GIAN Course on Distributed Network Algorithms

# The Power of Locality

## Case Study: Graph Coloring

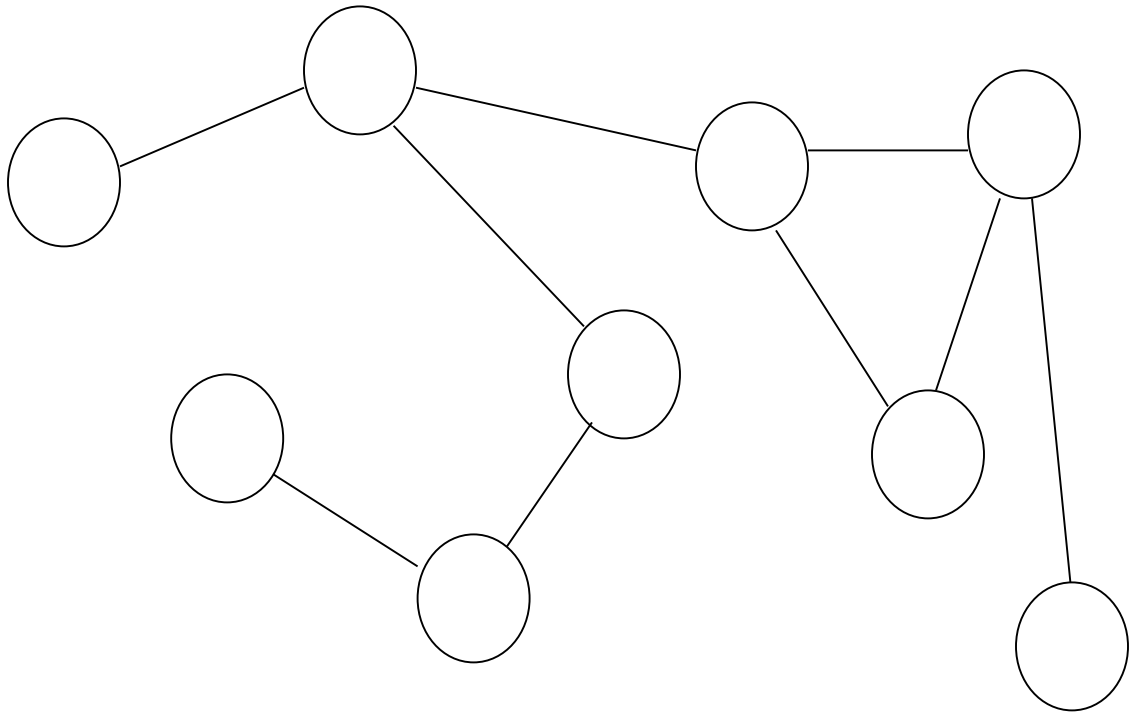
GIAN Course on Distributed Network Algorithms

# The Power of Locality

## Case Study: Graph Coloring

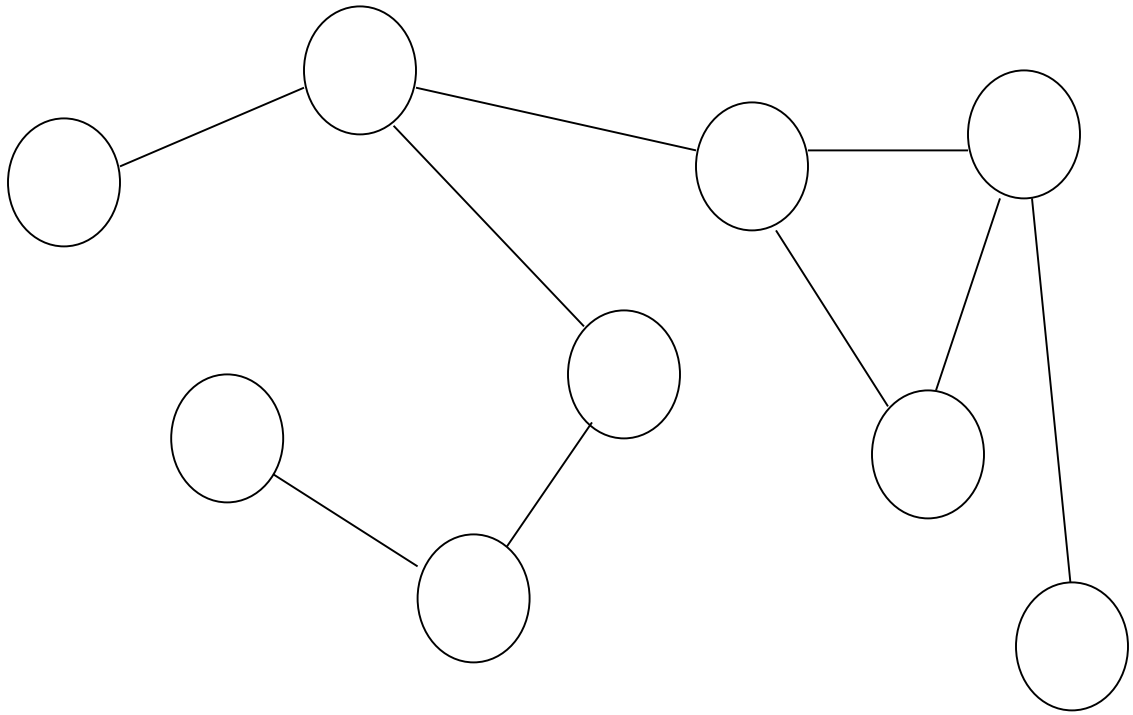
A well-known problem in TCS, now from a distributed perspective!

# Case Study: Graph Coloring

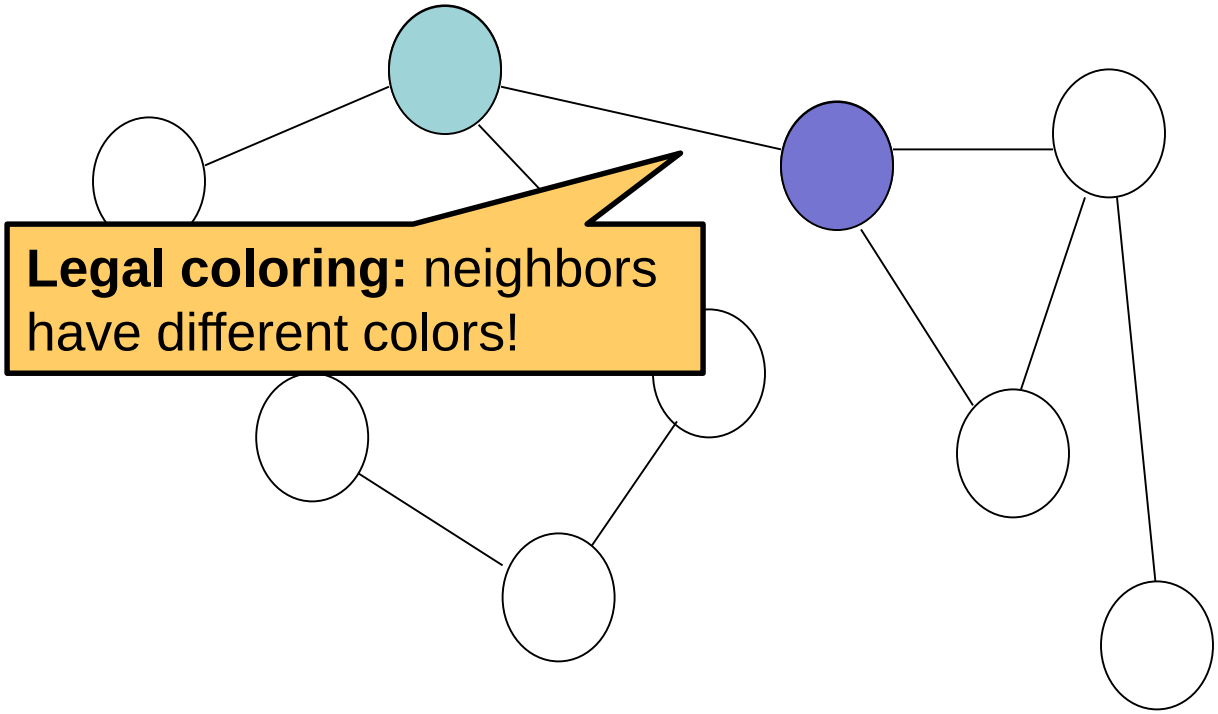


# Case Study: Graph Coloring

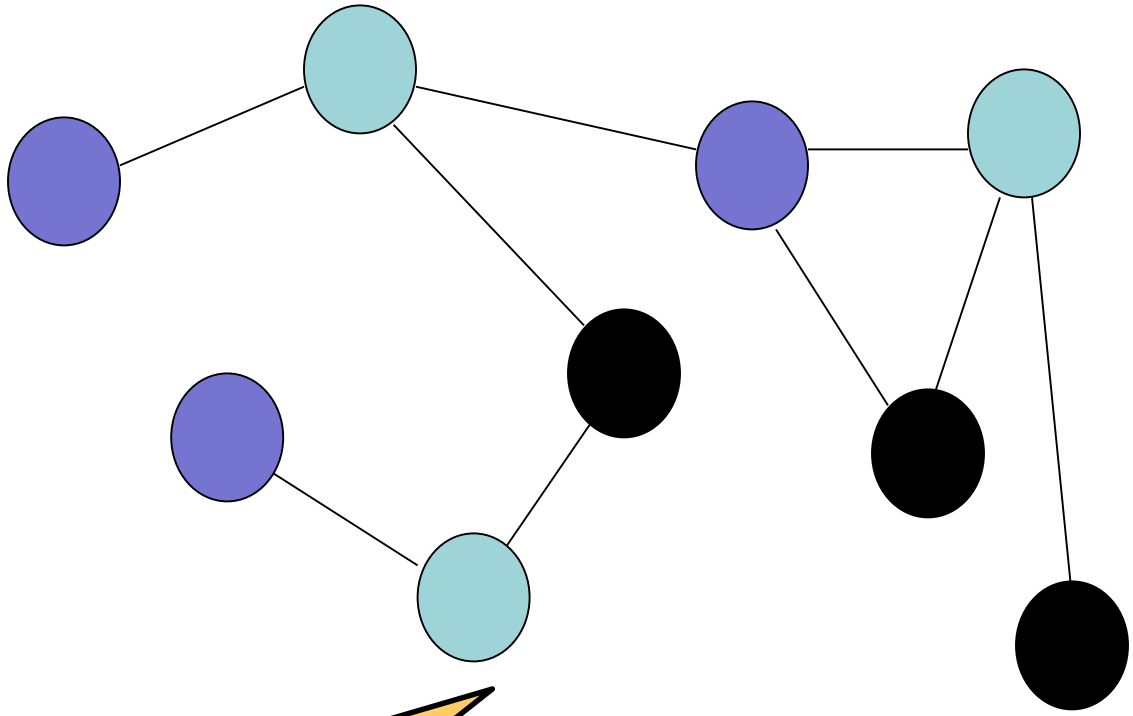
**Task:** Assign colors to nodes.



# Case Study: Graph Coloring



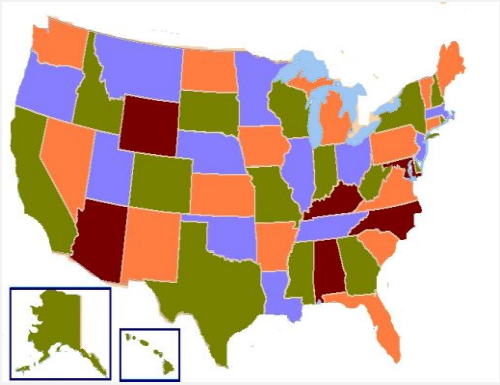
# Case Study: Graph Coloring



**Optimal coloring:** Minimum number of colors (aka chromatic number).

# Applications

## Country Maps

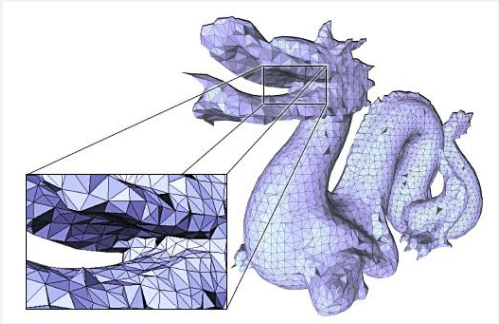


- **Color map**: Neighboring states should have different colors!
- Famous 4-color theorem: any map (or **planar graph**) can be painted with four colors!

## Medium Access

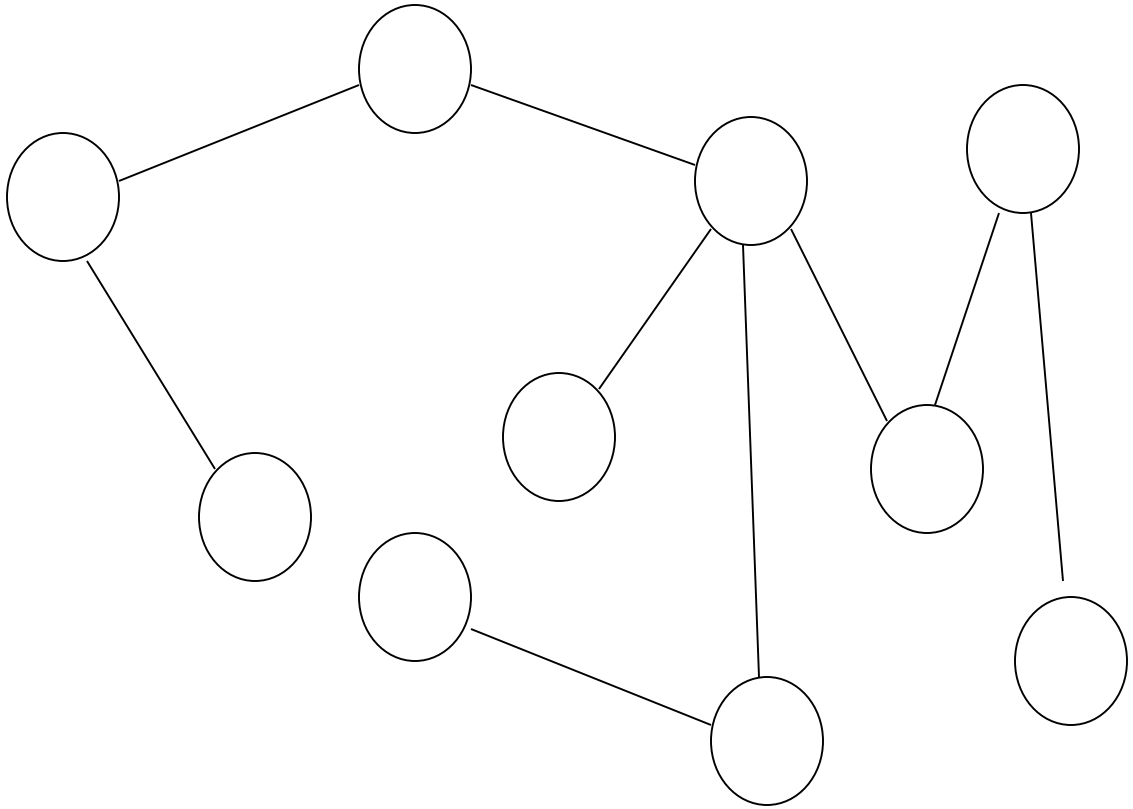
- Interference-free, efficient utilization of **spectrum**
- Neighboring cells should have **different frequencies**!
- Colors = frequencies, channels, etc.

