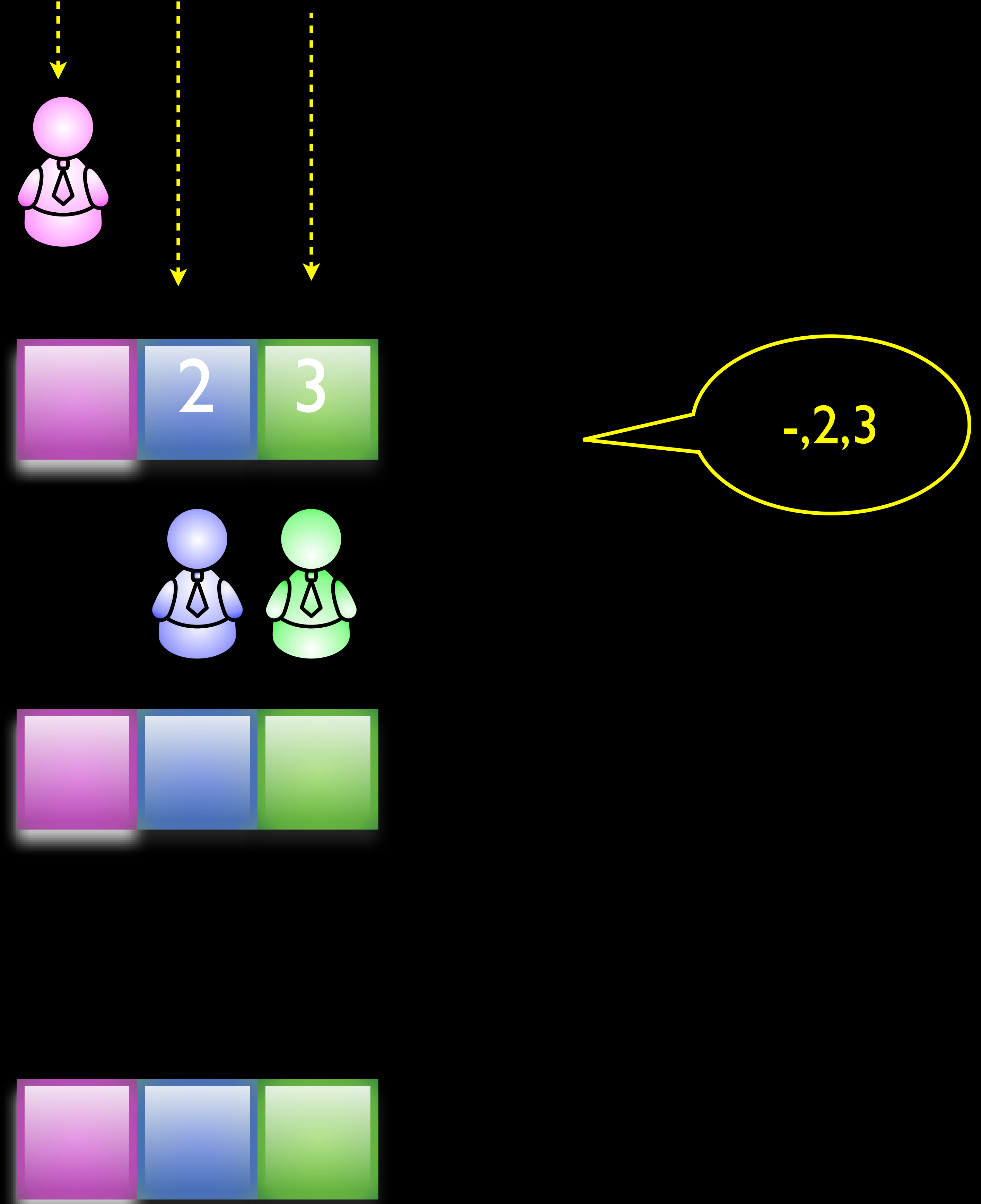
- arrive in arbitrary order
- last one sees all



