# Vertex Coloring

**Partha Sarathi Mandal**
Department of Mathematics
IIT Guwahati

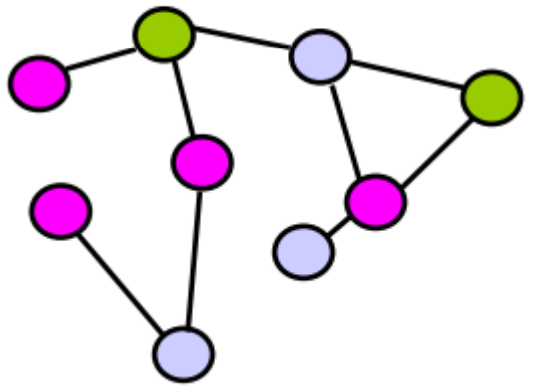
Thanks to

**Dr. Stefan Schmid** for the slides

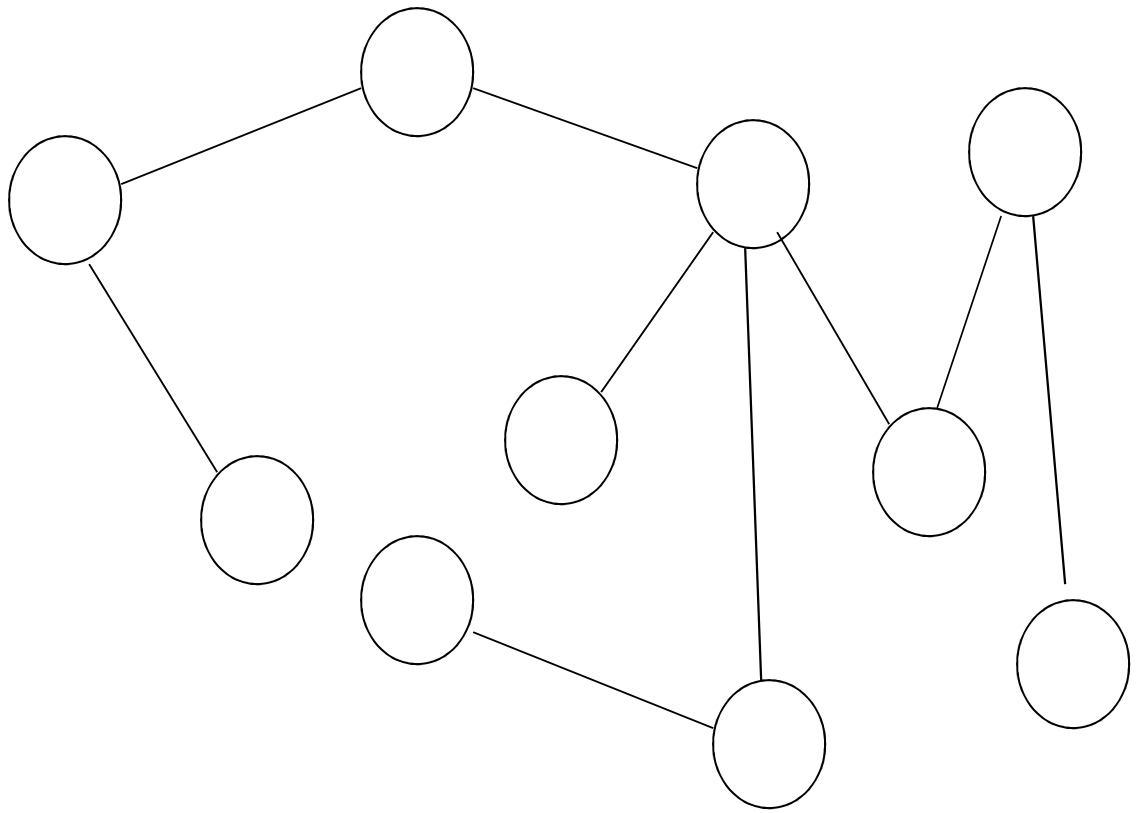
# Graph Coloring

## Vertex Coloring

Nodes should color themselves such that no adjacent nodes have same color – but minimize # colors!