## Image Processing

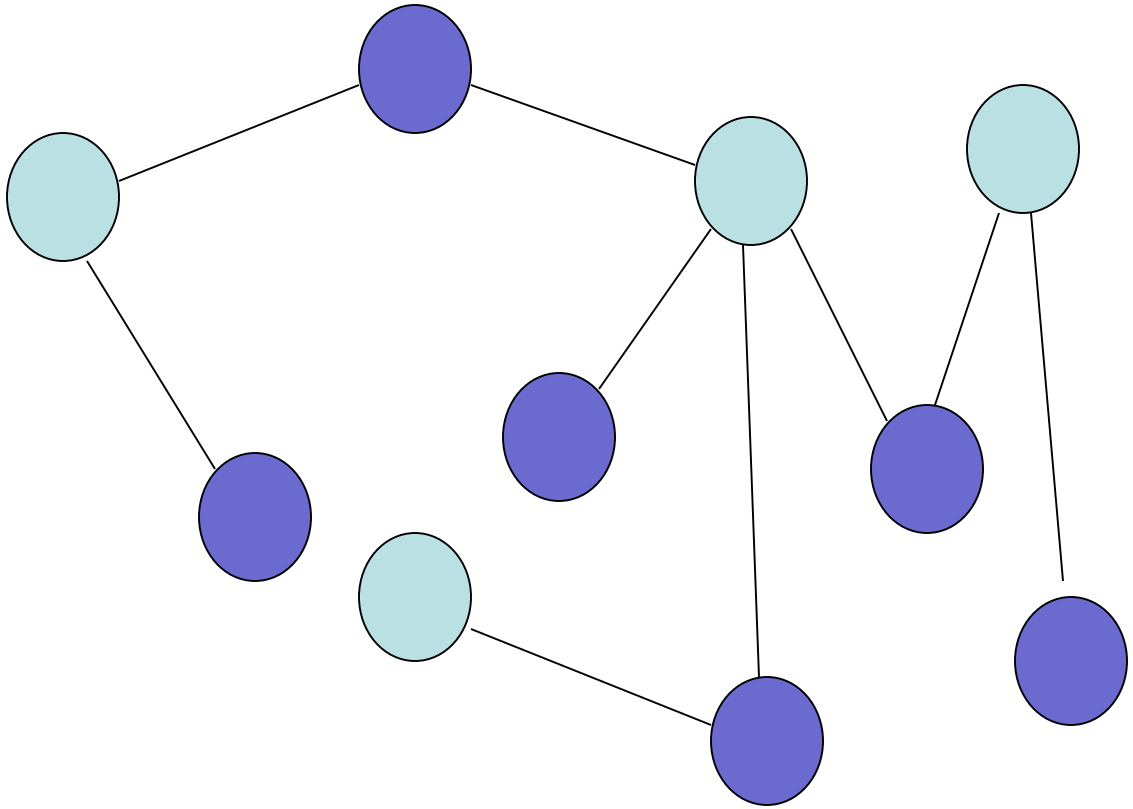


- **Chromatic scheduling** for physical simulation
- Goal: **process nodes of same color in parallel** without determinacy race: no mutual exclusion needed

# Legal coloring? Chromatic number?

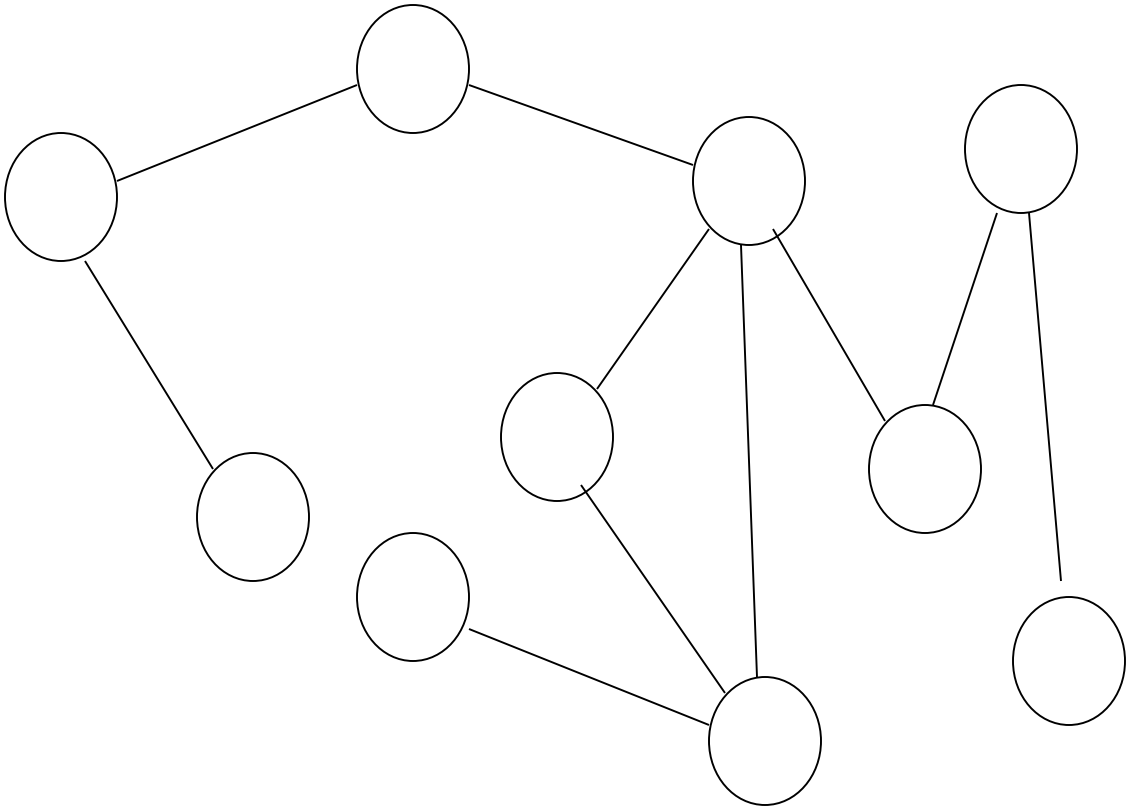


# Legal coloring? Chromatic number?

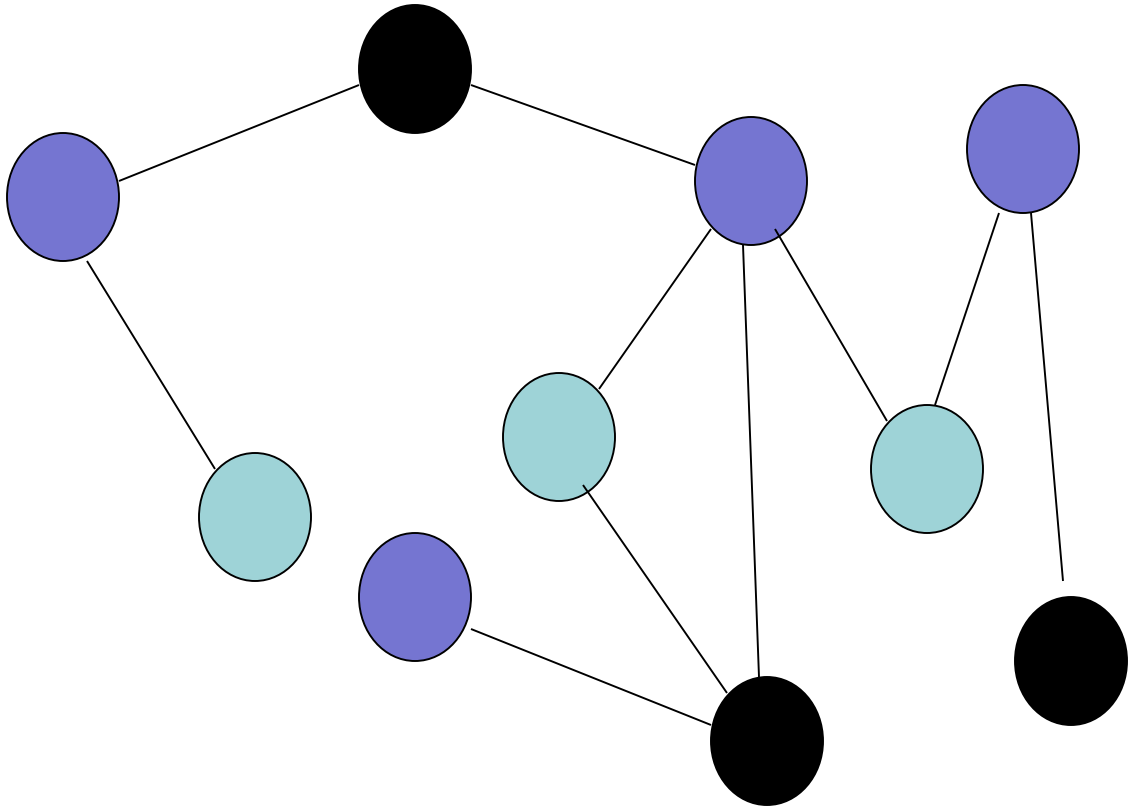


Tree! 2 colors are enough...

# What about this example?



# What about this example?



3 colors needed and enough...

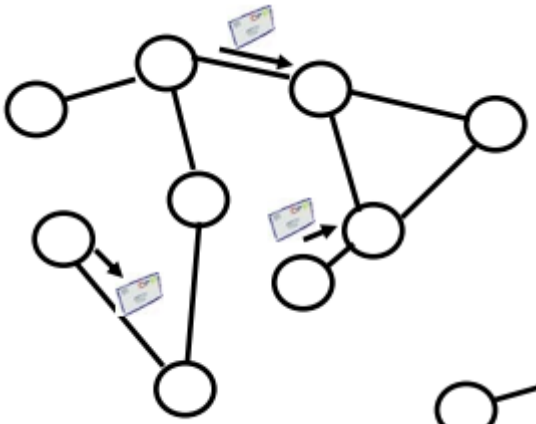
# How to color a graph in a distributed manner?

Time to introduce the LOCAL model of distributed computing!

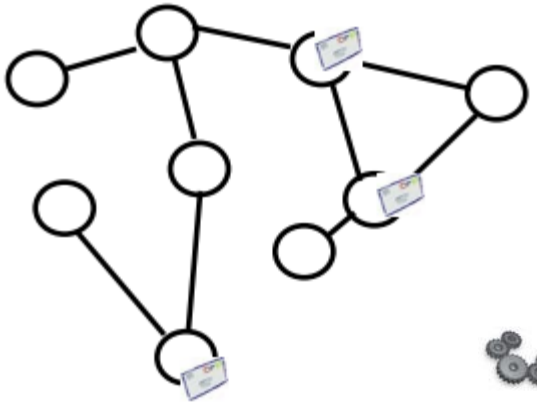
# The LOCAL Model: A Convenient **Synchronous Model**

... in each (synchronous) round!

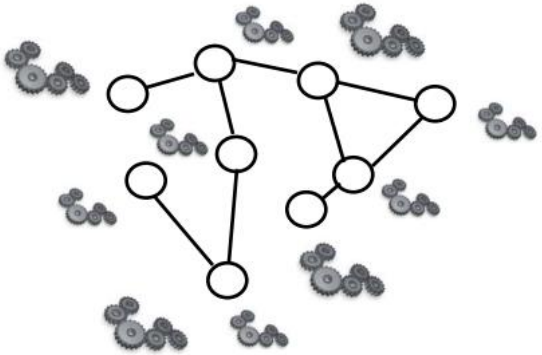
Send...



... receive...



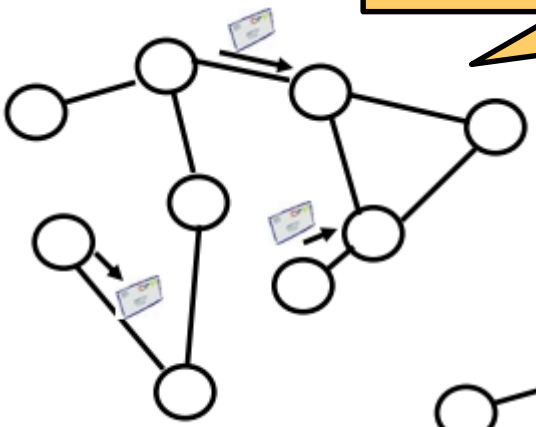
... compute.



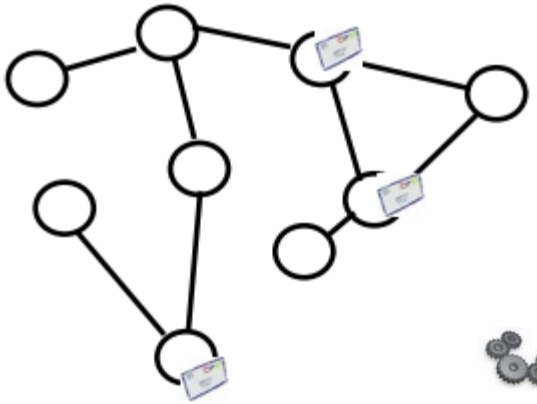
# The LOCAL Model:

We will see in this course: while the LOCAL is a **convenient model** to reason about and design distributed algorithms with…

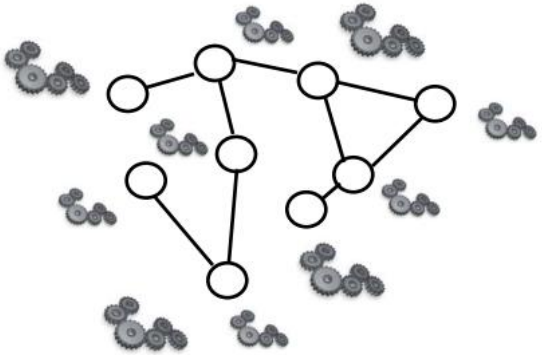
Send...



... receive...



... compute.



... in each round!

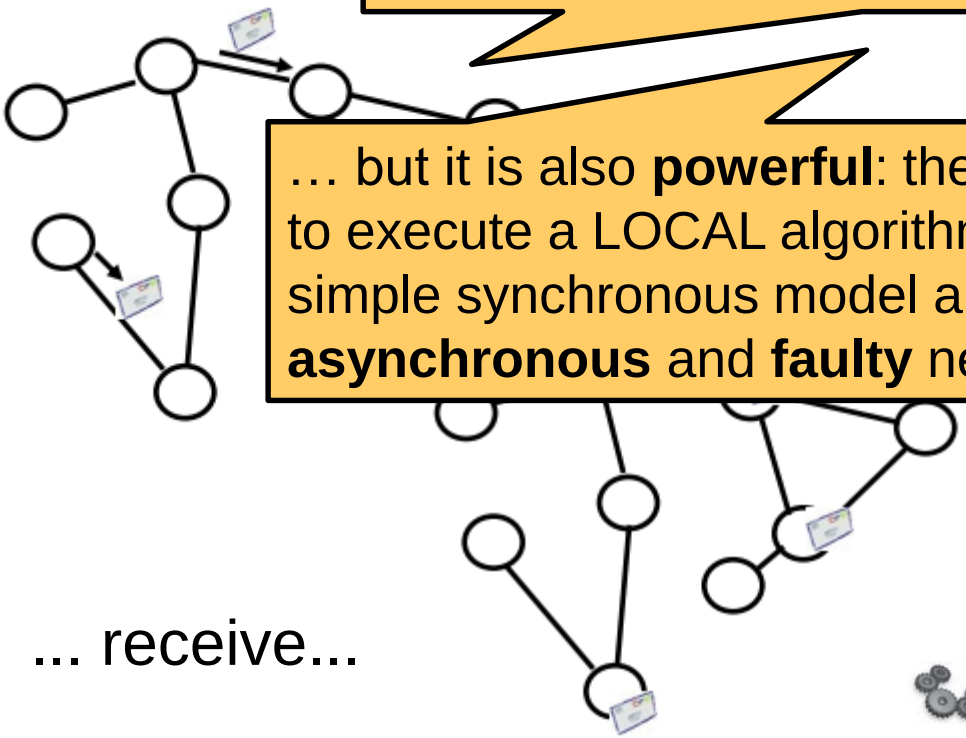
# The LOCAL Model:

We will see in this course: while the LOCAL is a **convenient model** to reason about and design distributed algorithms with…

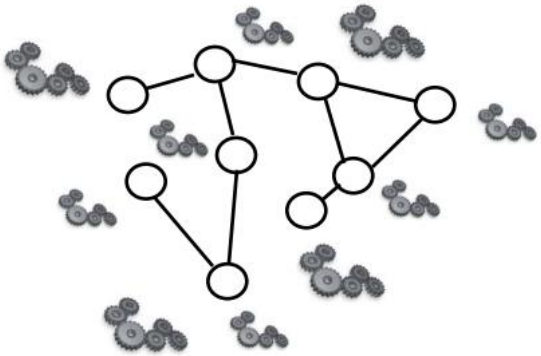
in each

… but it is also **powerful**: there are techniques to execute a LOCAL algorithm designed in the simple synchronous model also in **asynchronous** and **faulty** networks!