- arrive in arbitrary order
- last one sees all

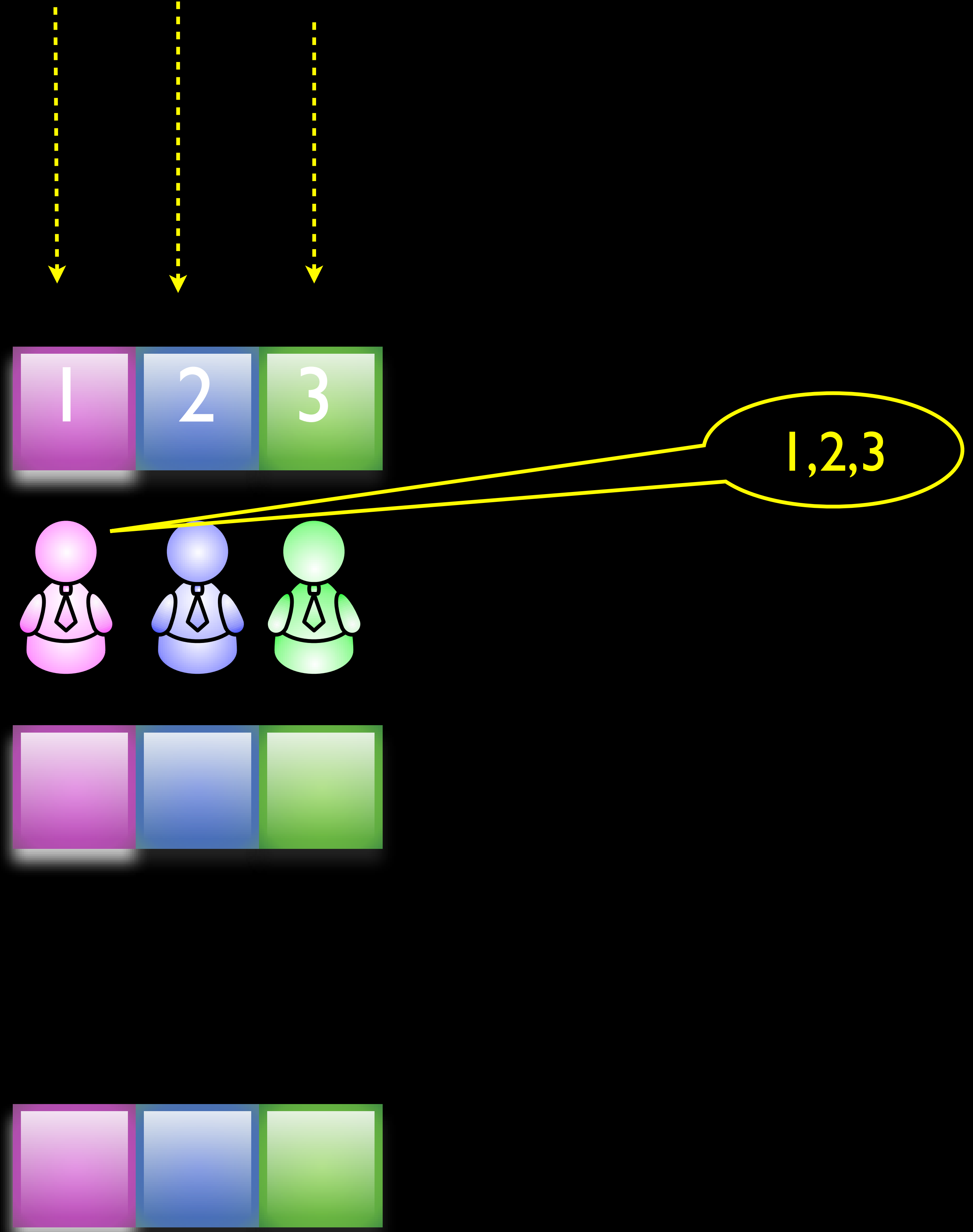


- arrive in arbitrary order
- last one sees all

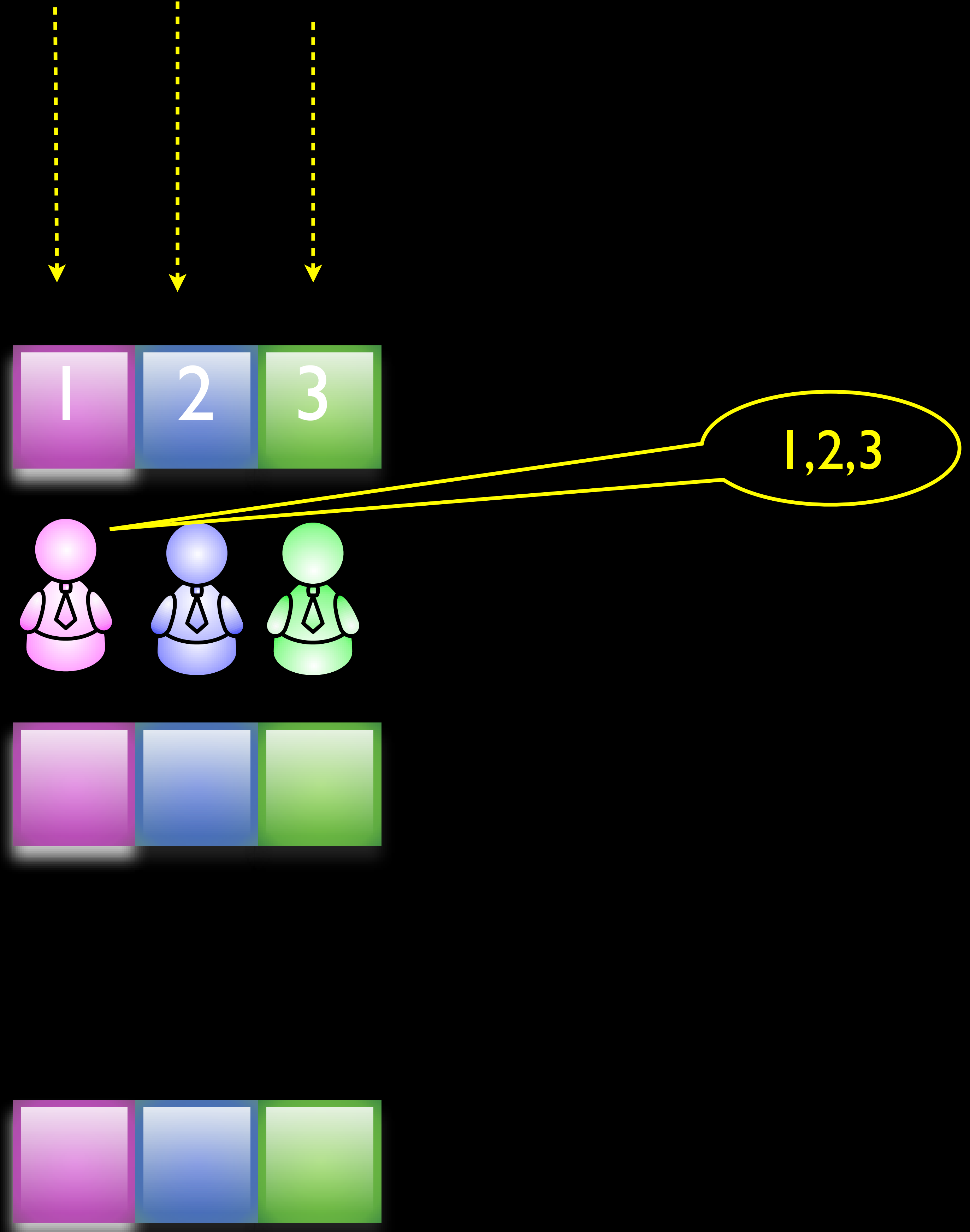


- arrive in arbitrary order
- last one sees all

- arrive in arbitrary order
- last one sees all

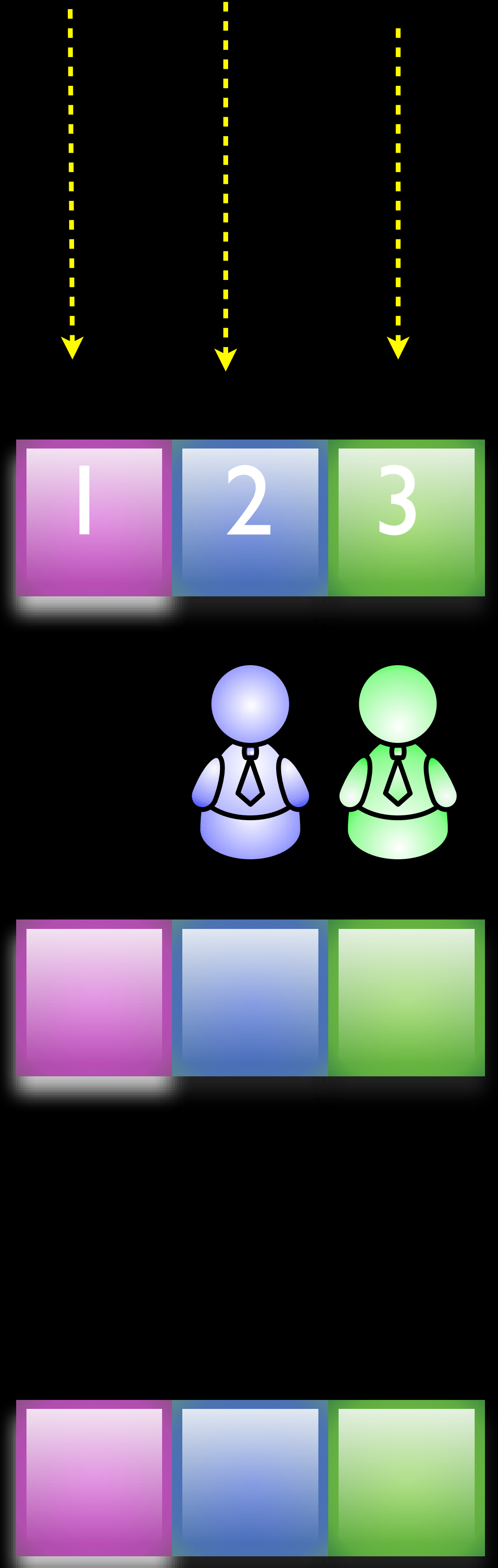


- arrive in arbitrary order
- last one sees all

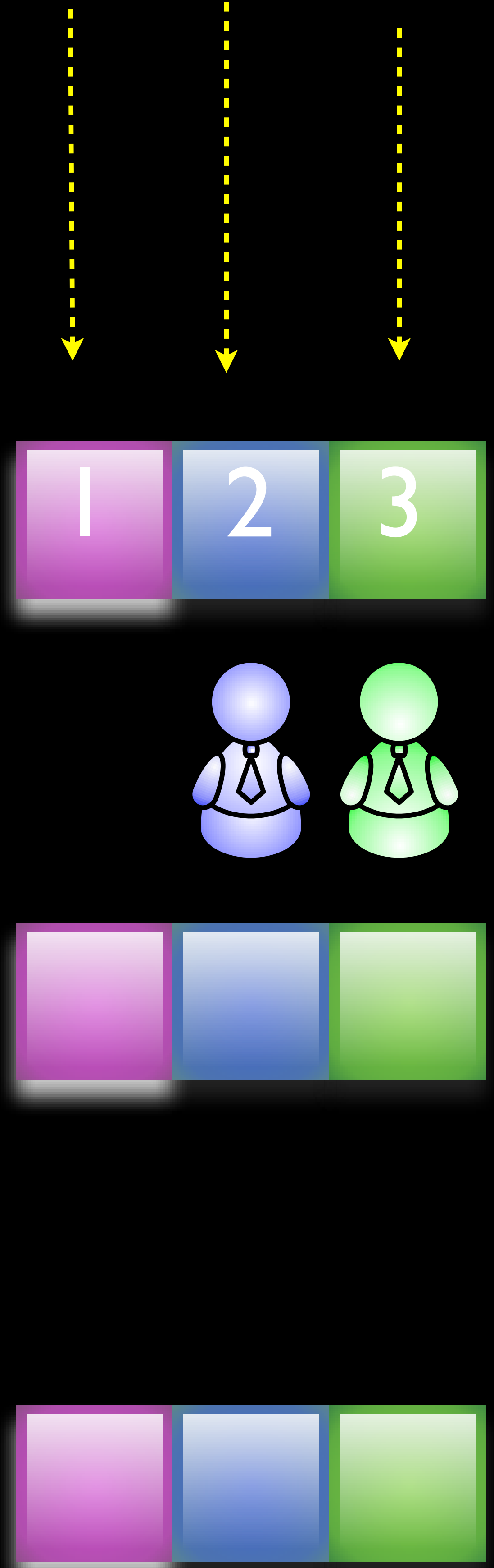


- arrive in arbitrary order
- last one sees all

returns 1,2,3

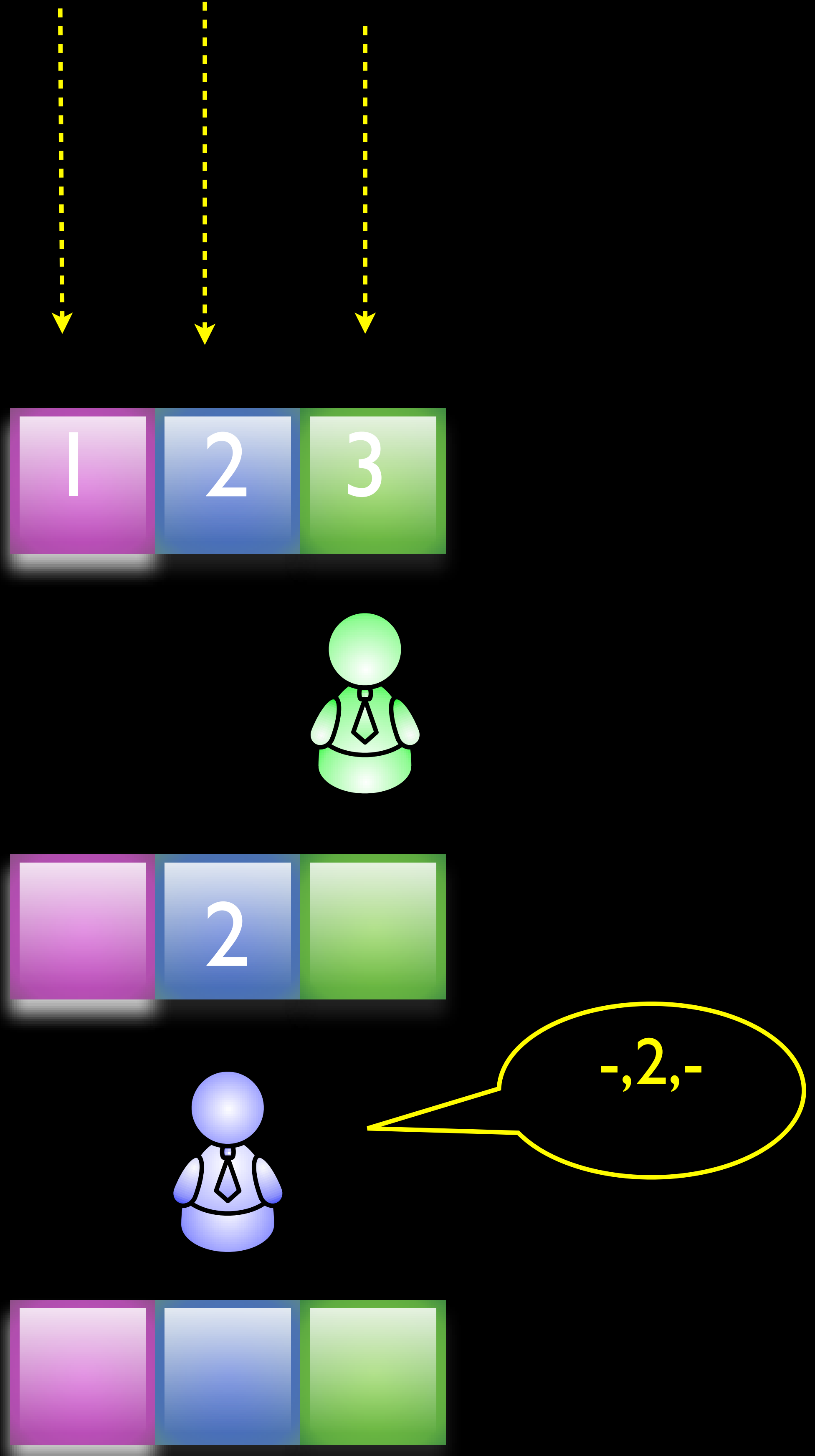


- remaining 2 go to next memory

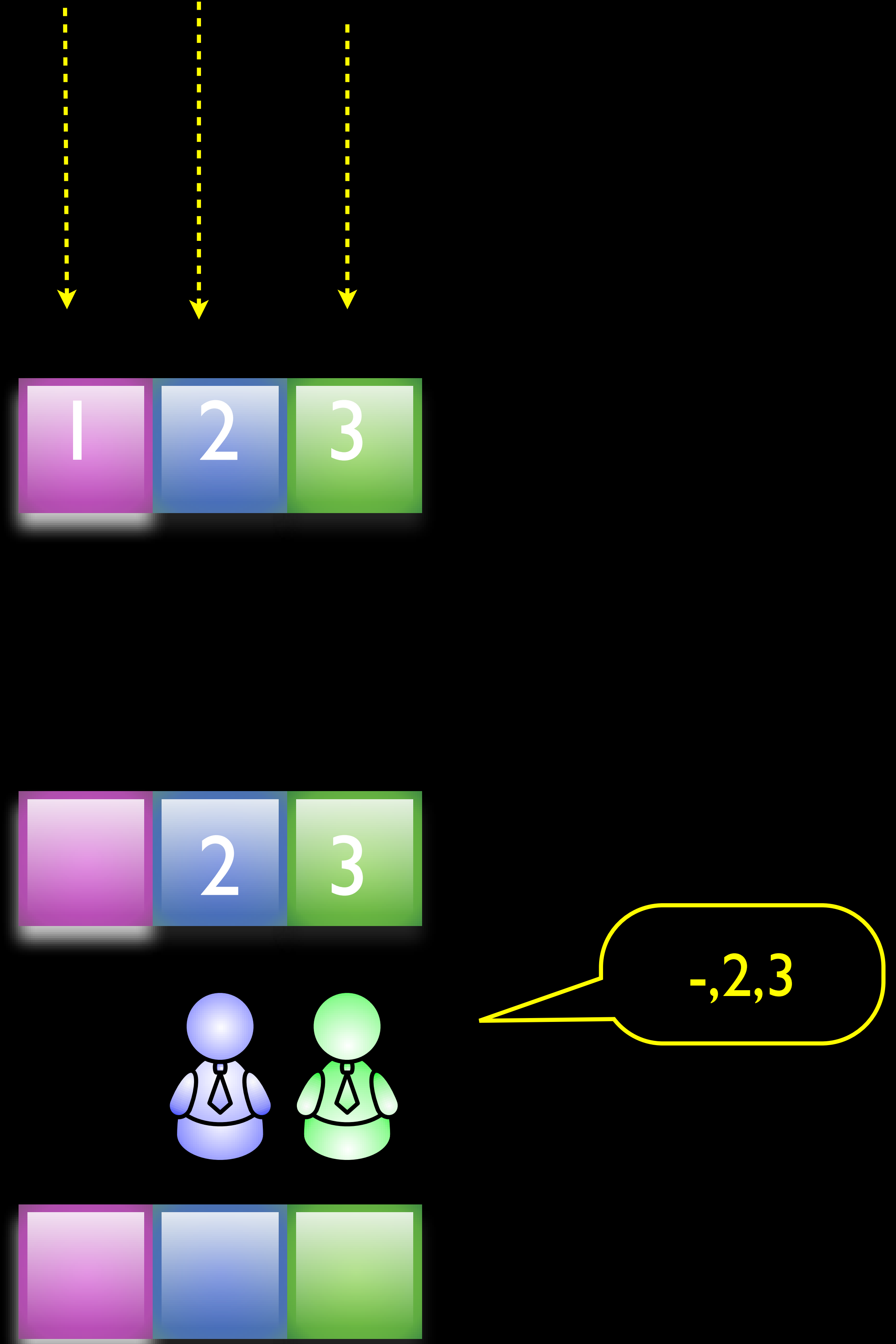


- remaining 2 go to next memory

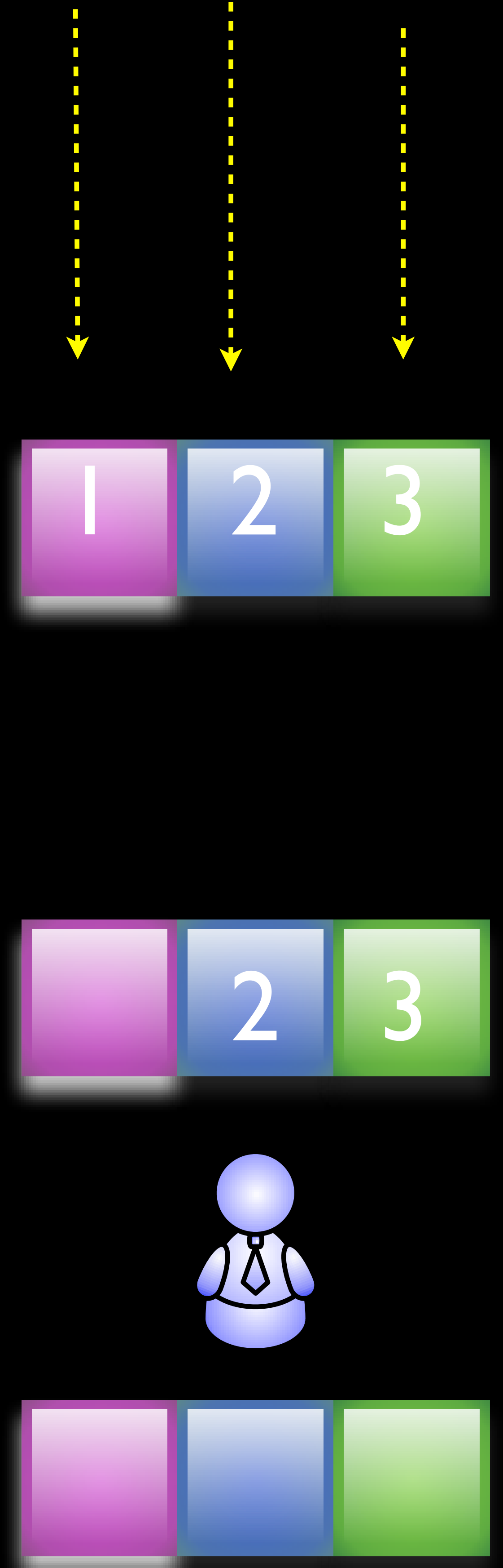
• remaining 2 go to next memory



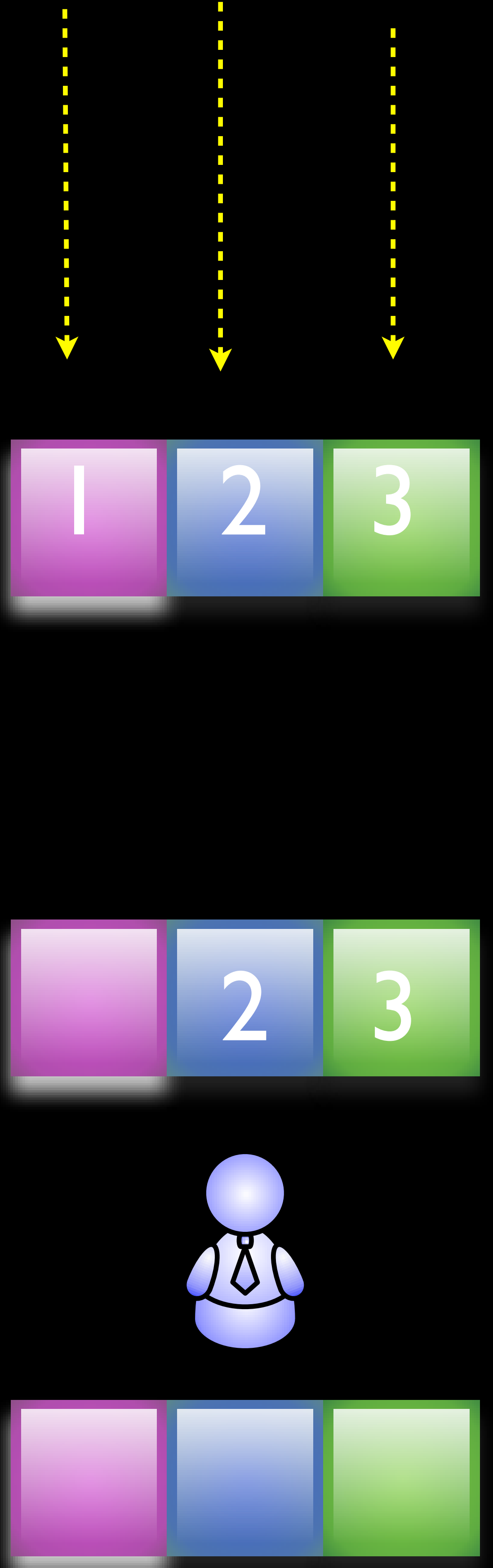
- 3rd one returns -,2,3

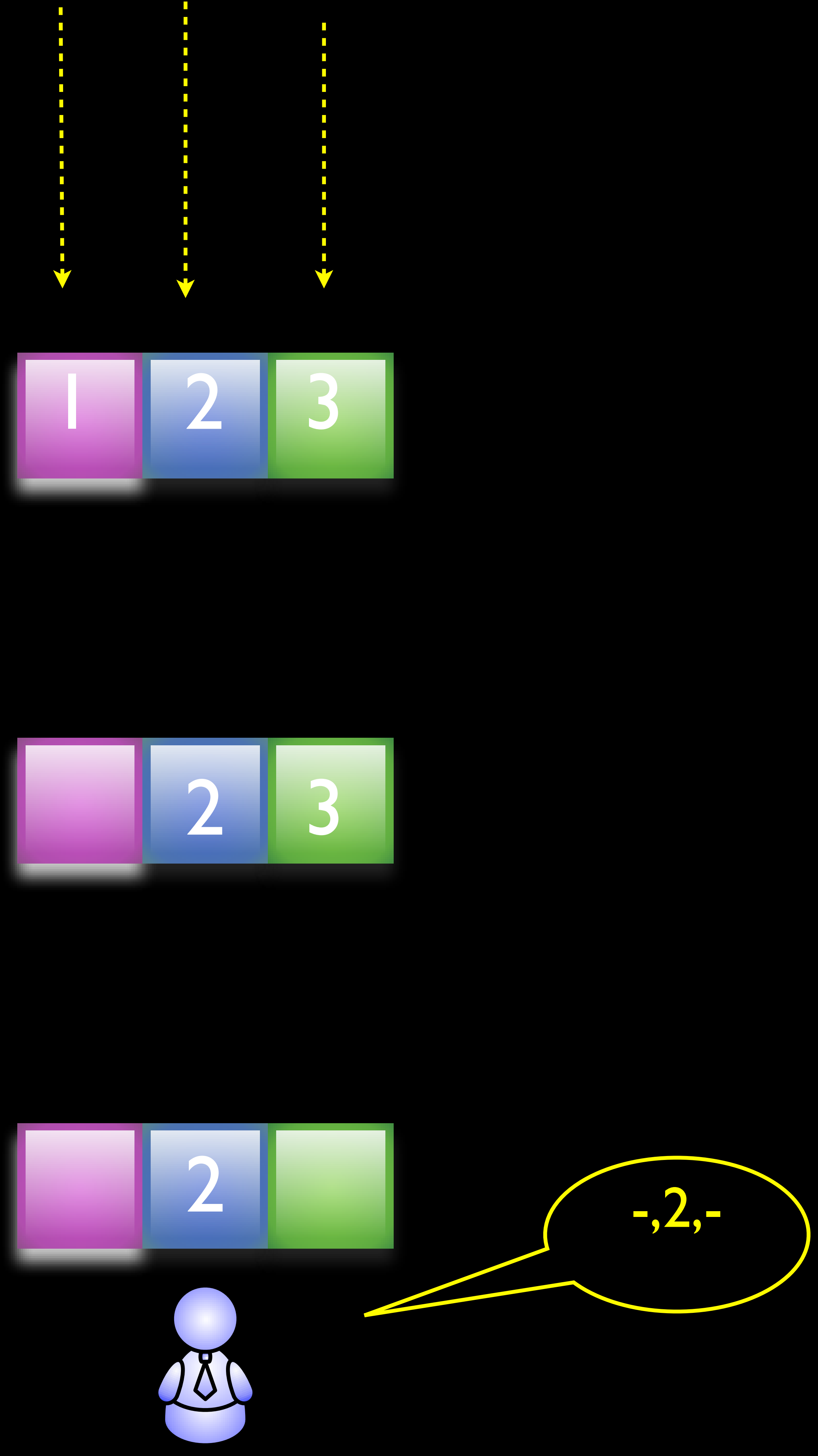


- 3rd one returns -,2,3

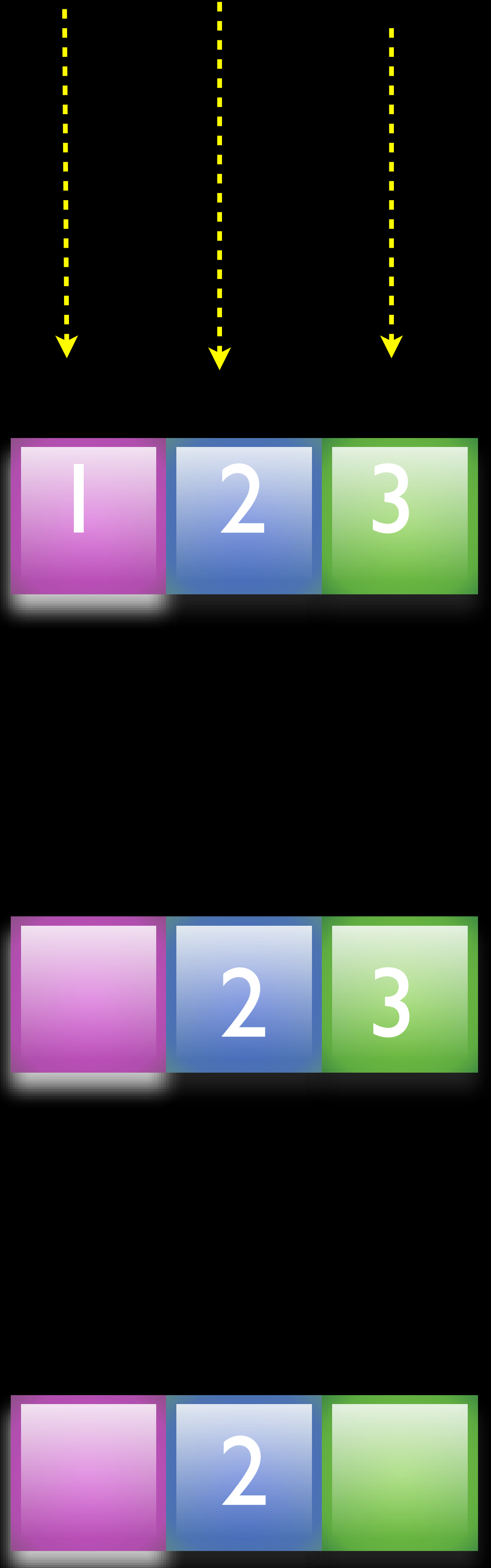


•2nd one goes alone



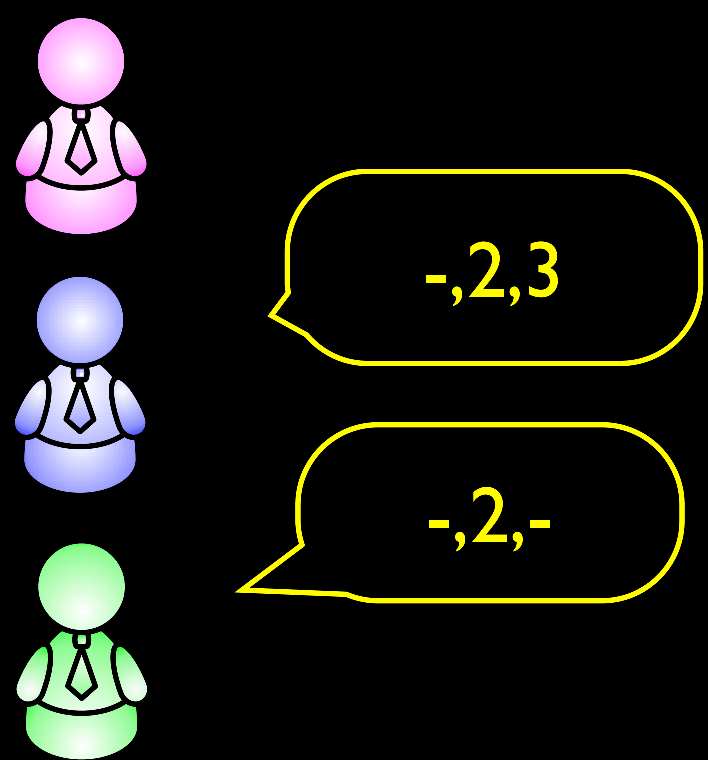


1
2
3
2
3
2
-,2,-

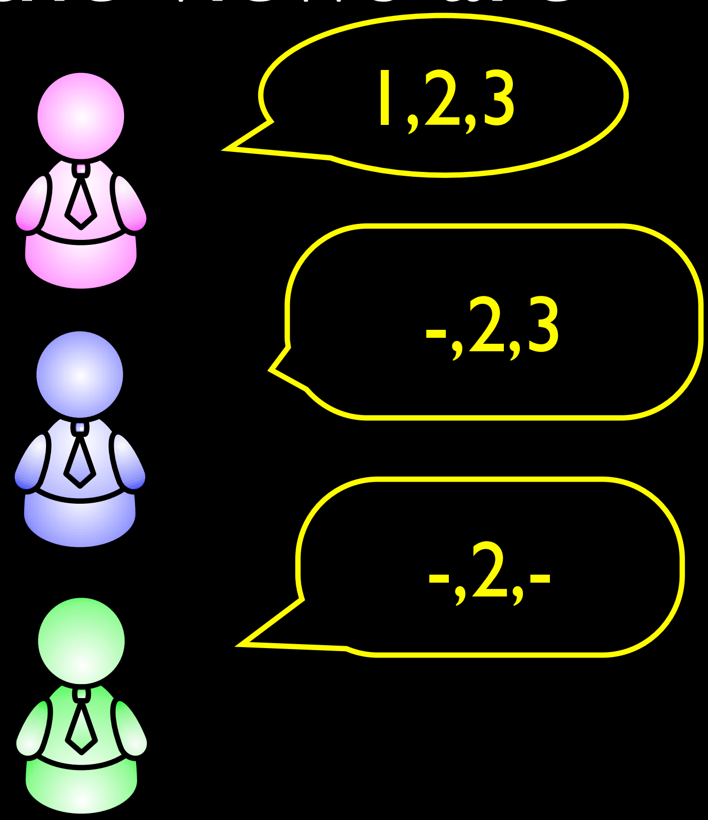


- returns -,2,-

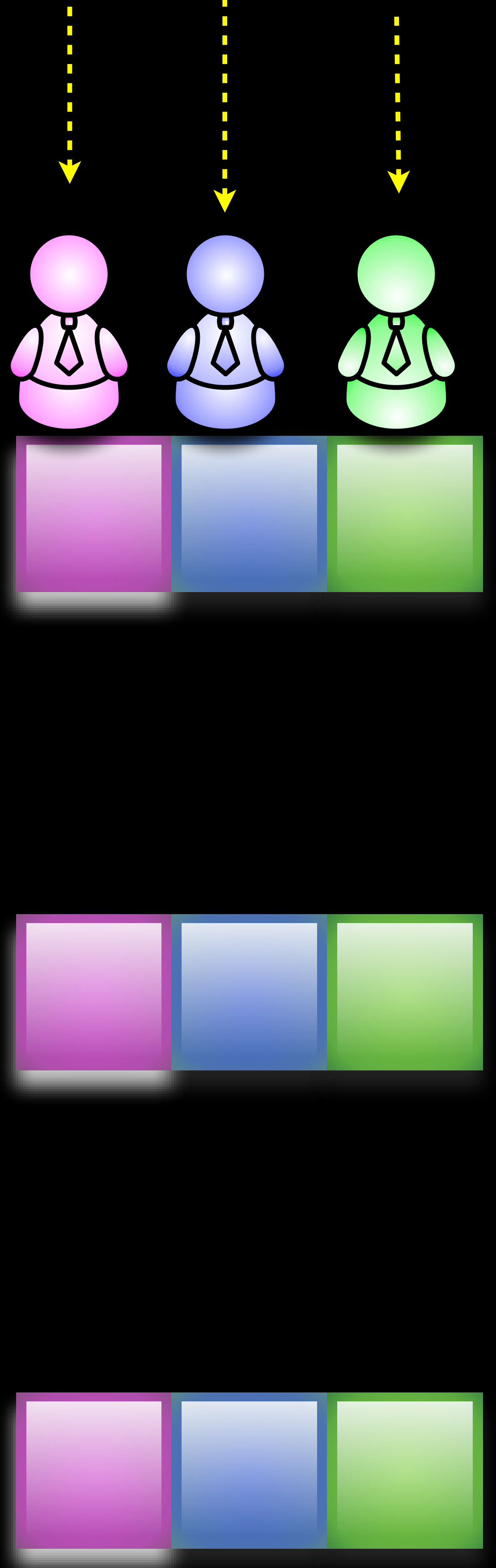
so in this run,
the views are



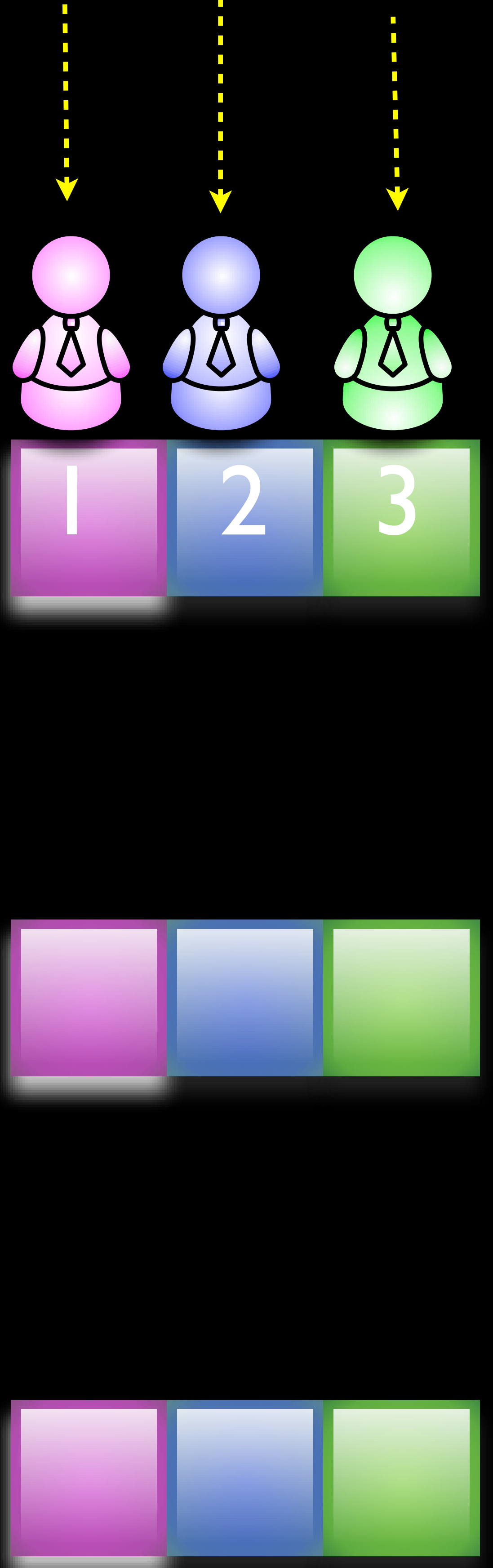
so in this run,
the views are



another run

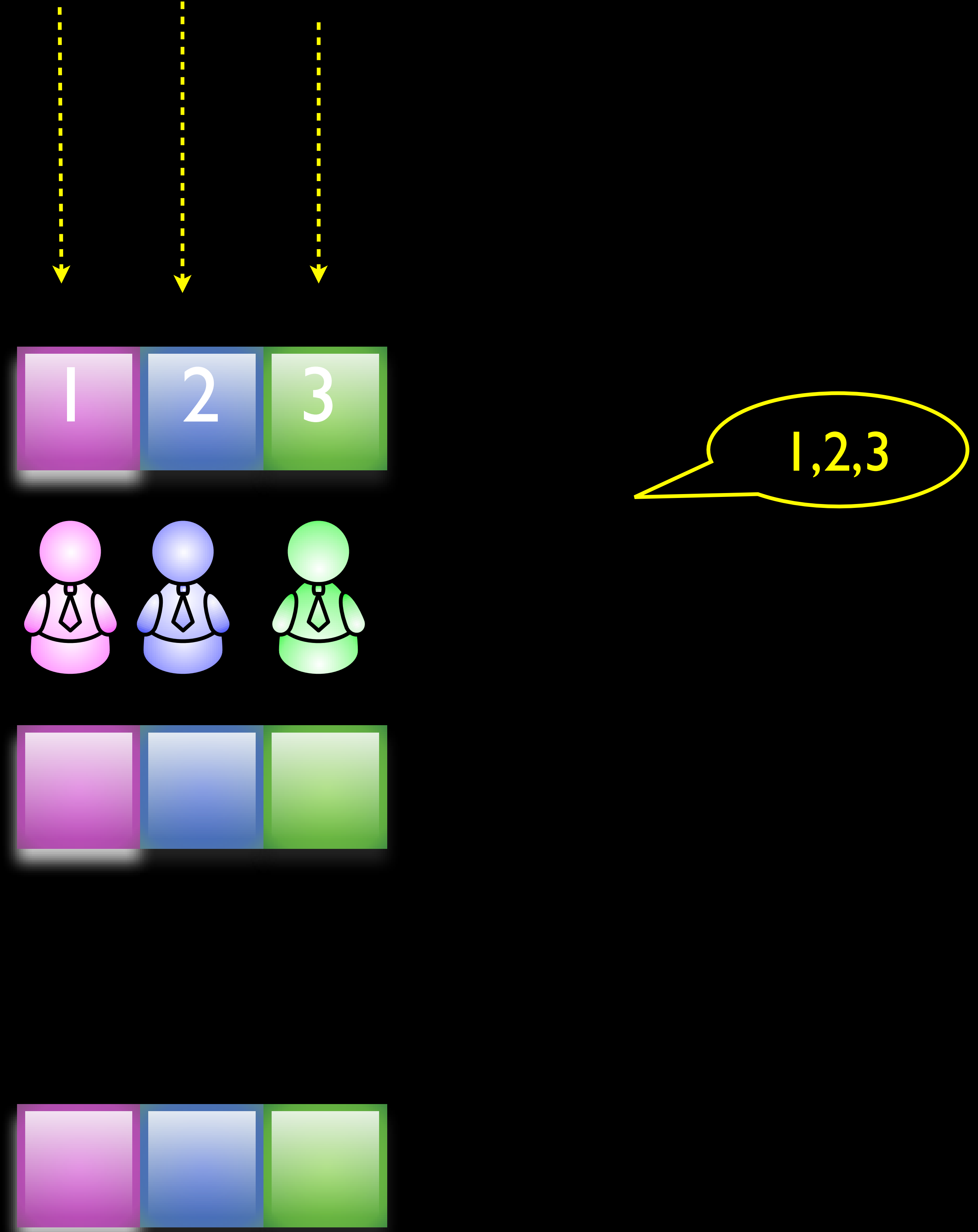


- arrive in arbitrary order



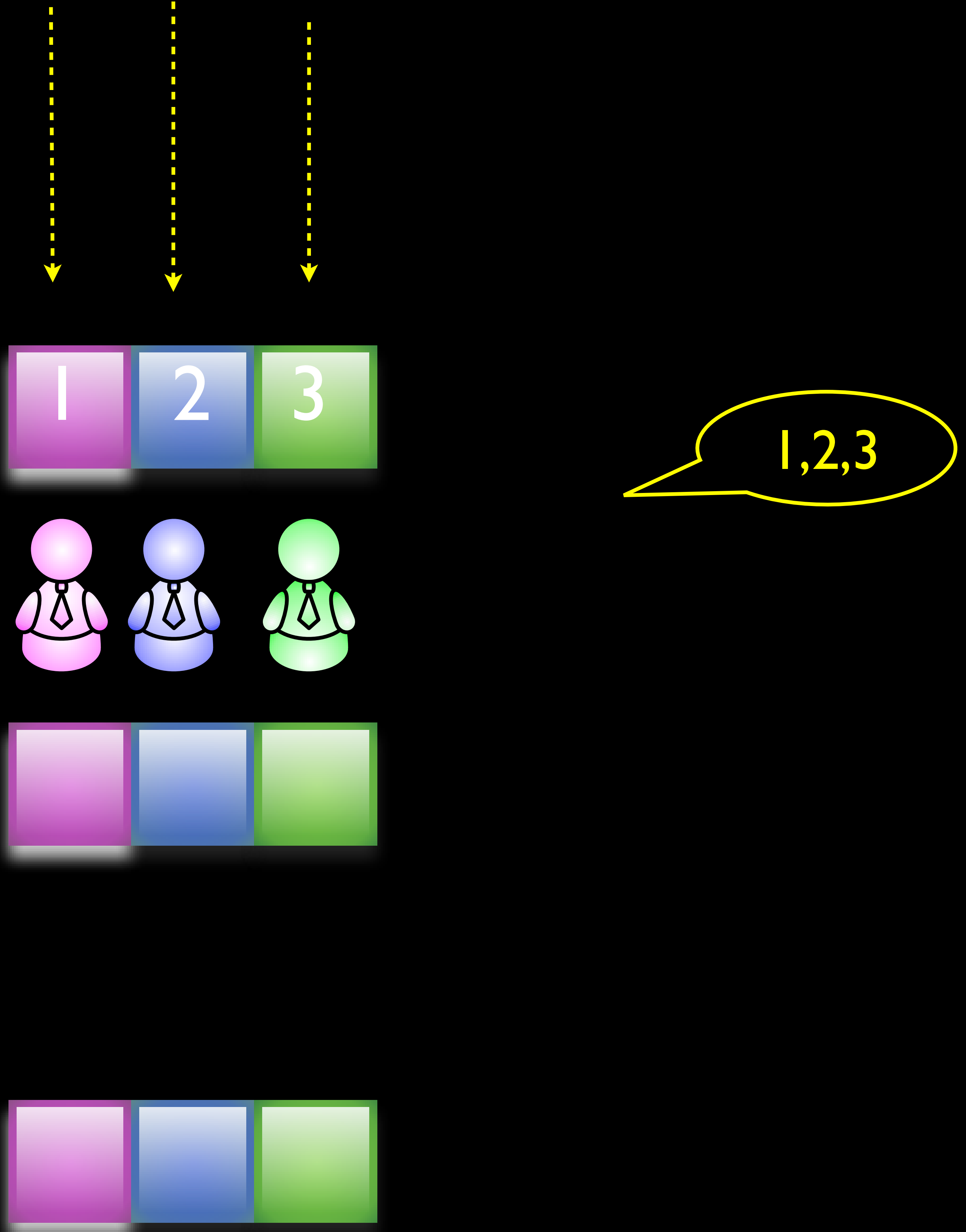
- all see all

- all see all



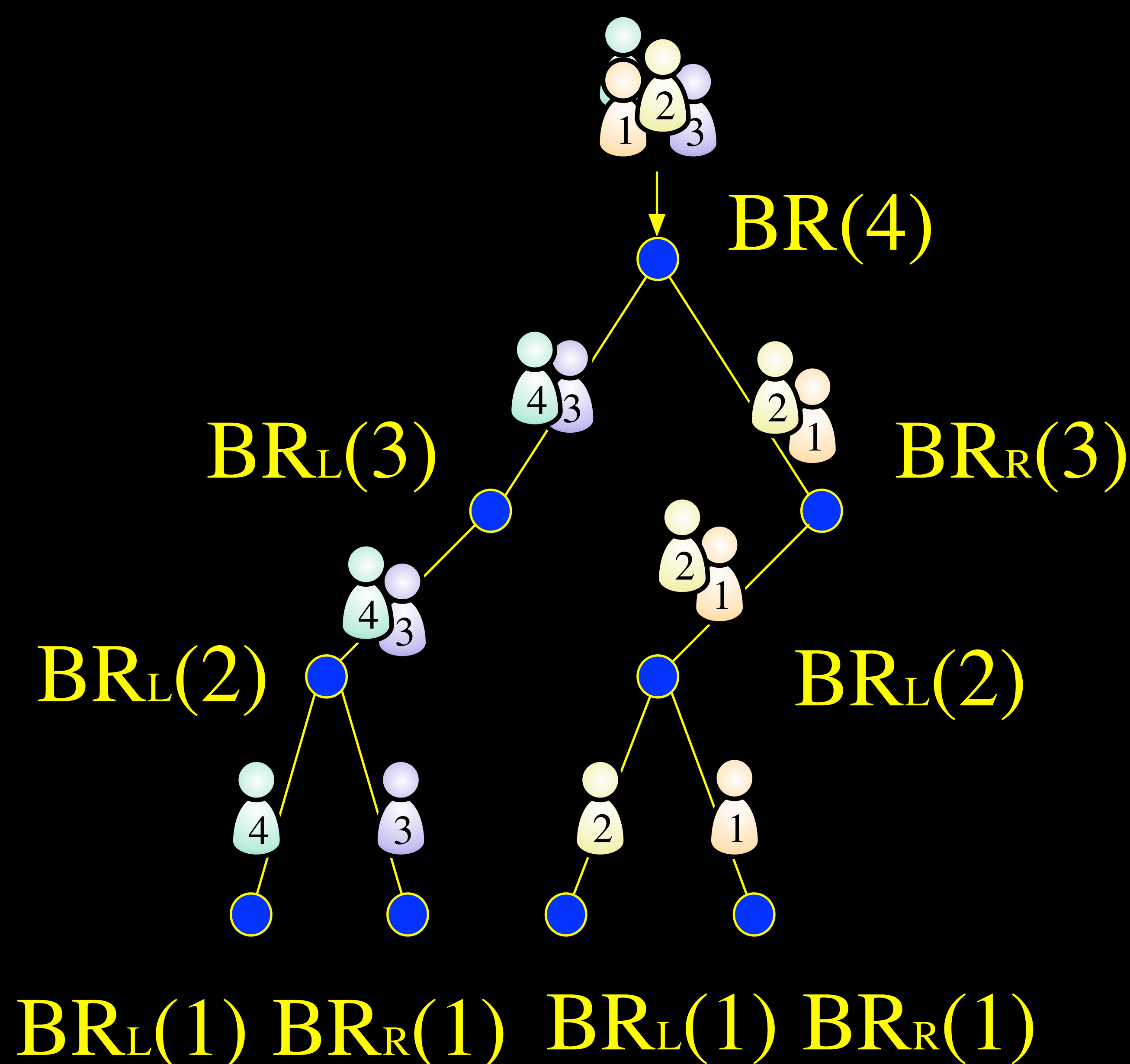
- and in this case, no recursive call,
- they all return with 1,2,3

- and in this case, no recursive call,
- they all return with 1,2,3



Renaming
and
binary branching recursion

Branching recursion



Renaming

Renaming

- Processes choose new names, as few as possible

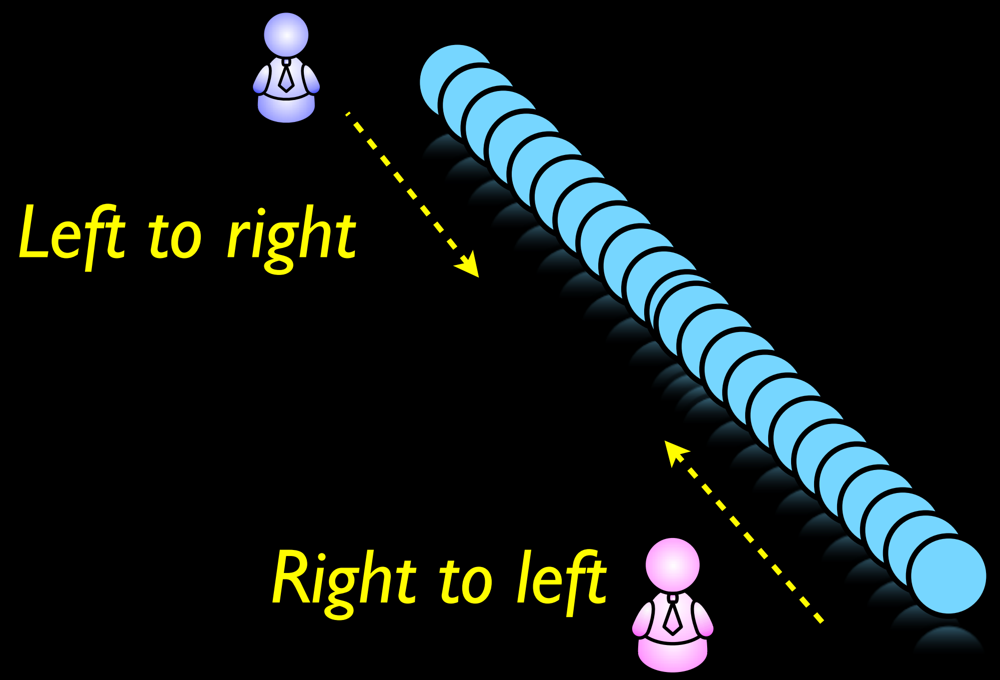
Renaming

- Processes choose new names, as few as possible
- There is a wait-free algorithm for *2n-1* names

Renaming

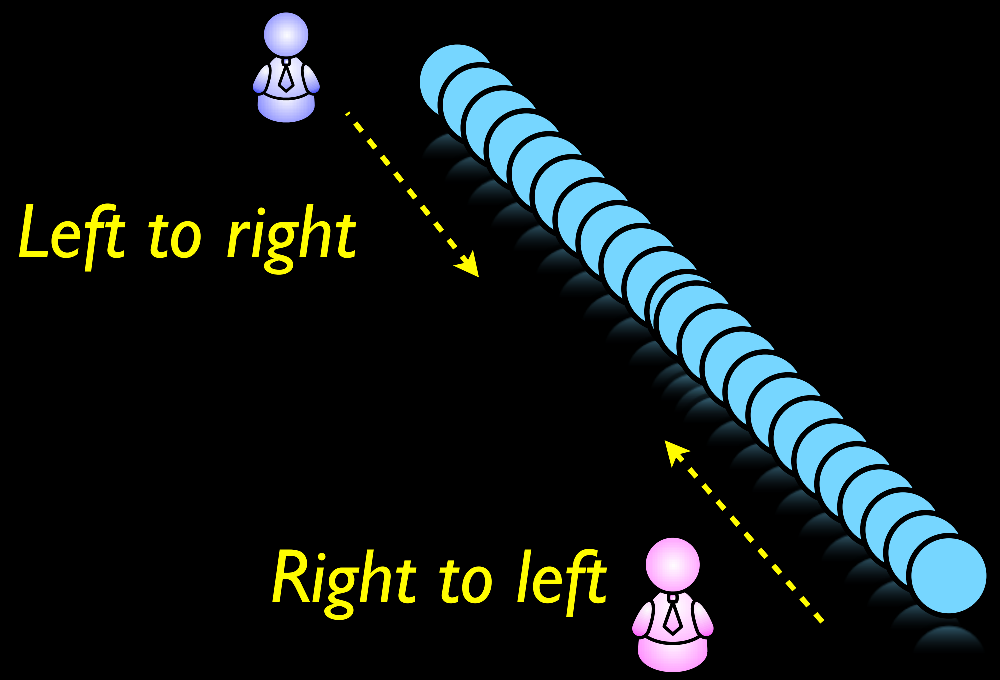
- Processes choose new names, as few as possible
- There is a wait-free algorithm for $2n-1$ names
- and impossible for fewer names (except in some exceptional cases)

Recursive renaming



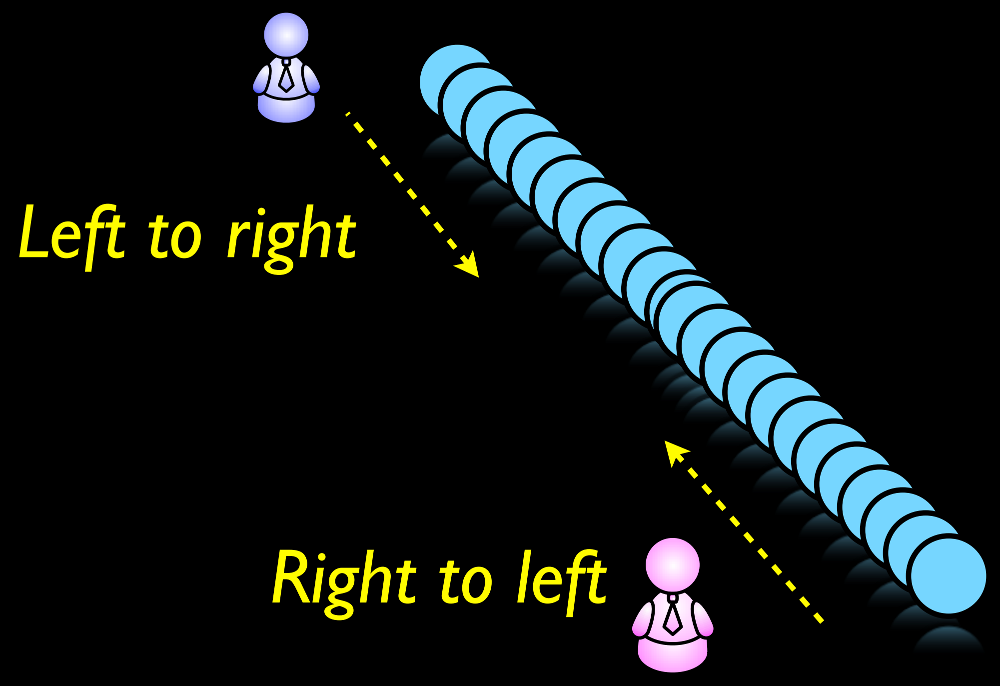
Recursive renaming

- Use weak splitter



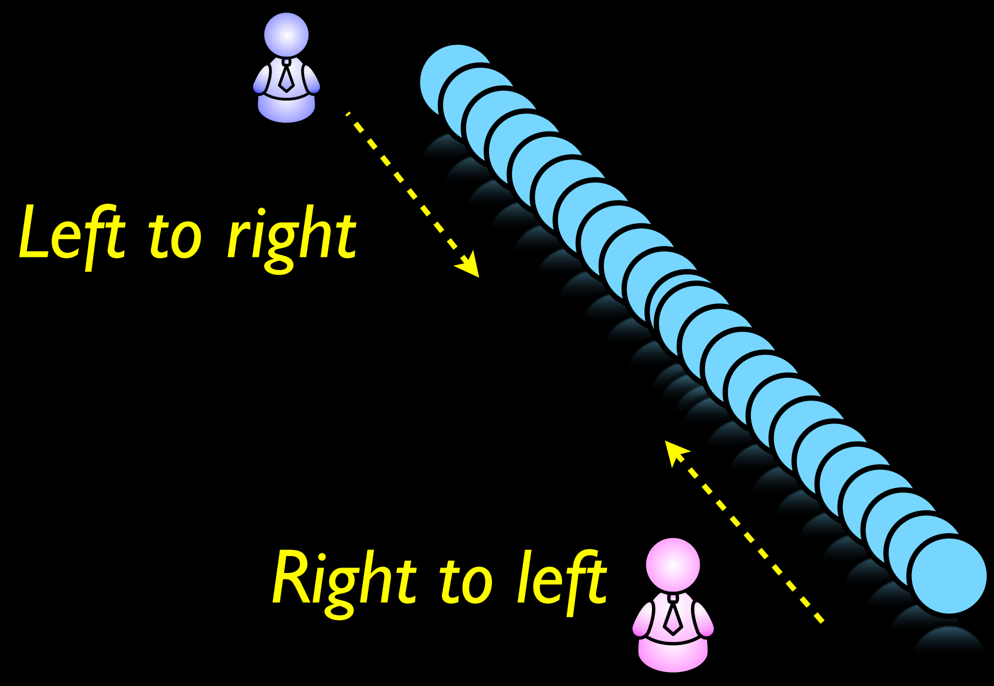
Recursive renaming

- Use weak splitter
- Some go left, and solve renaming recursively from left to right, the others do it from right to left