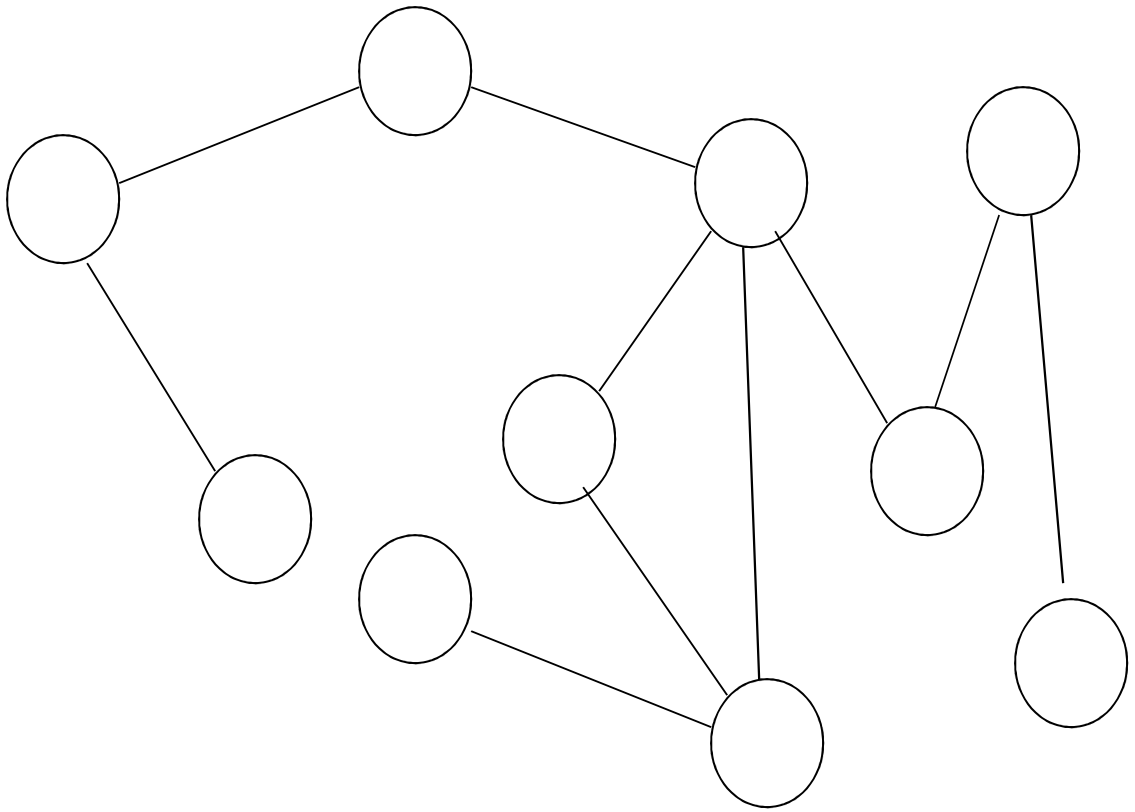


# How to color? Chromatic number?



Tree! Two colors enough...

# And now?



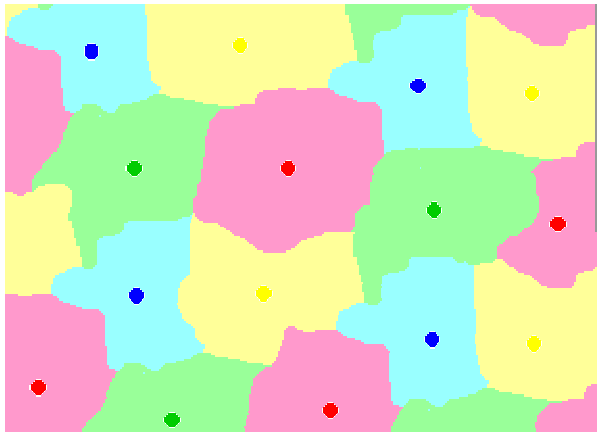
Three colors enough...

# Graph Coloring

Why color a network?

# Graph Coloring

**Medium access:** reuse frequencies in wireless networks at certain spatial distance such that there is “no“ interference.



**Break symmetries:** more generally...

Note: gives independent sets... How?

# Simple Coloring Algorithm? (Not distributed!)

## Greedy Sequential

while (uncolored vertices v left):
    color v with minimal color that does not conflict with neighbors

**Analysis?**
**# rounds/steps?**
**# colors?**

# Simple Coloring Algorithm? (Not distributed!)

## Greedy Sequential

while (uncolored vertices v left):
color v with minimal color that does not conflict with neighbors

**# steps**

At most n steps: walk through all nodes...

**# colors**

$\Delta+1$, where $\Delta$ is max degree.
Because: there is always a color free in $\{1, ..., \Delta+1\}$

**Note: many graphs can be colored with less colors!**
**Examples?**

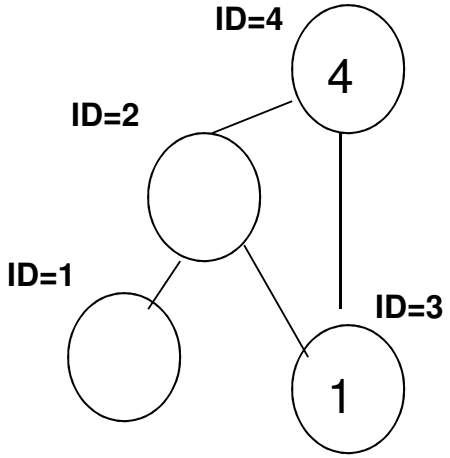
# How to do it in a distributed manner?

Now distributed!

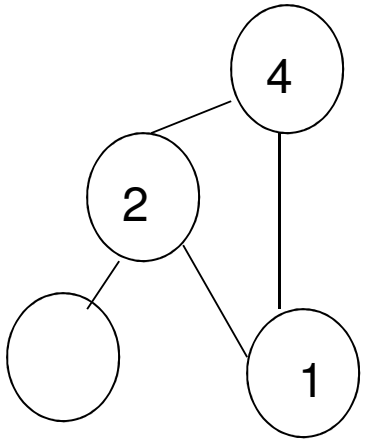
## First Free

Assume initial coloring (e.g., unique ID=color)

1. Each node uses smallest available color in neighborhood




Assume: two neighbors never choose color at the same time...