Send...



... receive...



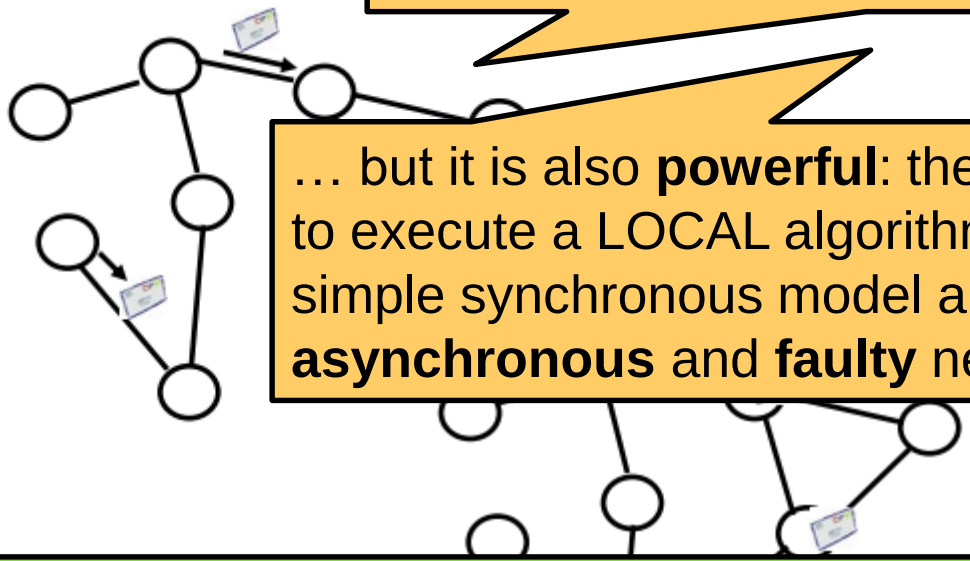
... compute.

# The LOCAL Model:

We will see in this course: while the LOCAL is a **convenient model** to reason about and design distributed algorithms with…

in each

Send...



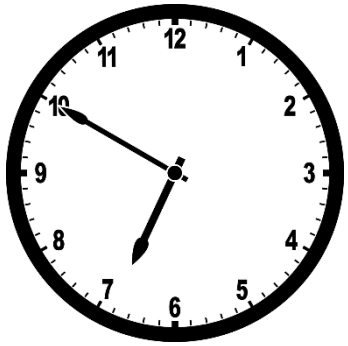
… but it is also **powerful**: there are techniques to execute a LOCAL algorithm designed in the simple synchronous model also in **asynchronous** and **faulty** networks!

For example: LOCAL algorithms can be automatically compiled for an asynchronous environment (using synchronizers) or into a robust (namely self-stabilizing) algorithm!

# Performance Metrics for Distributed Algorithms

- **Time Complexity:**

  Number of communication rounds

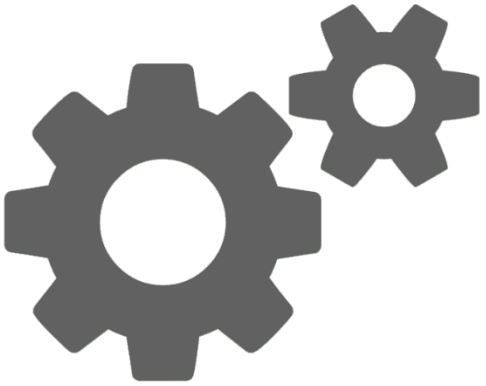


- **Message Complexity:**

  Number of messages sent

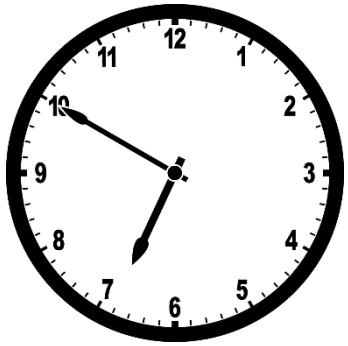
- **Local Computation:**

  Complexity of local computations



# Performance Metrics for Distributed Algorithms

What else?

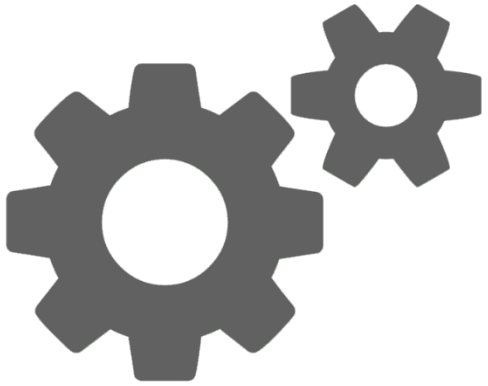


- **Time Complexity:**

  Number of communication rounds

- **Message Complexity:**

  Number of messages sent



- **Local Computation:**

  Complexity of local computations

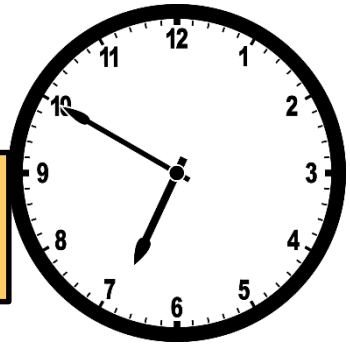
# Performance Metrics for Distributed Algorithms

What else?

- **Time Comple**

  Num

Quality of solution: Approximation ratio for example („price of locality“).

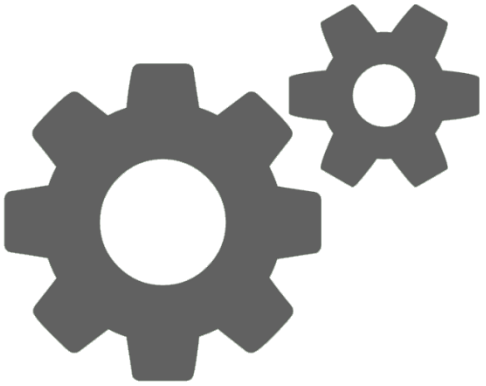


- **Message Complexity:**

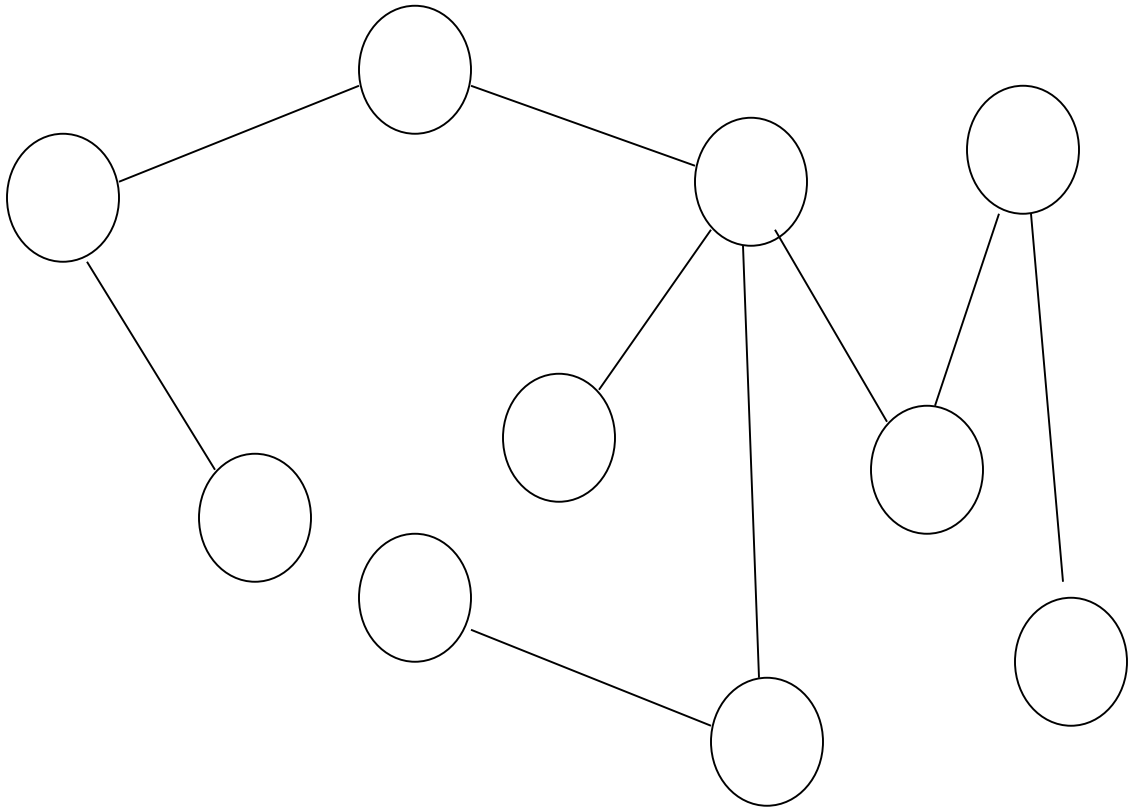
  Number of messages sent

- **Local Computation:**

  Complexity of local computations



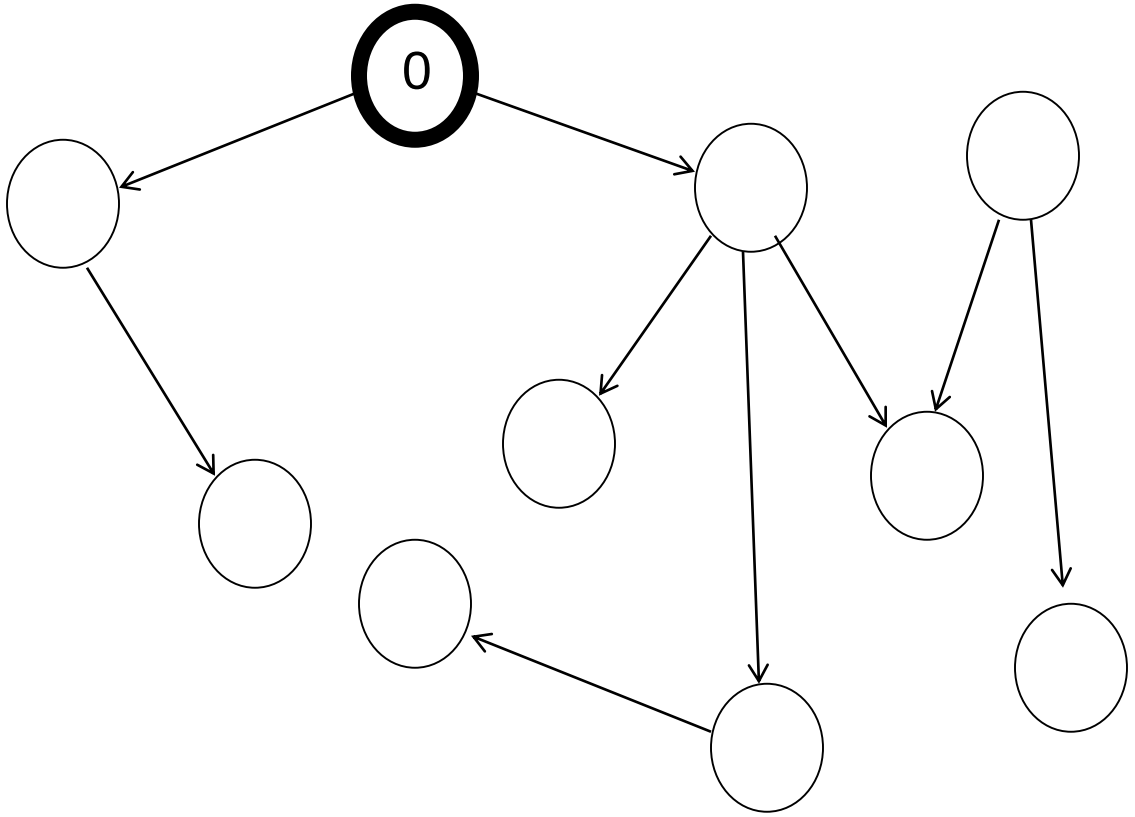
# Let us start simple: How to color a *tree* in a distributed manner?



# Even simpler: How to color a *rooted tree* in a distributed manner?

Simplification:

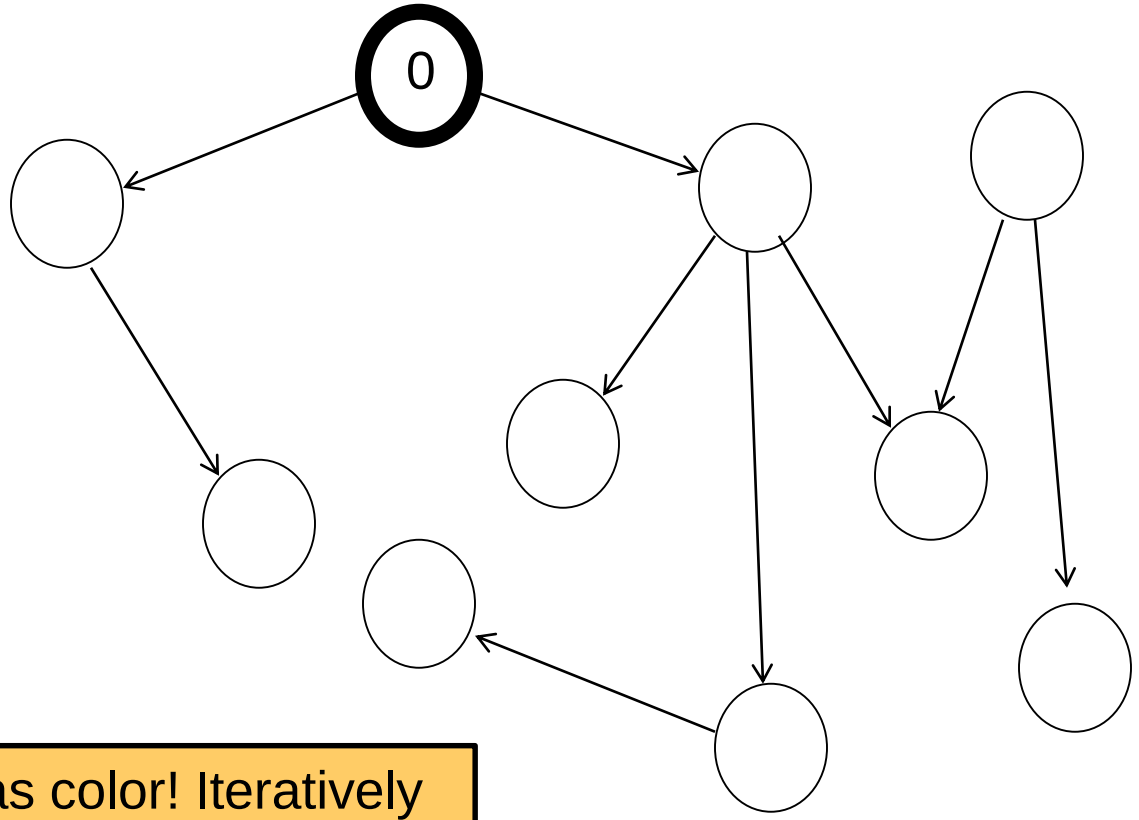
- ❏ Assume rooted
- ❏ Root ID 0



# Even simpler: How to color a *rooted tree* in a distributed manner?

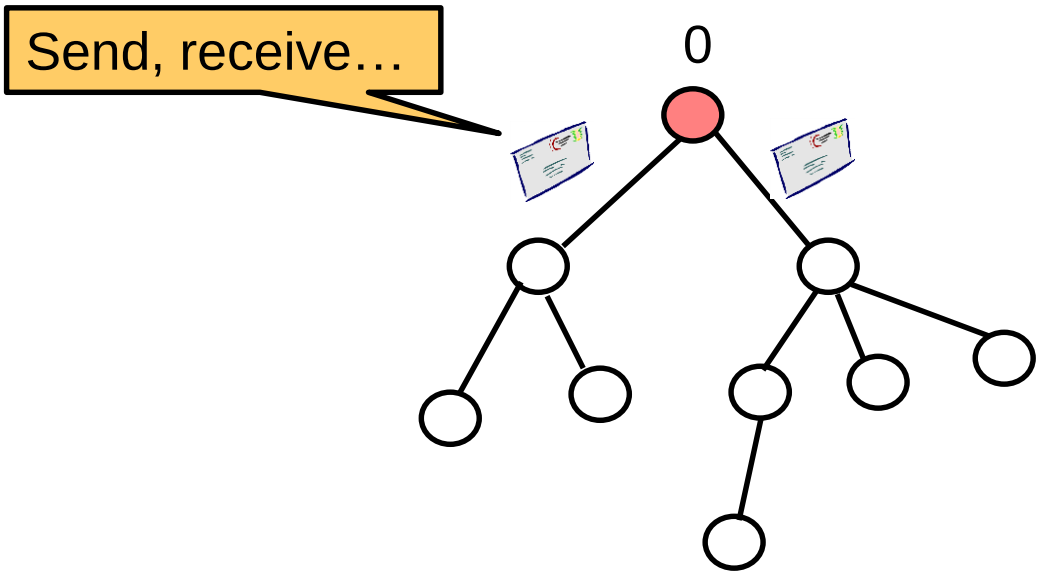
Simplification:

- ❏ Assume rooted
- ❏ Root ID 0



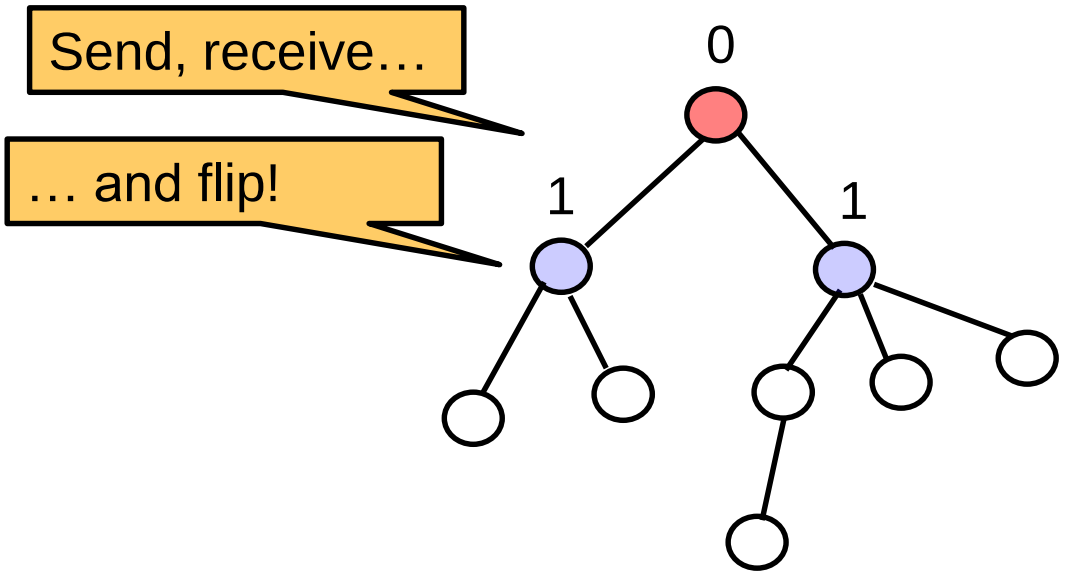

Idea: interpret root bit as color! Iteratively communicate colors to children and take opposite color from parent!

# Slow Distributed Tree Coloring: Example



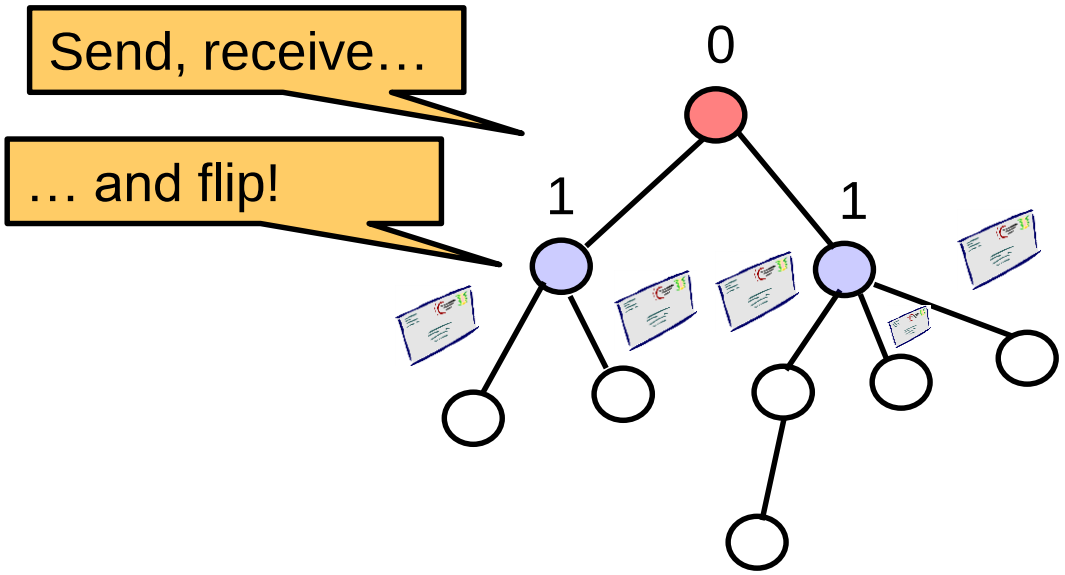

**Round 1**

# Slow Distributed Tree Coloring: Example



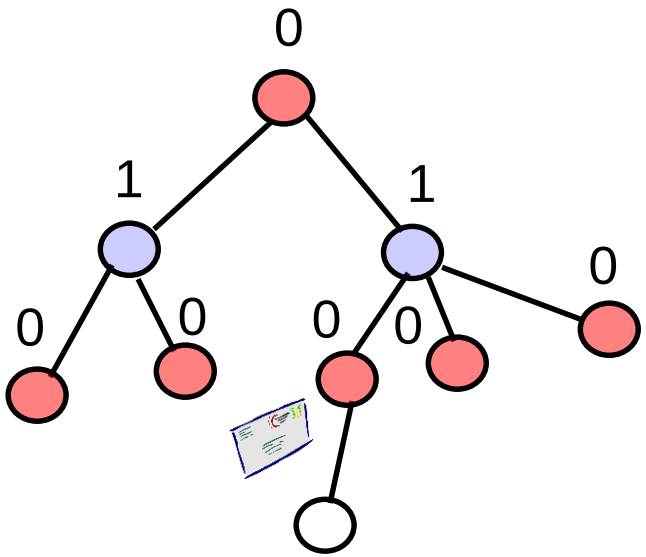

**Round 1**

# Slow Distributed Tree Coloring: Example



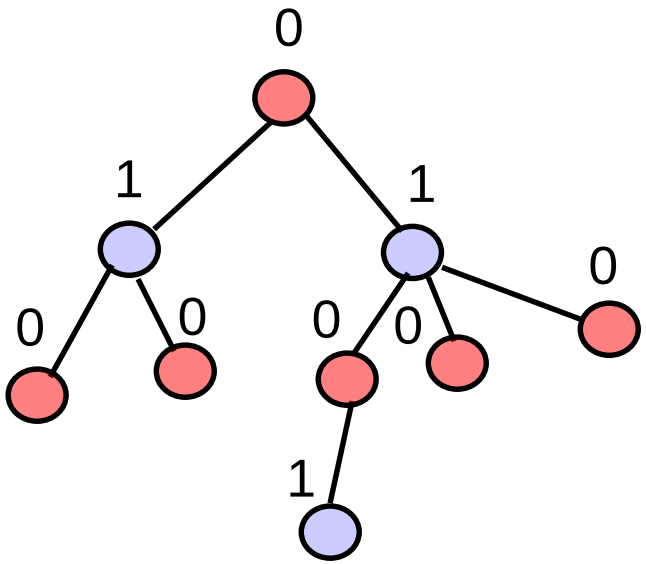

**Round 2**

# Slow Distributed Tree Coloring: Example




**Round 3**

# Slow Distributed Tree Coloring: Example



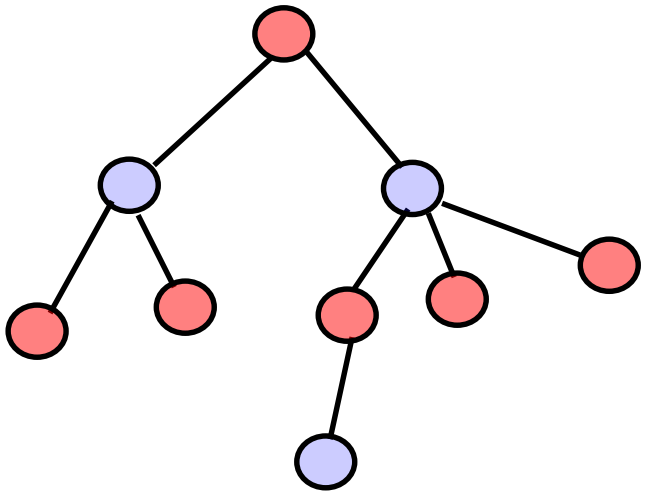

**Round 3**

# Slow Tree Algo

If root: color 0, send 0 to children
Otherwise: each node *v*:

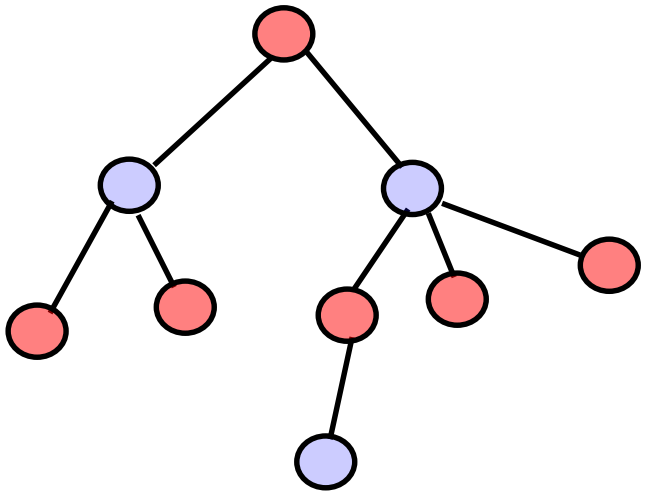
- Wait for message *x* from parent
- Choose color *y=1-x*
- Send *y* to children

# Slow Tree: Analysis



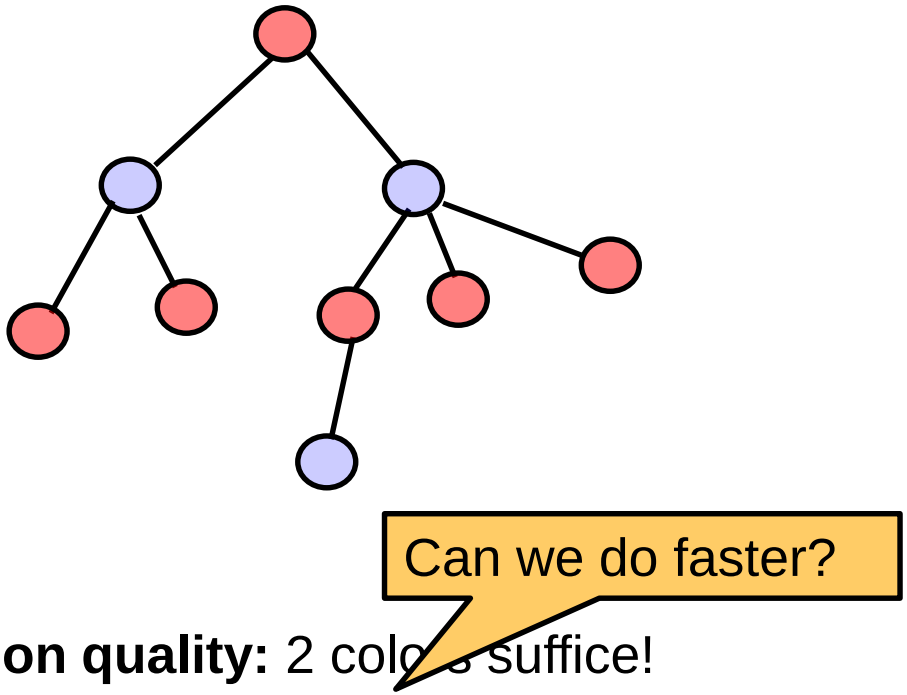
- **Approximation quality:** # colors?
- **Time complexity:** # rounds?
- **Message complexity:** # messages?
- **Local complexity:** local computations?

# Slow Tree: Analysis



- **Approximation quality:** 2 colors suffice!
- **Time complexity:** O(n), depth of the tree
- **Message complexity:** O(n)
- **Local complexity:** trivial, just flip!

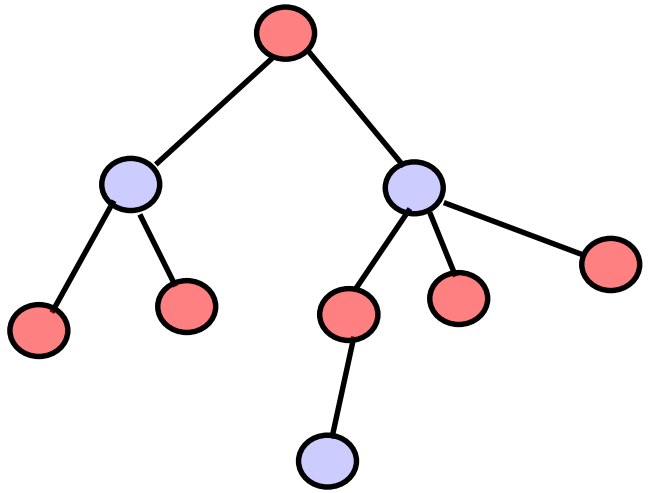
# Slow Tree: Analysis



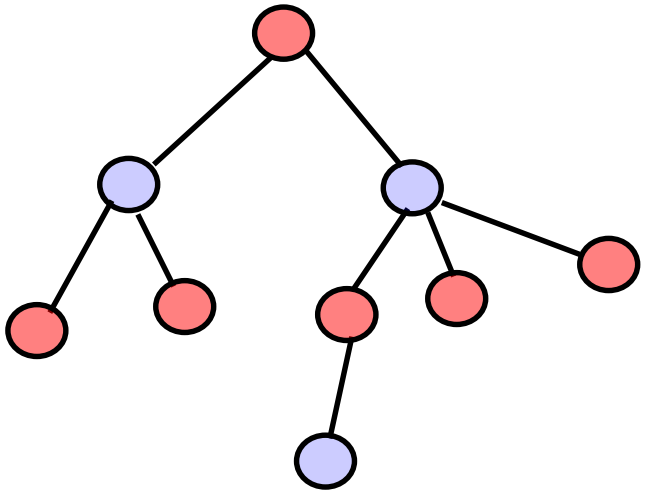

- **Approximation quality:** 2 colors suffice!
- **Time complexity:** O(n), depth of the tree
- **Message complexity:** O(n)
- **Local complexity:** trivial, just flip!