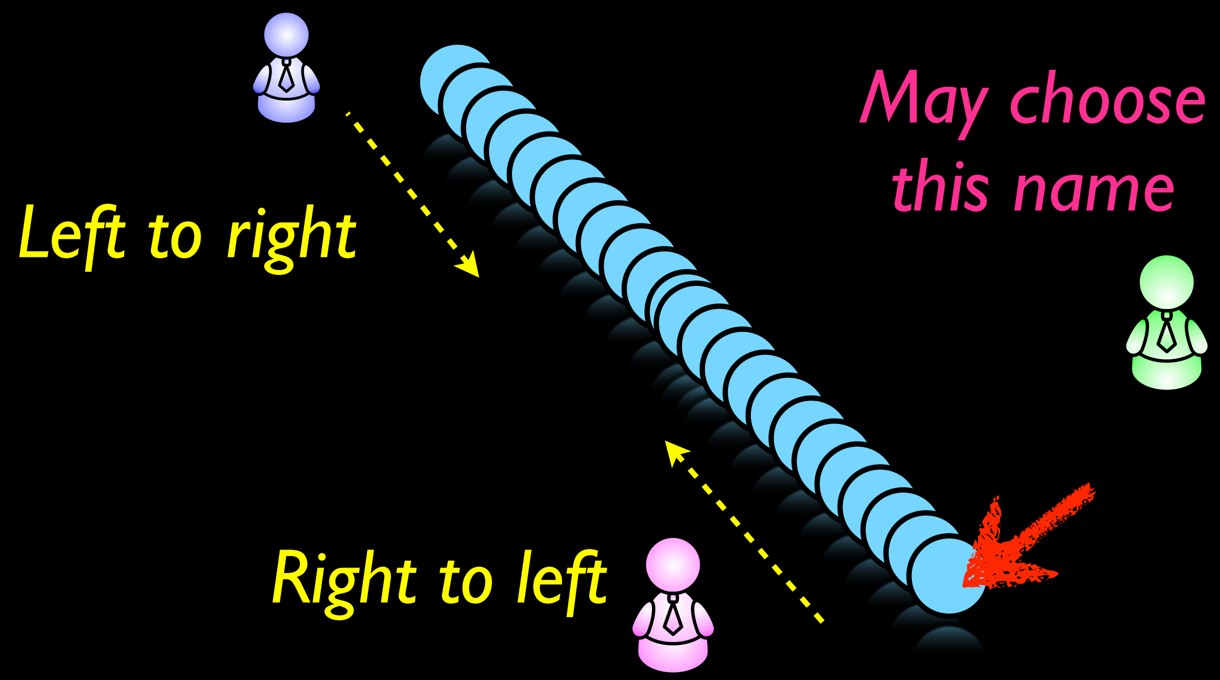
Recursive renaming

- Use weak splitter
- Some go left, and solve renaming recursively from left to right, the others do it from right to left
- one may stop with a new name



Recursive renaming

- Use weak splitter
- Some go left, and solve renaming recursively from left to right, the others do it from right to left
- one may stop with a new name



Renaming

- Algorithm Renaming $_{id}$($n, First, D$):
  - write id, read all registers
  - $Last = First + D(2n\text{-}2)$
  - if |read-set| = n, and id = max read-set then return $Last$
  - else return RenamingLR($n\text{-}1, Last\text{-}1, \text{-}D$)
- else
  - RenamingRL $_{id}$ ($n\text{-}1, First, D$)

Renaming

- Algorithm Renaming $_{id}$ $(n, First, D)$:
  - write id, read all registers
  - $Last = First + D(2n\text{-}2)$
  - if |read-set| = n, and id = max read-set then return $Last$
  - else return RenamingLR($n\text{-}1, Last\text{-}1, \text{-}D$)

at least one sees all

- else
  - RenamingRL $_{id}$ $(n\text{-}1, First, D)$

Renaming

- Algorithm Renaming$_{id}$($n, First, D$):
  - write id, read all registers
  - $Last = First + D(2n-2)$

    at least one sees all

  - if |read-set| = n, and id = max read-set then return $Last$

    at most $n-1$ call this

  - else return RenamingLR($n-1, Last-1, -D$)
- else
  - RenamingRL$_{id}$ ($n-1, First, D$)

Renaming

- Algorithm Renaming $_{id}$ ($n, First, D$):
  - write id, read all registers
  - $Last = First + D(2n-2)$
  - if |read-set| = n, and id = max read-set then return $Last$
  - else return RenamingLR($n-1, Last-1, -D$)
- else
  - RenamingRL $_{id}$ ($n-1, First, D$)

at least one sees all

at most $n-1$ call this

at most $n-1$ call this

Recursive algorithms facilitate impossibility proofs

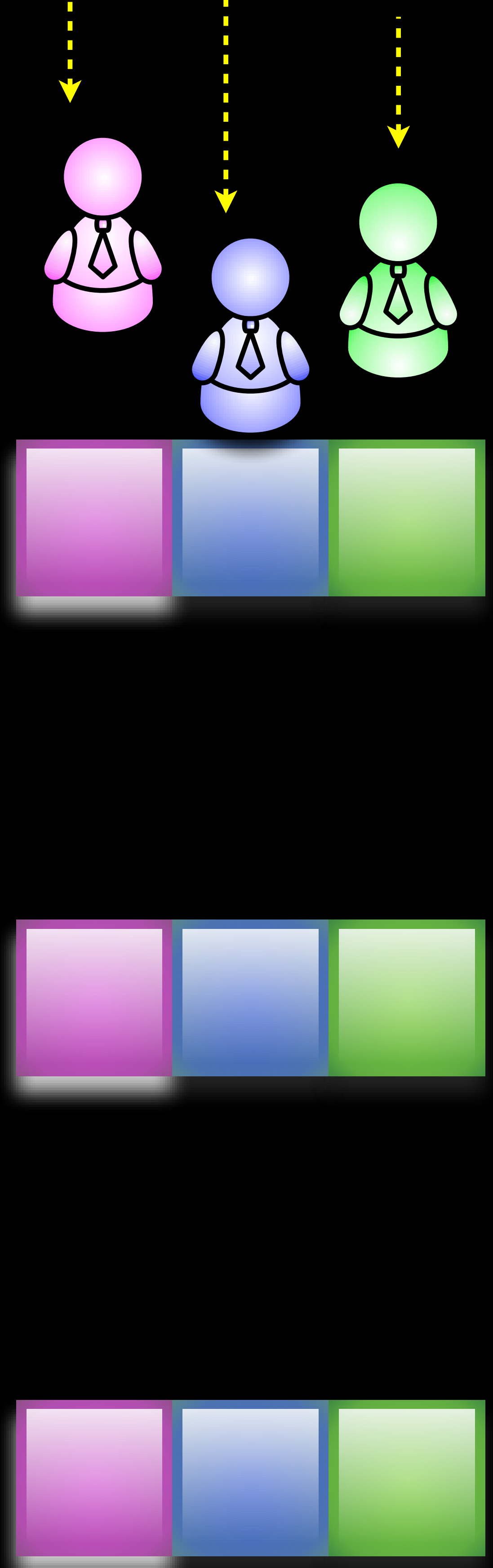
Emma Louise Jones

Recursive $\Rightarrow$ iterated

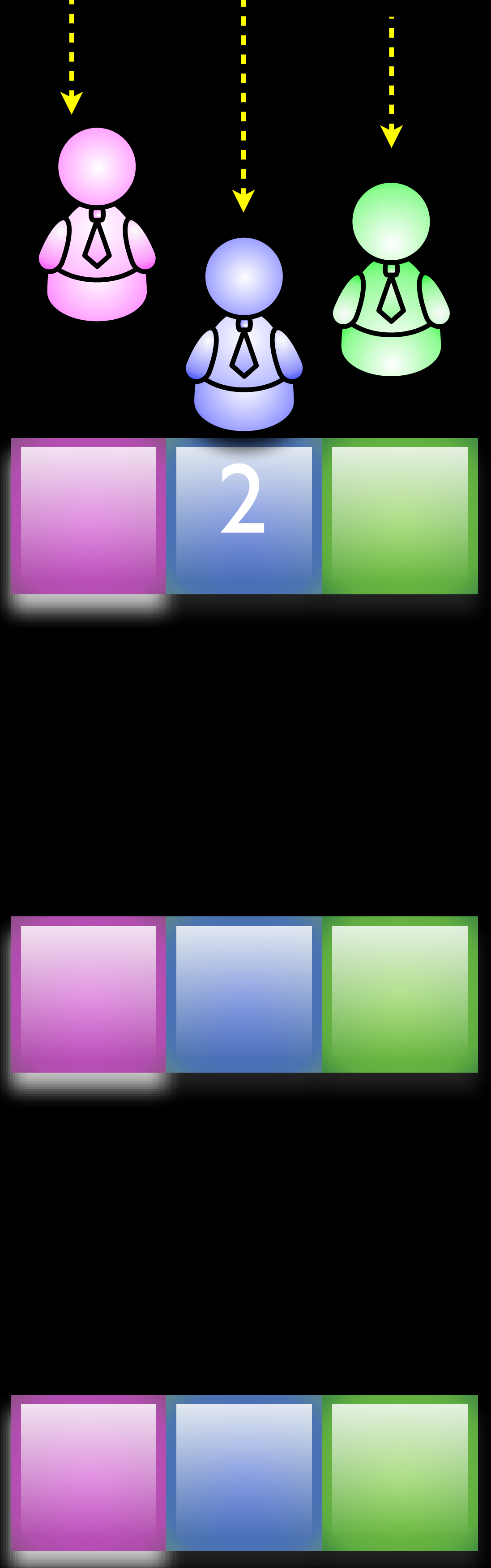
- when we unfold the recursion, we get an iterated run
- because each recursive call works with a fresh memory

every copy is new

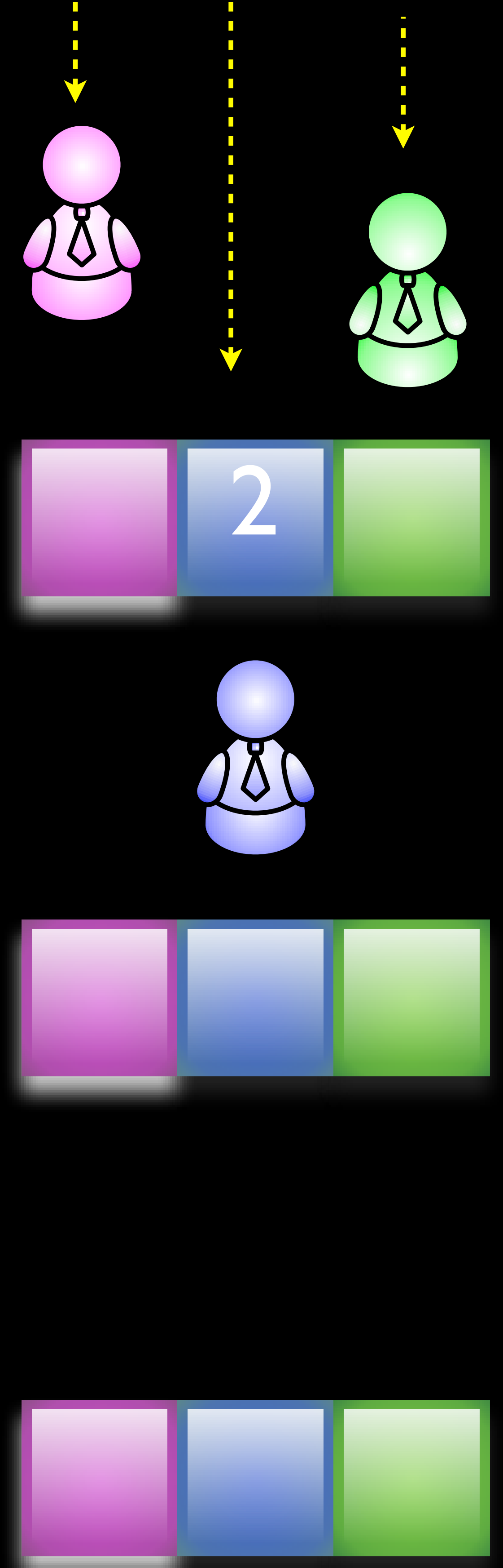
- arrive in arbitrary order
- last one sees all



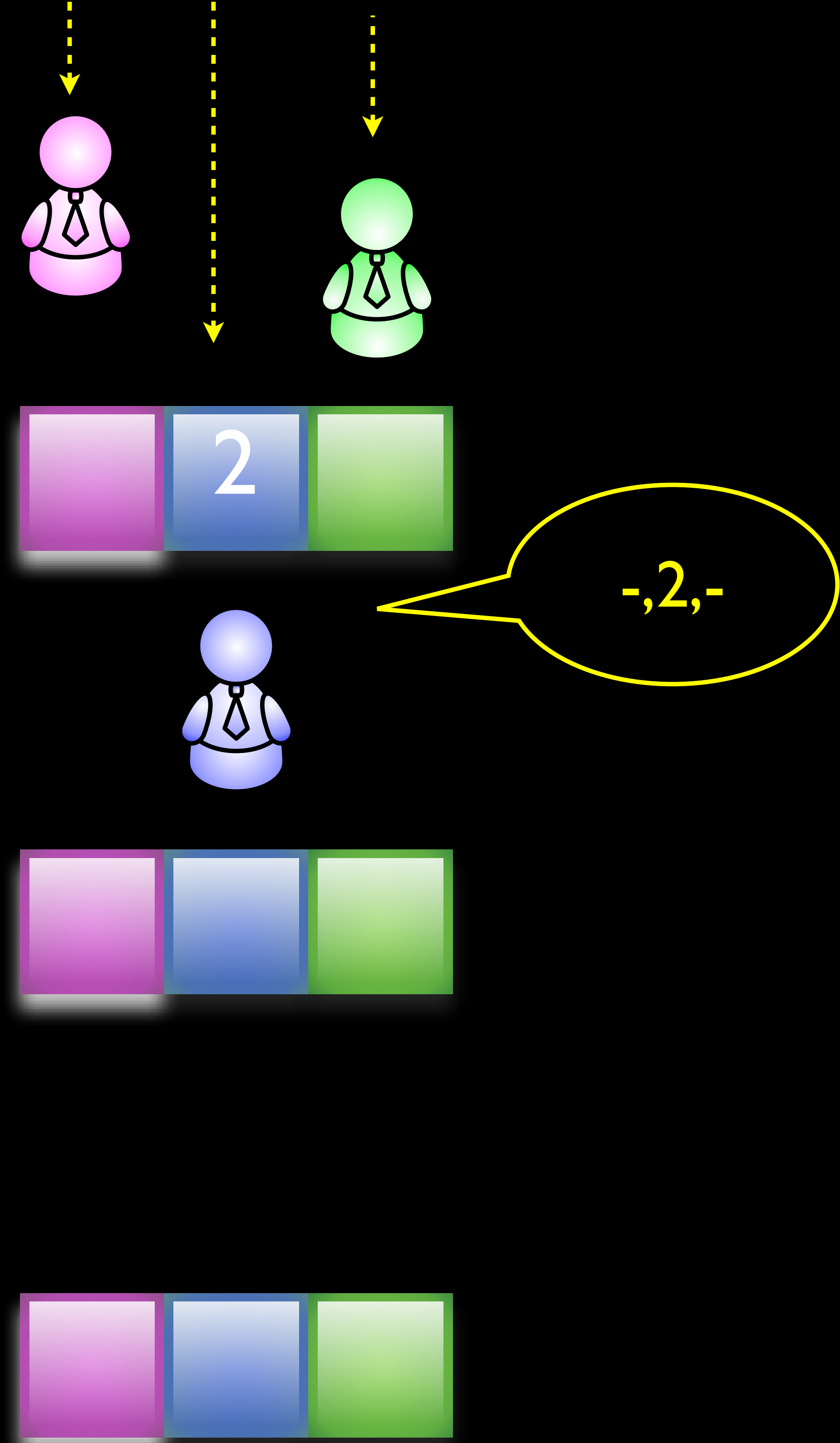
- arrive in arbitrary order
- last one sees all



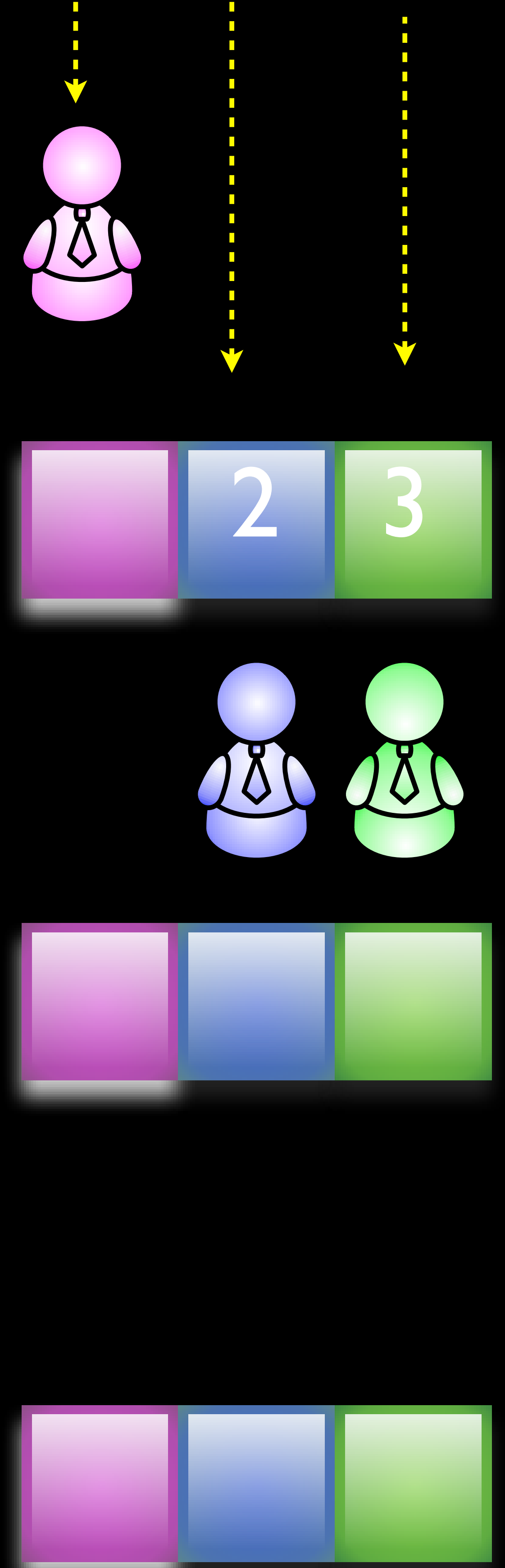
- arrive in arbitrary order
- last one sees all



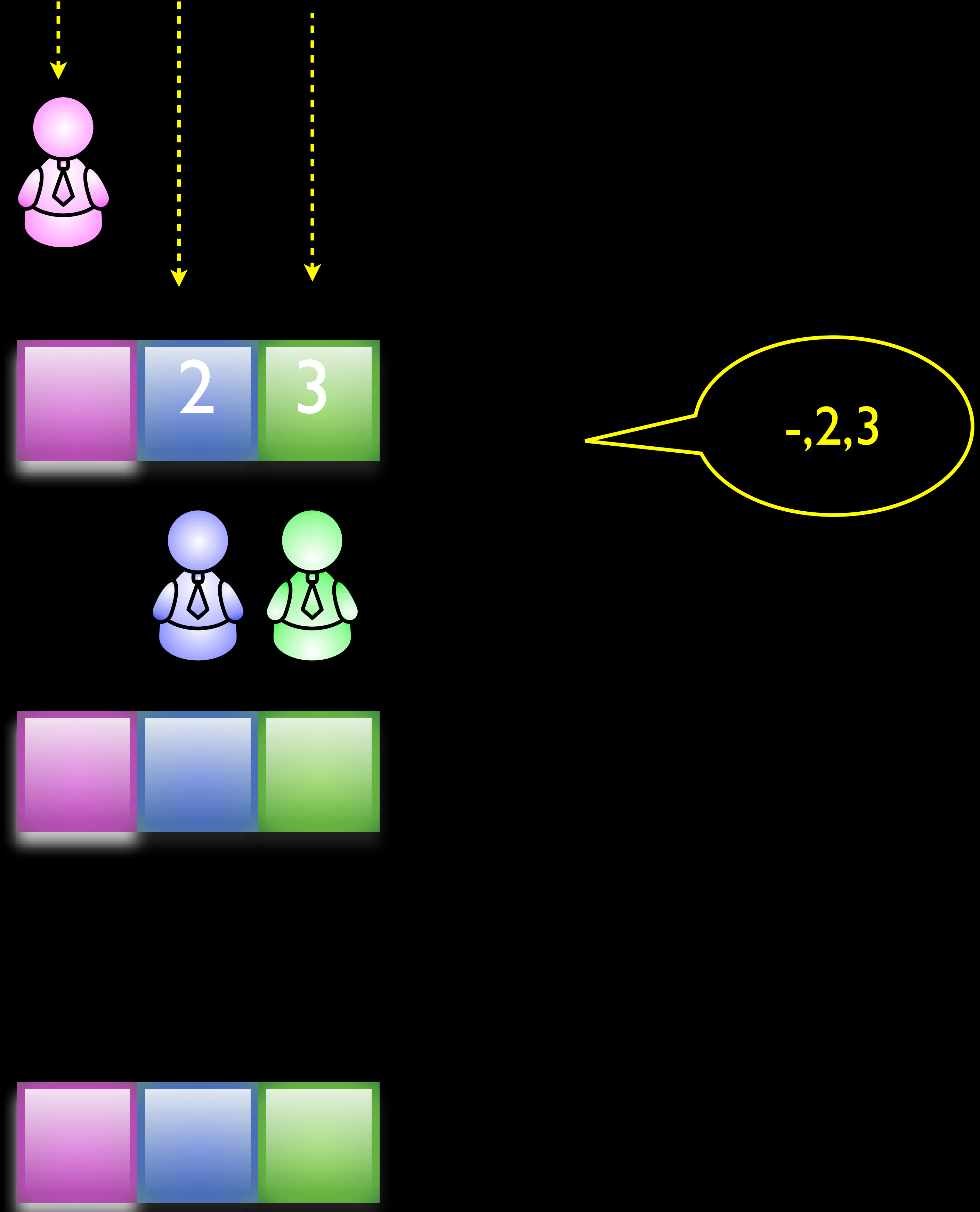
- arrive in arbitrary order
- last one sees all