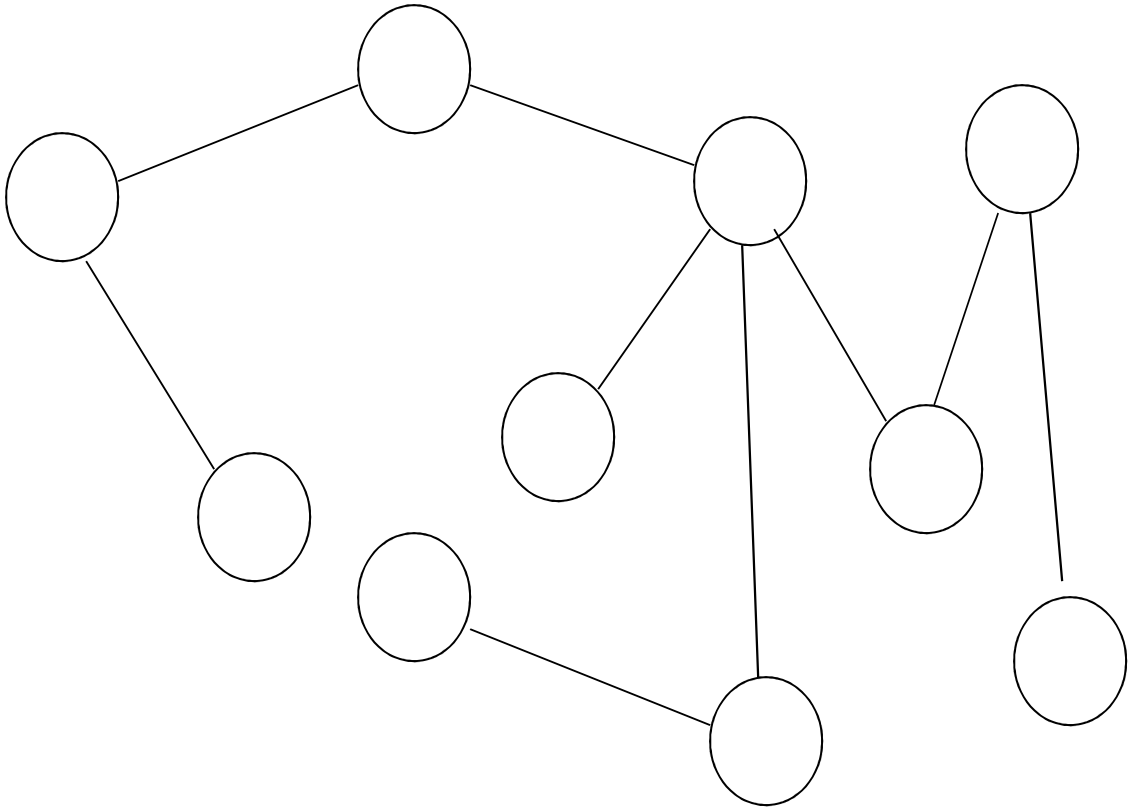

## Reduce

Initial coloring = IDs
Each node v:

1. v sends ID to neighbors (idea: sort neighbors!)
2. while (v has uncolored neighbor with higher ID)
    1. v sends “undecided“ to neighbors
3. v chooses free color using **First Free**
4. v sends decision to neighbors

**Analysis? Not parallel!**

**Let us focus on trees now....**
**Chromatic number?**
**Algo?**



## Slow Tree

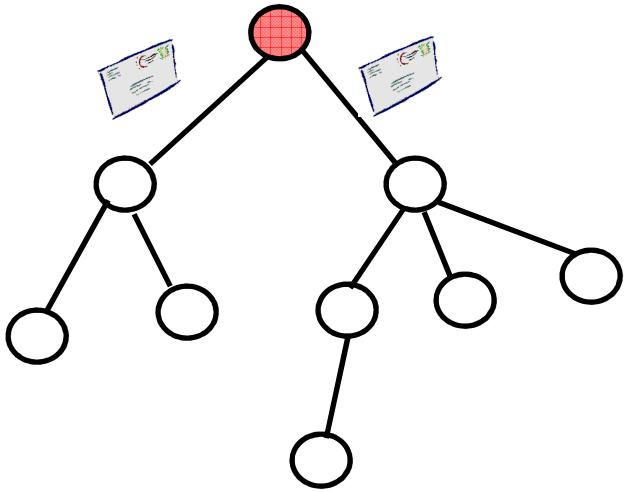
1. Color root 0, send to kids

Each node v does the following:

- Receive message x from parent
- Choose color y=1-x
- Send y to kids

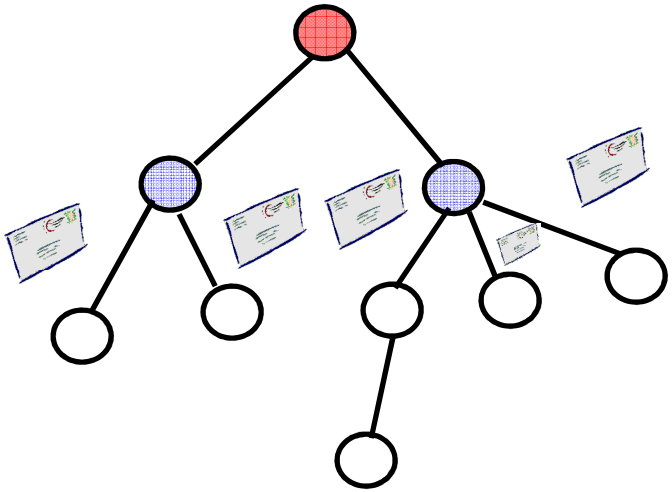
# Slow Tree

Two colors suffice: root sends binary message down...



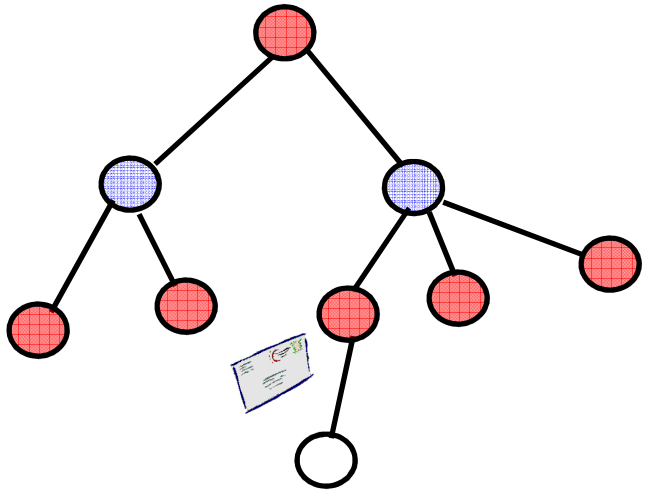
# Slow Tree

Two colors suffice: root sends binary message down...



