Topics in Complexity

Please evaluate this course on Axess!

Your feedback really does make a difference.

Applied Complexity Theory

- Complexity theory has enormous practical relevance across various domains in CS.
- In this lecture, we'll explore three of them:
  - **Hardness of approximation**.
  - **Commitment schemes**.
  - **Zero-knowledge proofs**.

Approximation Algorithms

Decision vs. Function Problems

- All of the problems we have encountered this quarter are **decision problems** with yes/no answers.
- Many interesting questions do not have yes/no answers:
  - What is 1 + 1?
  - How many steps does it take this TM to halt on this string?
  - Which seat can I take?
- These questions are called **function problems** and require some object to be found.

NP-Hard Function Problems

- Some function problems are **NP**-hard: there is a polynomial-time reduction from any problem in **NP** to those problems.
- Examples:
  - What is the largest independent set in a graph?
  - What is the length of the longest path in a graph?
  - What is the minimum number of colors required to color a graph?
- A polynomial-time solver for any of these could be used to build a polynomial-time decider for any problem in **NP**.

NP-Hard Function Problems

- The **maximum independent set problem** (called **MAX-INDSET**) is

  Given a graph *G*, find an independent set *S* in *G* of the largest possible size.

- MAX-INDSET is **NP**-hard by a reduction from independent set:
  - M = "On input $\langle G, k\rangle$:
    - Find a maximum independent set in G, call it S.
    - If $|S| \geq k$, accept; otherwise, reject."

NP-Hard Function Problems

- Because they can be used to solve any problem in **NP**, **NP**-hard function problems are believed to be computationally infeasible.
  - If *any* **NP**-hard function problem has a polynomial-time solution, then **P = NP**.
  - Since the **P = NP** question is still open, no NP-hard function problems are known to have polynomial-time solutions.

Approximation Algorithms

- An **approximation algorithm** is an algorithm for yielding a solution that is “close” to the correct solution to a problem.
- The definition of “close” depends on the problem:
  - Maximum independent set: Find an independent set that is not “much smaller” than the maximum independent set.
  - Longest path: Find a long path that is not “much smaller” than the longest path.
  - Graph coloring: Find a coloring of the graph that does not use “many more” colors than the optimal coloring.

How Good is an Approximation?

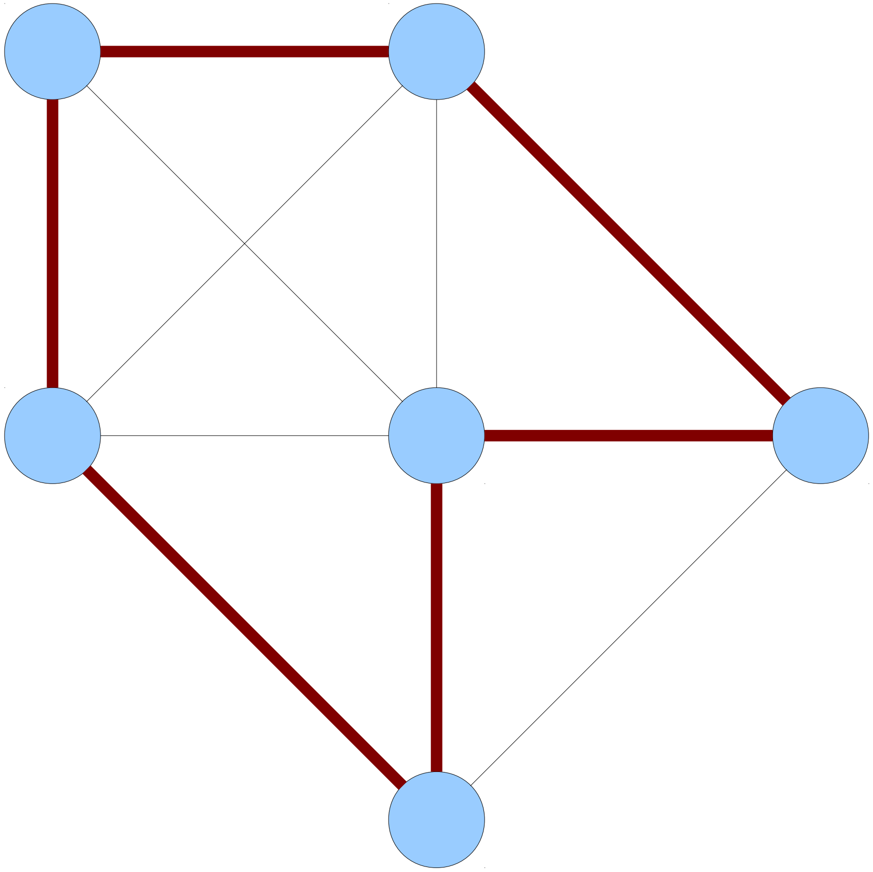
- Approximation algorithms are only useful if there is some connection between the approximate answer and the real answer.
- We say that an approximation algorithm is a ***k*-approximation** if its answer is always within a factor of *k* of the optimal solution.
  - A **2-approximation** to the graph coloring problem always finds a coloring that uses at most twice the optimal number of colors.
  - A **2/3-approximation** to the longest path problem always finds a path that is at least 2/3 as long as the optimal path.