- arrive in arbitrary order
- last one sees all

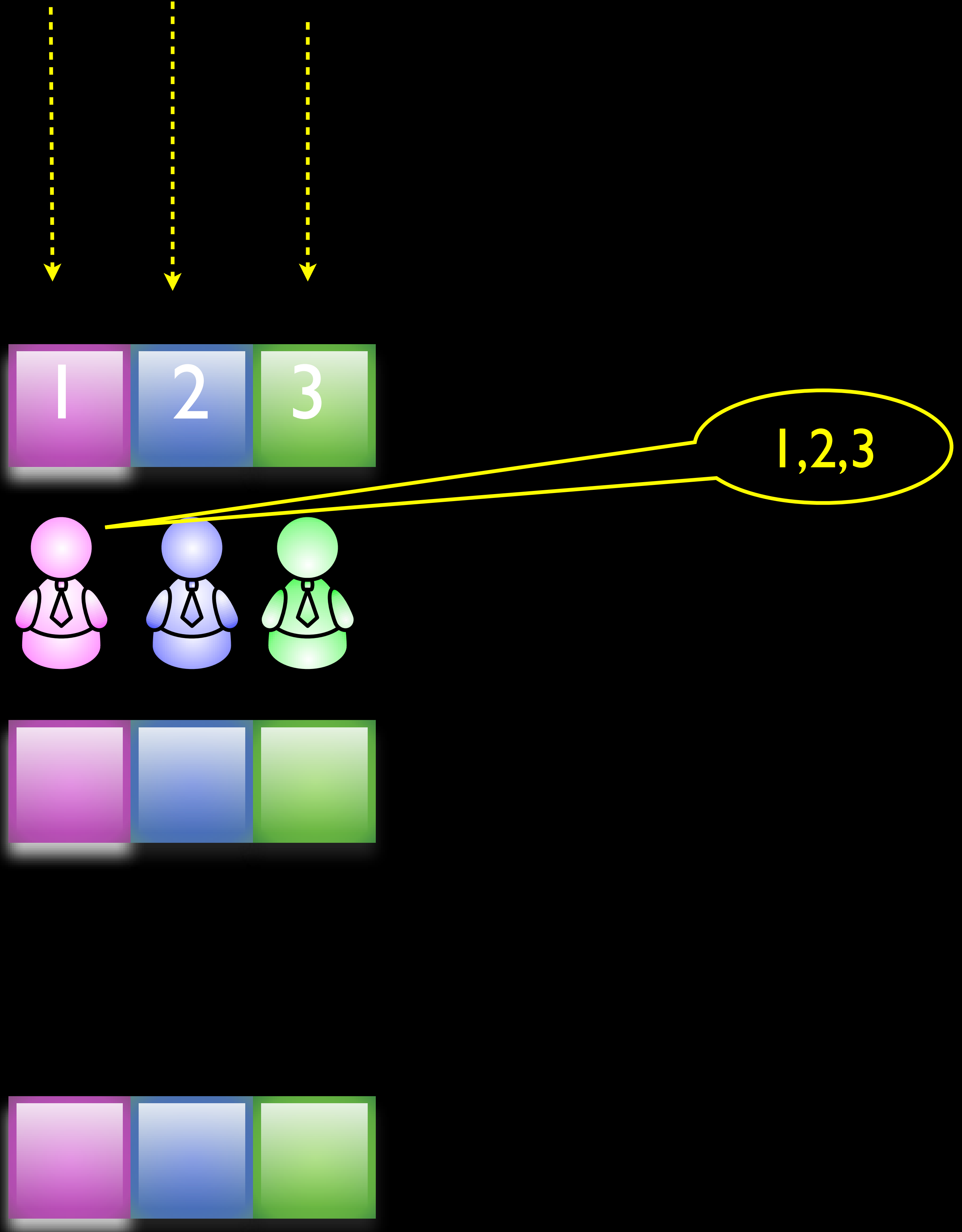


- arrive in arbitrary order
- last one sees all

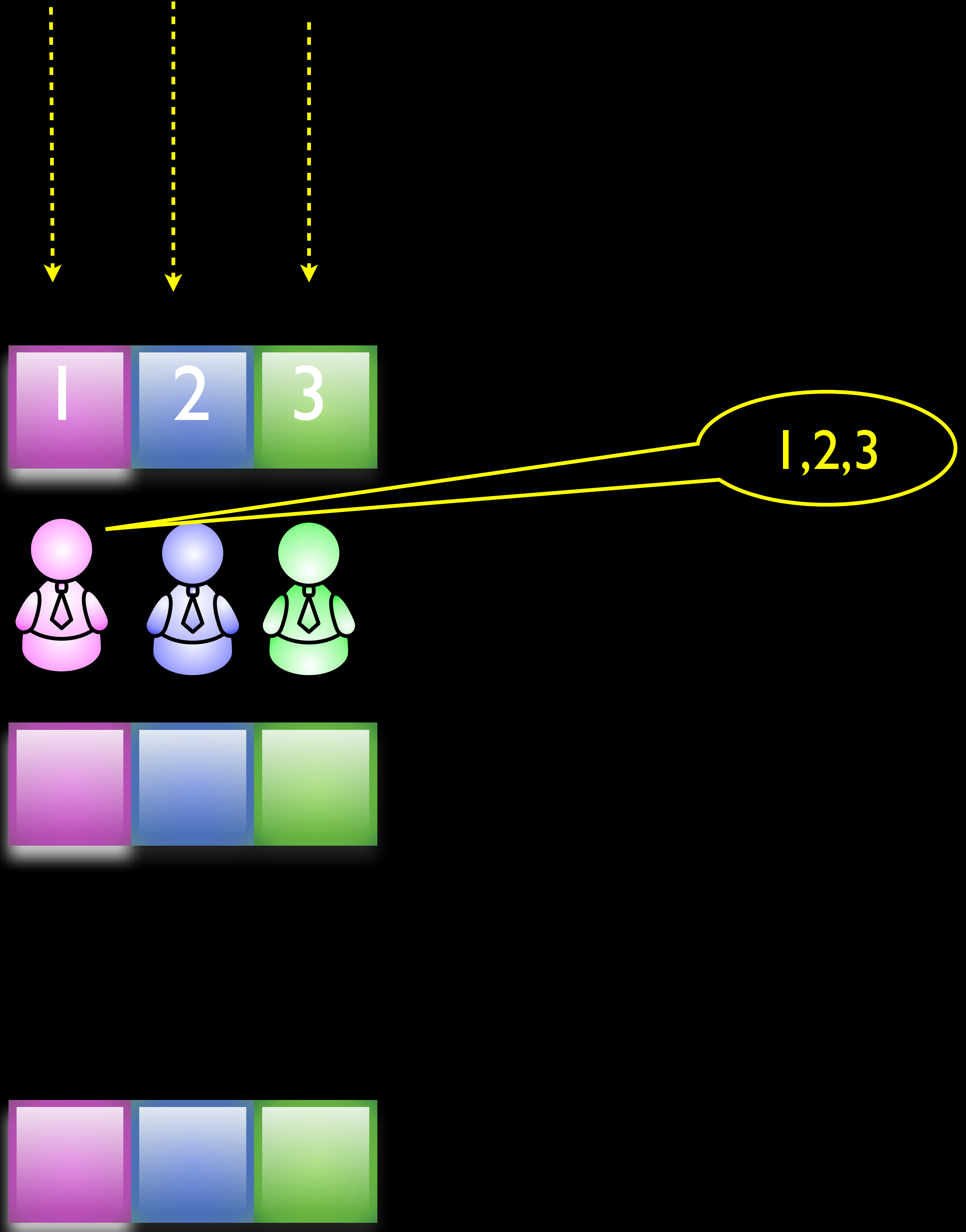


- arrive in arbitrary order
- last one sees all

- arrive in arbitrary order
- last one sees all

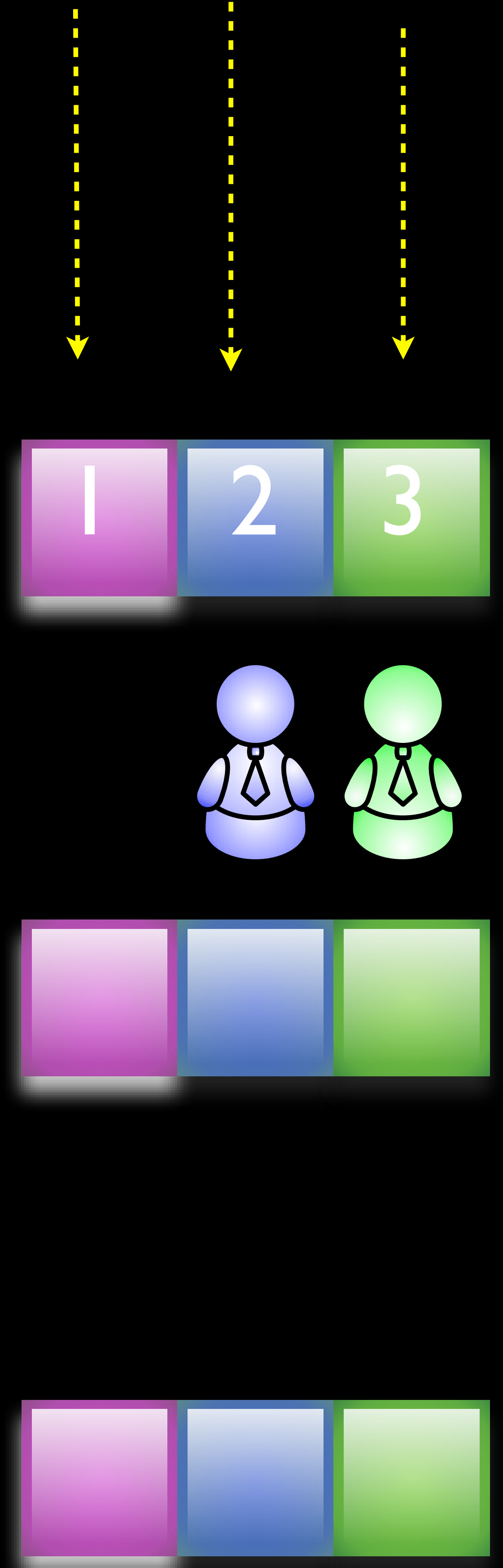


- arrive in arbitrary order
- last one sees all

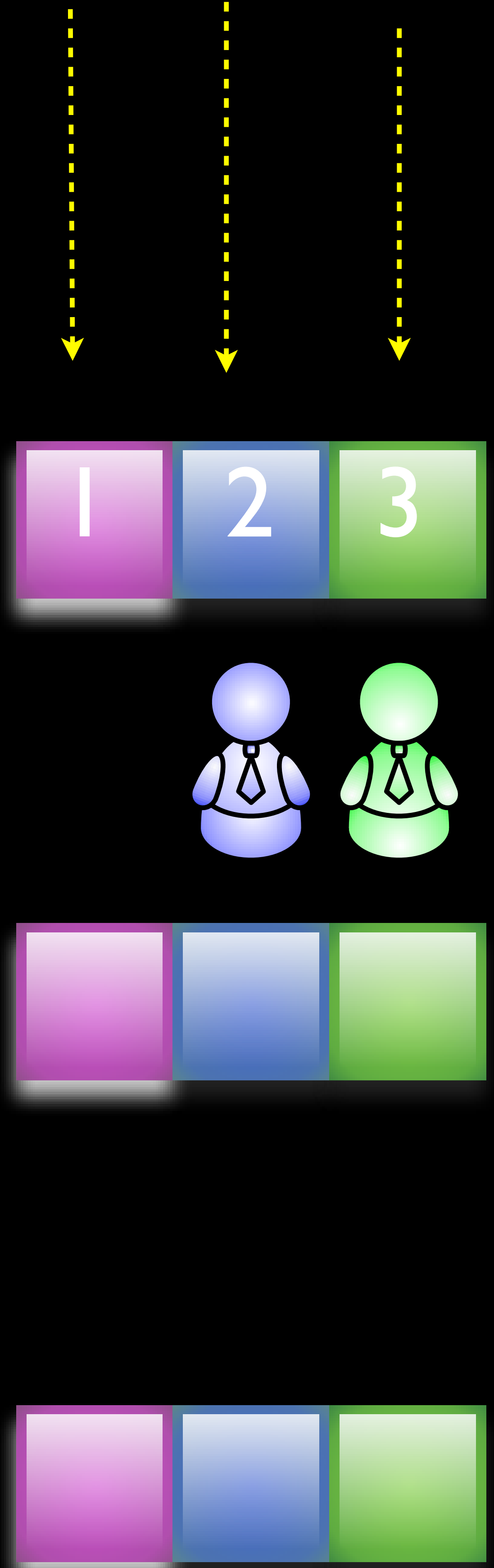


- arrive in arbitrary order
- last one sees all

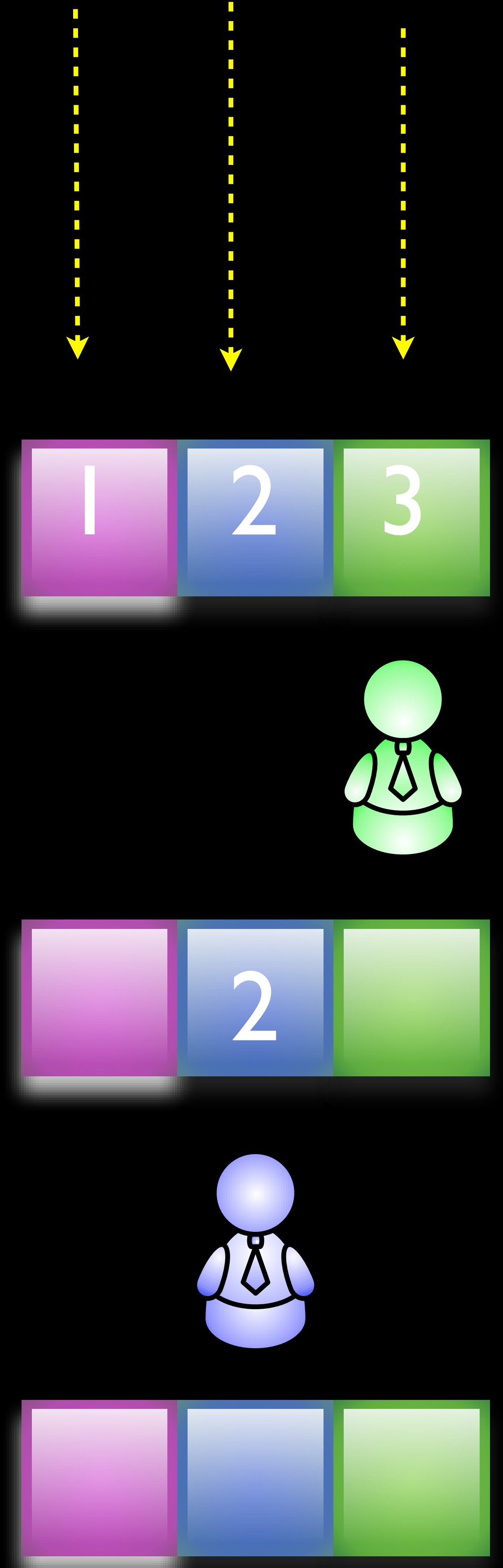
returns 1,2,3



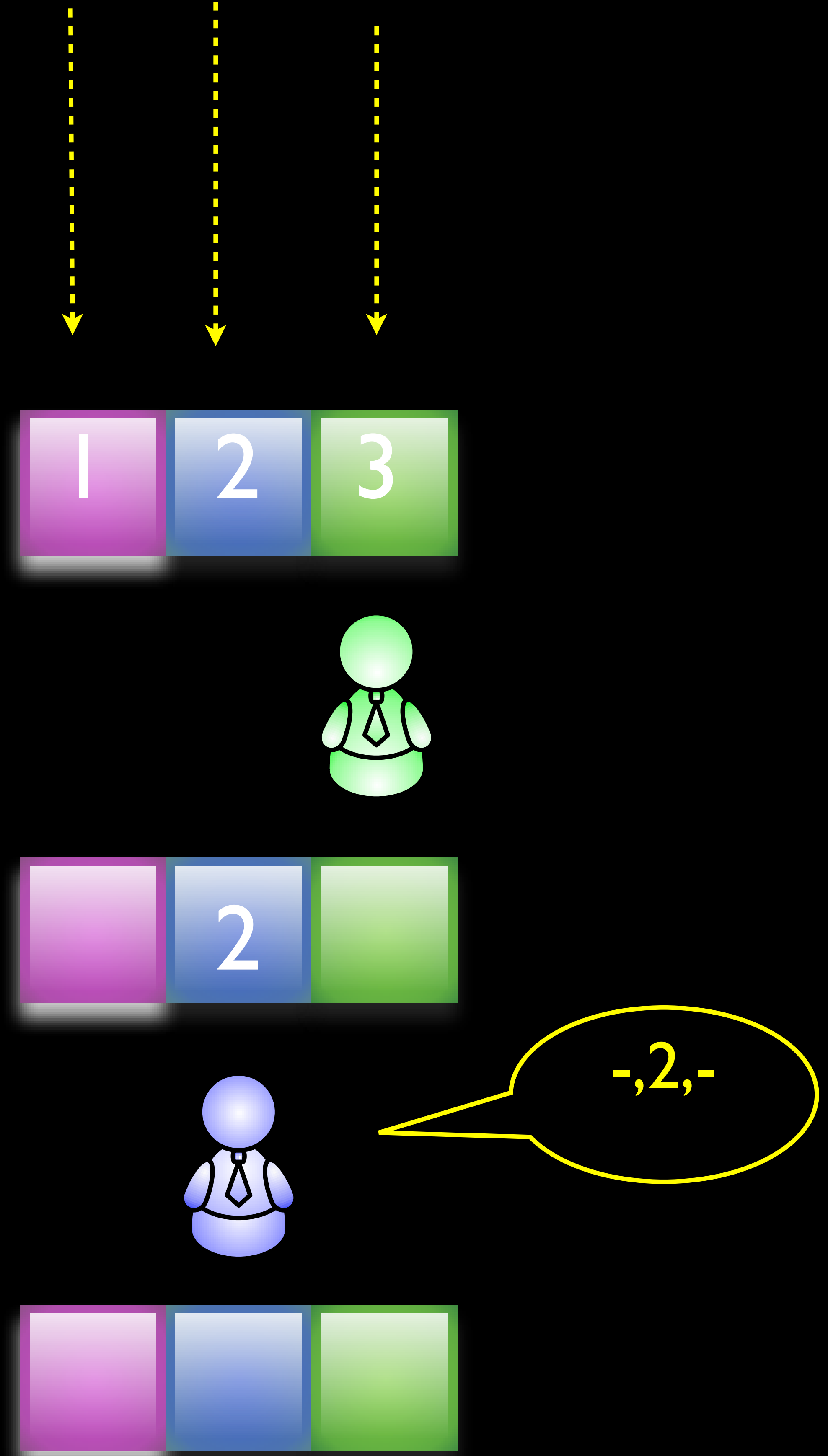
- remaining 2 go to next memory



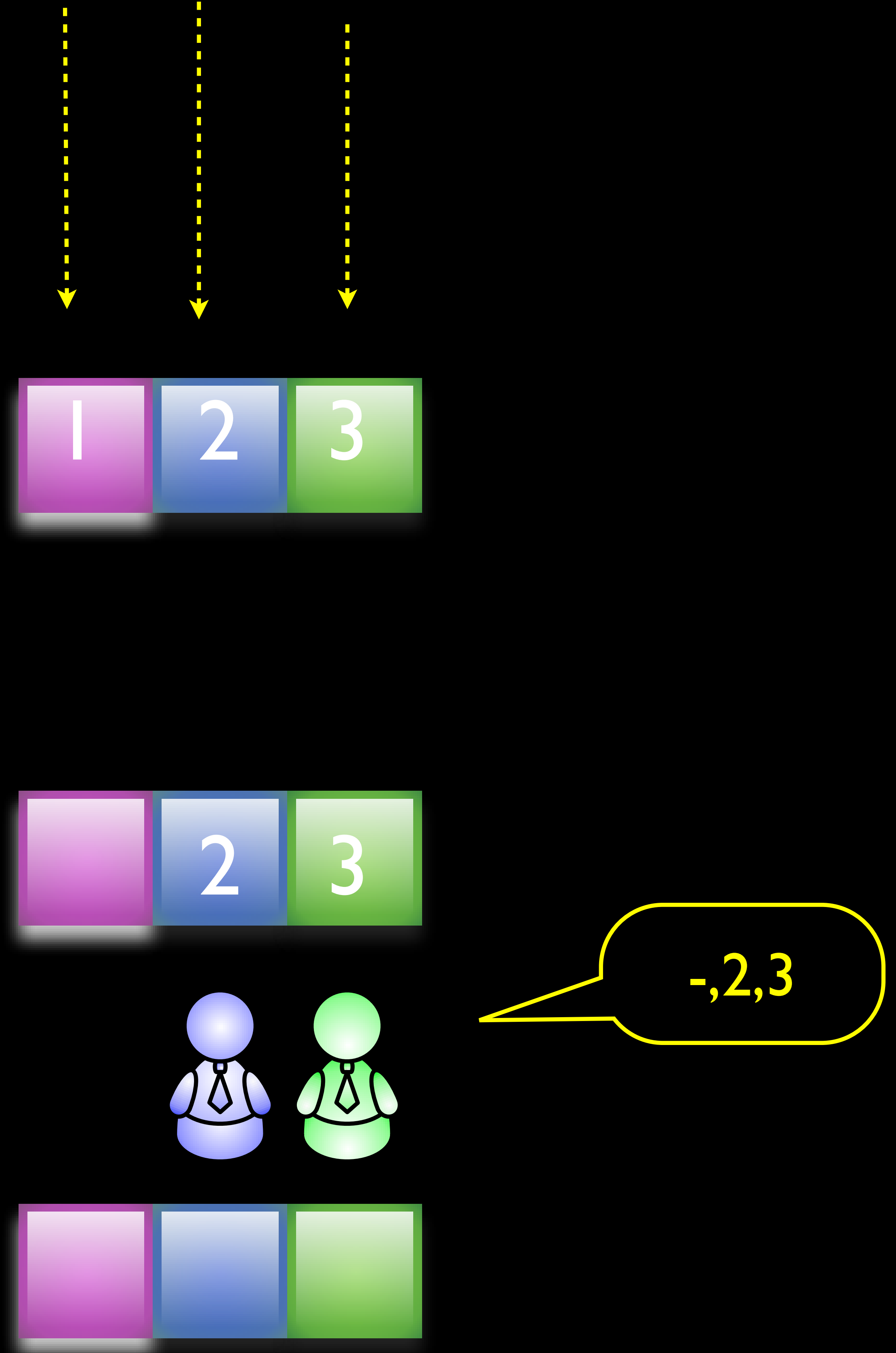
• remaining 2 go to next memory



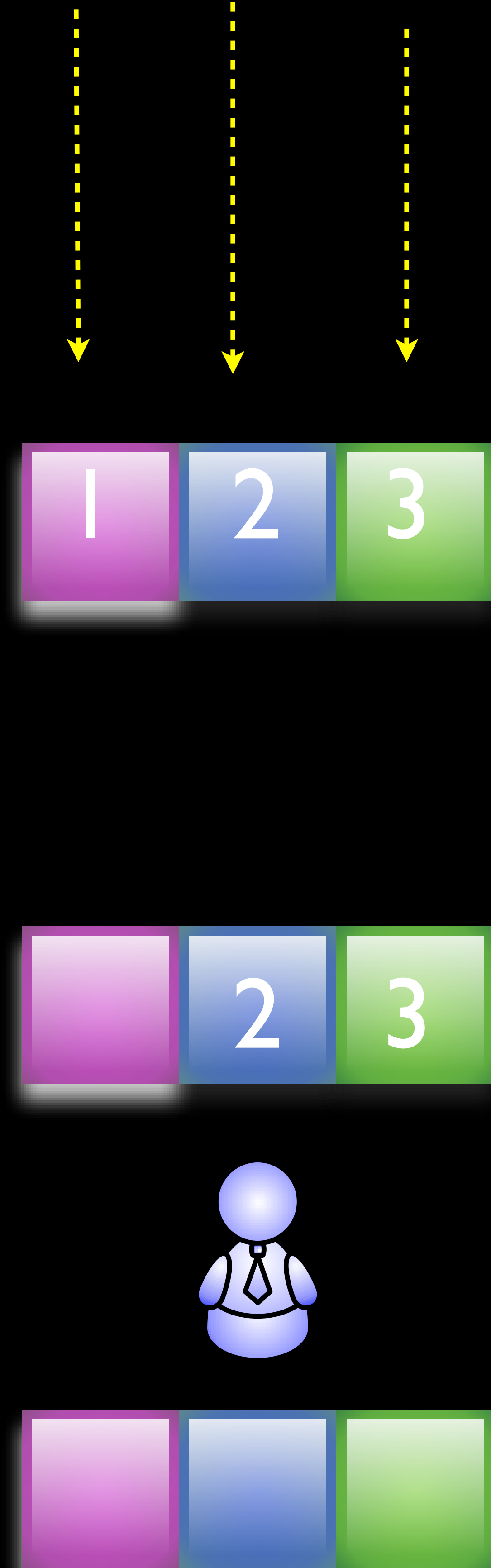
• remaining 2 go to next memory



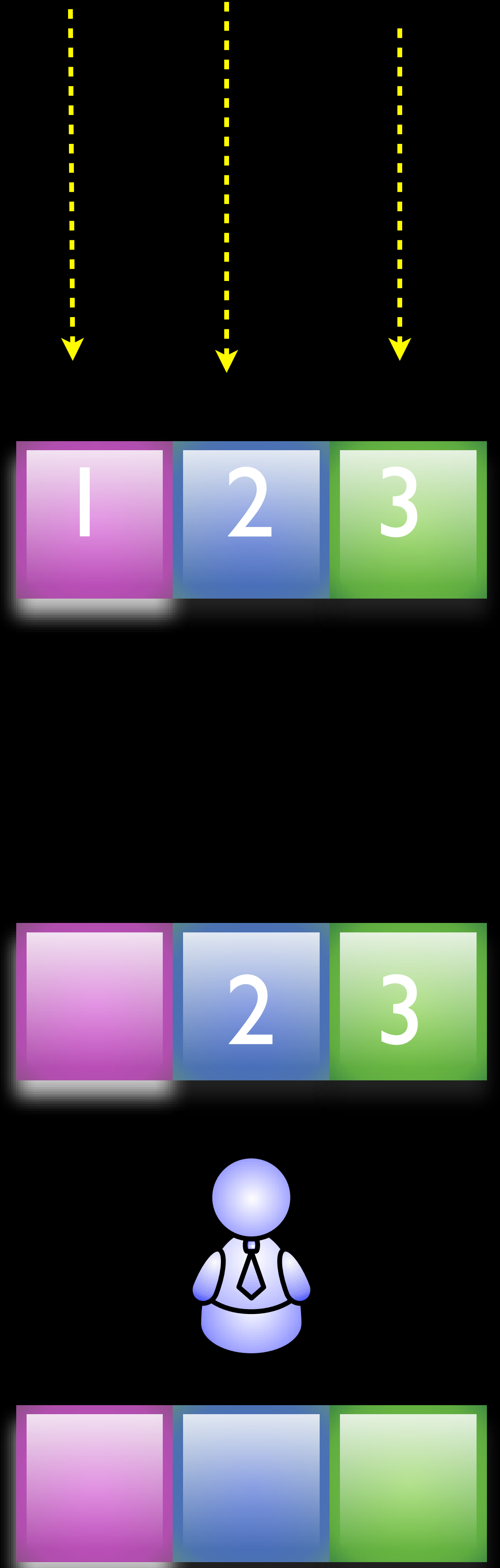
- 3rd one returns -,2,3

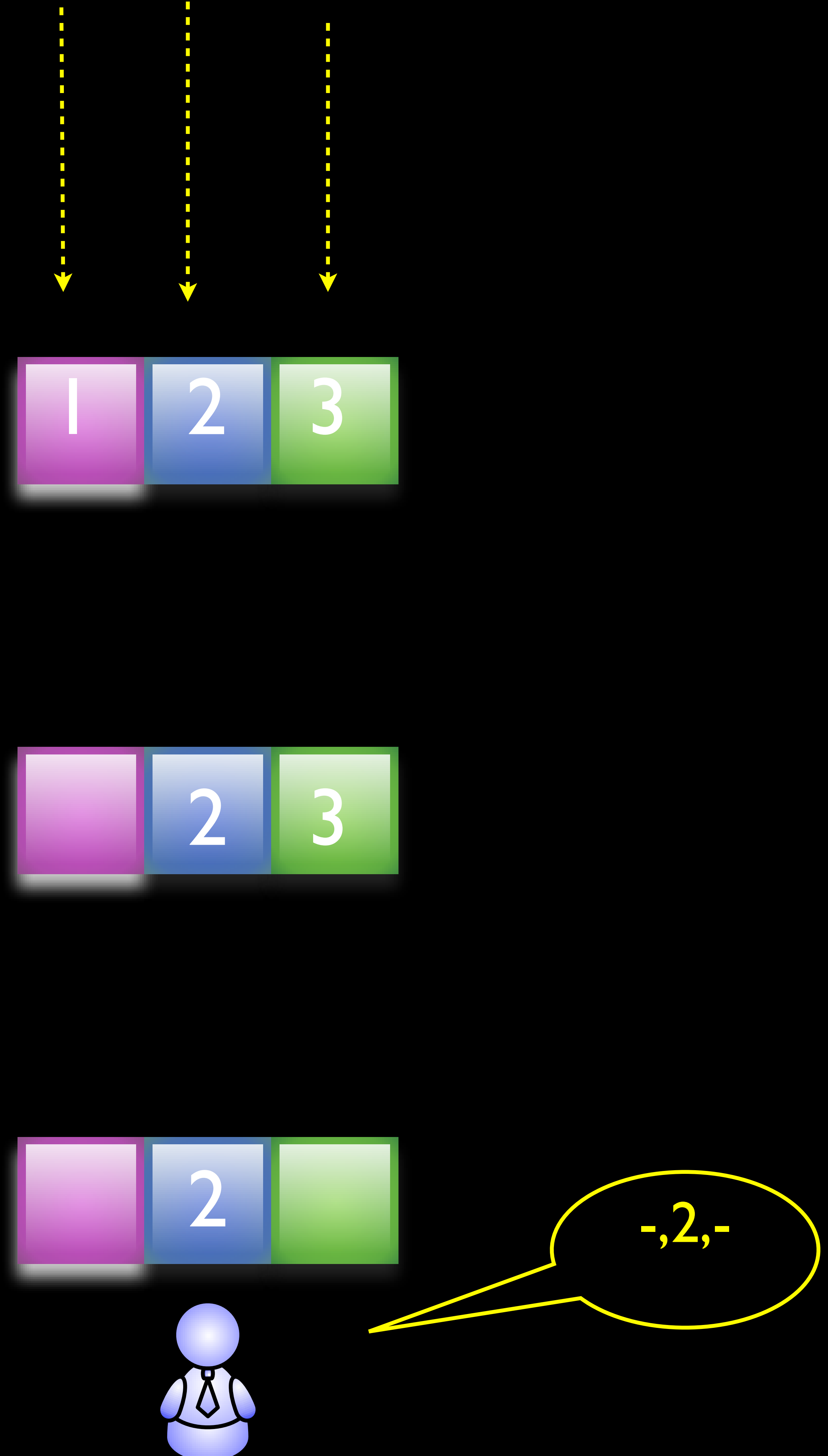


- 3rd one returns -,2,3

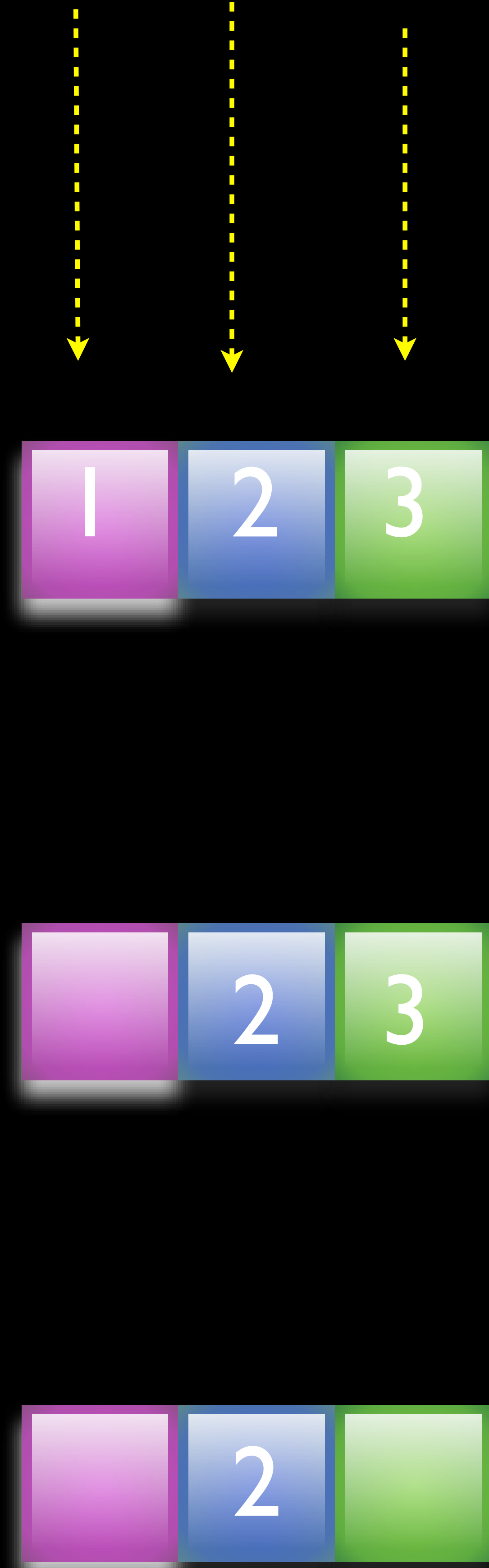


• 2nd one goes alone

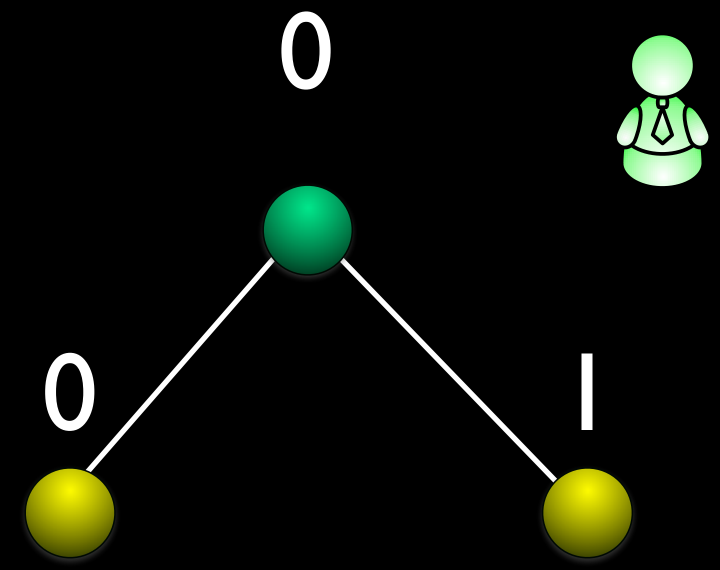



1
2
3
2
3
2
-,2,-

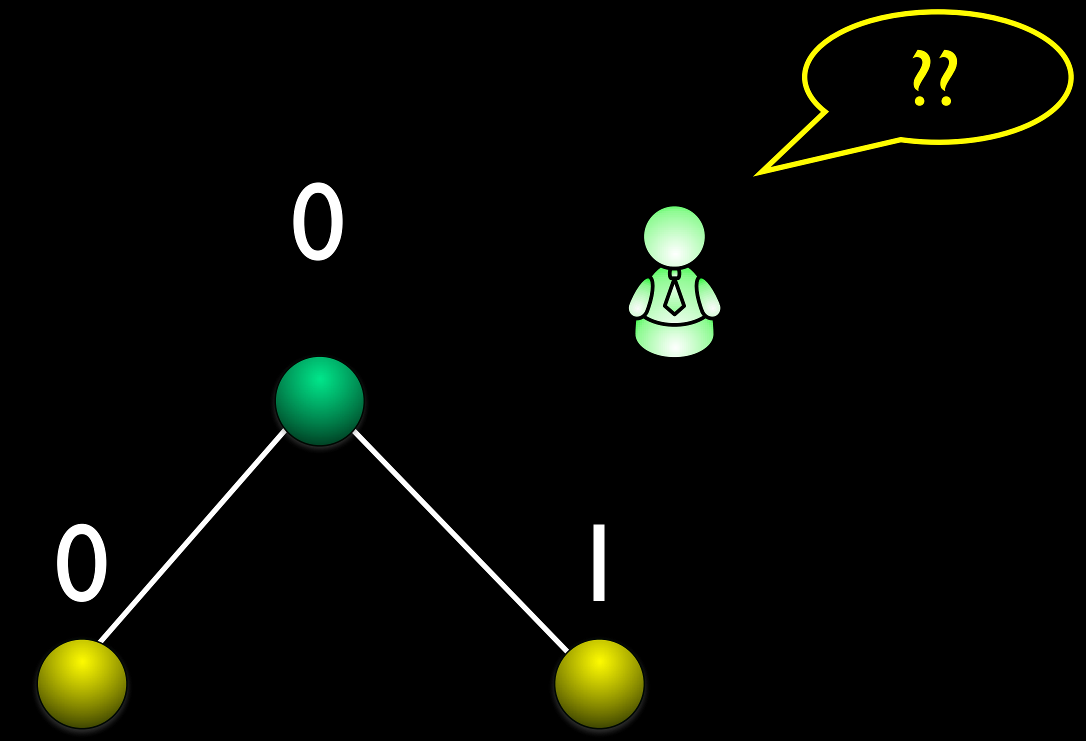
- returns -,2,-



limitations come from indistinguishability

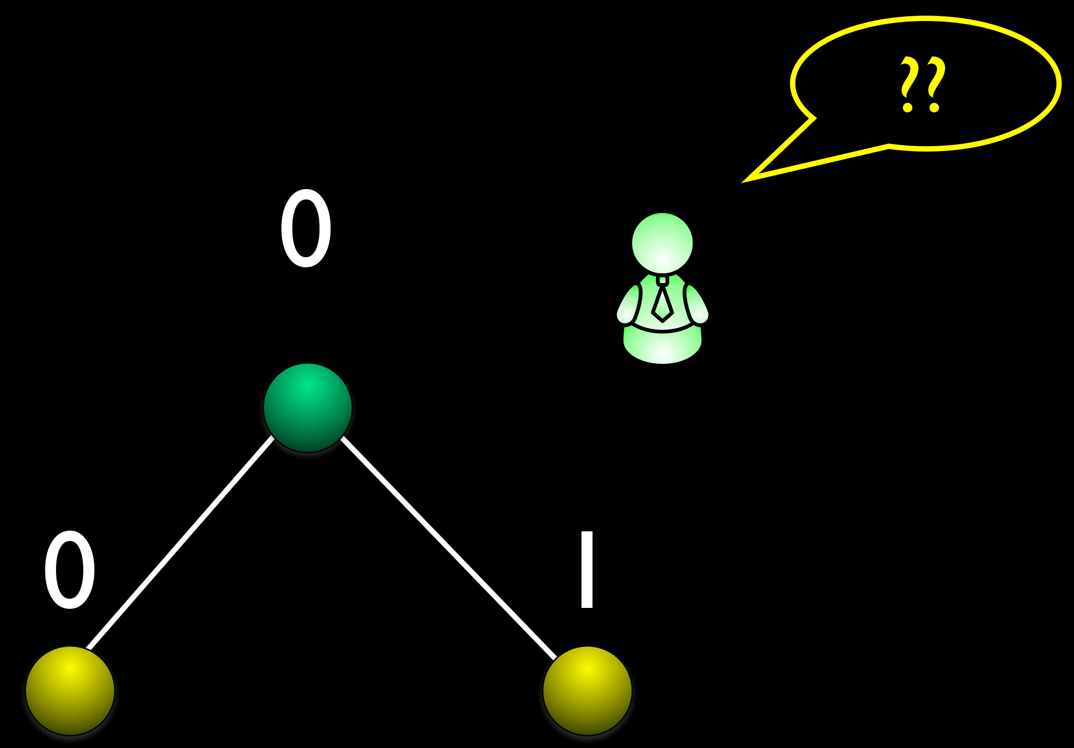


limitations come from indistinguishability



limitations come from indistinguishability

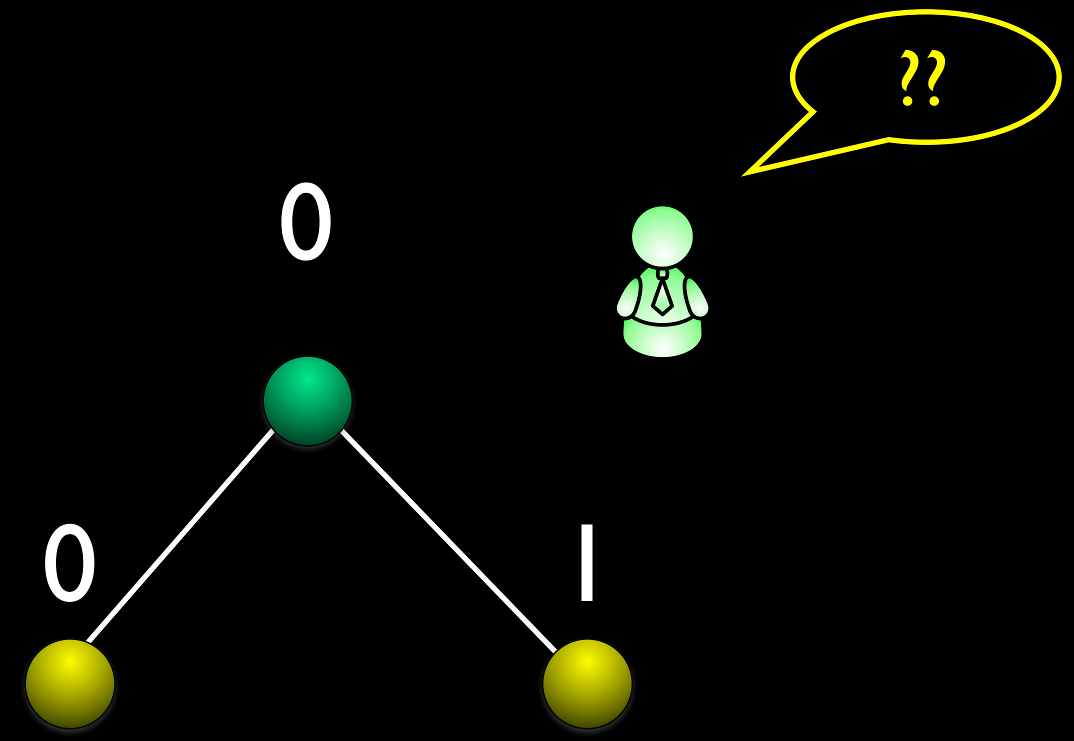
- The most essential distributed computing issue is that a process has only a local perspective of the world



limitations come from indistinguishability

- The most essential distributed computing issue is that a process has only a local perspective of the world