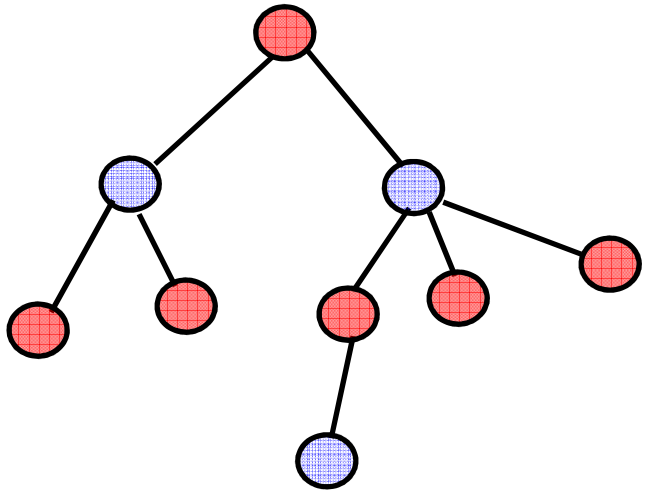
# Slow Tree

Two colors suffice: root sends binary message down...



# Slow Tree

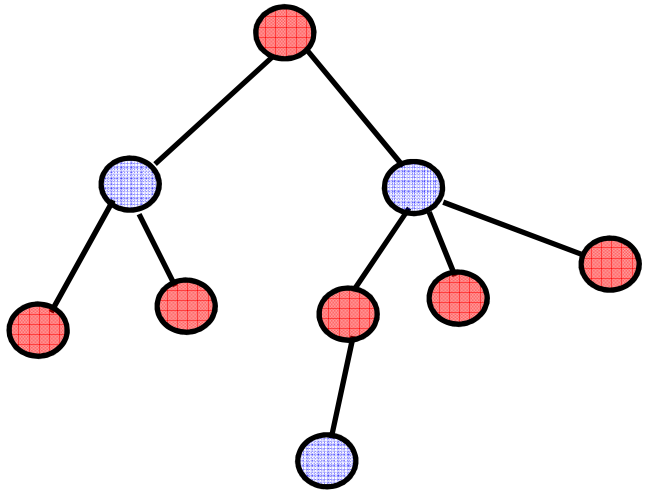
Two colors suffice: root sends binary message down...



Time complexity?
Message complexity?
Local compuations?
Synchronous or asynchronous?

# Slow Tree

Two colors suffice: root sends binary message down...



Time complexity? depth · n

Message complexity? n-1

Local compuations?

Synchronous or asynchronous? both!

# Discussion

Time complexity? depth · n

Message complexity? n-1

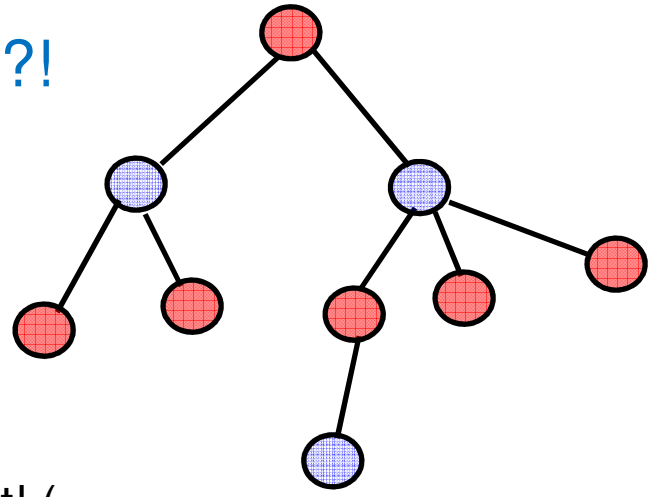
Local compuations?

Synchronous or asynchronous? both!

**Can we do better?**

# Local Vertex Coloring for Tree?

Can we do faster than diameter of tree?!



Yes! With **constant** number of colors in

**log*(n) time!!**

One of the fastest non-constant time algos that exist! (...
besides inverse Ackermann function or so)

(log = divide by two, loglog = ?, log* = ?)

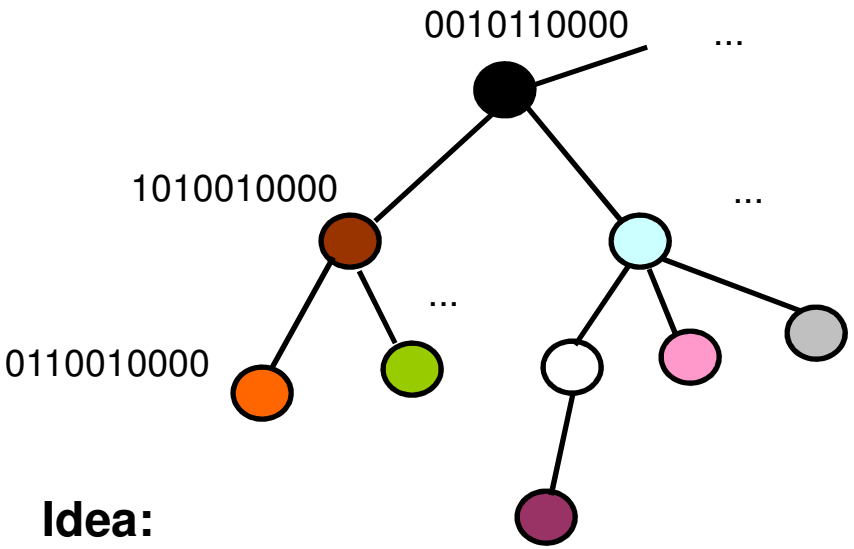
**log* (# atoms in universe) = 5**

Why is this good? If something happens (dynamic network),
back to good state in a sec!

There is a lower bound of log-star too, so that's optimal!