# Ultra Fast Distributed Tree Coloring

- Yes we can!
- 3-coloring in O(log* n) rounds



# Ultra Fast Distributed Tree Coloring



- Yes we can, **assuming unique IDs**
- 3-coloring in O(log* n) rounds
- Idea: based on ID manipulations
  - Idea: interpret ID as color

**Initially: legal but very expensive coloring!**
**How can we quickly reduce the ID space?**

# Intuition: n vs log* n

log n: How many times do I have to **:2** until <2?

n, n/2, n/4, n/8, …, 8, 4, 2, 1

log n

# Intuition: n vs log* n

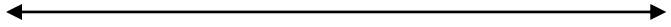
log n: How many times do I have to **:2** until <2?

n, n/2, n/4, n/8, …, 8, 4, 2, 1

log n

loglog n: How many times do I have to $\sqrt{x}$ until <2?

n, $\sqrt{n}$, $\sqrt{\sqrt{n}}$, $\sqrt{\sqrt{\sqrt{n}}}$, …, <2



loglog n

# Intuition: n vs log* n

log n: How many times do I have to **:2** until <2?

n, n/2, n/4, n/8, …, 8, 4, 2, 1

log n

loglog n: How many times do I have to $\sqrt{x}$ until <2?

n, $\sqrt{n}$, $\sqrt{\sqrt{n}}$, $\sqrt{\sqrt{\sqrt{n}}}$, …, <2

loglog n

log* n: How many times do I have to log x until <2?

n, log n, loglog n, logloglog n, …, <2

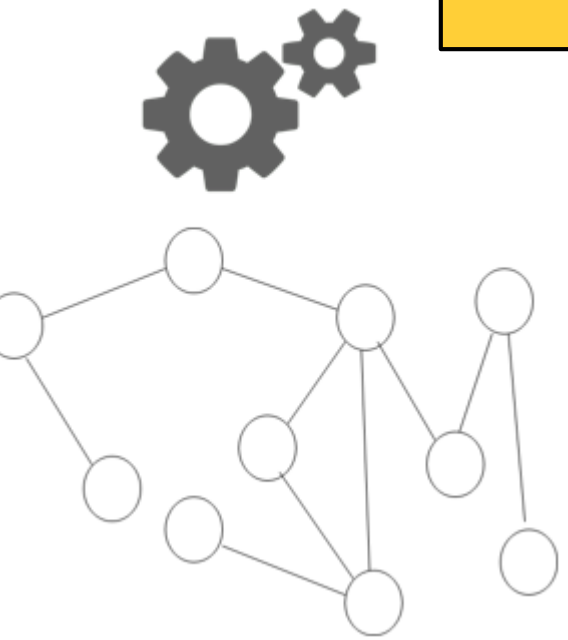
log* n


n = atoms in universe ≈ $10^{80}$
log*(atoms in universe) ≈ 5

## Slow Algo

**No parallelism!**

**e.g., our slow tree algorithm executed on the line.**

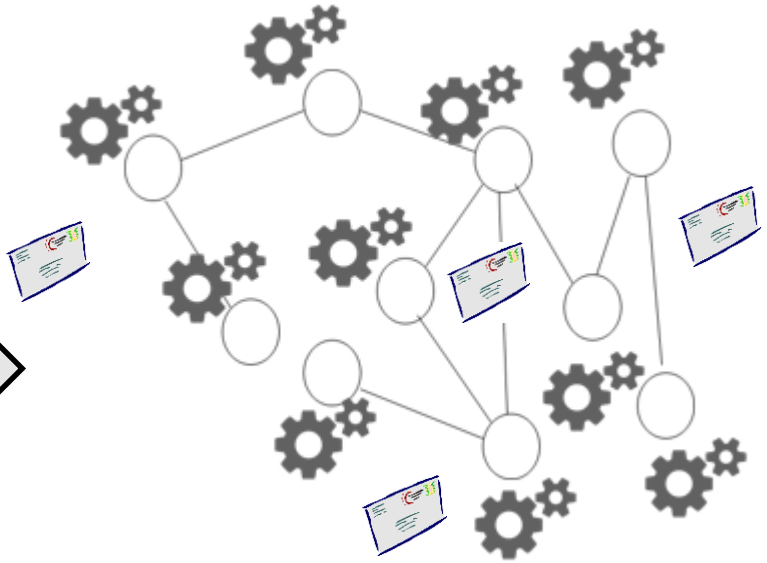


Time: n

A big im-provement!

## Fast Algo

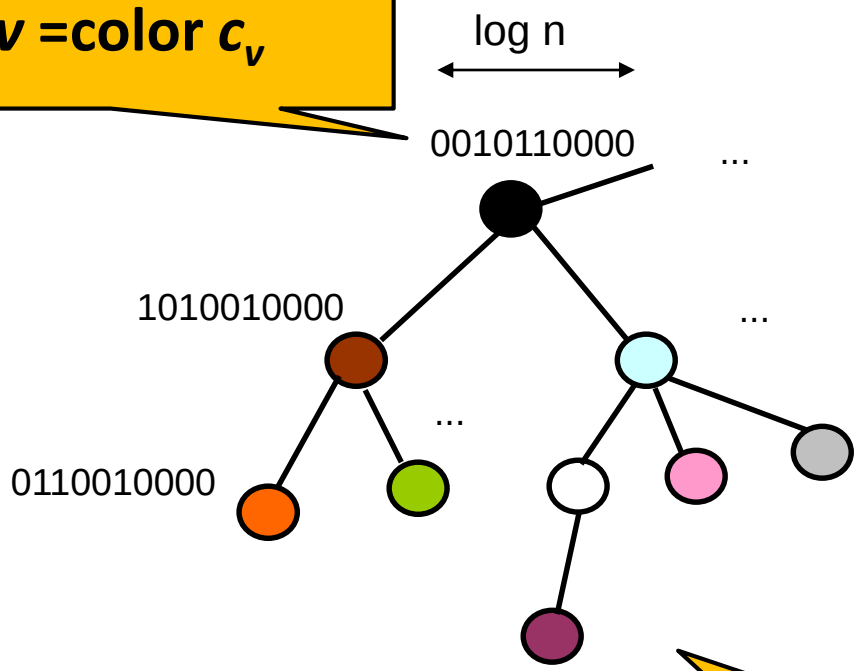
**Efficient parallel manipulations!**



Time: log* n

# Log*-Time Coloring with Label Manipulation

**Initially ID = bit label of node $v$ =color $c_v$**

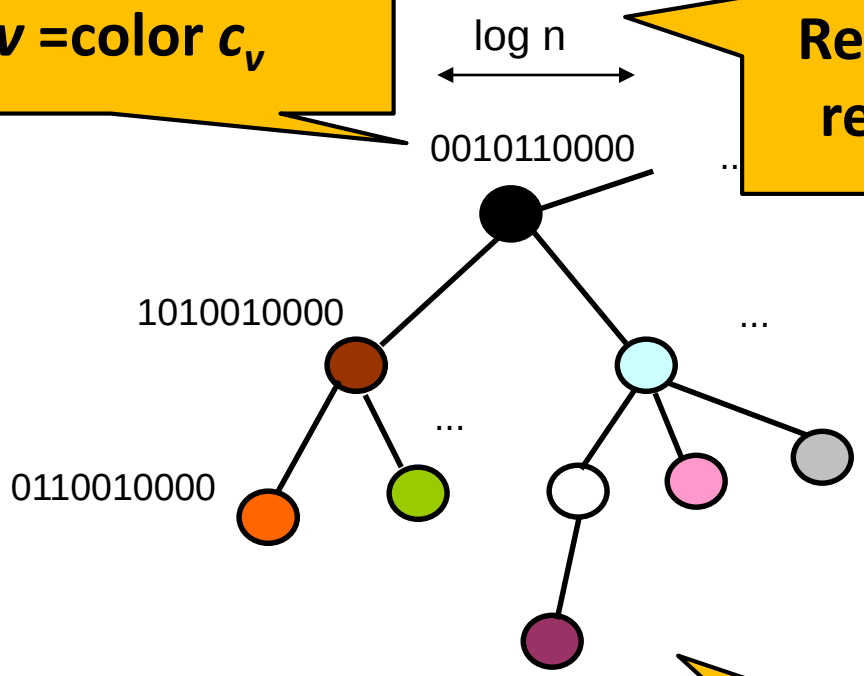



**initially**

**Great: Legal coloring!**
**But: expensive coloring!**

# Log*-Time Coloring with Label Manipulation

**Initially ID = bit label of node $v$ =color $c_v$**

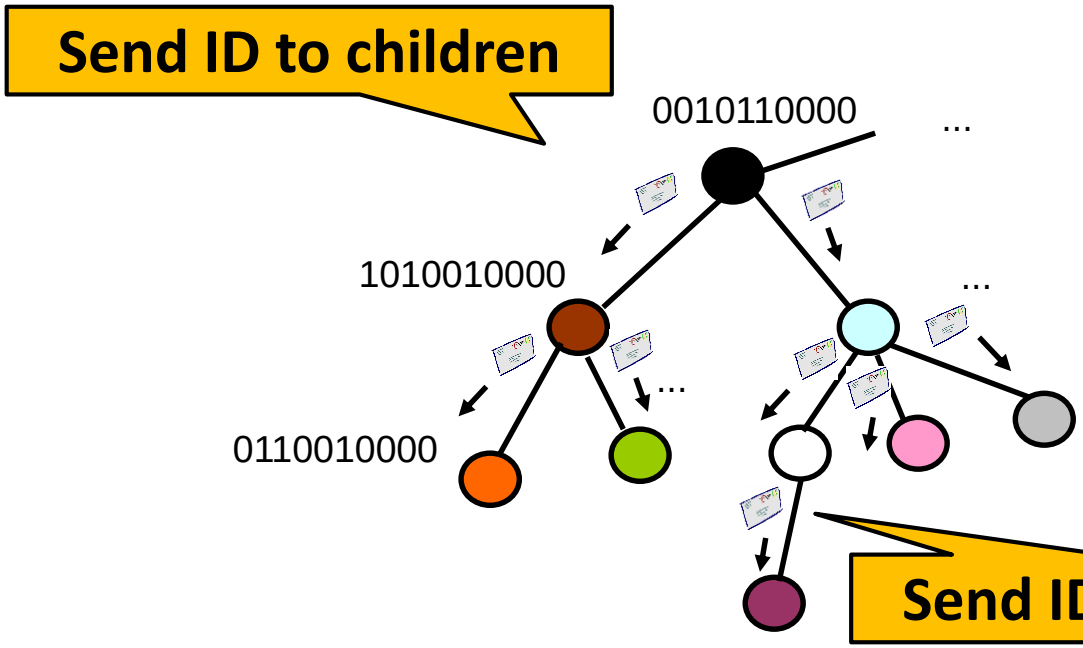



**Recall: need log n bits to represent n unique IDs**

**initially**

**Great: Legal coloring!**
**But: expensive coloring!**

# What happens in Round 1?

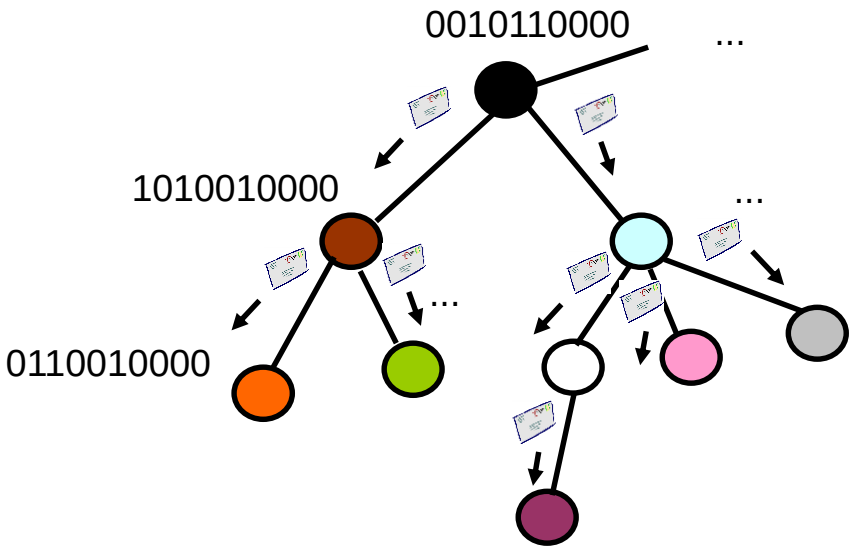



**Round 1**

**Algorithm:** in round i, node v:

1. Send my $c_v$ to children (in parallel!)
2. Receive parent ID/color $c_p$

# What happens in Round 1?




**Round 1**

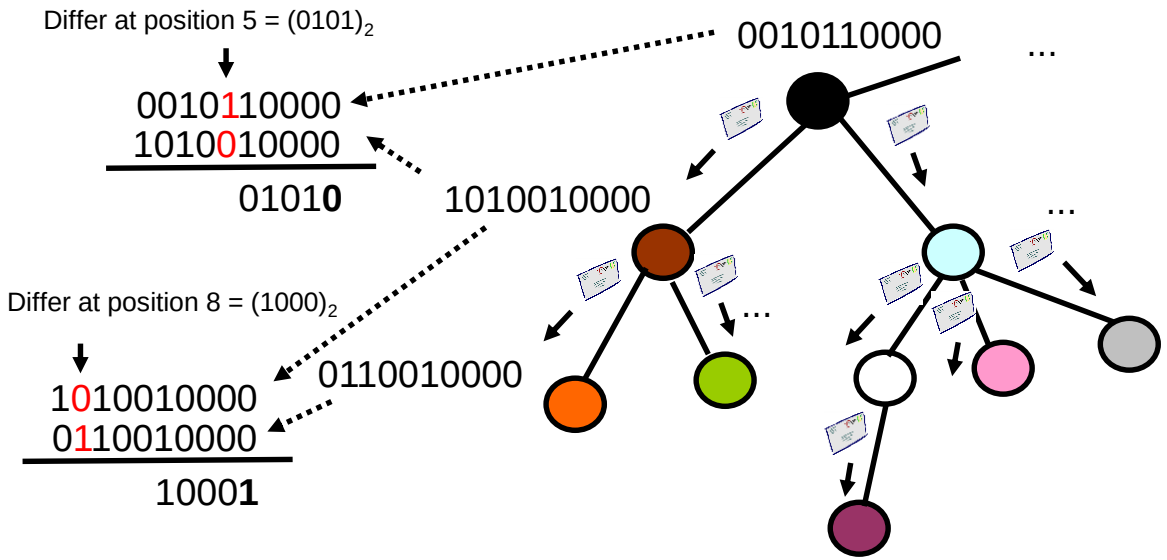
**Algorithm:** in round i, node v:

1. Send my $c_v$ to children (in parallel!)
2. Receive parent ID/color $c_p$
3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v = i \,||\, c_v(i)$

**ID = color for next round: the position!**

# What happens in Round 1?

**Example:**

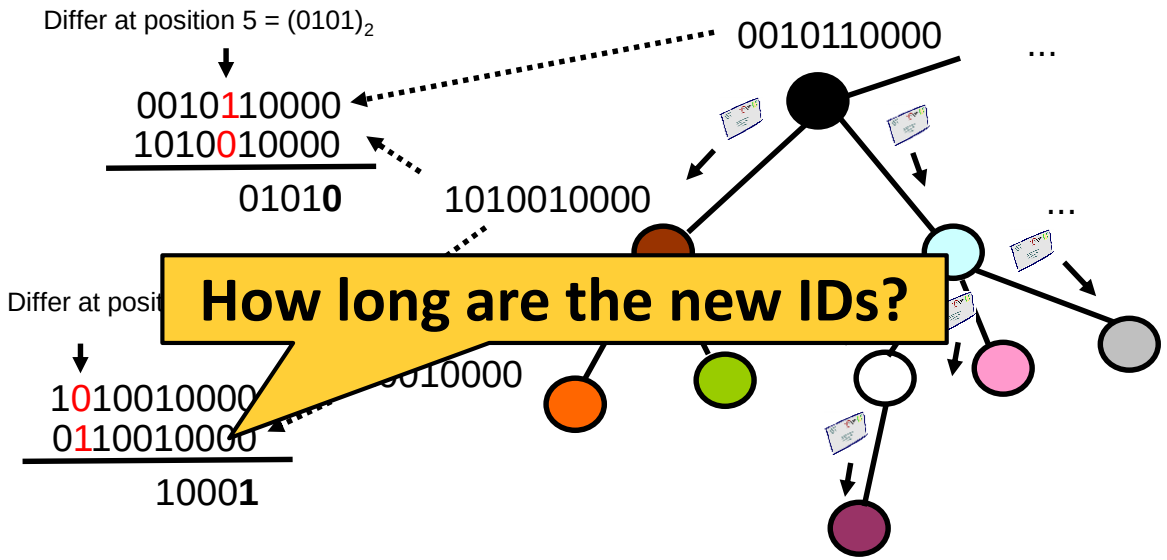



**Round 1**

**Algorithm:** in round i, node v:

1. Send my $c_v$ to children (in parallel!)
2. Receive parent ID/color $c_p$
3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v$ = i || $c_v$(i)