- Represent with a vertex labeled with id (green) and a local state this perspective

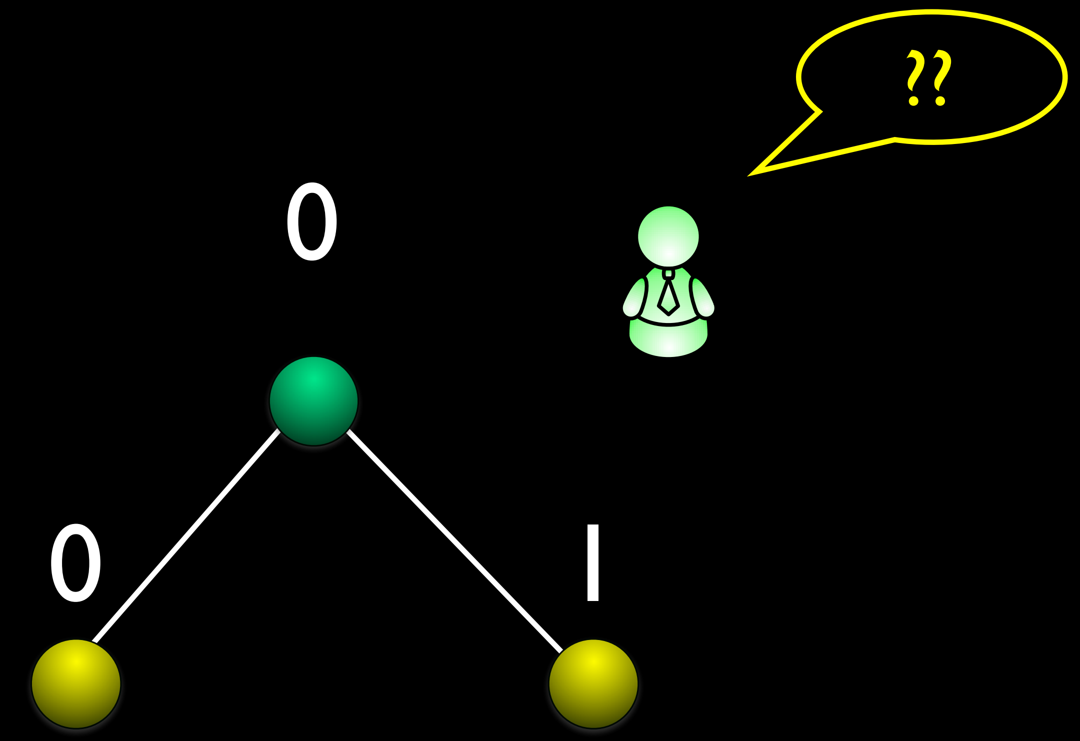


limitations come from indistinguishability

- The most essential distributed computing issue is that a process has only a local perspective of the world

- Represent with a vertex labeled with id (green) and a local state this perspective

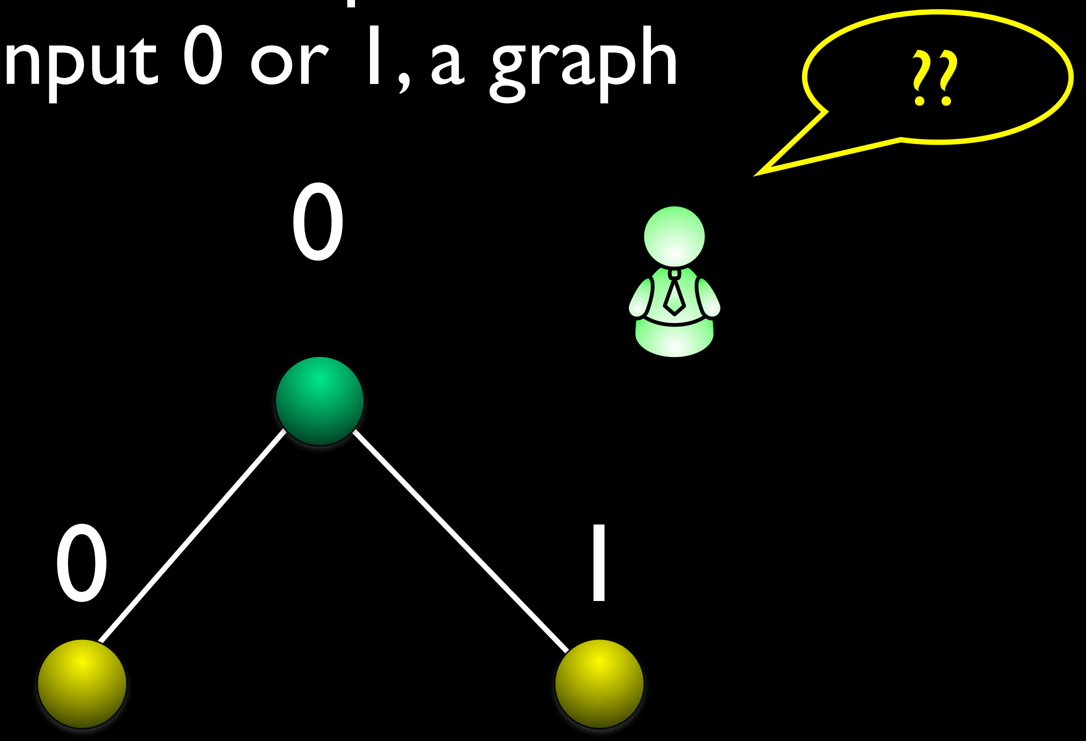
- E.g., its input is 0



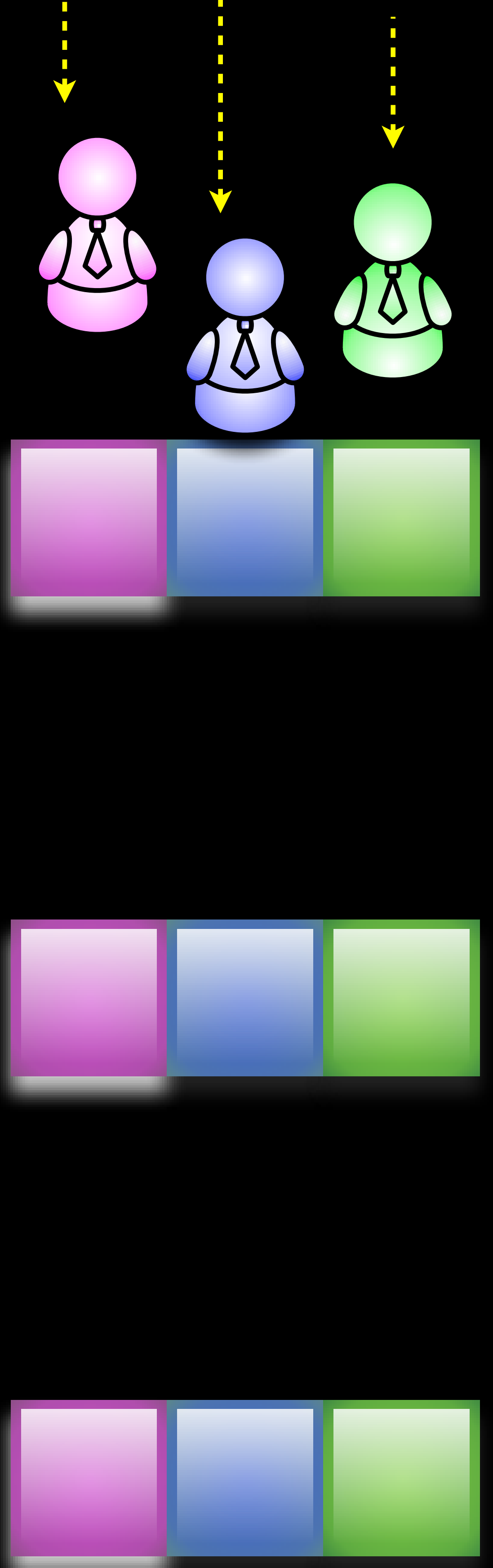
limitations come from indistinguishability

- The most essential distributed computing issue is that a process has only a local perspective of the world
- Represent with a vertex labeled with id (green) and a local state this perspective
- E.g., its input is 0

- Process does not know if another process has input 0 or 1, a graph

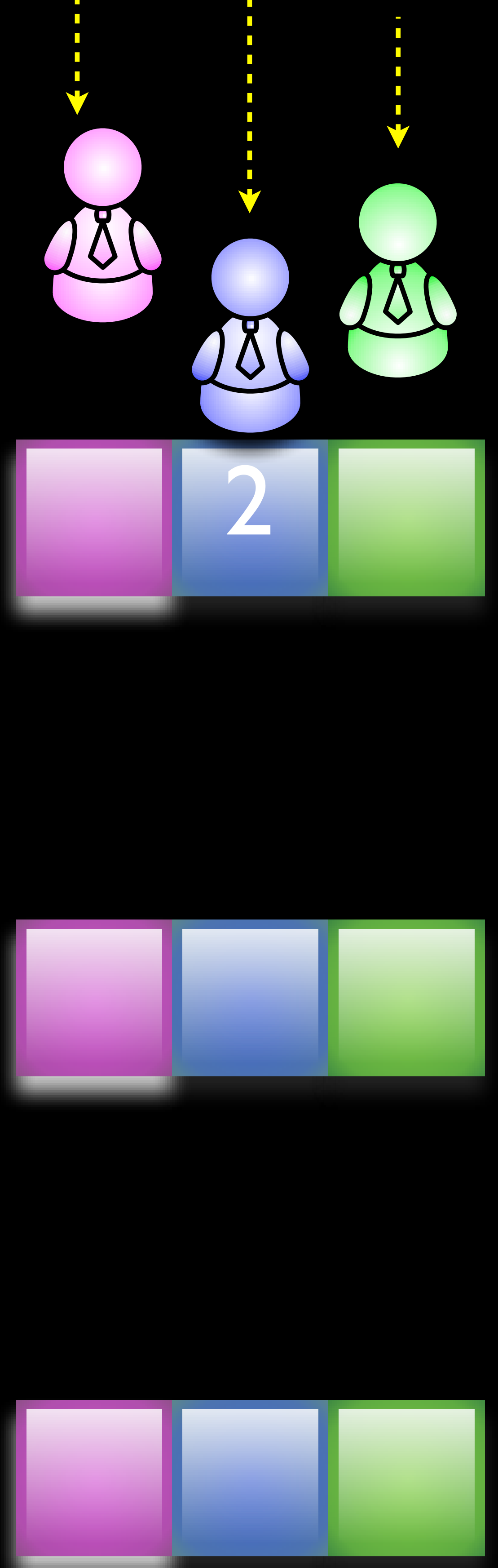


Indistinguishability graph for 2 processes

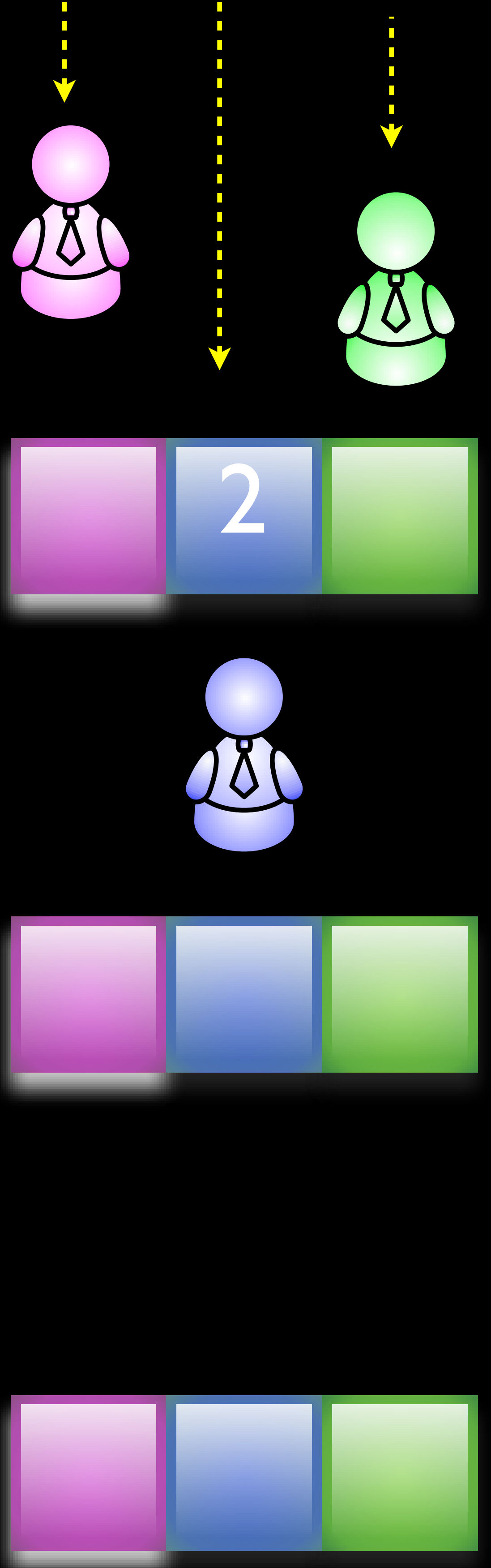


- focus on 2 processes

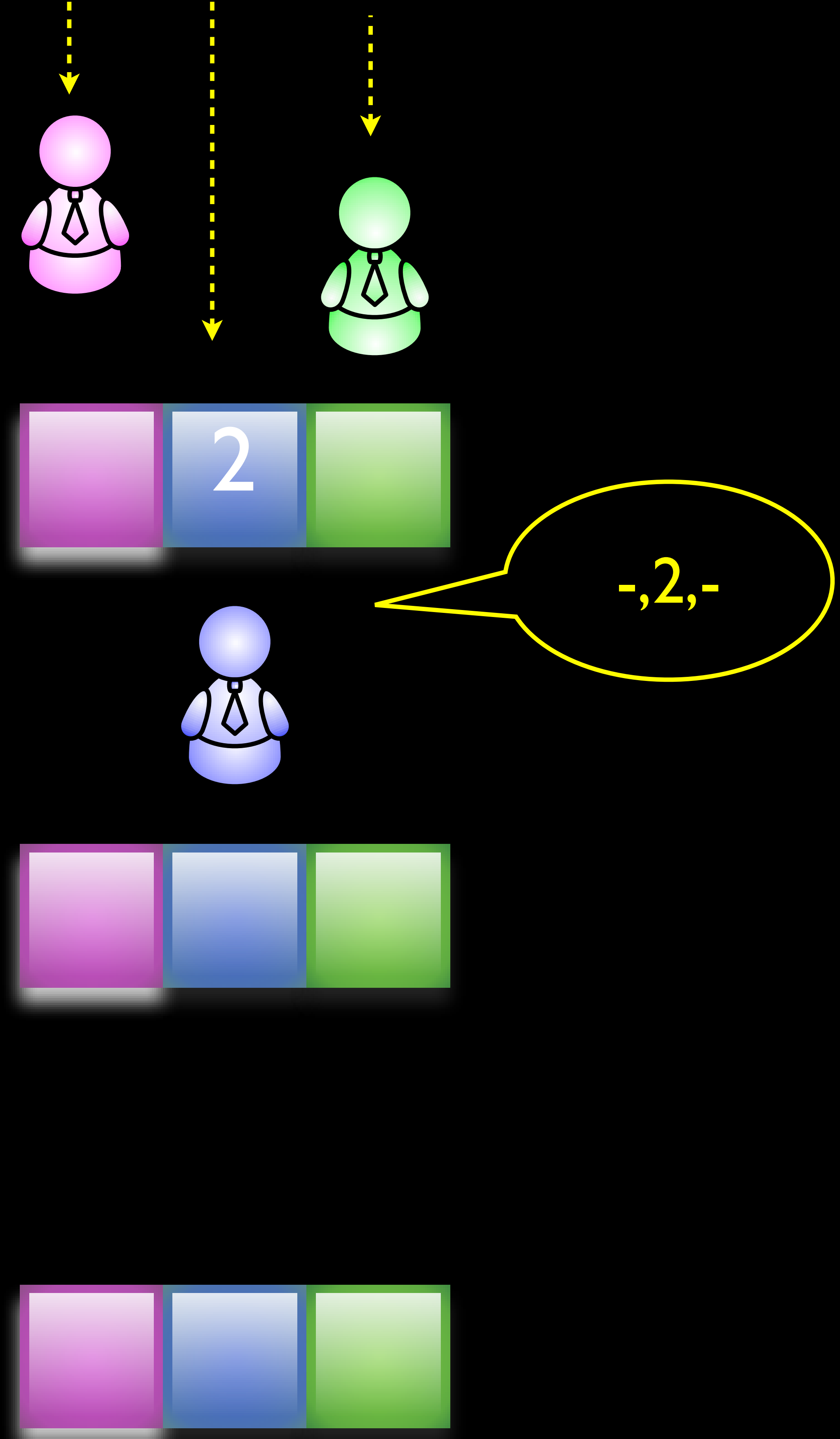
- there may be more that arrive after



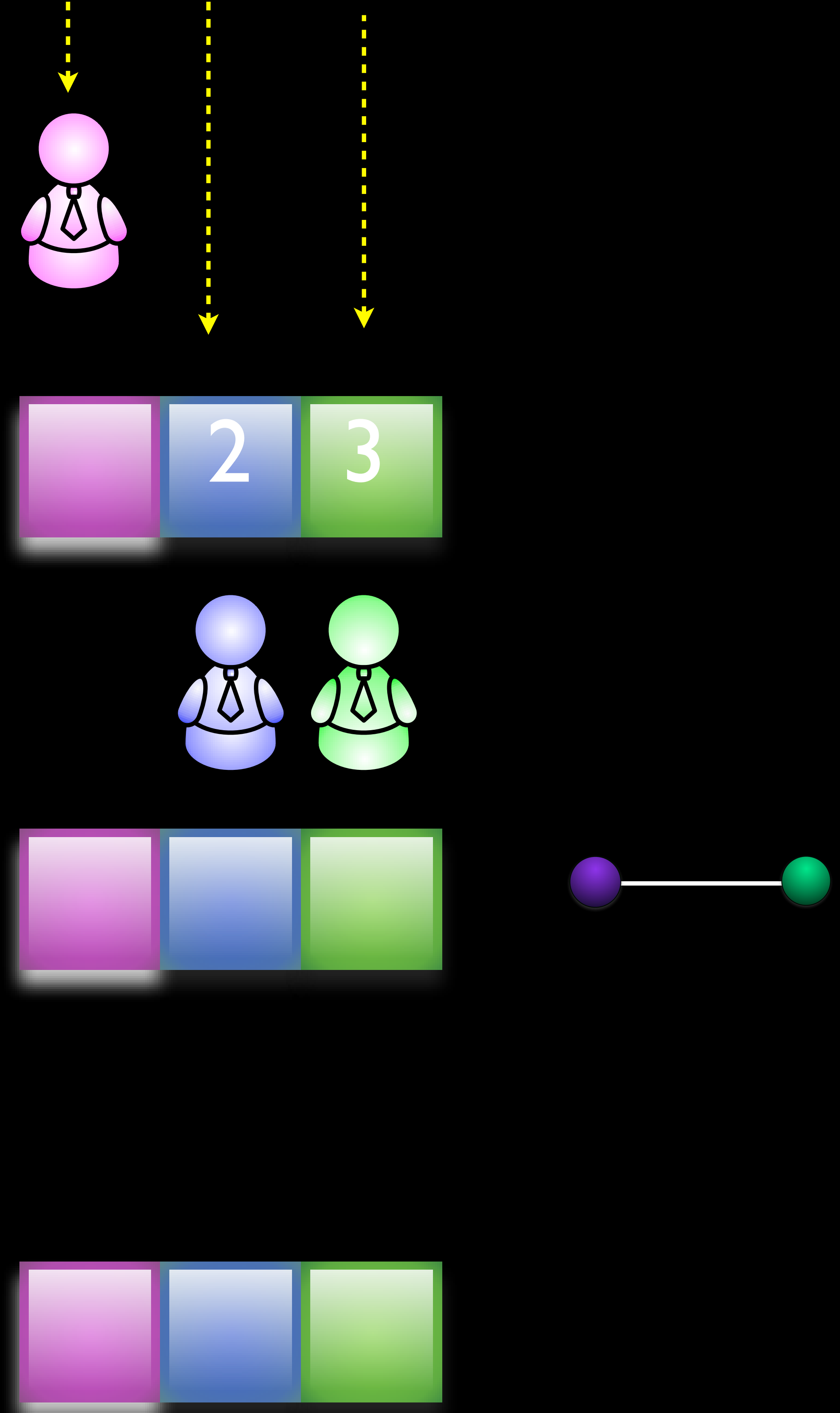
sees only itself



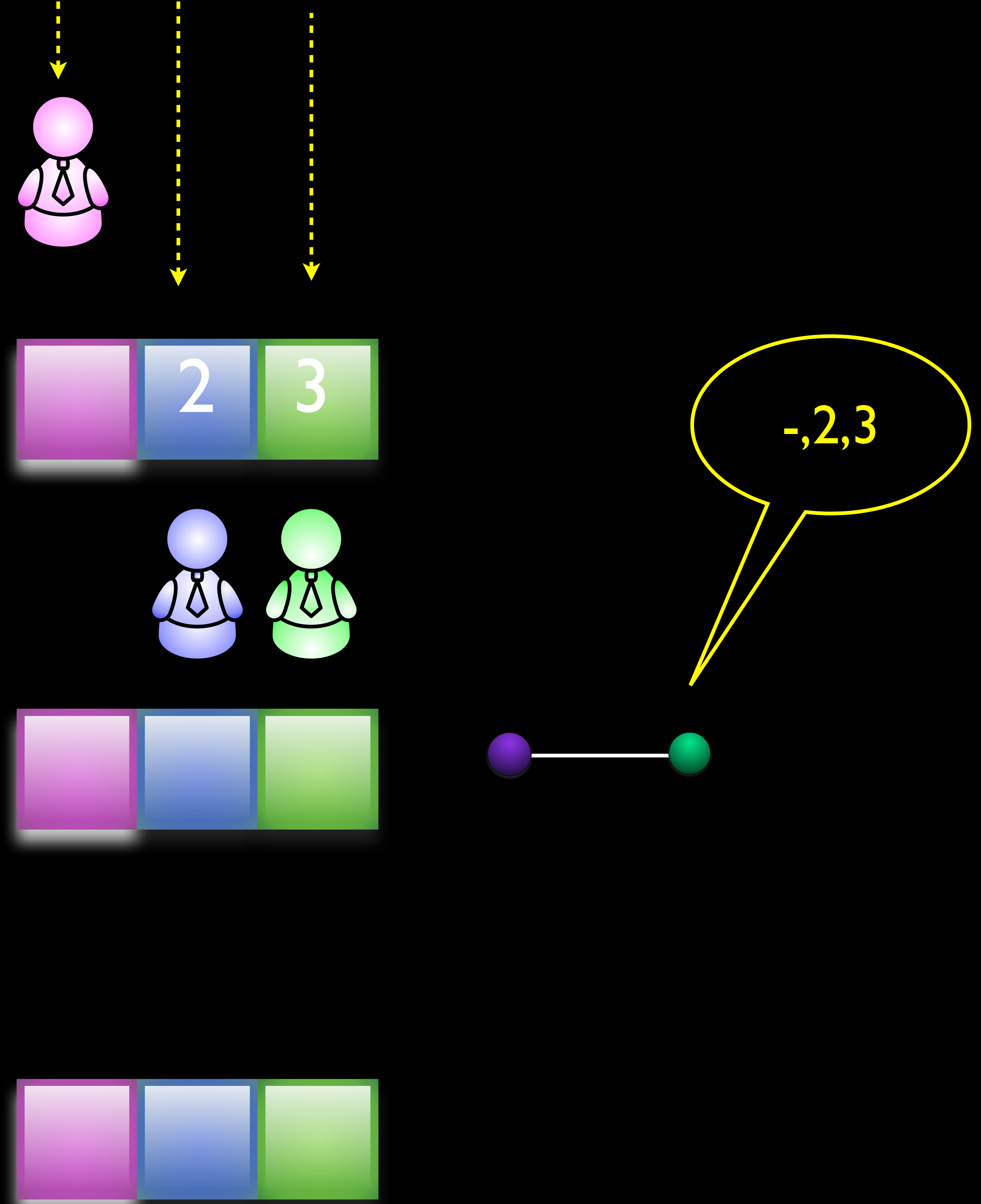
sees only itself



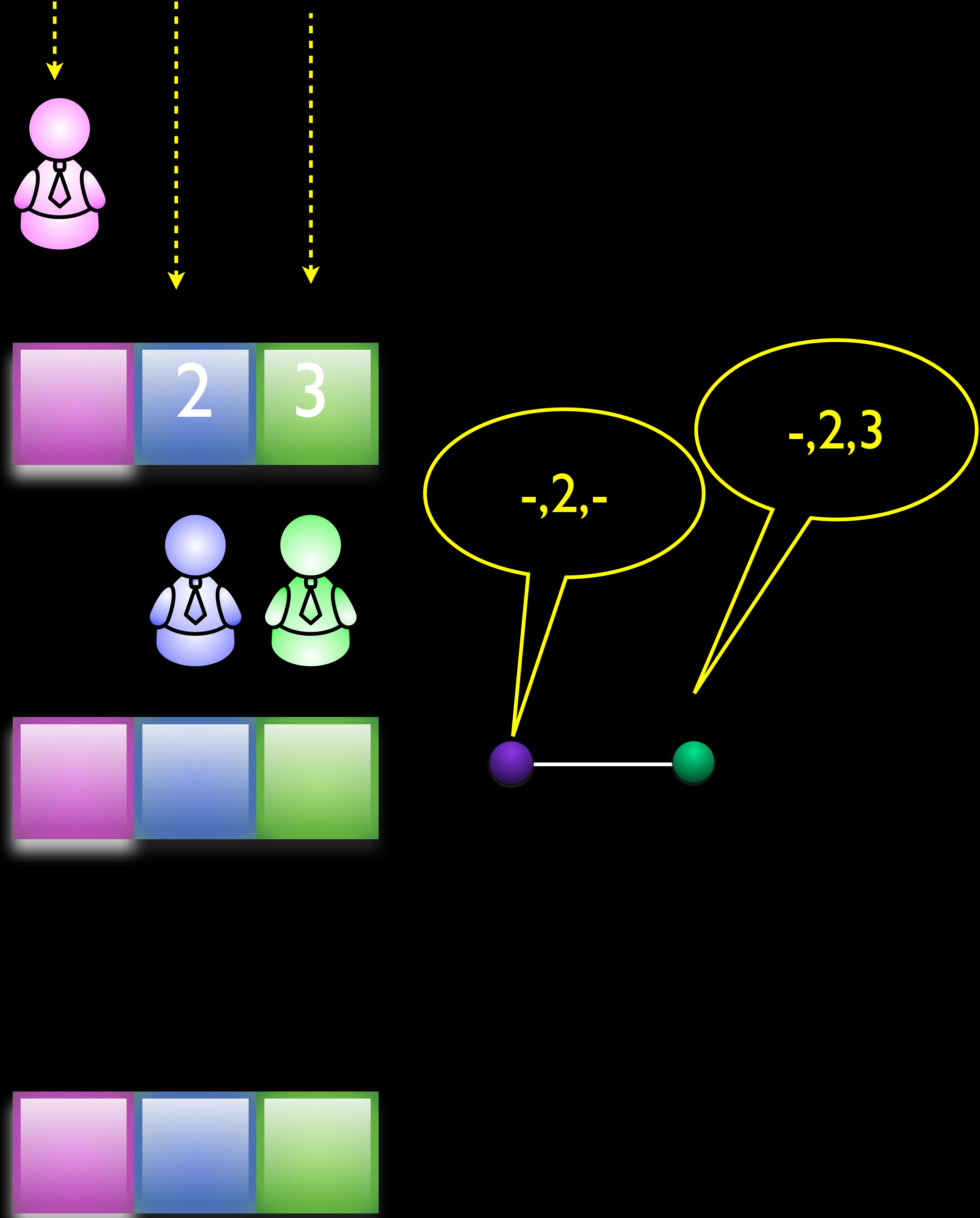
- green sees both
- but ...