# How does it work?

Initially: each node has unique log(n)-bit ID = legal coloring
(interpret ID as color => n colors)




**Idea:**

root should have label 0 (fixed)

in each step: send ID to $c_v$ to all children;

receive $c_p$ from parent and interpret as little-endian bit string:
$c_p = c(k)...c(0)$

let i be smallest index where $c_v$ and $c_p$ differ

set new $c_v$ = i (as bit string) || $c_v(i)$

until $c_v \in \{0,1,2,...,5\}$ (at most 6 colors)

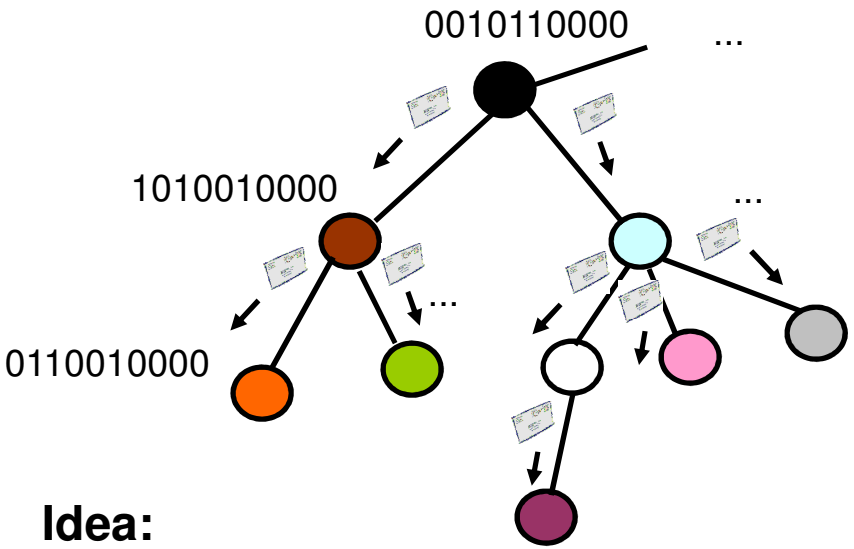
# 6-Colors

Assume legal initial coloring
Root sets itself color 0
Each other node v does (in parallel):

1. Send $c_v$ to kids
2. Repeat (until $c_w \in \{0,...,5\}$ for all w):
    1. Receive $c_p$ from parent
    2. Interpret $c_v/c_p$ as little-endian bitstrings c(k)...c(1)c(0)
    3. Let i be smallest index where $c_v$ and $c_p$ differ
    4. New label is: **$i||c_v(i)$**
    5. Send $c_v$ to kids

# How does it work?

Initially: each node has unique log(n)-bit ID = legal coloring
(interpret ID as color => n colors)




**Round 1**

**Idea:**

root should have label 0 (fixed)

in each step: send ID to $c_v$ to all children;

receive $c_p$ from parent and interpret as little-endian bit string:
$c_p = c(k)...c(0)$

let i be smallest index where $c_v$ and $c_p$ differ

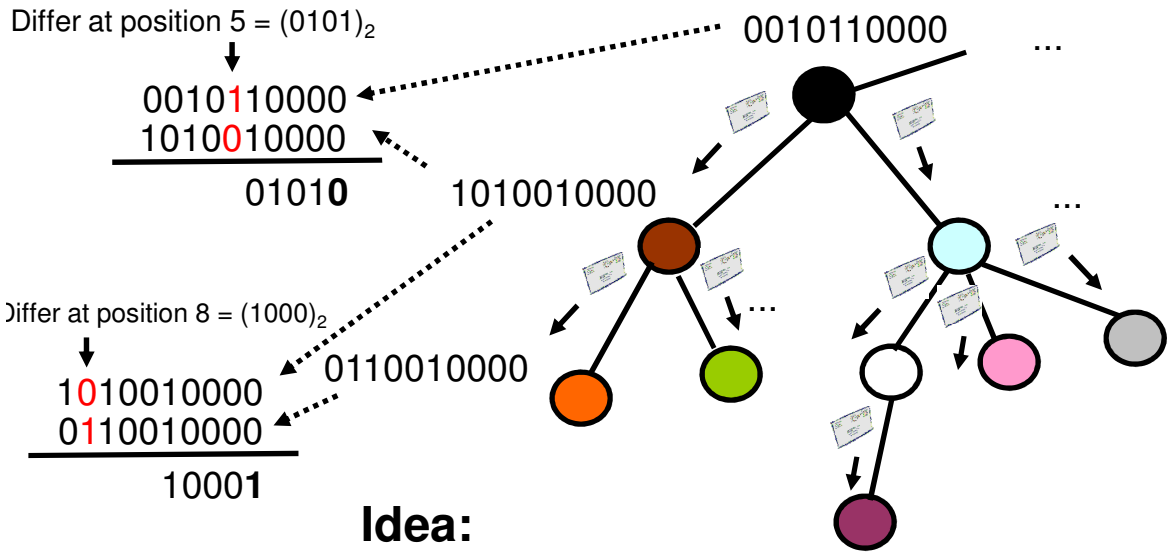
set new $c_v$ = i (as bit string) || $c_v(i)$

until $c_v \in \{0,1,2,...,5\}$ (at most 6 colors)

# How does it work?

Initially: each node has unique log(n)-bit ID = legal coloring

(interpret ID as color => n colors)




**Round 1**

**Idea:**

root should have label 0 (fixed)

in each step: send ID to $c_v$ to all children;

receive $c_p$ from parent and interpret as little-endian bit string:
$c_p = c(k)...c(0)$

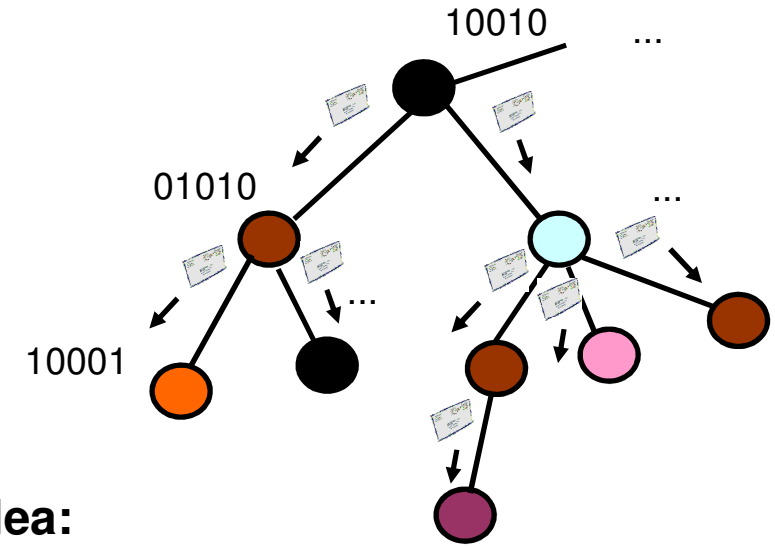
let i be smallest index where $c_v$ and $c_p$ differ

set new $c_v$ = i (as bit string) || $c_v(i)$

until $c_v \in \{0,1,2,...,5\}$ (at most 6 colors)