# What happens in Round 1?

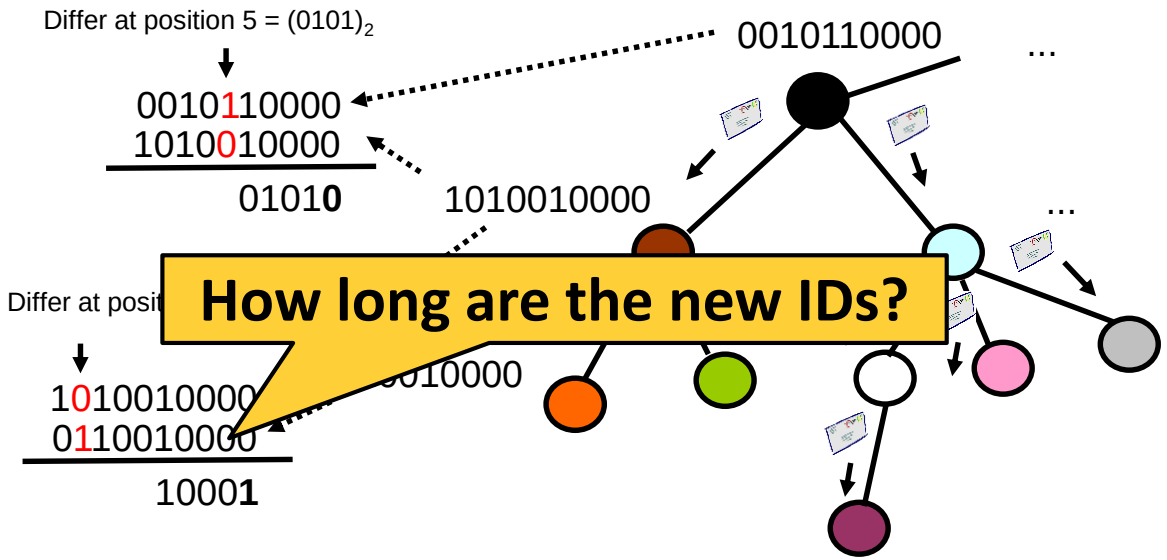



**Round 1**

**Algorithm:** in round i, node v:

1. Send my $c_v$ to children (in parallel!)
2. Receive parent ID/color $c_p$
3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v$ = i || $c_v$(i)

# What happens in Round 1?



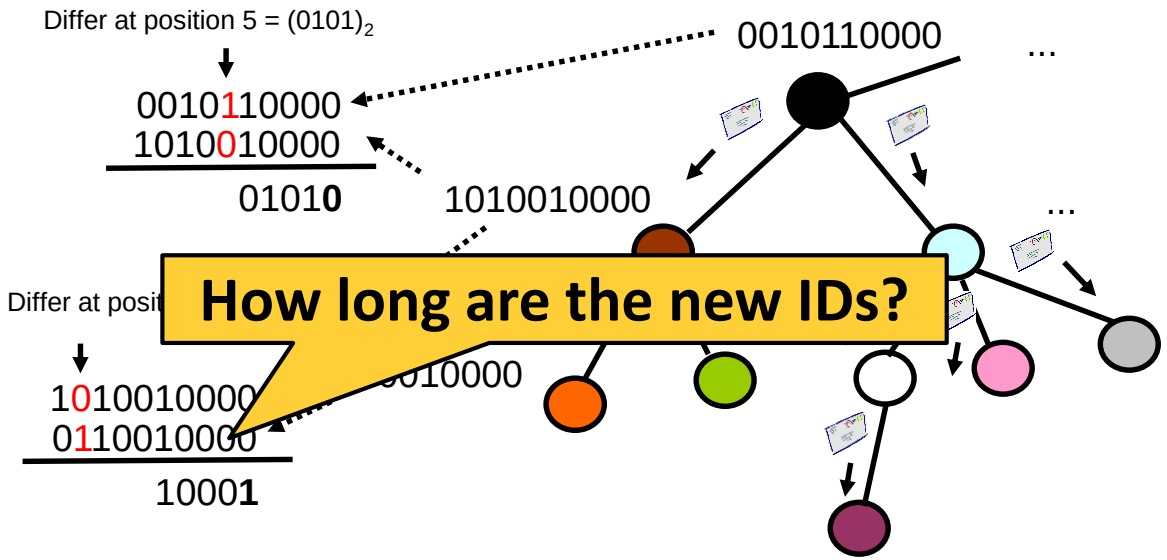

**Round 1**

**Describing position in x-bit string takes log x bits, so: loglog n bits**

allel!)

3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v$ = i || $c_v$(i)

# What happens in Round 1?




**Round 1**

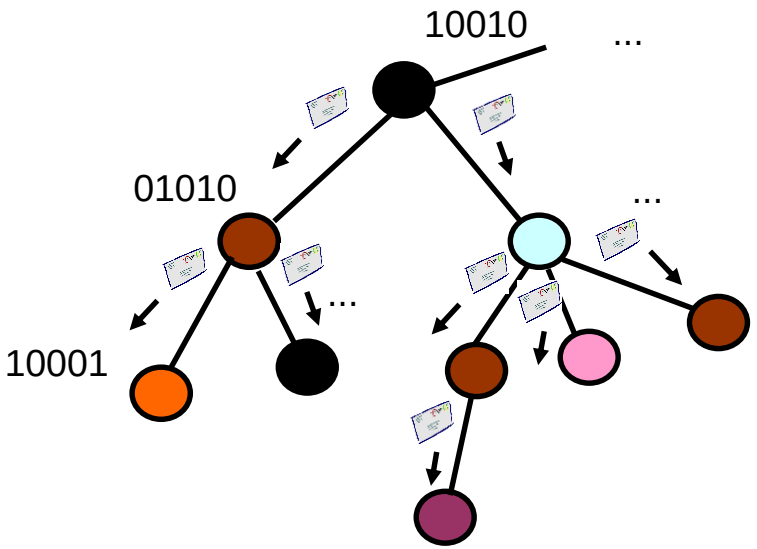
**Describing position in x-bit string takes log x bits, so: loglog n bits**

allel!)

3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v$ = i || $c_v$(i)

**+1 bit**

# Log*-Time Coloring with Label Manipulation

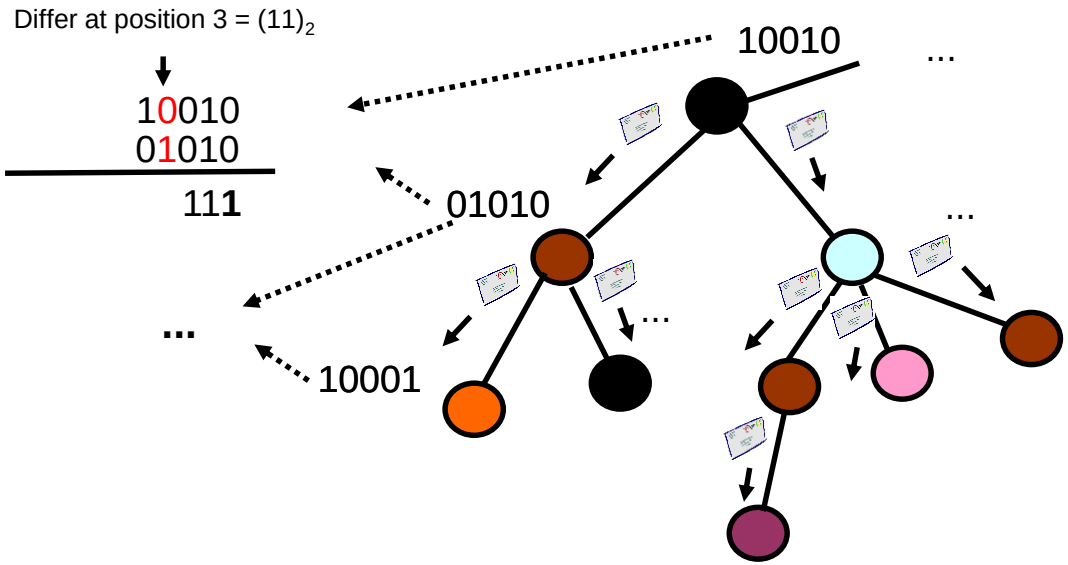



**Round 2**

**Algorithm:** in round i, node v:

1. Send my $c_v$ to children (in parallel!)
2. Receive parent ID/color $c_p$
3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v$ = i || $c_v$(i)

# Log*-Time Coloring with Label Manipulation

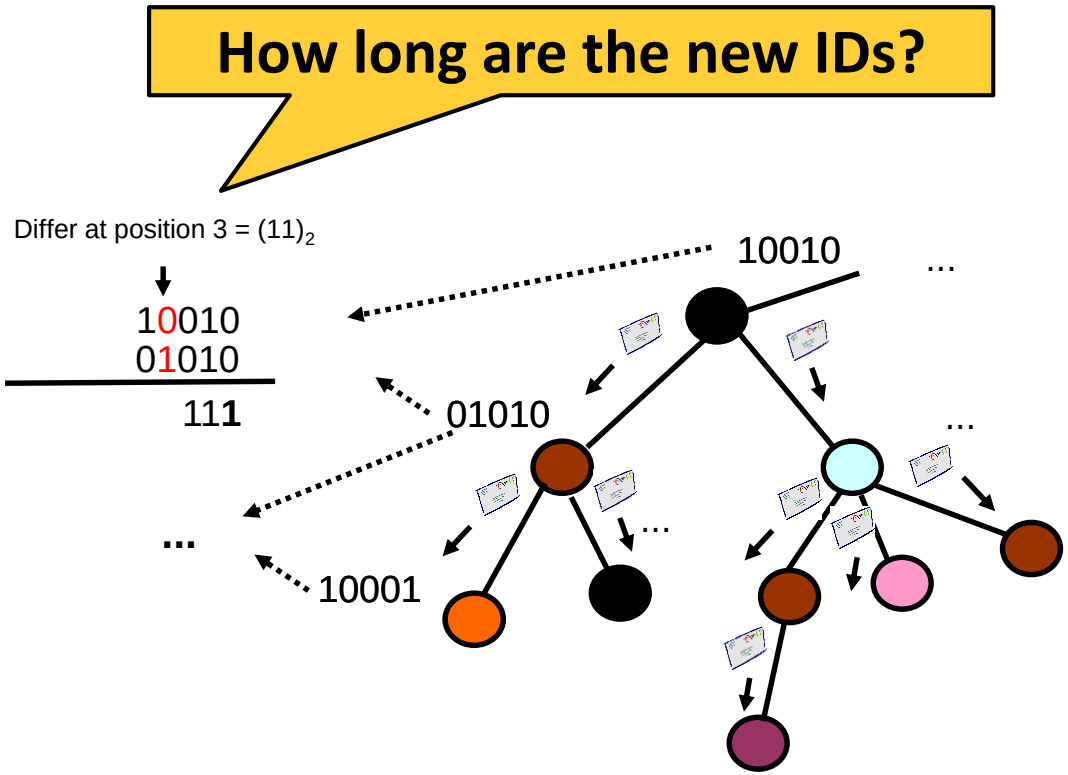



**Round 2**

**Algorithm:** in round i, node v:

1. Send my $c_v$ to children (in parallel!)
2. Receive parent ID/color $c_p$
3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v$ = i || $c_v$(i)

# Log*-Time Coloring with Label Manipulation



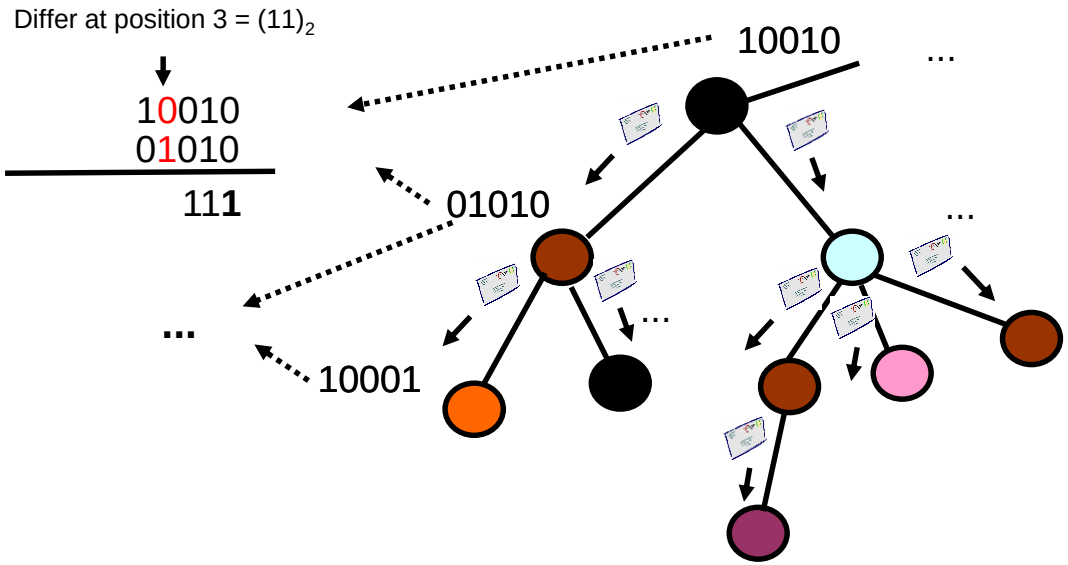

**Round 2**

**Algorithm:** in round i, node v:

1. Send my $c_v$ to children (in parallel!)
2. Receive parent ID/color $c_p$
3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v$ = i || $c_v$(i)

# Log*-Time Coloring with Label Manipulation

**How long are the new IDs?**




**Round 2**

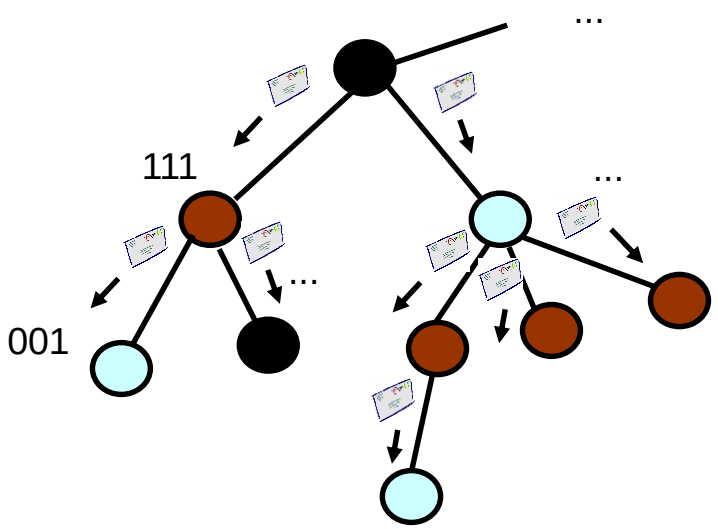
**Describing position in x-bit string takes log x bits, so: logloglog n bits**

allel!)

3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v$ = i || $c_v$(i)

**+1 bit**

# Log*-Time Coloring with Label Maniplulation

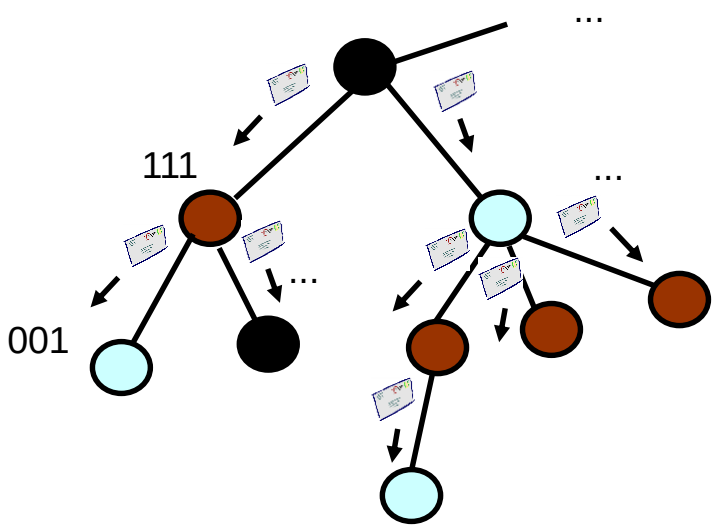



**Round 3**

**Algorithm:** in round i, node v:

1. Send my $c_v$ to children (in parallel!)
2. Receive parent ID/color $c_p$
3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v$ = i || $c_v$(i)

# Log*-Time Coloring with Label Maniplulation




**etc.!**

**Algorithm:** in round i, node v:

1. Send my $c_v$ to children (in parallel!)
2. Receive parent ID/color $c_p$
3. Let i be the smallest index where $c_v$ and $c_p$ differ (from right, binary)
4. My new $c_v$ = i || $c_v$(i)

# Analysis

- How long does it take until O(1) colors?