- green sees both
- but ...

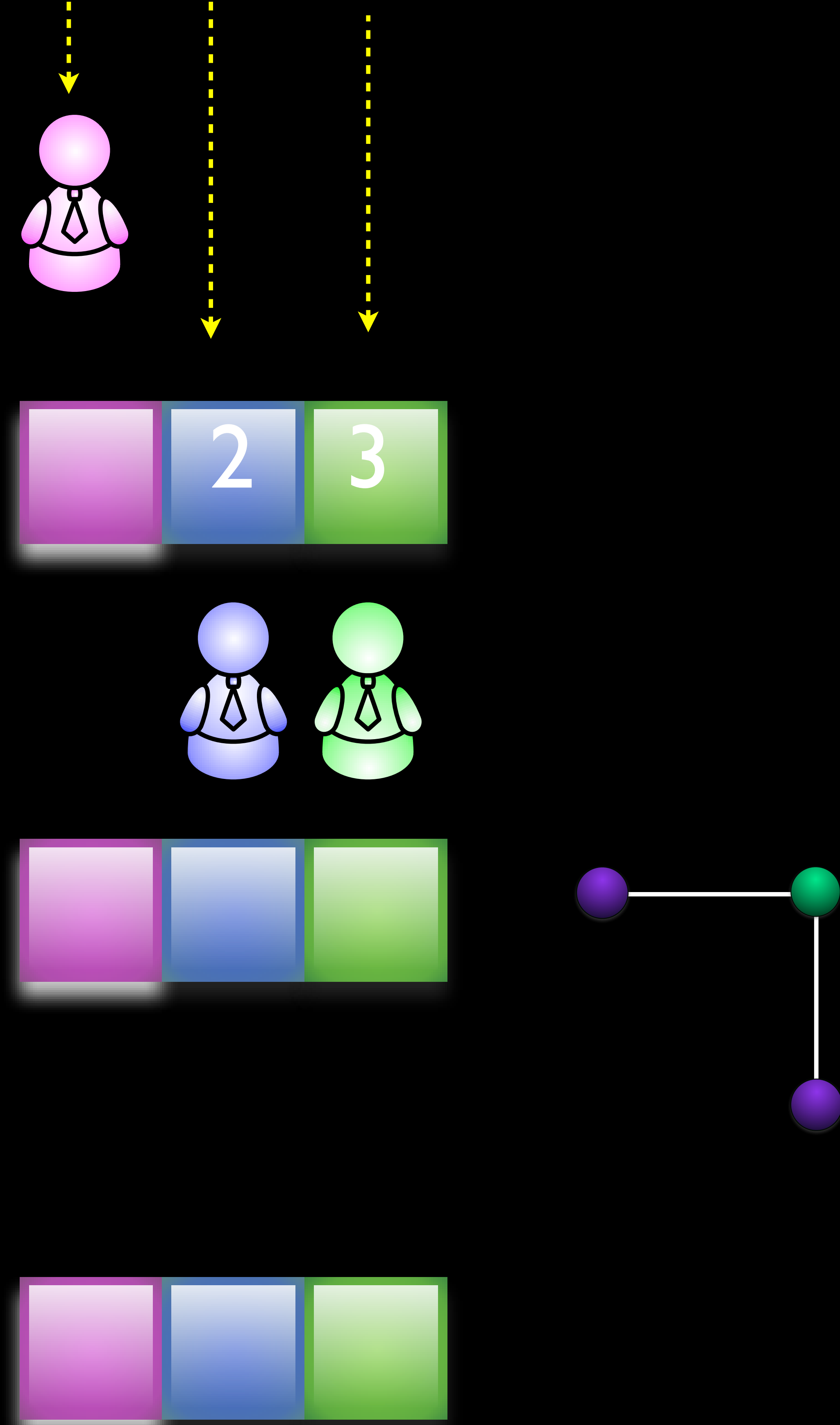


- green sees both
- but ...



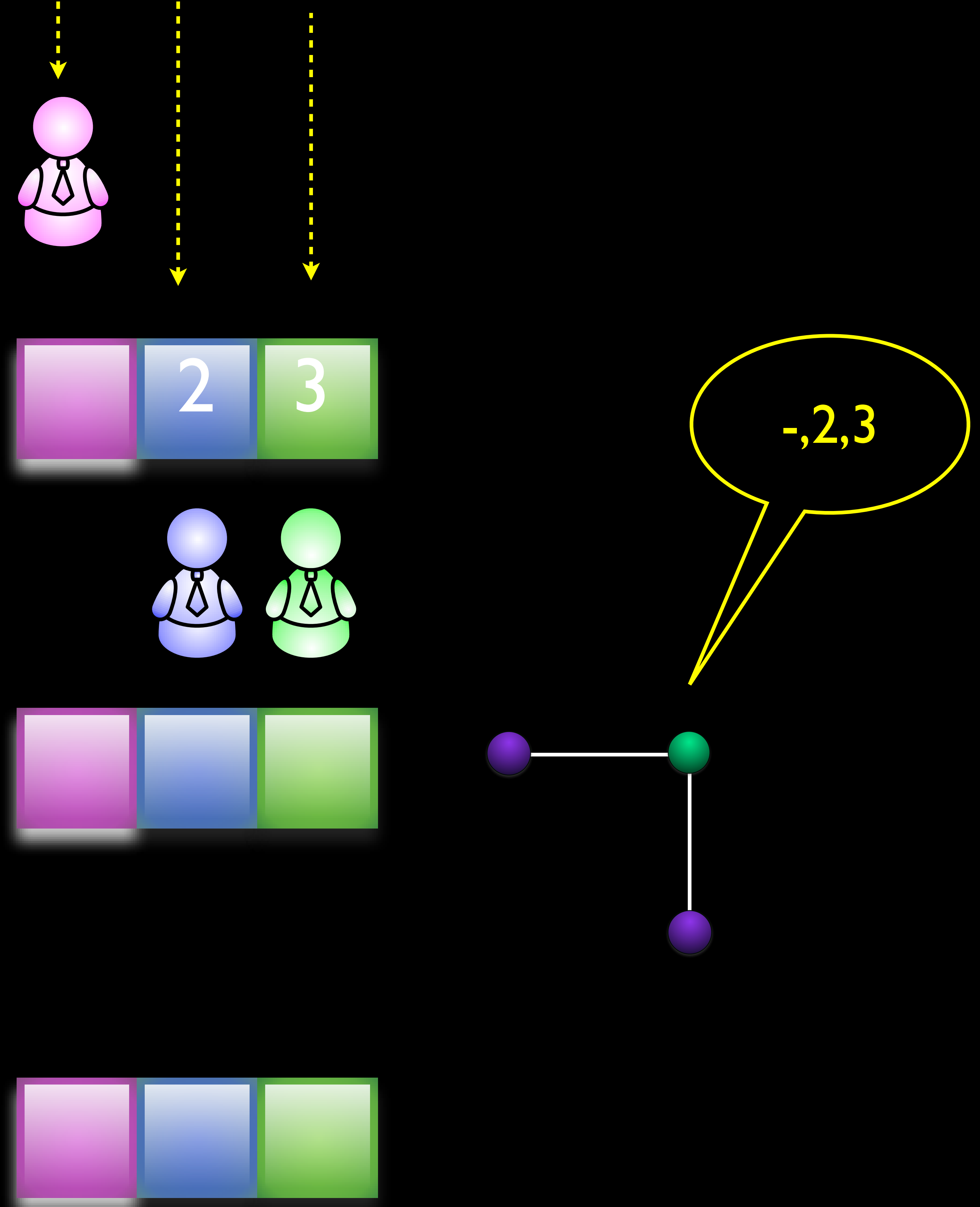
- green sees both

- but, doesn't know if seen by the other



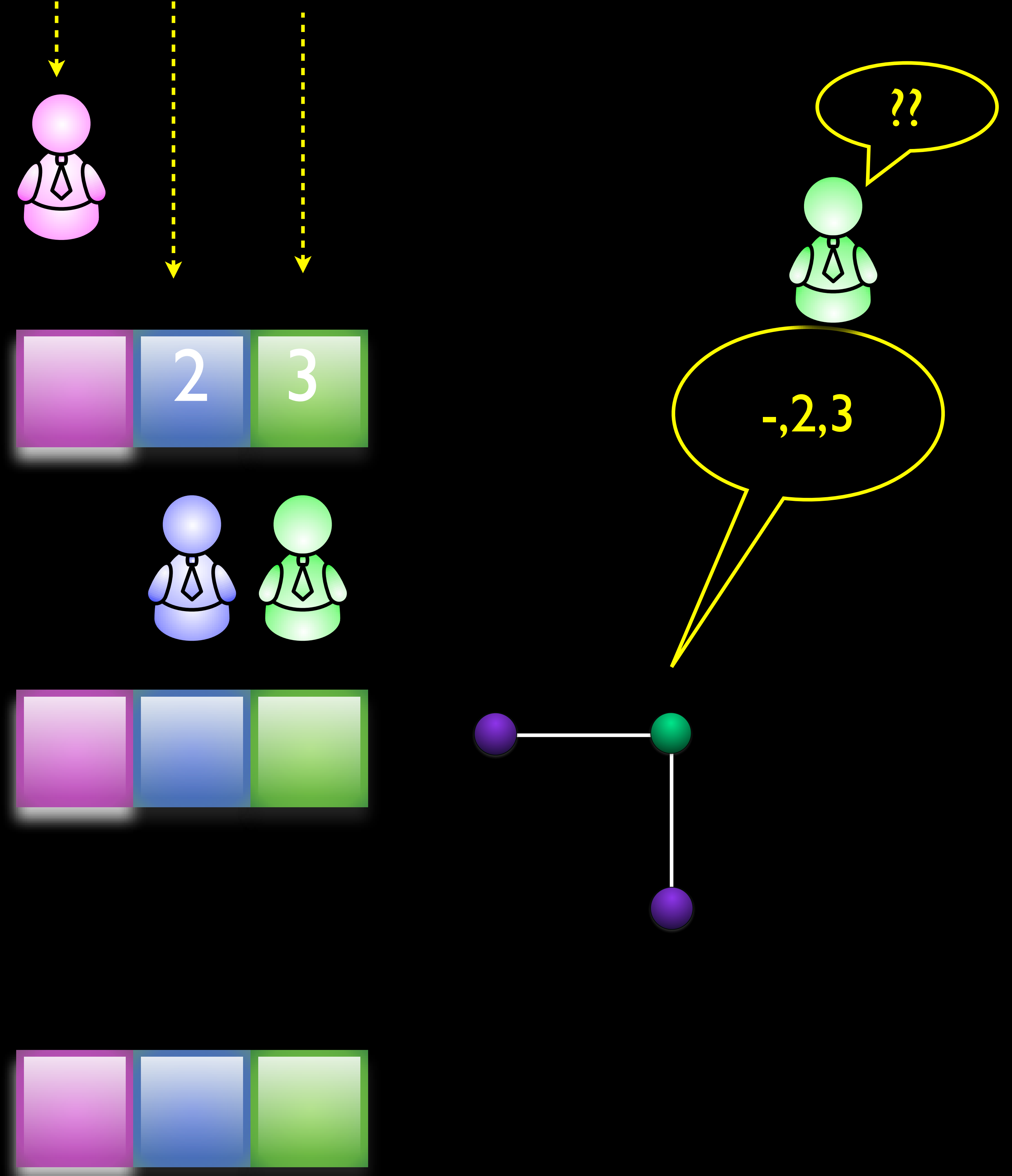
• green sees both

• but, doesn't know if seen by the other



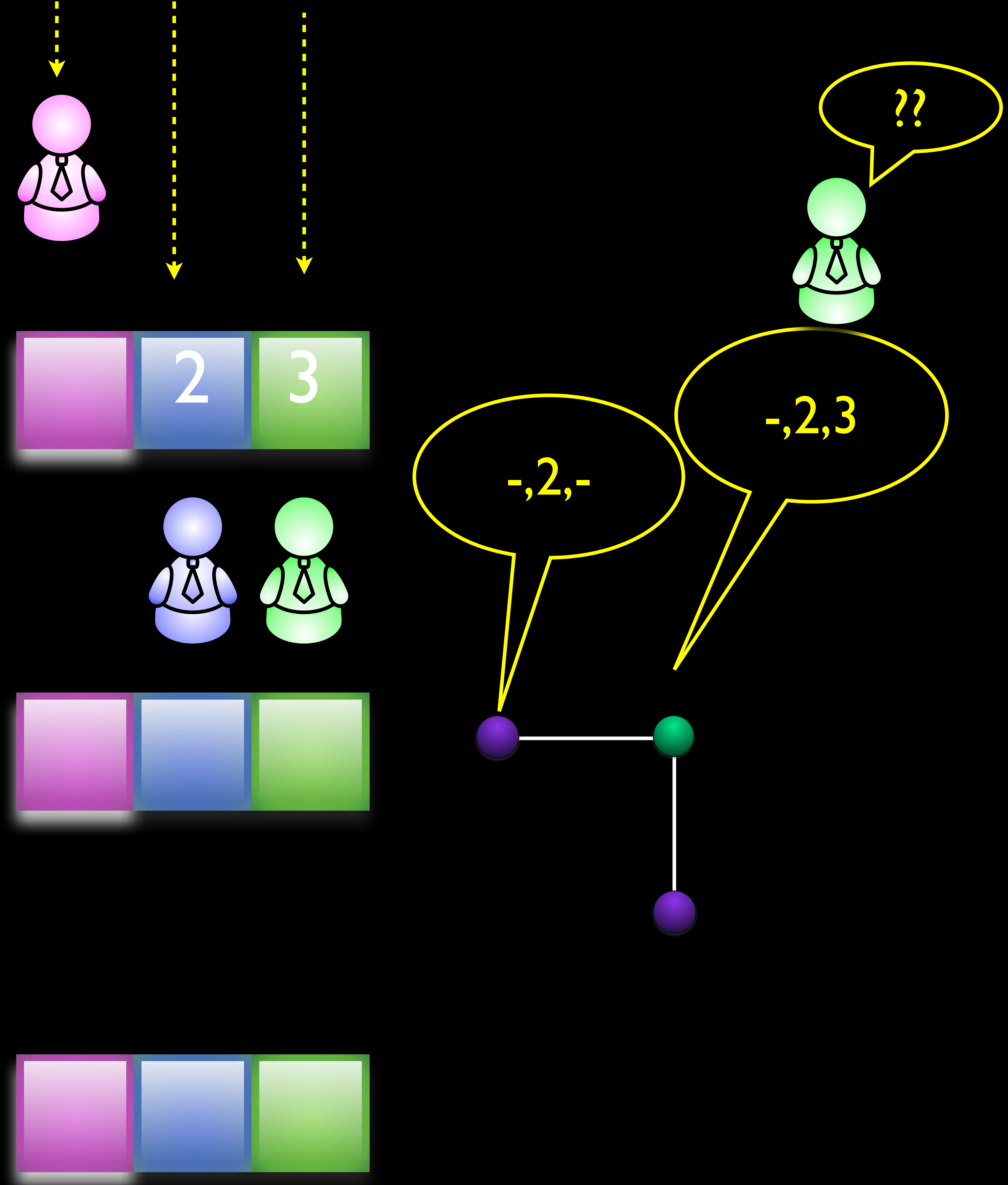
- green sees both

- but, doesn't know if seen by the other

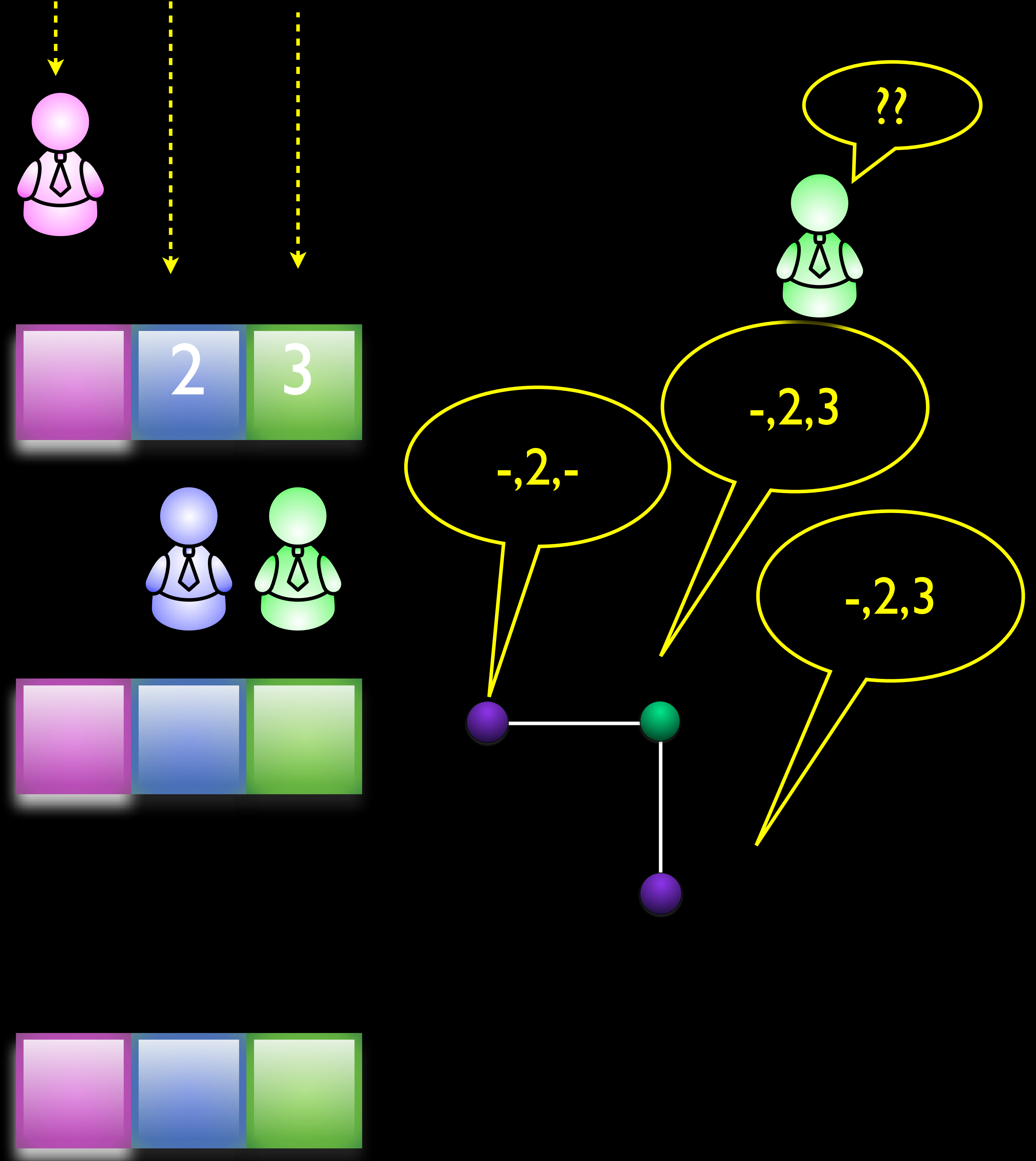


- green sees both

- but, doesn't know if seen by the other

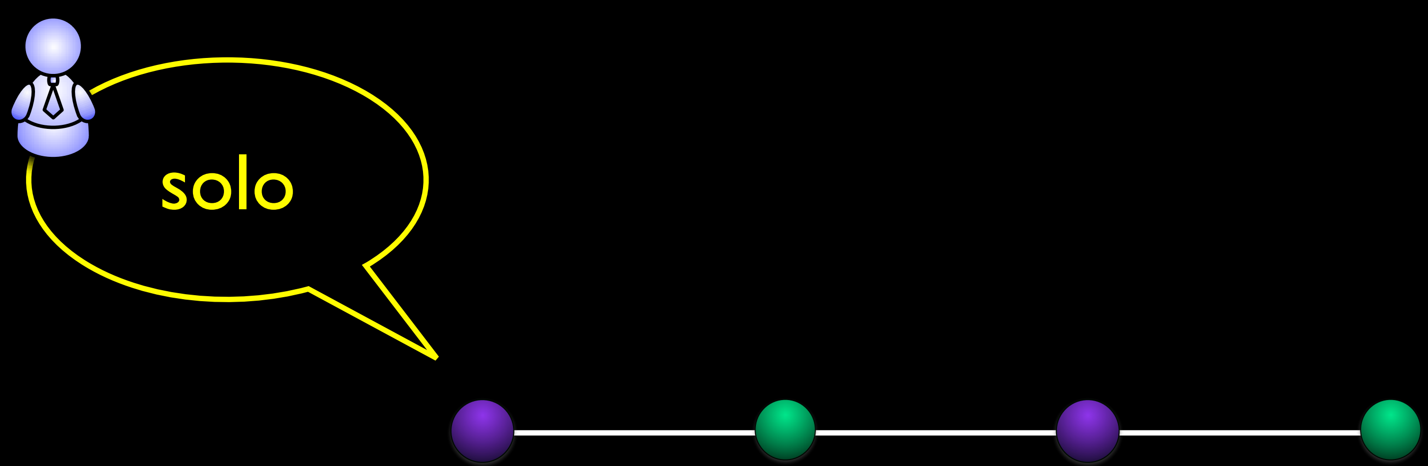


- green sees both
- but, doesn't know if seen by the other

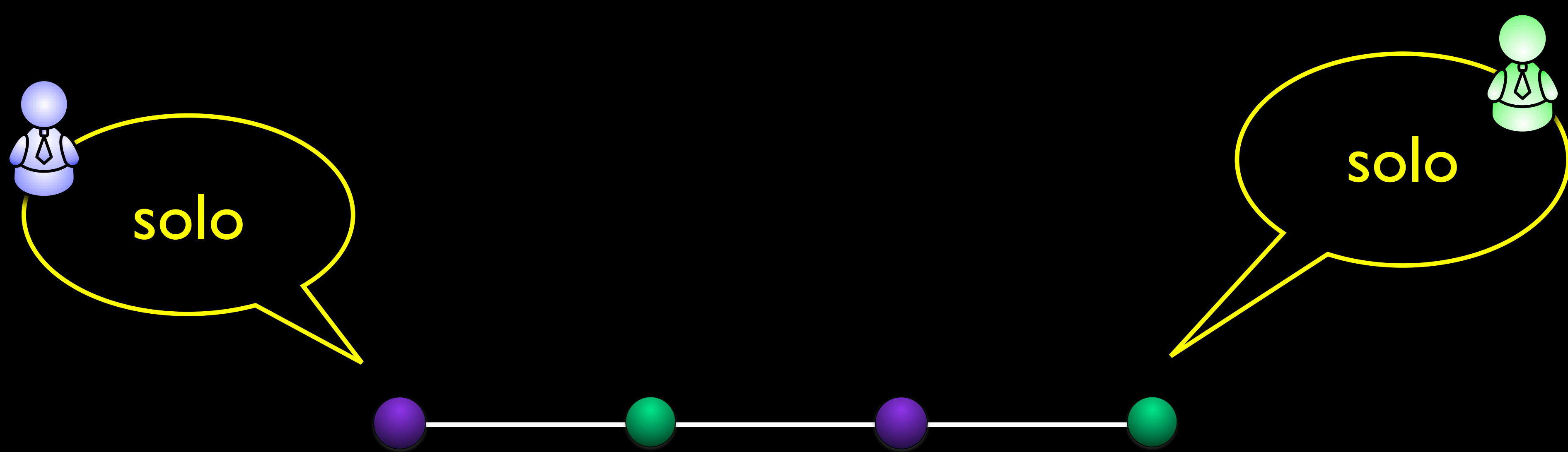


one round graph for 2 processes

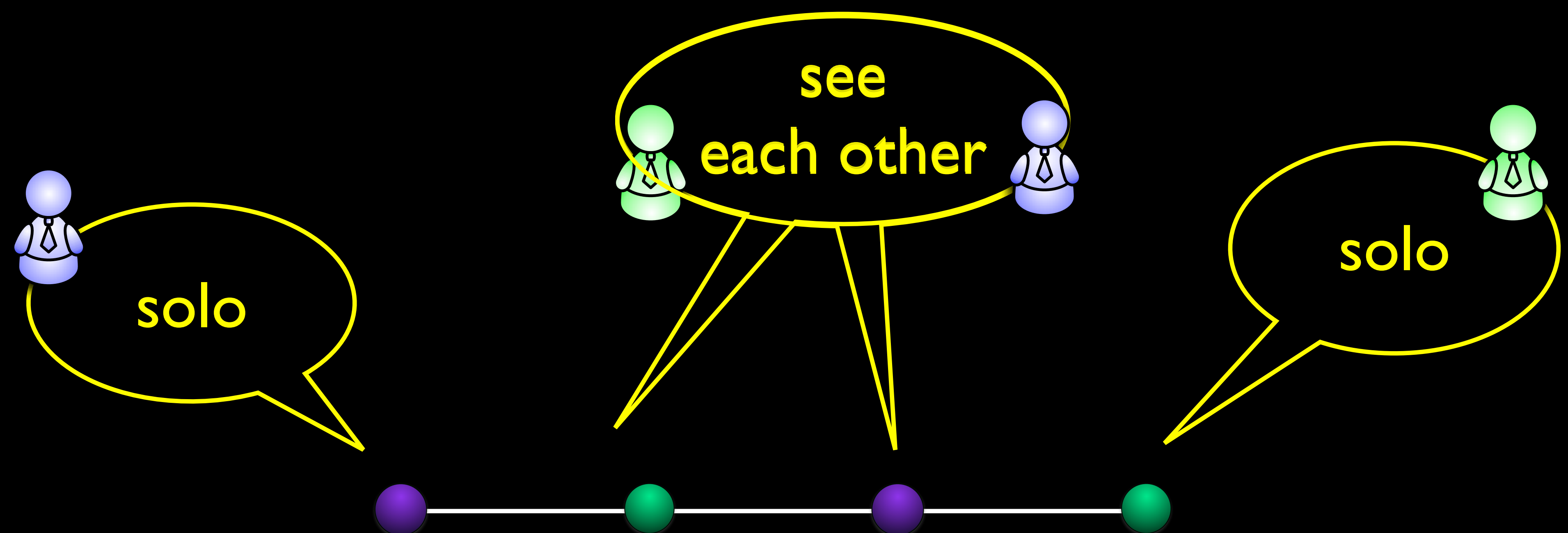
one round graph for 2 processes



one round graph for 2 processes



one round graph for 2 processes



iterated runs

for each run in round 1 there are the same 3 runs in the next round

round 1:

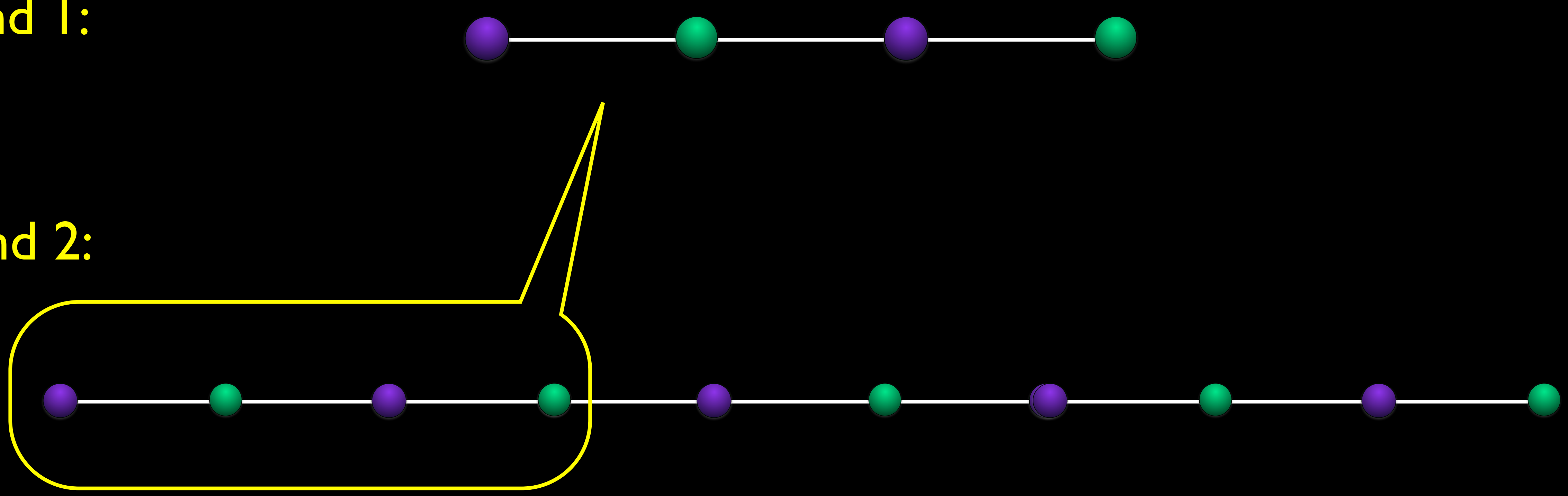
round 2:

iterated runs

for each run in round 1 there are the same 3 runs in the next round

round 1:

round 2:

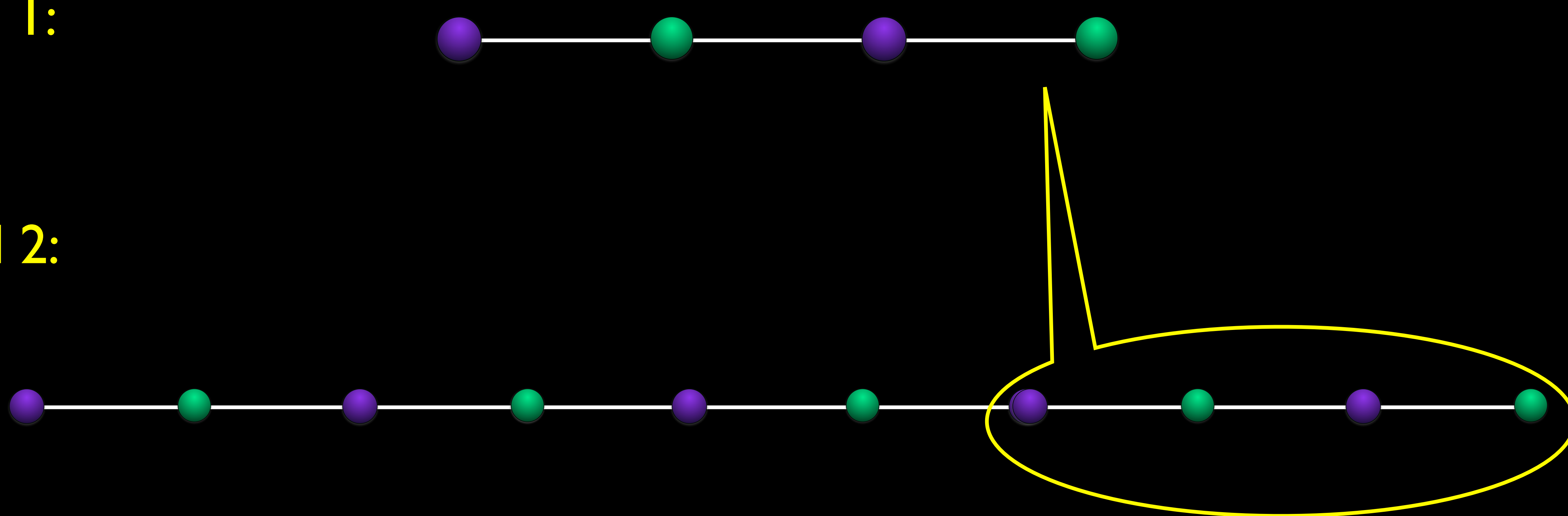


iterated runs

for each run in round 1 there are the same 3 runs in the next round

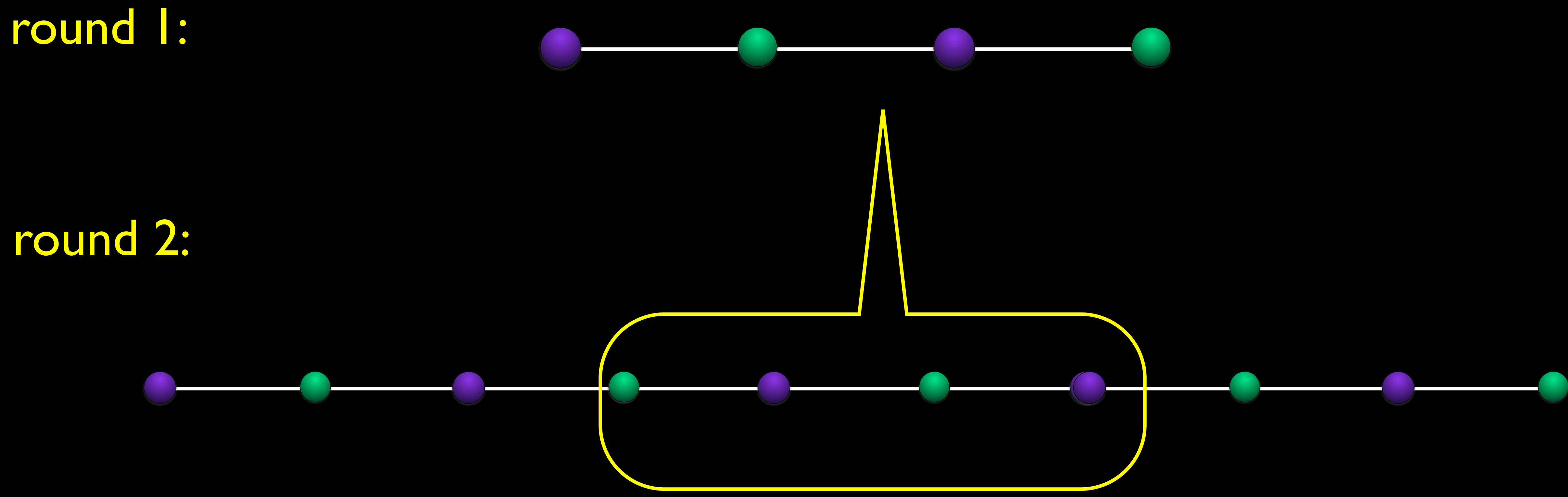
round 1:

round 2:



iterated runs

for each run in round 1 there are the same 3 runs in the next round



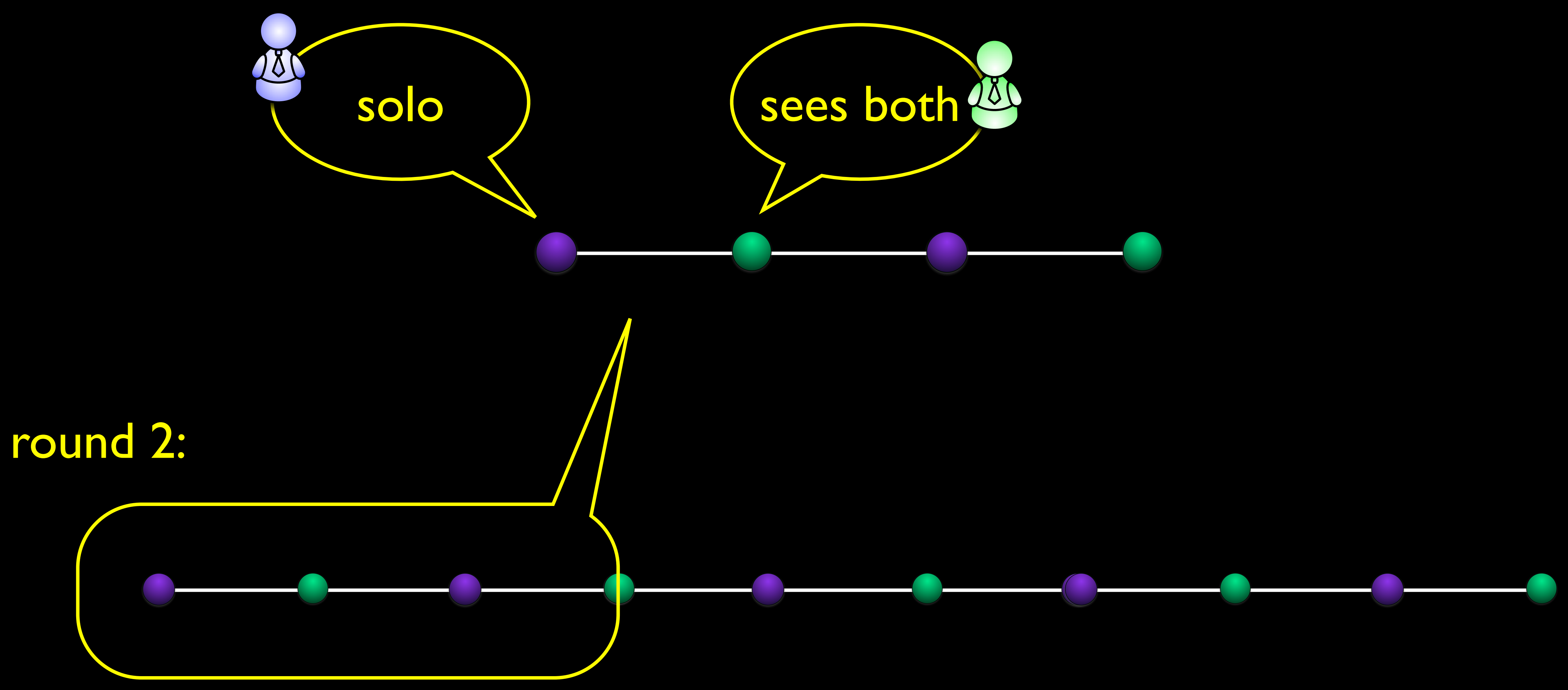
iterated runs

for each run in round 1 there are the same 3 runs in the next round

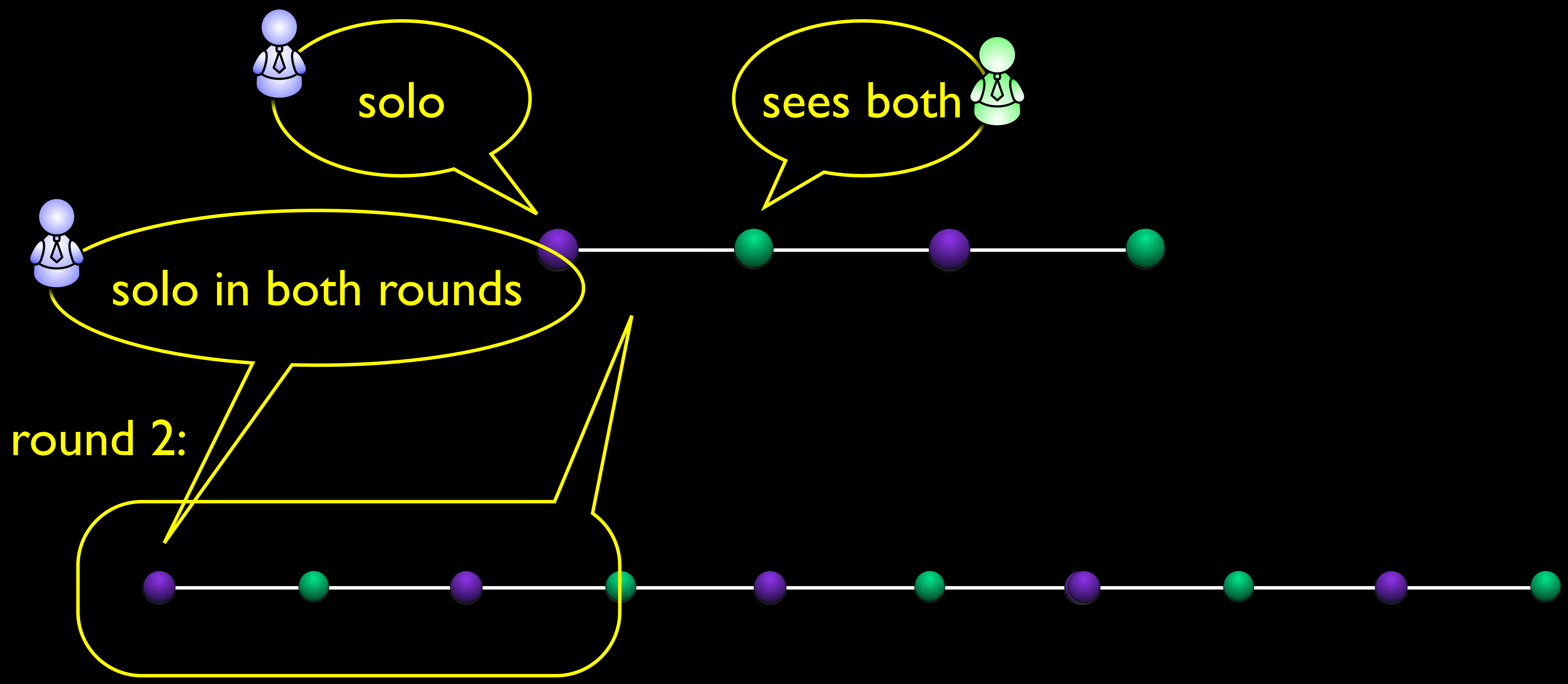
round 1:

round 2:

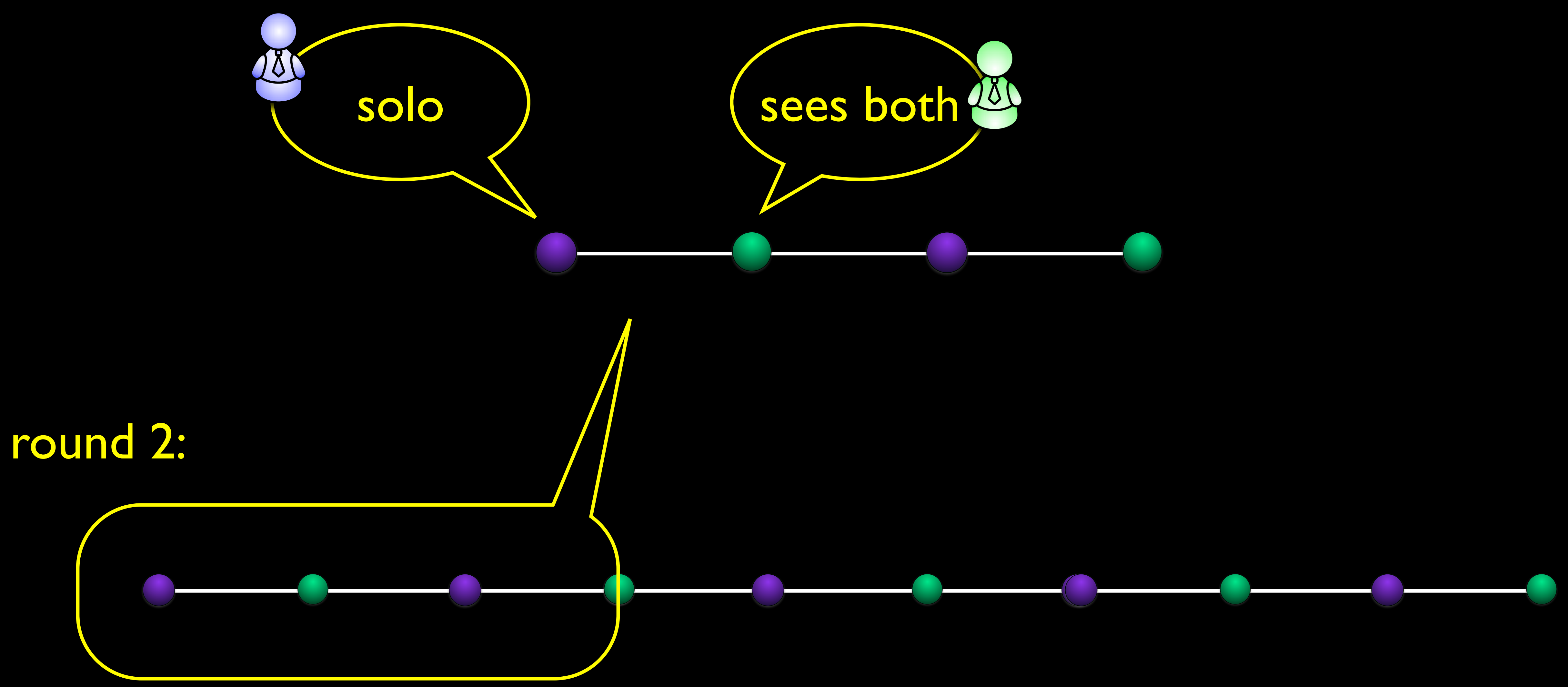
iterated runs



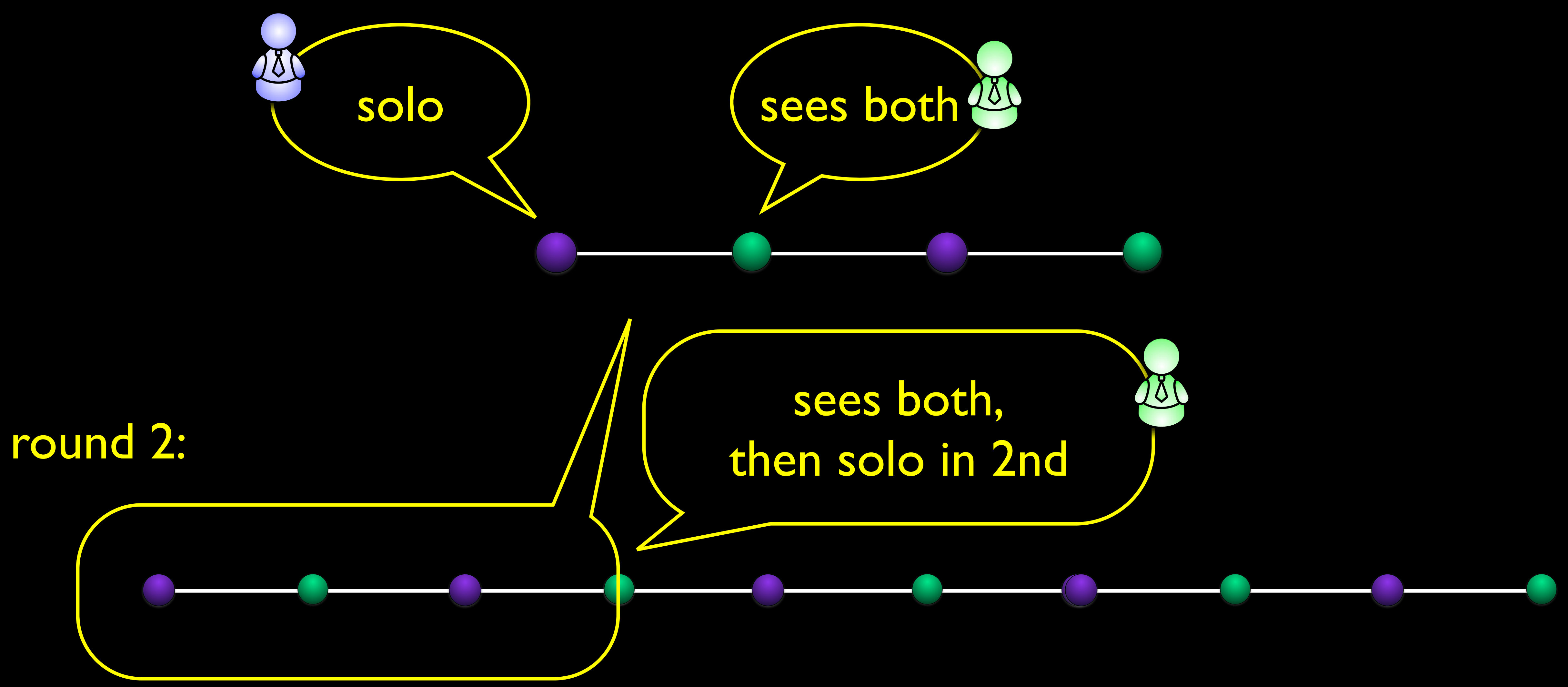
iterated runs



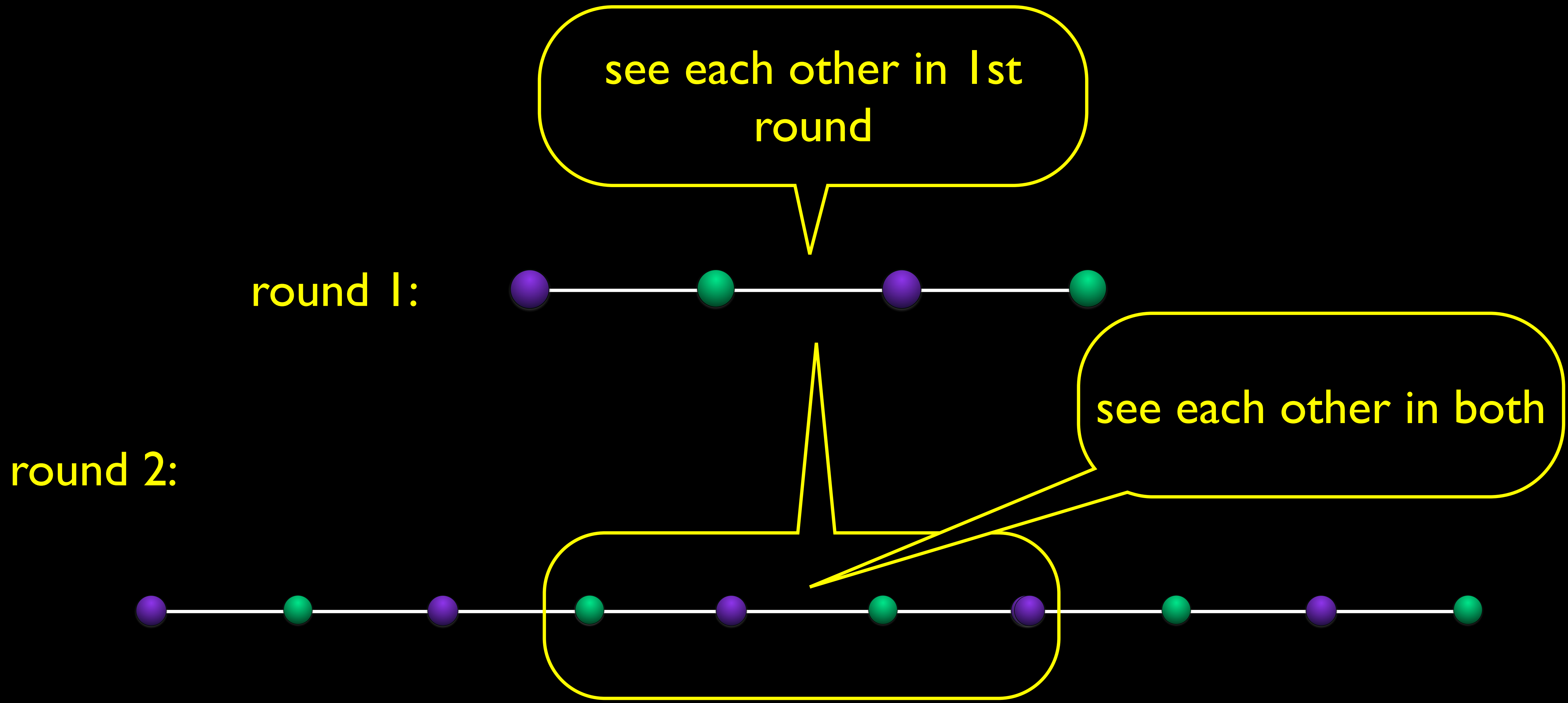
iterated runs



iterated runs



iterated runs

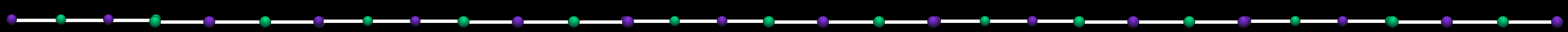


More rounds

round 1:

round 2:

round 3:



Theorem: protocol graph after *k* rounds

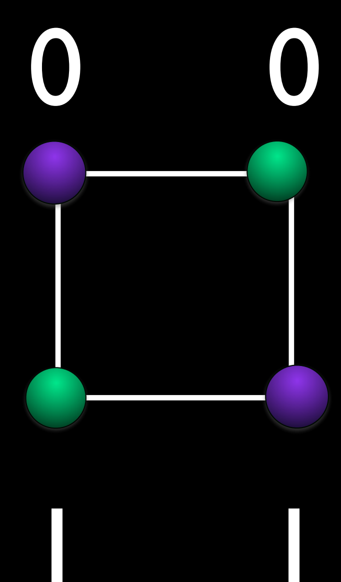
-longer
-but always connected

implications in terms of

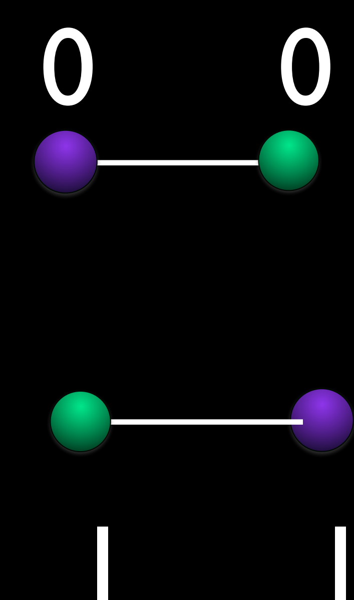
- task solvability
- complexity
- computability

representing tasks

binary consensus



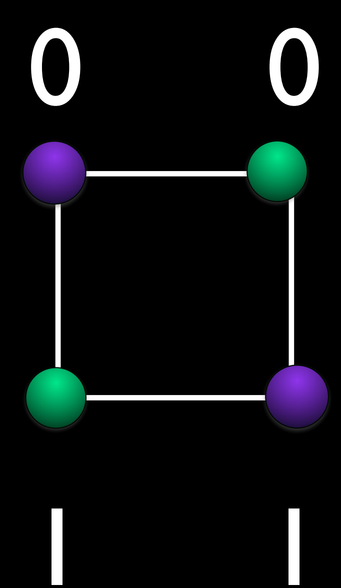
Input Graph



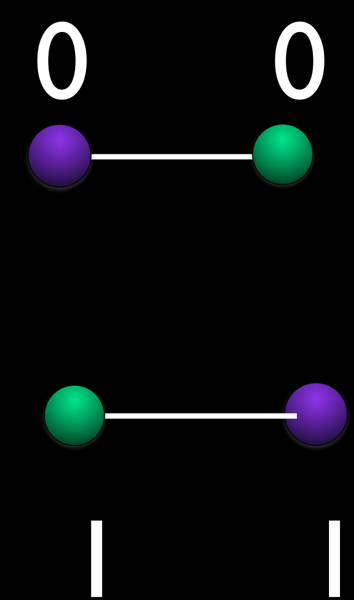
Output Graph

representing tasks

binary consensus



Input Graph



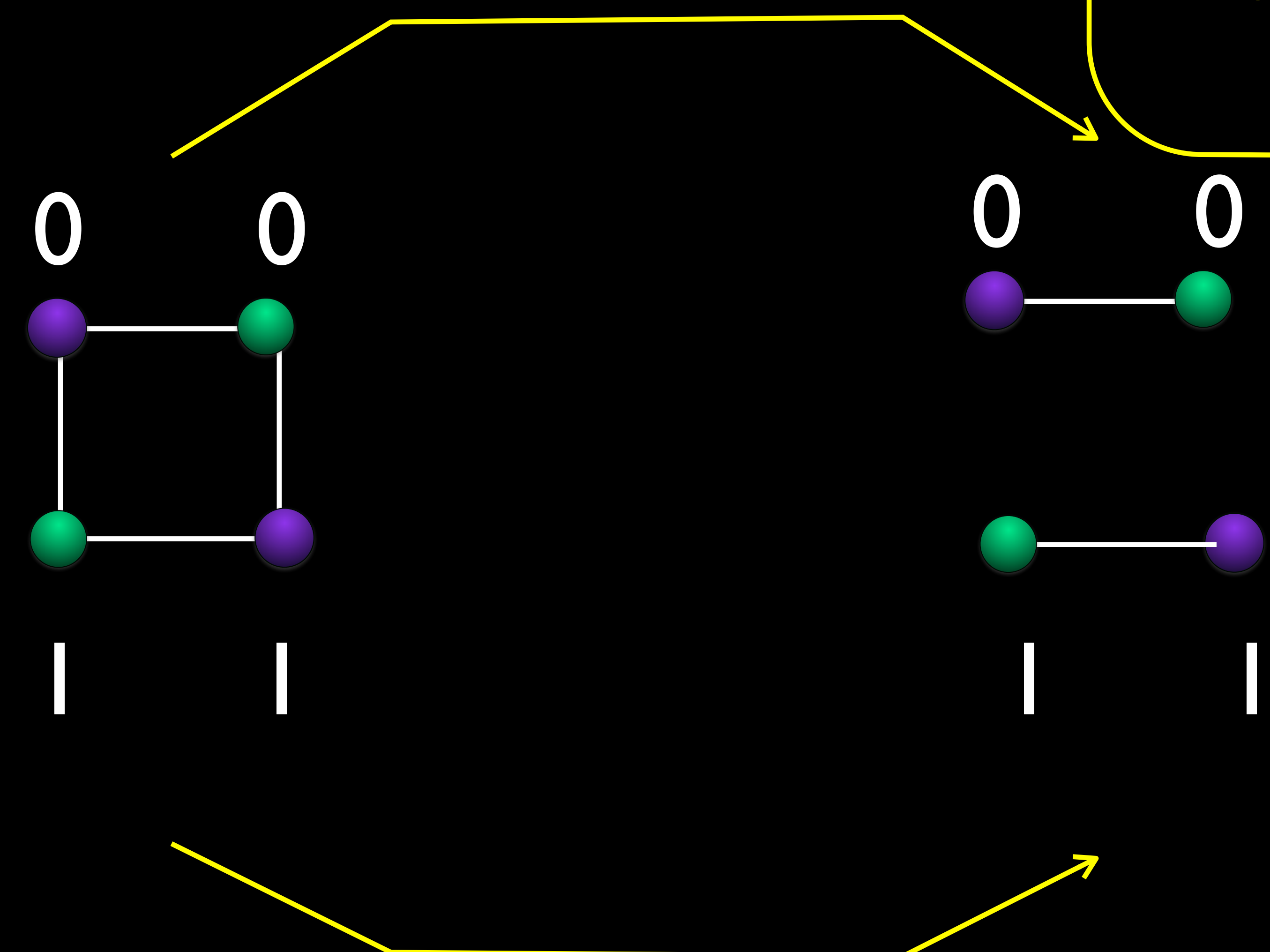
Output Graph

Input/output relation

representing tasks

binary consensus

start with same input
decide same output



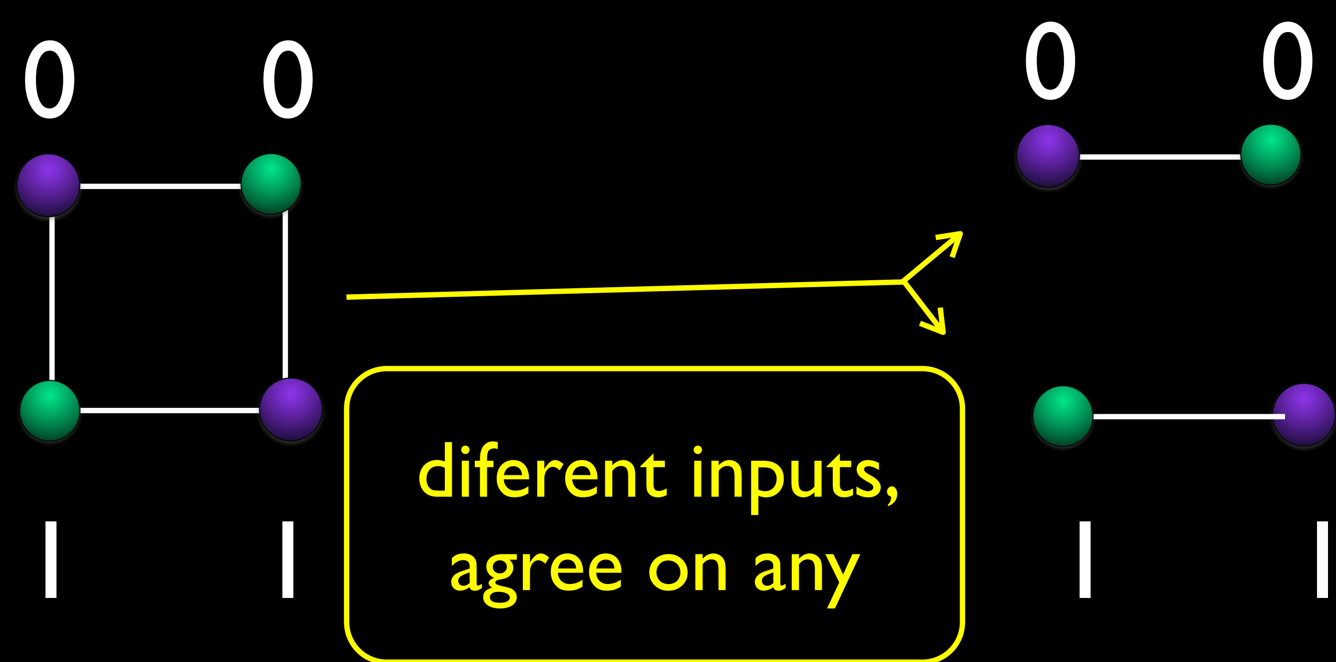
Input/output relation

Input Graph

Output Graph

representing tasks

binary consensus

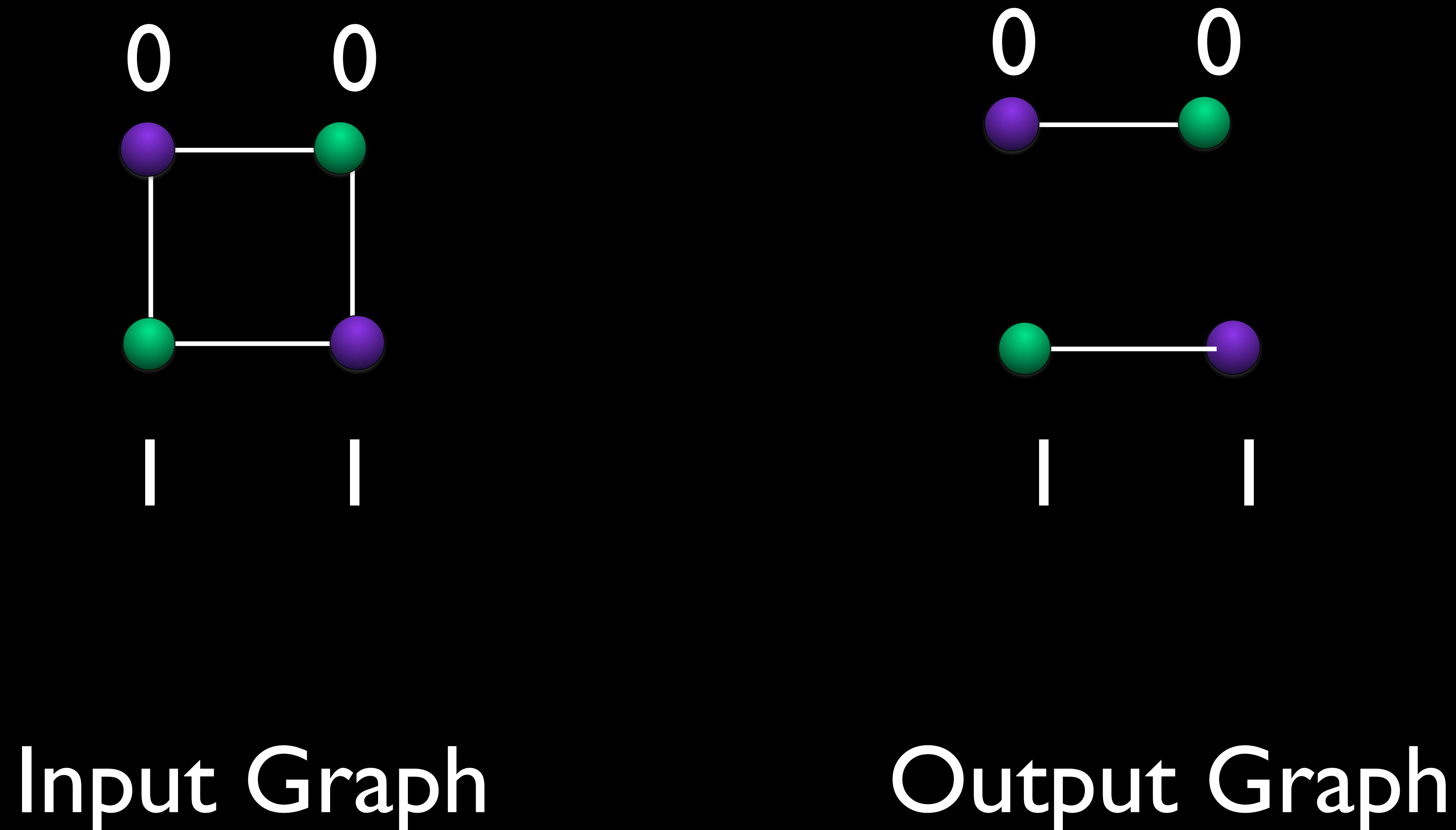


Input/output relation

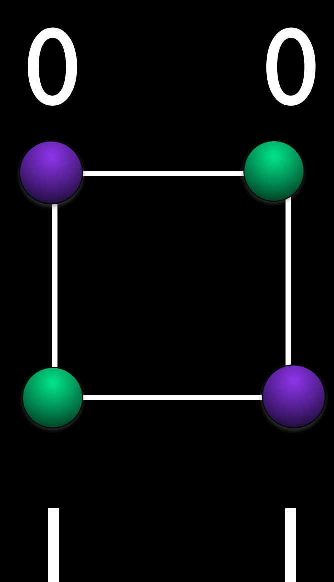
Input Graph

Output Graph

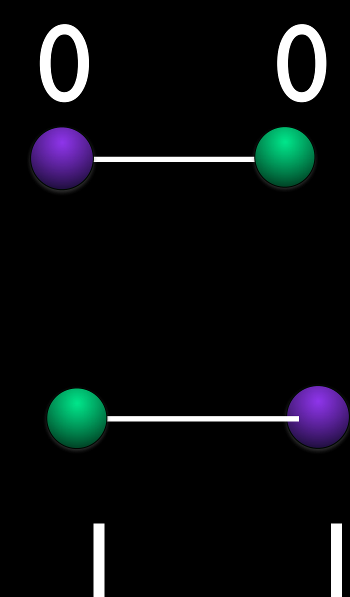
Binary consensus is not solvable due to connectivity