# How does it work?

Initially: each node has unique log(n)-bit ID = legal coloring
(interpret ID as color => n colors)




**Round 2**

**Idea:**

root should have label 0 (fixed)

in each step: send ID to $c_v$ to all children;

receive $c_p$ from parent and interpret as little-endian bit string:
$c_p = c(k)...c(0)$

let i be smallest index where $c_v$ and $c_p$ differ

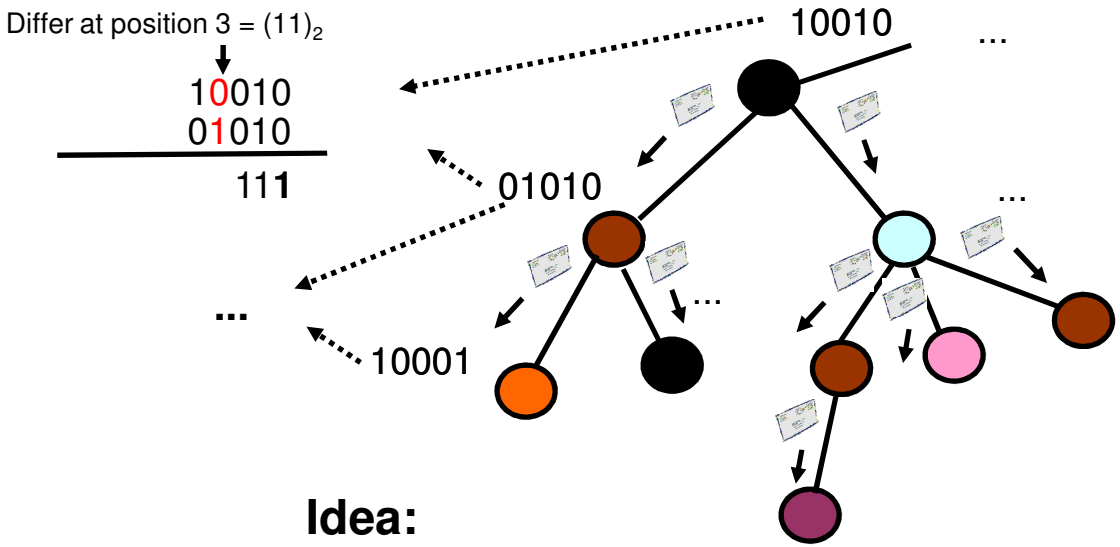
set new $c_v$ = i (as bit string) || $c_v(i)$

until $c_v \in \{0,1,2,...,5\}$ (at most 6 colors)

# How does it work?

Initially: each node has unique log(n)-bit ID = legal coloring

(interpret ID as color => n colors)




# Round 2

**Idea:**

root should have label 0 (fixed)

in each step: send ID to $c_v$ to all children;

receive $c_p$ from parent and interpret as little-endian bit string:
$c_p = c(k)...c(0)$

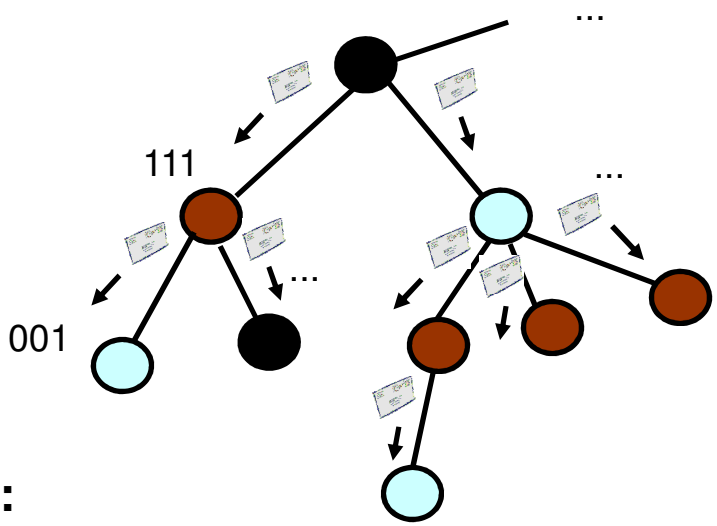
let i be smallest index where $c_v$ and $c_p$ differ

set new $c_v$ = i (as bit string) || $c_v(i)$

until $c_v \in \{0,1,2,...,5\}$ (at most 6 colors)

# How does it work?

Initially: each node has unique log(n)-bit ID = legal coloring
(interpret ID as color => n colors)




**Round 3, etc.**

**Idea:**

root should have label 0 (fixed)

in each step: send ID to $c_v$ to all children;

receive $c_p$ from parent and interpret as little-endian bit string:
$c_p = c(k)...c(0)$

let i be smallest index where $c_v$ and $c_p$ differ

set new $c_v$ = i (as bit string) || $c_v(i)$

until $c_v \in \{0,1,2,...,5\}$ (at most 6 colors)

# *Why* does it work?

## Why is this log* time?!

Idea: In each round, the size of the ID (and hence the number of colors) is reduced by a log factor: To index the bit where two labels of size n bits differ, log(n) bits are needed! Plus the one bit that is appended...

## Why is this a valid vertex coloring?!

Idea: During the entire execution, adjacent nodes always have different colors (invariant!) because: IDs always differ as new label is index of difference to parent plus own bit there (if parent would differ at same location as grand parent, at least the last bit would be different).