- Why is coloring always legal?

# Analysis

- How long does it take until O(1) colors?
  - # bits/colors reduced by a log-factor in each round
  - The definition of log*!

    log* n: How many times do I have to log x until <2?

- Why is coloring always legal?

**Algorithm:** My new $c_v$ = i || $c_v$(i)

# Analysis

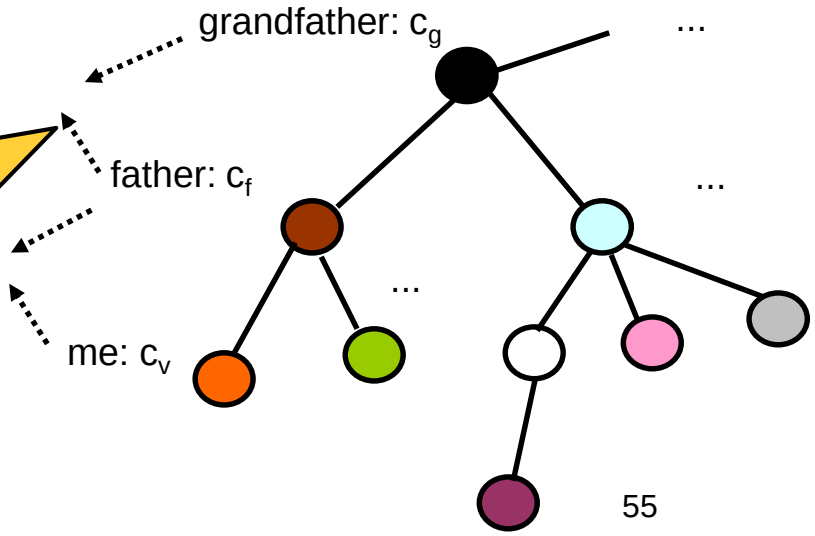
- How long does it take until O(1) colors?
  - # bits/colors reduced by a log-factor in each round
  - The definition of log*!

log* n: How many times do I have to log x until <2?

- Why is coloring always legal?

**Algorithm:** My new $c_v$ = i || $c_v$(i)

**By contradiction: To have same ID as parent, I need to differ at same position from parent as parent from grandparent. But then last bit must be different: there I took my *own* bit (and parent will do the same with its own bit which is different by definition)!**



# Summary of Algorithm

## 6-Colors

Assume: legal initial coloring, root with label $c_v$=0
Each other node v does (in parallel):

Send $c_v$ to kids
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

# Summary of Algorithm

## 6-Colors

Assume: legal initial coloring, root with label $c_r = 0$

Each other node v does (in pa

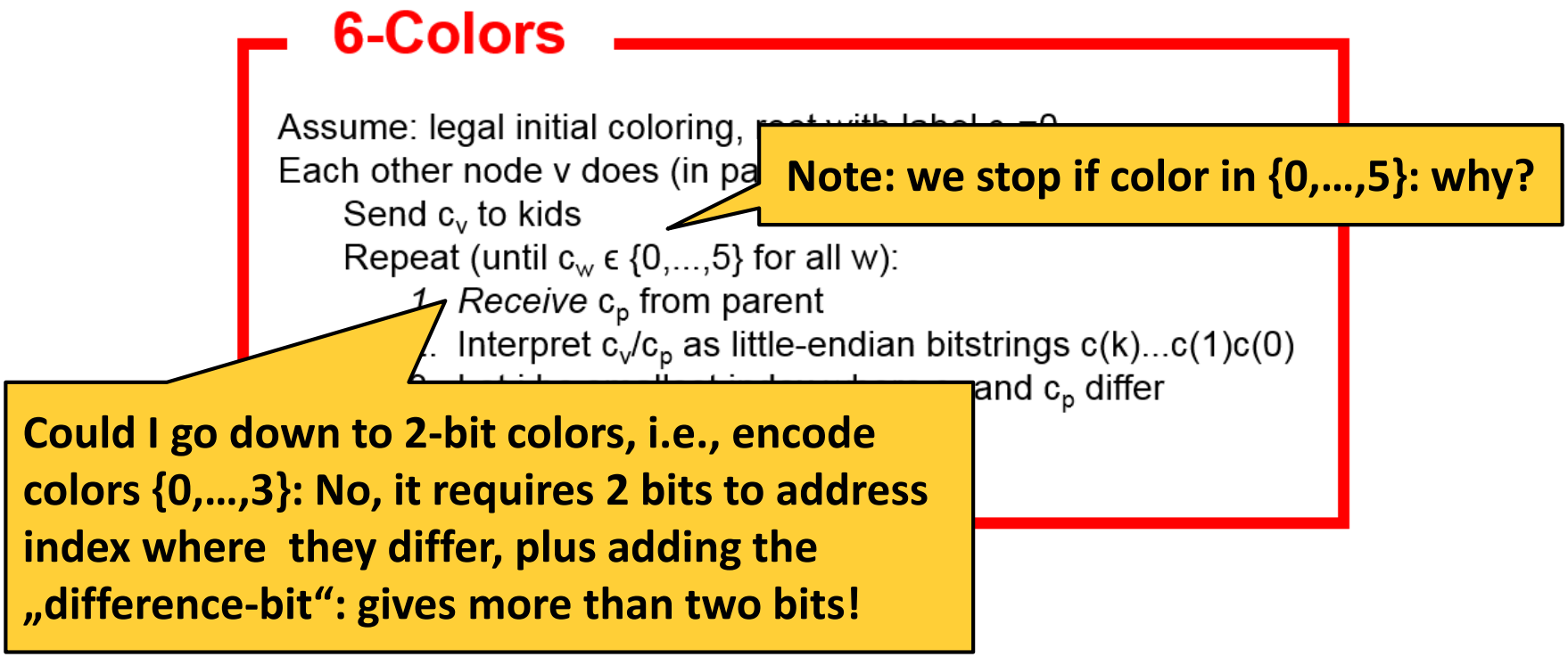
Send $c_v$ to kids

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: $\mathbf{i||c_v(i)}$
5. *Send* $c_v$ to kids

**Note: we stop if color in {0,…,5}: why?**

# Summary of Algorithm



# Summary of Algorithm

## 6-Colors

Assume: legal initial coloring, [illegible]

Each other node v does (in pa[illegible]

Send $c_v$ to kids

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)

[illegible] and $c_p$ differ

**Note: we stop if color in {0,…,5}: why?**

**Could I go down to 2-bit colors, i.e., encode colors {0,…,3}: No, it requires 2 bits to address index where they differ, plus adding the „difference-bit“: gives more than two bits!**

**For 3-bit colors {0,…,7} this still works: e.g., $7=(111)_2$ can be described with 3 bits, and position index (0,1,2) requires two bits, plus one „difference-bit“ gives three again**

# Summary of Algorithm

## 6-Colors

Assume: legal initial coloring, [illegible]
Each other node v does (in pa[illegible]
Send $c_v$ to kids
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)
[illegible] and $c_p$ differ

**Note: we stop if color in {0,…,5}: why?**

**Could I go down to 2-bit colors, i.e., encode colors {0,…,3}: No, it requires 2 bits to address index where they differ, plus adding the „difference-bit“: gives more than two bits!**

**For 3-bit colors {0,…,7} this still works: e.g., 7=$(111)_2$ can be described with 3 bits, and position index (0,1,2) requires two bits, plus one „difference-bit“ gives three again**

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

With 6-COLORS algorithm we can get down to 6 colors.

What about improving it to 3 or even 2 colors?

With 6-COLORS algorithm we can get down to 6 colors.

What about improving it to 3 or even 2 colors?

Homework

# Remark: Optimality

One can show that no local algorithm can O(1)-color a graph faster than in O(log* n).

# Remark: Optimality

One can show that no local algorithm can O(1)-color a graph faster than O(log* n).

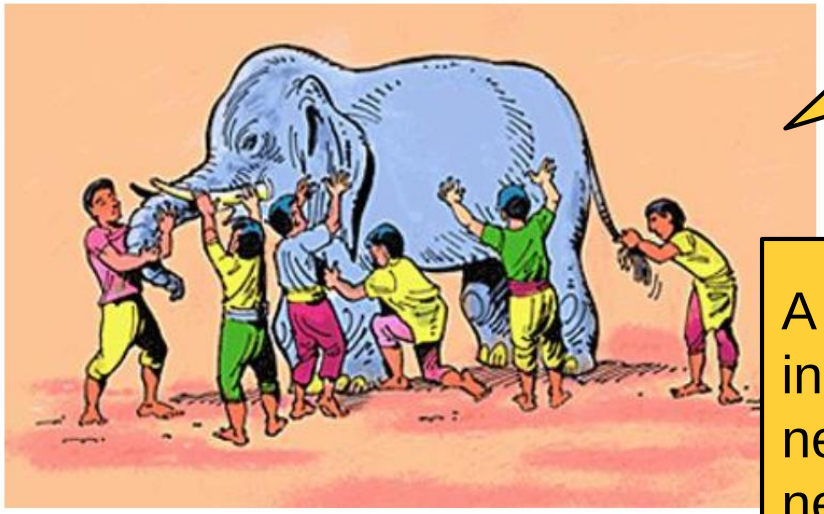
In fact:
in 0 rounds: ≥ n colors
in 1 round: ≥ log n colors
in 2 rounds: ≥ loglog n colors
etc.!

# Remark: Optimality

One can show that no local algorithm can O(1)-color a graph faster than O(log* n).

In fact:
in 0 rounds: ≥ n colors
in 1 round: ≥ log n colors
in 2 rounds: ≥ loglog n colors
etc.!

**Proof idea:** Recall the elephant!



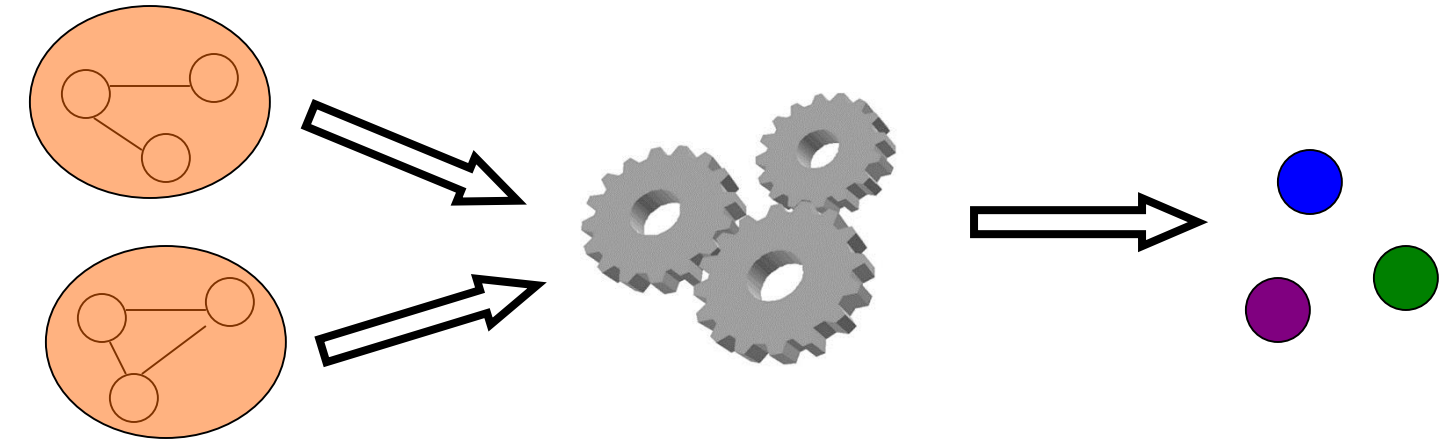
A local coloring algorithm can be seen as a function:
**f: neighborhood → color**

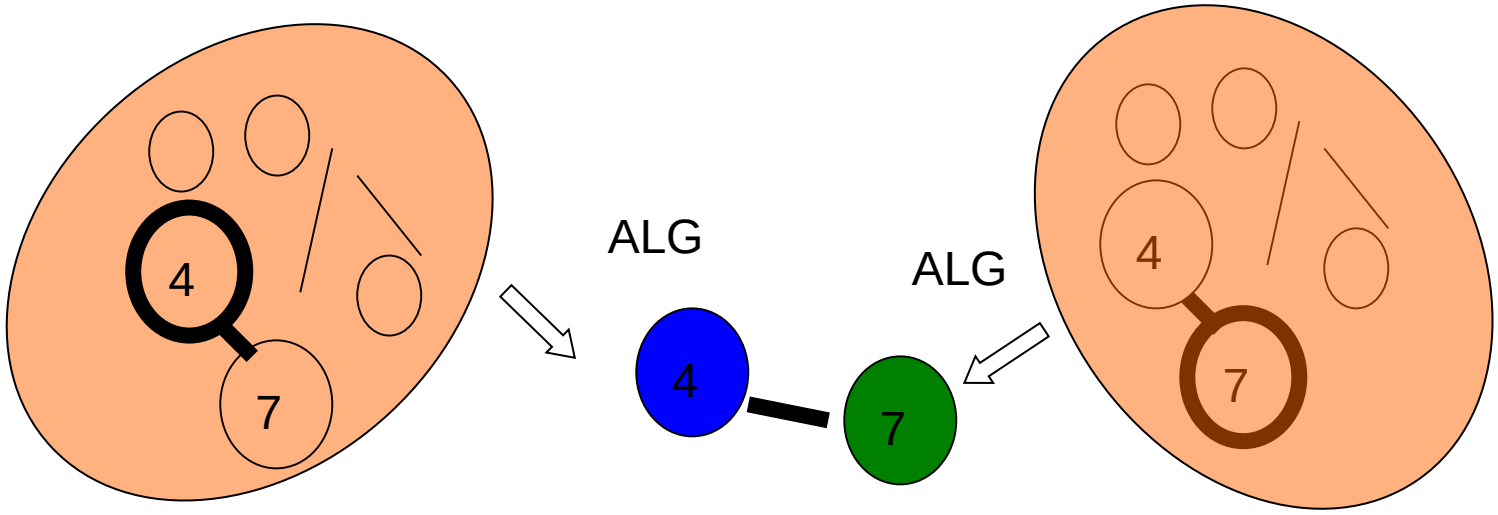
A deterministic algorithm needs to decide in the same way, given same neighborhood. Only with communication neighborhoods start look different and require less conservative colors.

# Lower Bound

Can reduce problem of finding **runtime lower bound** to determine chromatic number of special **neighborhood dependency graphs**!

**Set of neighborhoods** **Local coloring algo** **Vertex coloring**





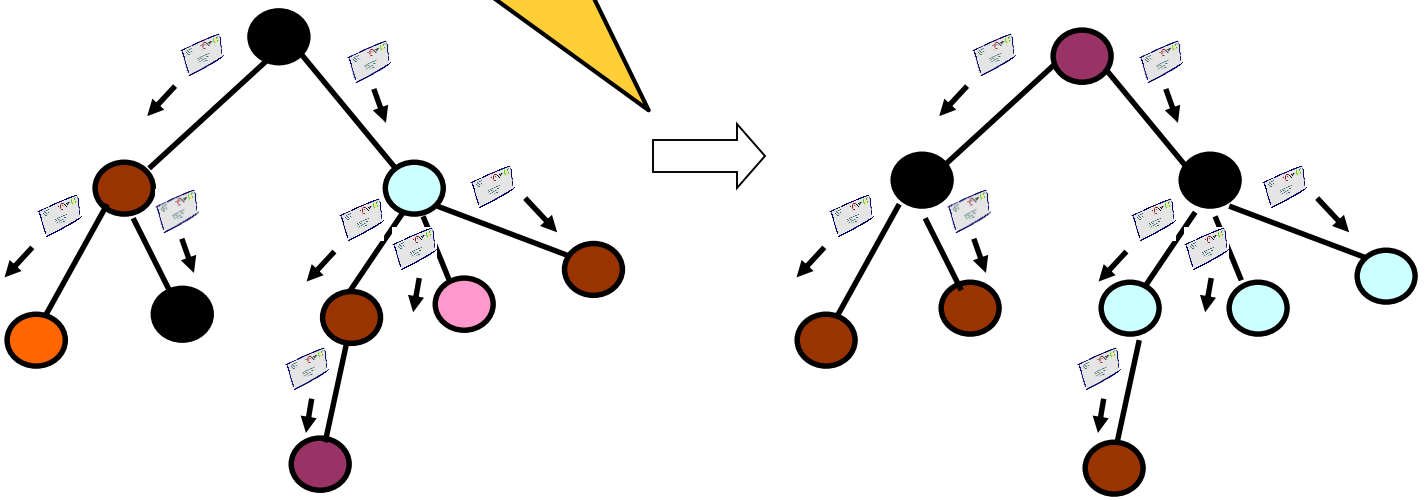
With 6-COLORS algorithm we can get down to 6 colors.