Binary consensus is not solvable due to connectivity



Input Graph

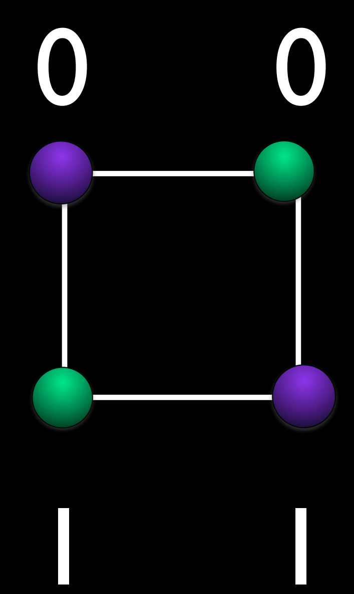


Output Graph

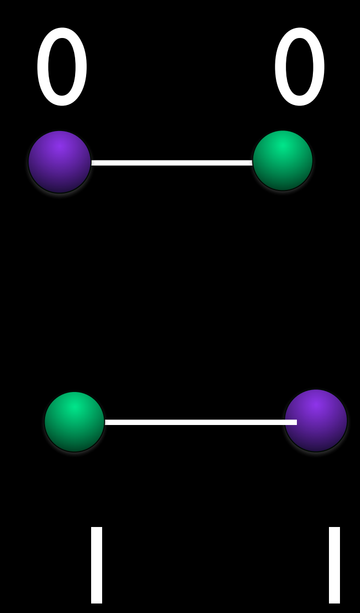
Input/output relation

Binary consensus is not solvable due to connectivity

Each edge is an initial configuration of the protocol



Input Graph

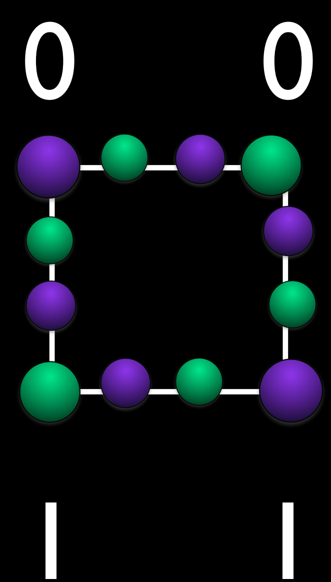


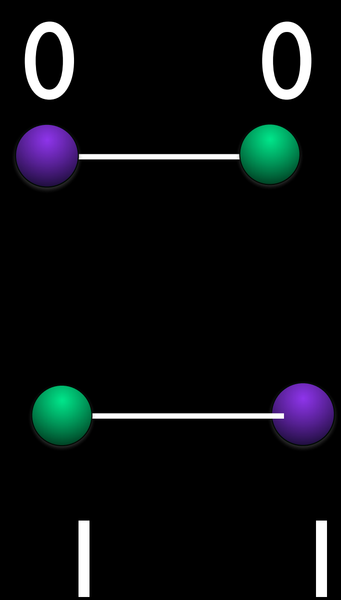
Output Graph

Input/output relation

Binary consensus is not solvable due to connectivity

subdivided after 1 round





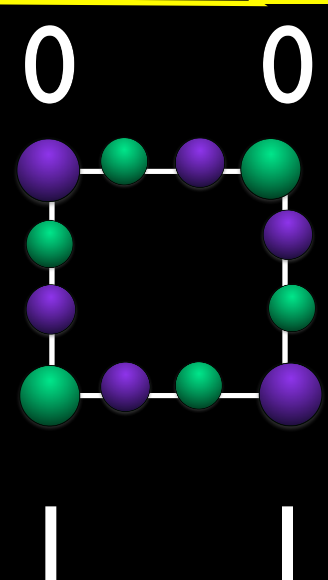
Input/output relation

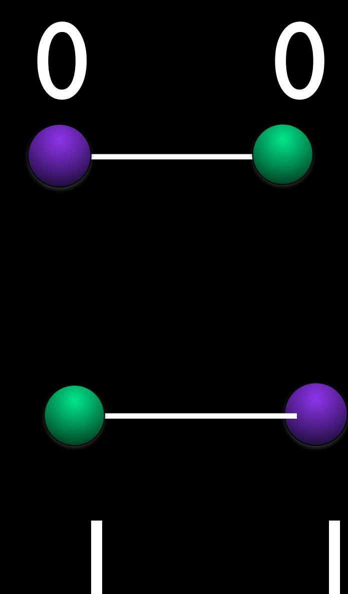
Input Graph

Output Graph

Binary consensus is not solvable due to connectivity

no solution in 1 round



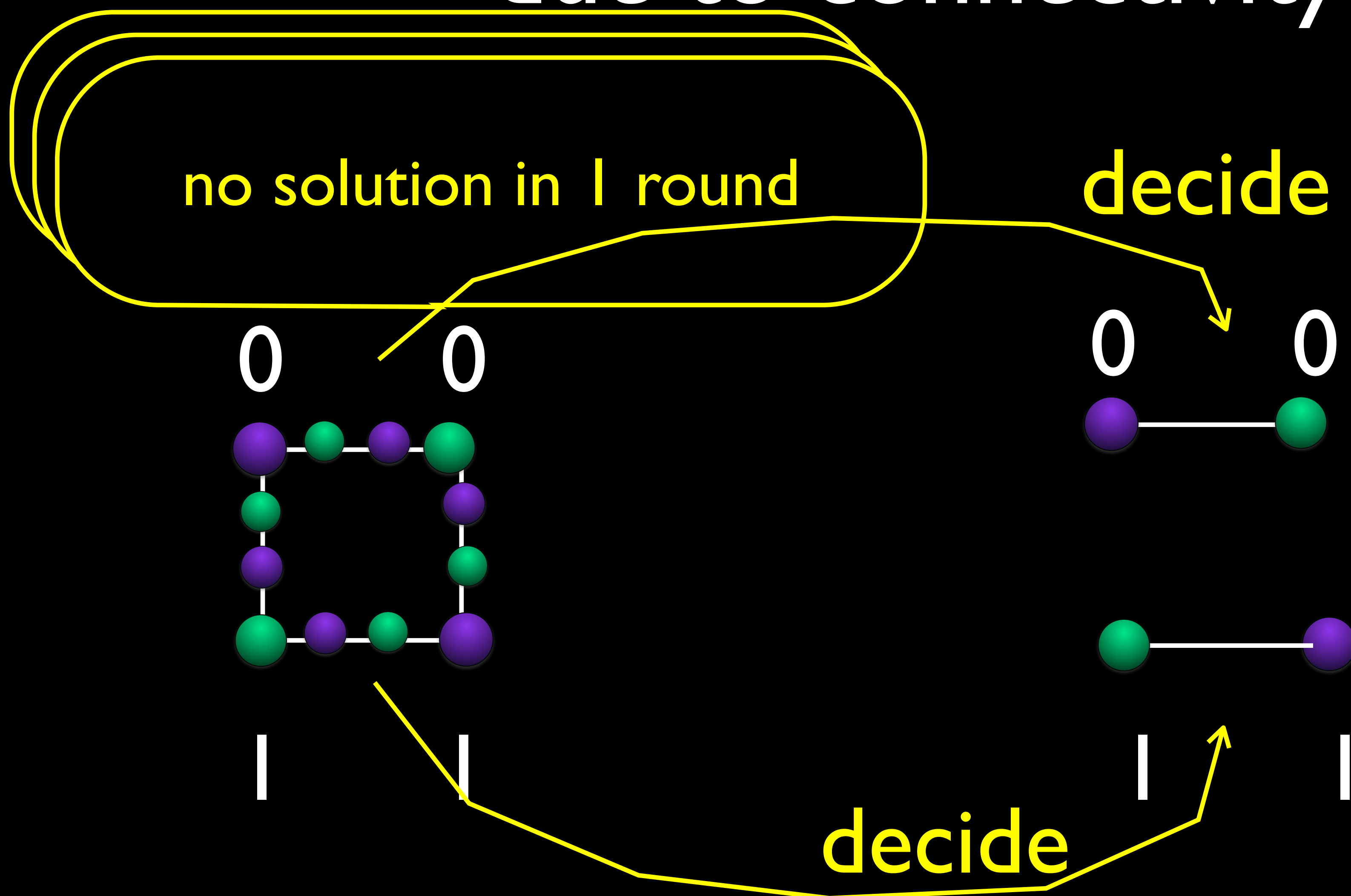


Input/output relation

Input Graph

Output Graph

Binary consensus is not solvable due to connectivity

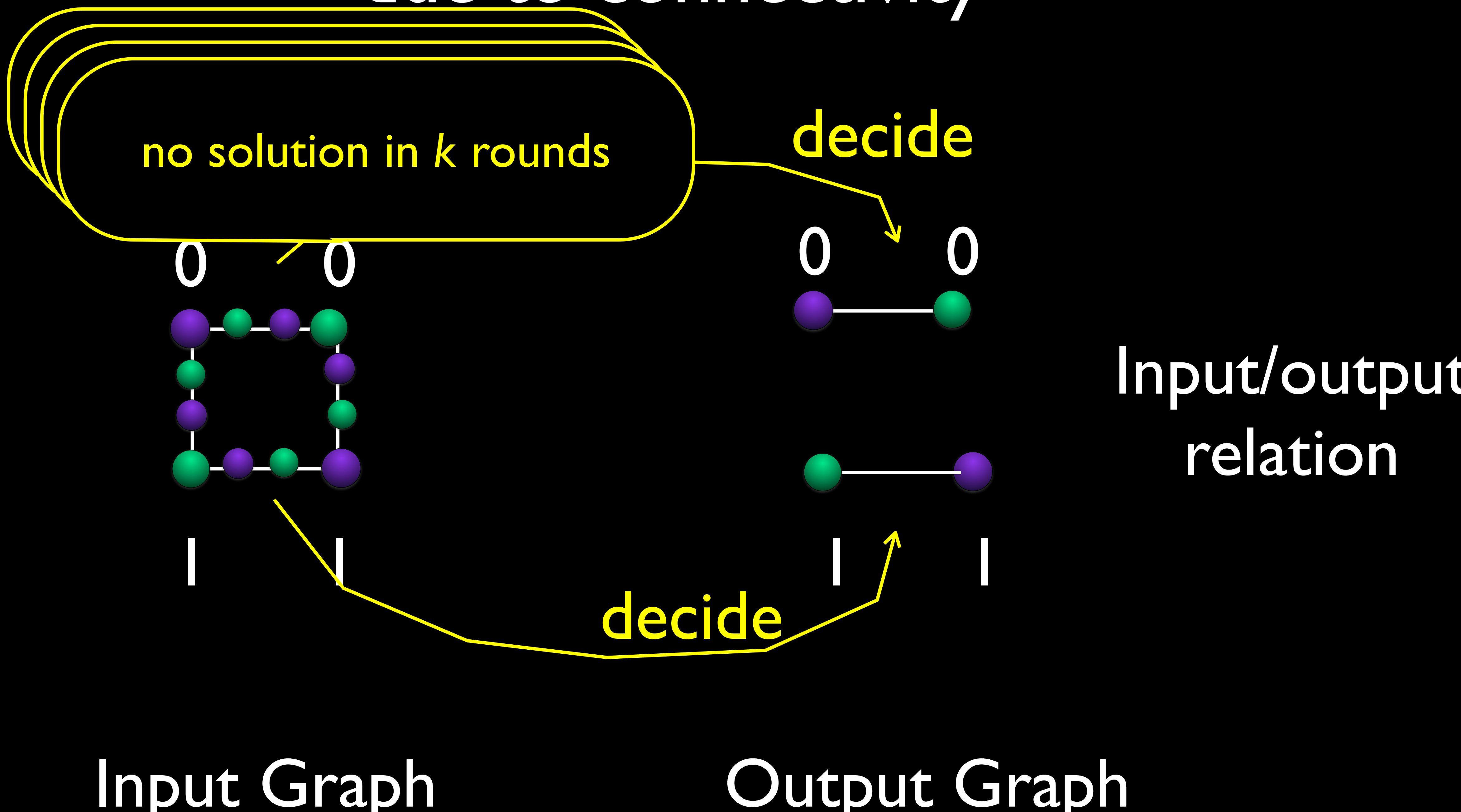


Input/output relation

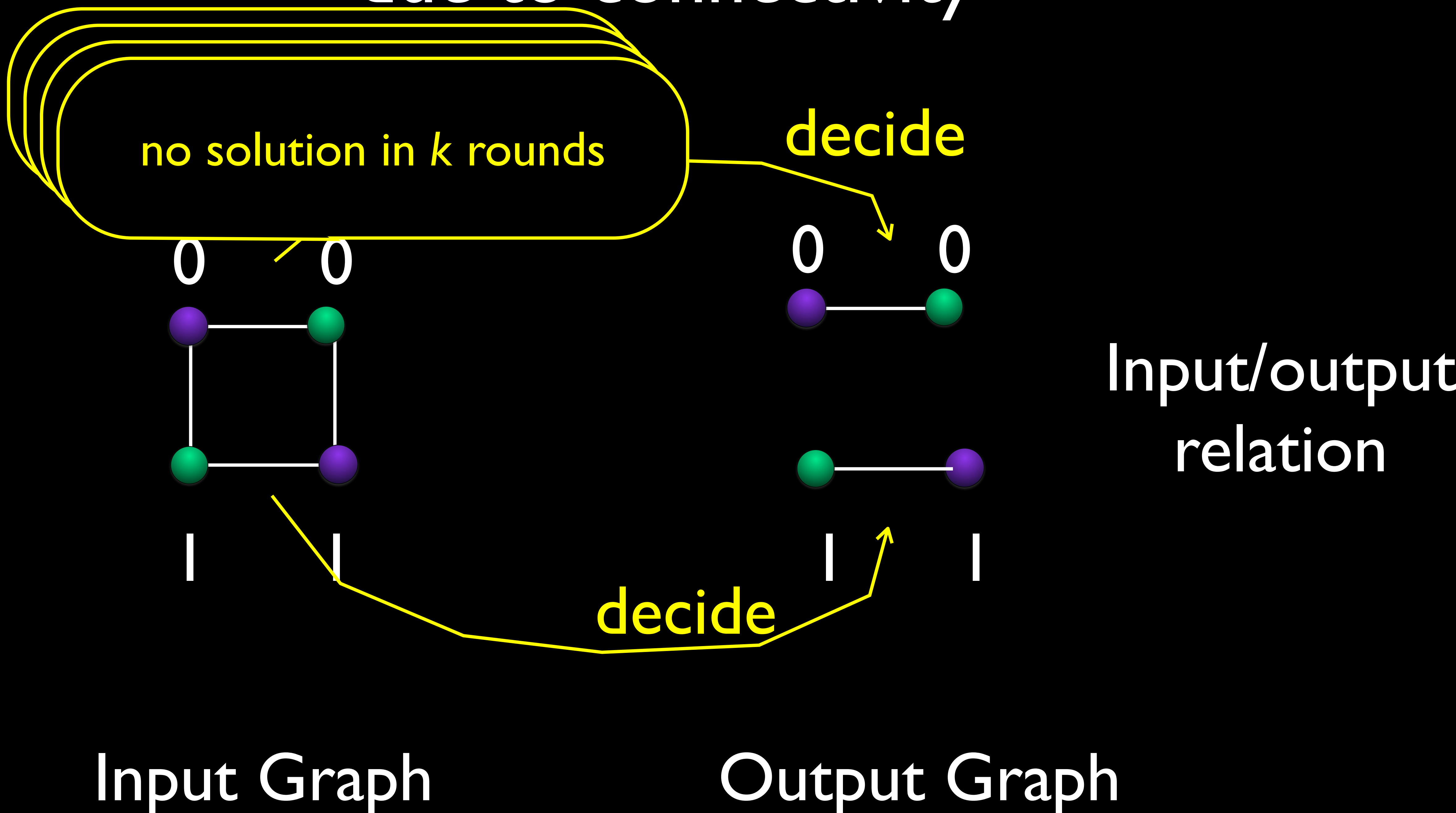
Input Graph

Output Graph

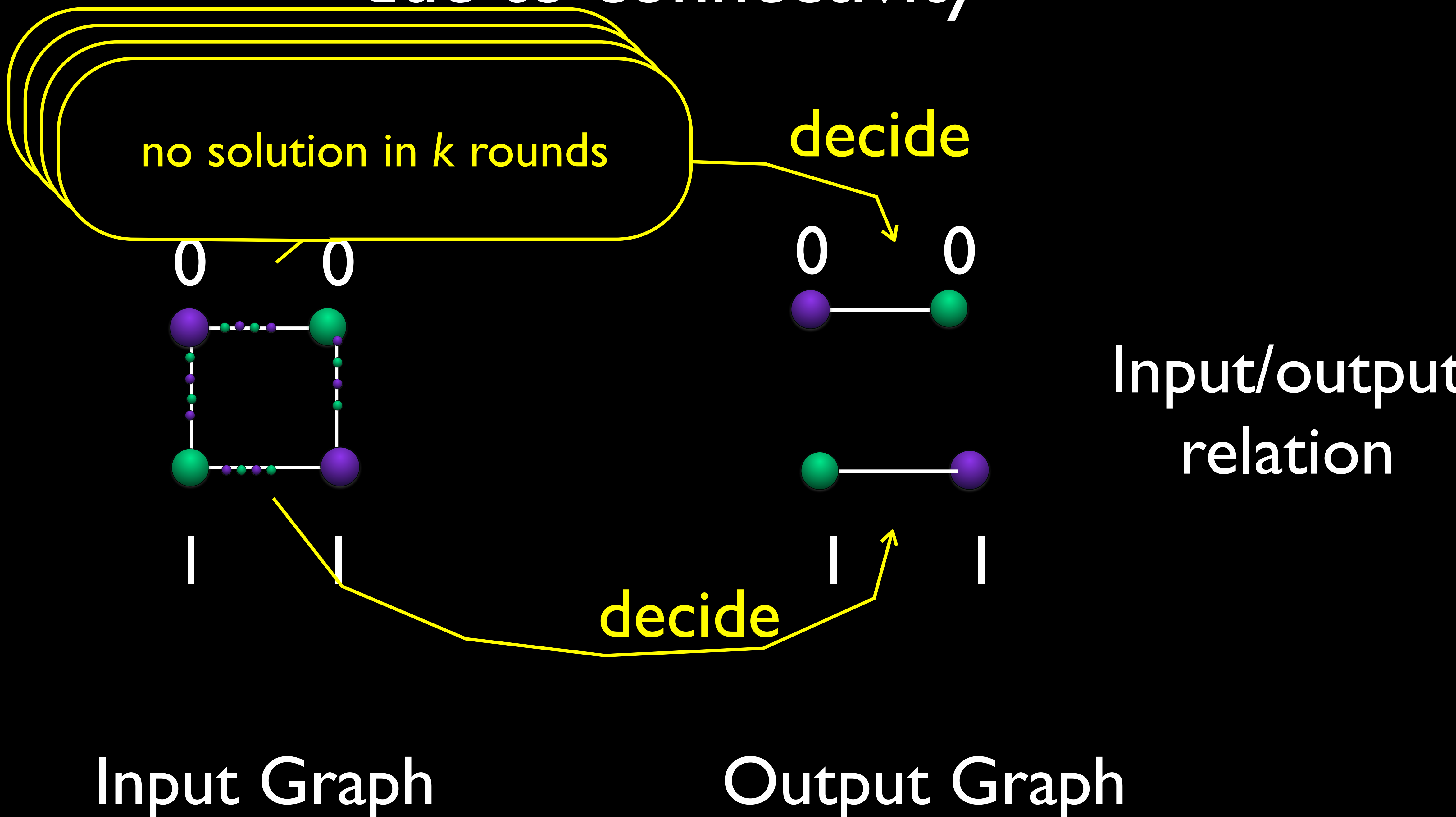
Binary consensus is not solvable due to connectivity



Binary consensus is not solvable due to connectivity



Binary consensus is not solvable due to connectivity



Runs for 2 processes

round 1:

round 2:

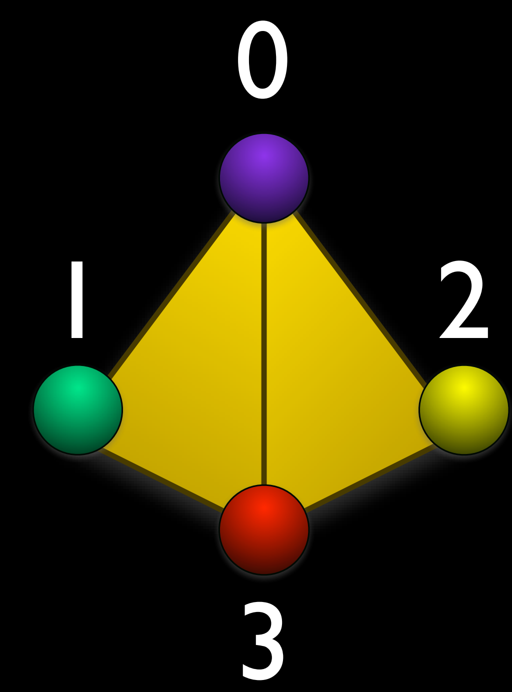
round 3:

Theorem: protocol graph after *k* rounds

- -longer
- -but always connected

Runs for *n* processes

- 4 local states in some execution
- 3-dim simplex
- e.g. inputs 0,1,2,3



Theorem: protocol complex after *k* rounds

- recursively subdivided
- but always *n*-connected

ContextFree SpiralTree

Conclusions

Conclusions

Conclusions

- In SSS‘2010 we present recursive algorithms for snapshots, immediate snapshots, renaming and swap

Conclusions

- In SSS‘2010 we present recursive algorithms for snapshots, immediate snapshots, renaming and swap
- linear, binary branching and multi-branching recursion

Conclusions

- In SSS'2010 we present recursive algorithms for snapshots, immediate snapshots, renaming and swap
- linear, binary branching and multi-branching recursion
- Recursion is useful:

Conclusions

- In SSS'2010 we present recursive algorithms for snapshots, immediate snapshots, renaming and swap
- linear, binary branching and multi-branching recursion
- Recursion is useful:
  - some new algorithms,

Conclusions

- In SSS‘2010 we present recursive algorithms for snapshots, immediate snapshots, renaming and swap
- linear, binary branching and multi-branching recursion
- Recursion is useful:
  - some new algorithms,
  - facilitates analysis

Conclusions

Conclusions

- Recursion is also interesting for lower bounds, due to recursive structure of the iterated models obtained

Conclusions

- Recursion is also interesting for lower bounds, due to recursive structure of the iterated models obtained
- In OPODIS‘2010 we show how to transform a distributed algorithm to iterated

Conclusions

- Recursion is also interesting for lower bounds, due to recursive structure of the iterated models obtained
- In OPODIS‘2010 we show how to transform a distributed algorithm to iterated
- A survey in LATIN‘2010 (LNCS 6034)

Conclusions

- Recursion is also interesting for lower bounds, due to recursive structure of the iterated models obtained
- In OPODIS'2010 we show how to transform a distributed algorithm to iterated
- A survey in LATIN'2010 (LNCS 6034)
- Connection to topology

Open questions

Open questions

- Can every distributed algorithm be written in a recursive form?

Open questions

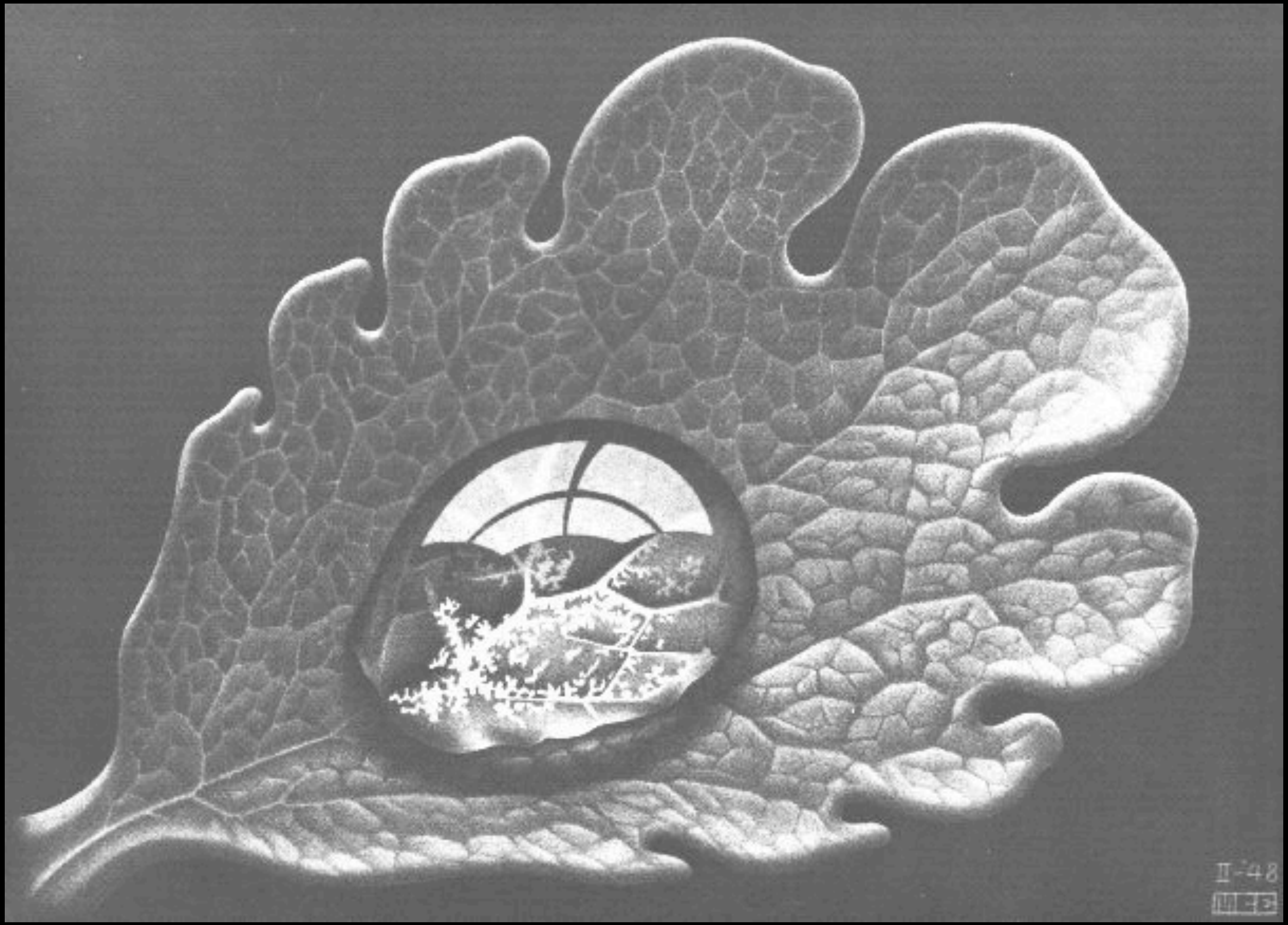
- Can every distributed algorithm be written in a recursive form?
- Our algorithms, based on splitters, have depth $O(n)$, and quadratic step complexity. Other type of recursive algorithms?

Open questions

- Can every distributed algorithm be written in a recursive form?
- Our algorithms, based on splitters, have depth $O(n)$, and quadratic step complexity. Other type of recursive algorithms?
- Programming languages for recursive algorithms?

Open questions

- Can every distributed algorithm be written in a recursive form?
- Our algorithms, based on splitters, have depth $O(n)$, and quadratic step complexity. Other type of recursive algorithms?
- Programming languages for recursive algorithms?
- Many other interesting question ....


II-'48

Thank you