## Why $c_w \in \{0,...,5\}$?! Why not more or less?

# *Why* does it work?

## Why is this log* time?!

Idea: In each round, the size of the ID (and hence the number of colors) is reduced by a log factor: To index the bit where two labels of size n bits differ, log(n) bits are needed! Plus the one bit that is appended...

## Why is this a valid vertex coloring?!

Idea: During the entire execution, adjacent nodes always have different colors (invariant!) because: IDs always differ as new label is index of difference to parent plus own bit there (if parent would differ at same location as grand parent, at least the last bit would be different).

## Why $c_w \in \{0,...,5\}$?! Why not more or less?

Idea: {0,1,2,3} does not work, as two bits are required to address index where they differ, plus adding the "**difference-bit**" gives more than two bits...

Idea: {0,1,2,...,7} works, as $7=(111)_2$ can be described with 3 bits, and to address index (0,1,2) requires two bits, plus one "**difference-bit**" gives three again.

Moreover: colors 110 (for color "**6**") and 111 (for color "**7**") are not needed, as we can do another round! (IDs of three bits can only differ at positions 00 (for "**0**"), 01 (for "**1**"), 10 (for "**2**")

# Everything super?

## When can I terminate?

Not a local algorithm like this! Node cannot know when *all* other nodes have colors in that range!

Kid should not stop before parent stops! Solution: wait until parent is finished?

No way, this takes linear time in tree depth!

Ideas?

If nodes know n, they can stop after the (deterministic) execution time...

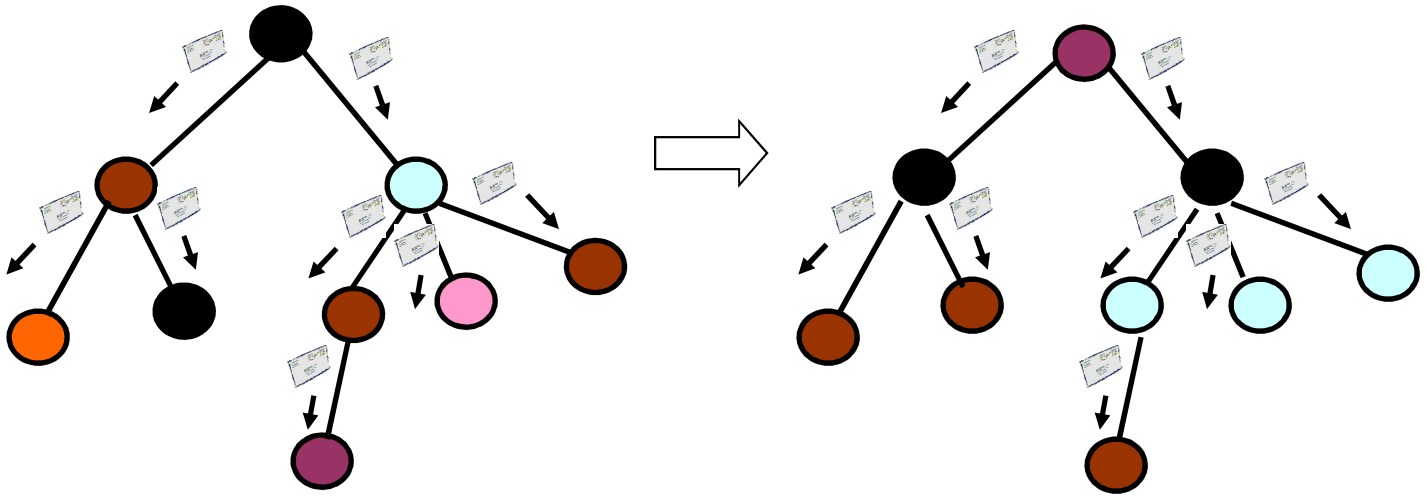
Other ideas? Maybe an exercise...

## Six colors is good: but we know that tree can be colored with two only!

How can we improve coloring quickly?

## Shift Down

Root chooses a new (different) color from {0; 1; 2}
Each node v concurrently does:
    recolor v with color of parent



Property?
Preserves coloring legality!
Siblings become monochromatic!
(Make siblings “independent“.)

## 6-to-3

Each other node v does (in parallel):

1. Run "**6-Colors**" for log*(n) rounds
2. For x=5,4,3:
   1. Perform **Shift Down**
   2. If ($c_v$=x) choose new color $c_v \in \{0,1,2\}$ according "**first free**" principle

Why still log*?

Rest is fast....

Why {3,4,5} recoloring not in same step?

Make sure coloring remains legal....

Cancel remaining colors one at a time

(nodes of same color independent)!

Why does it work?