What about improving it to 2 colors?

Impossible: takes linear time.
What about 3 colors?

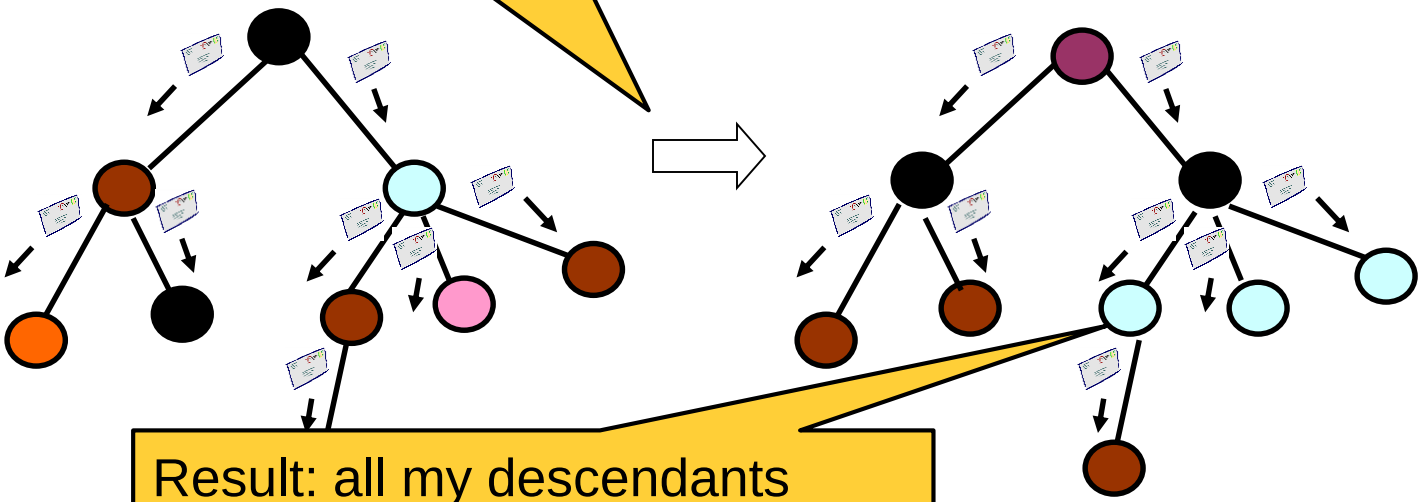
# Observation: Shift Down

Let us note a simple trick: shift colors down by one level makes siblings „independent“. And preserves legal coloring…



# Observation: Shift Down

Let us note a simple trick: shift colors down by one level makes siblings „independent“. And preserves legal coloring…



Result: all my descendants have same color! At most 2 colors are occupied: father and descendants! 3rd color free!

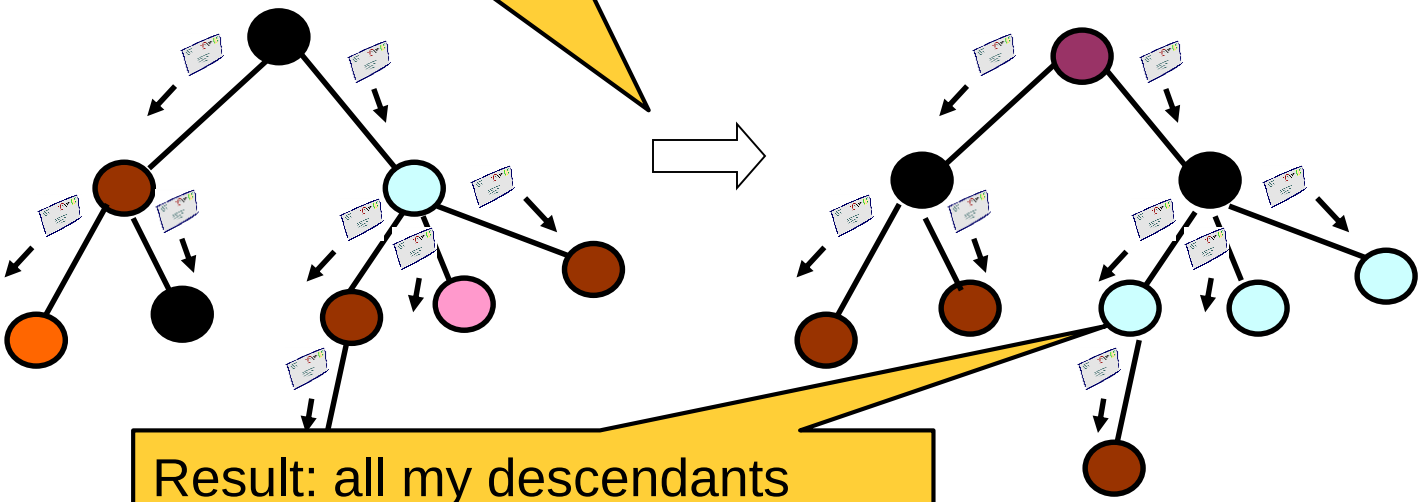
# Observation: Shift Down

**Shift Down**

Each node v concurrently does:
recolor v with color of parent

Formally…

Let us note a simple trick: shift colors down by one level makes siblings „independent“. And preserves legal coloring…



Result: all my descendants have same color! At most 2 colors are occupied: father and descendants! 3rd color free!

## 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 $c_v$ ϵ {0,1,2} according „**first free**“ principle

## 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 $c_v$ ϵ {0,1,2} according „**first free**“ principle

Why still log* n time?

## 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 $c_v$ ϵ {0,1,2} according „**first free**“ principle

Why still log* n time?

Just 3 more rounds!

## 6-to-3

Why not do in same step?

Each 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 $c_v$ ϵ {0,1,2} according „**first free**“ principle

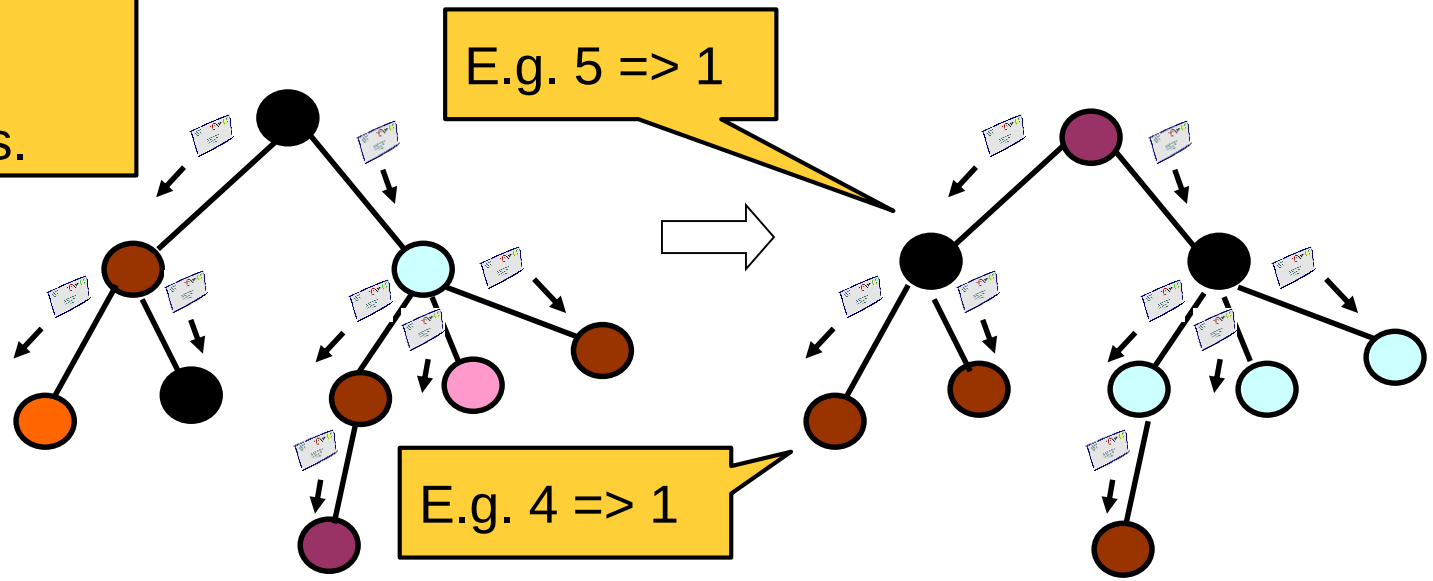
## 6-to-3

Why not do in same step?

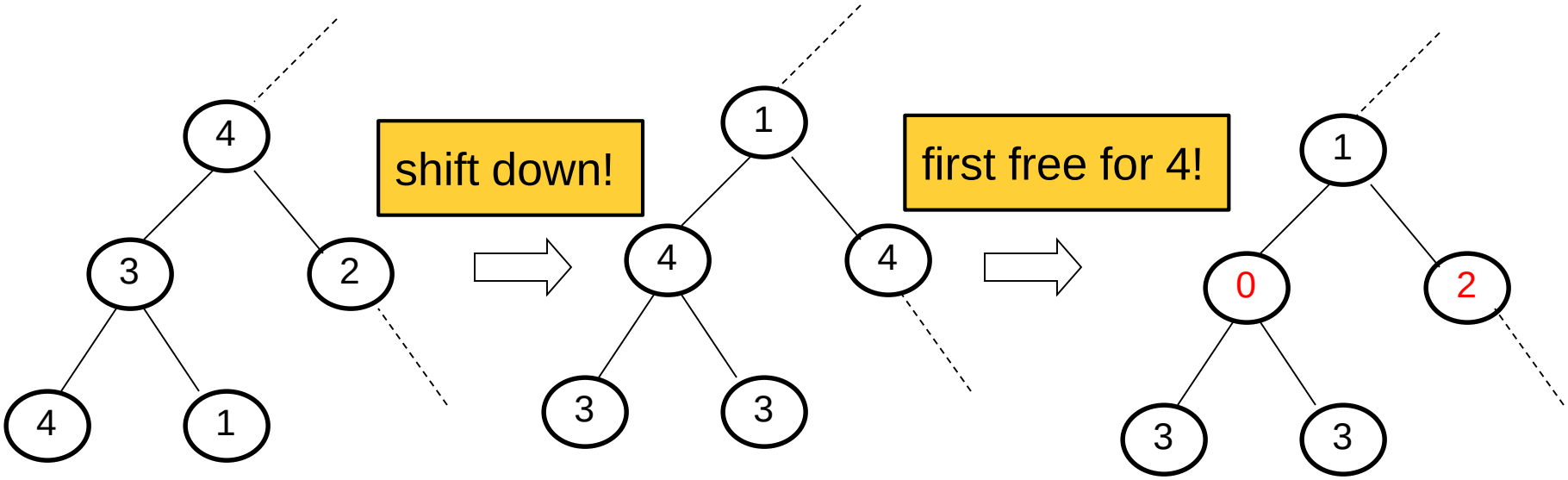
Each node v does (in parallel):

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 $c_v$ ϵ {0,1,2} according „**first free**“ principle

Could be harmful: same 3rd color! Need to do it for independent sets.

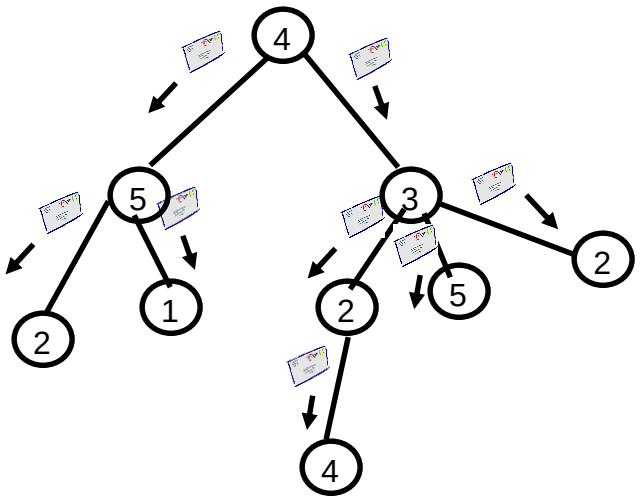


# Example: Shift Down + Drop Color 4

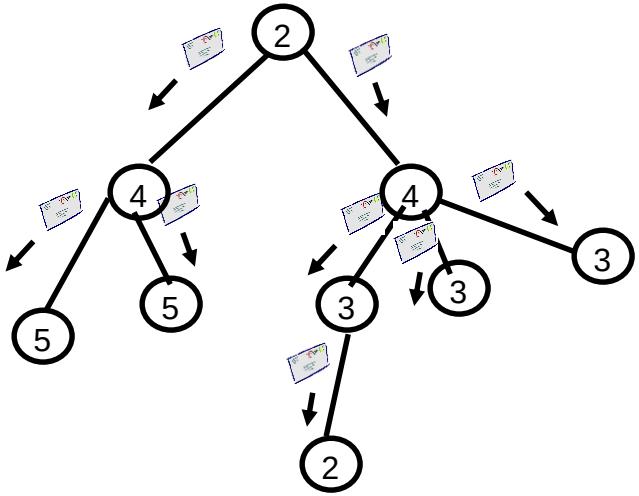



**Siblings no longer have same color: must do shift down again first!**

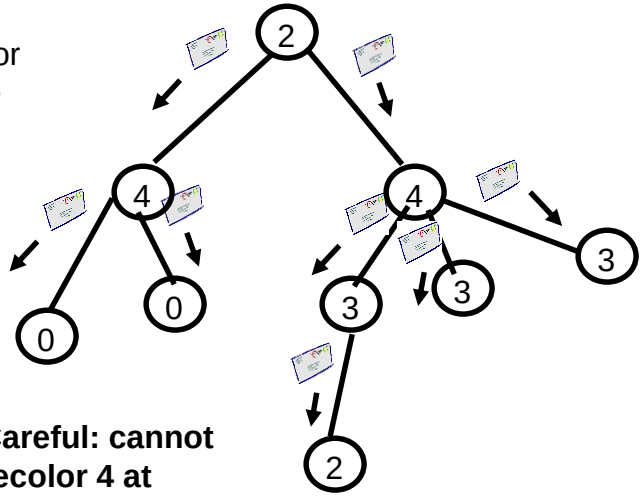
# Example: 6-to-3




**shift down**

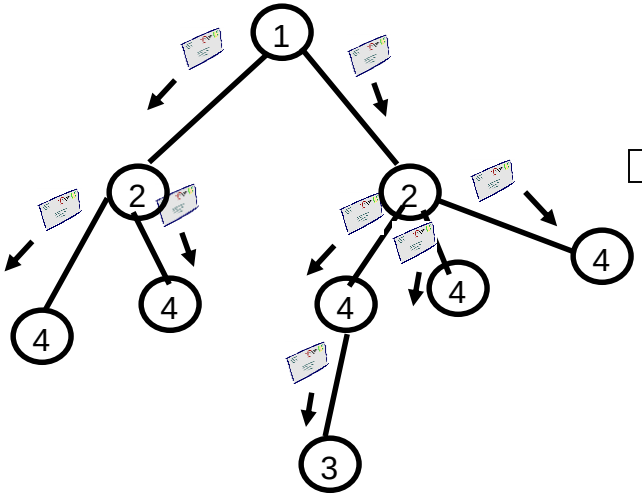



new color for 5: **first free**




**Careful: cannot recolor 4 at same time!**

shift down



# Concluding Remarks

**Can we reduce to 2 colors?**

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

**Simple on purpose: results more general!**

log* runtime is also possible on more general graphs
Many results: see ACM PODC conference!

## Where can I learn more?

- Lecture notes
- Distributed Computing book by David Peleg
- ACM Survey by Jukka Suomela