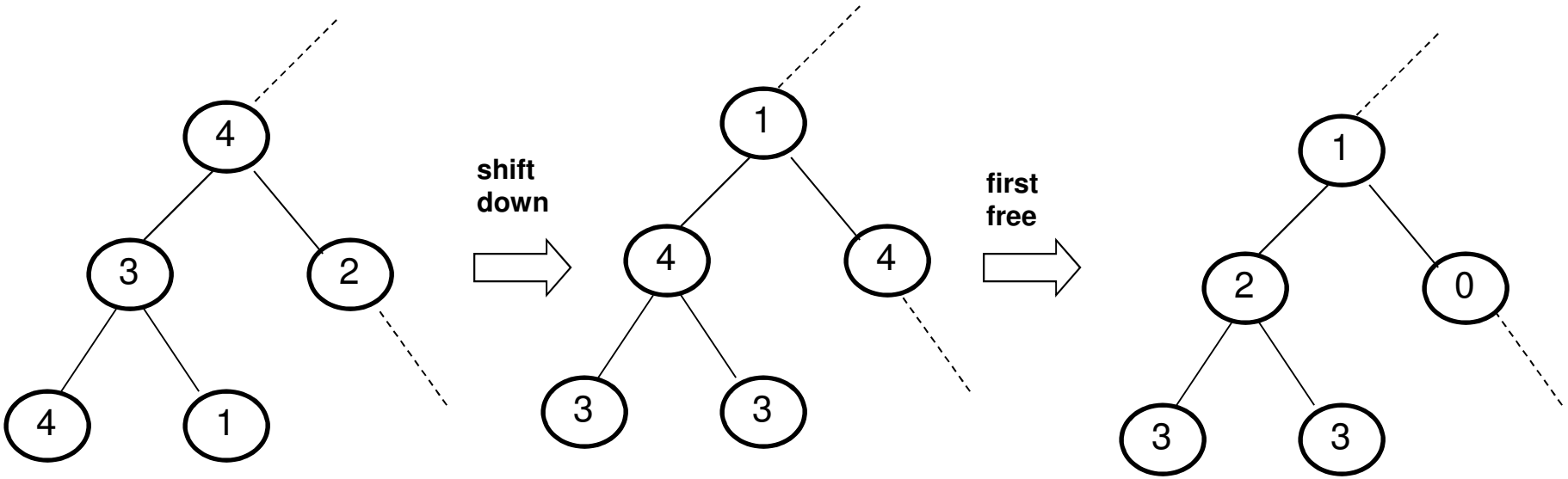
One of the three colors must be free!

(Need only two colors in tree, and due to shift down, one color is occupied by parent, one by children!)

We only recolor nodes simultaneously which are not adjacent.

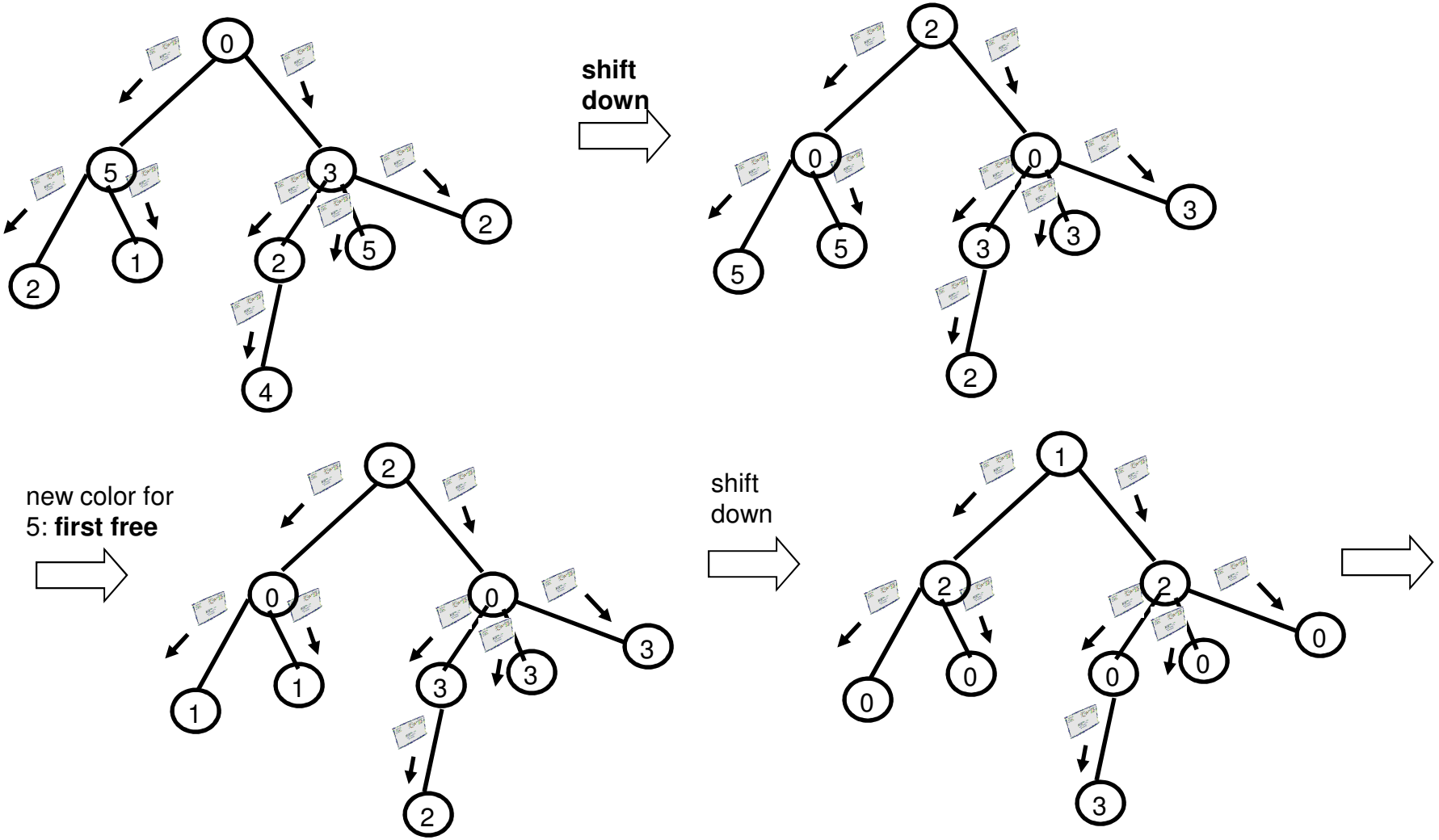
And afterwards no higher color is left...

# Example: Shift Down + Drop Color 4




**Siblings no longer have same color => must do shift down again first!**

# Example: 6-to-3



# Discussion

**Can we reduce to 2 colors?**

Not without increasing runtime significantly!
(Linear time, more than exponentially worse!)

**Other topologies?**

Yes, similar ideas to $O(\Delta)$-color general graphs
with constant degree $\Delta$ in log* time!
How?

**Lower bounds?**

Yes. ☺
In particular, runtime of our algorithm is asymptotically optimal.