Why Approximations Matter

- Recall from last time: the **job scheduling problem** is **NP**-hard:

  Given a set of tasks and a set of workers to perform the tasks, how do you distribute tasks to workers to minimize the total time required (assuming the workers work in parallel?)

- Although finding an optimal solution is **NP**-hard, there are polynomial-time algorithms for finding **4/3-approximate** solutions.
- If we just need a “good enough” answer, the **NP**-hardness of job scheduling is not a deterrent to its use.



A **Hamiltonian cycle** in an undirected graph G is a simple cycle that visits every node in G.

Hamiltonian Cycles

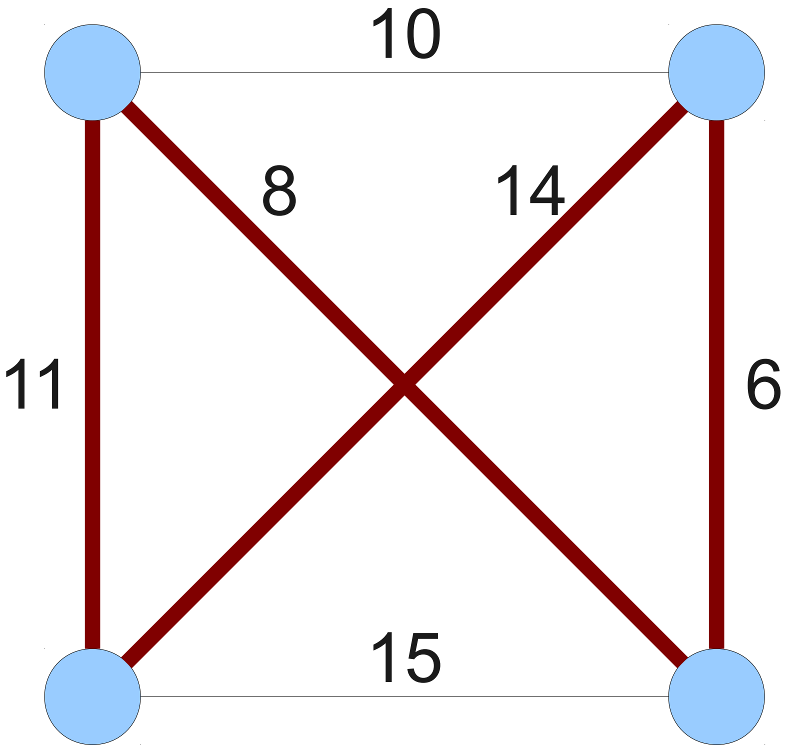
- The **undirected Hamiltonian cycle problem** is

  Given an undirected graph *G*,
  does *G* contain a Hamiltonian cycle?

- As a formal language:

  UHAMCYCLE = { ⟨*G*⟩ | *G* is an undirected graph containing a Hamiltonian cycle }

- **Important fact**: *UHAMCYCLE* is **NP**-complete.
  - Reduction from *UHAMPATH*.



Cost: **39**

(This is the optimal solution)

Given a complete, undirected, weighted graph *G*, the **traveling salesman problem** (**TSP**) is to find a Hamiltonian cycle in *G* of least total cost.

TSP, Formally

- Given as input
  - A **complete**, **undirected** graph *G*, and
  - a set of **edge weights**, which are positive integers,

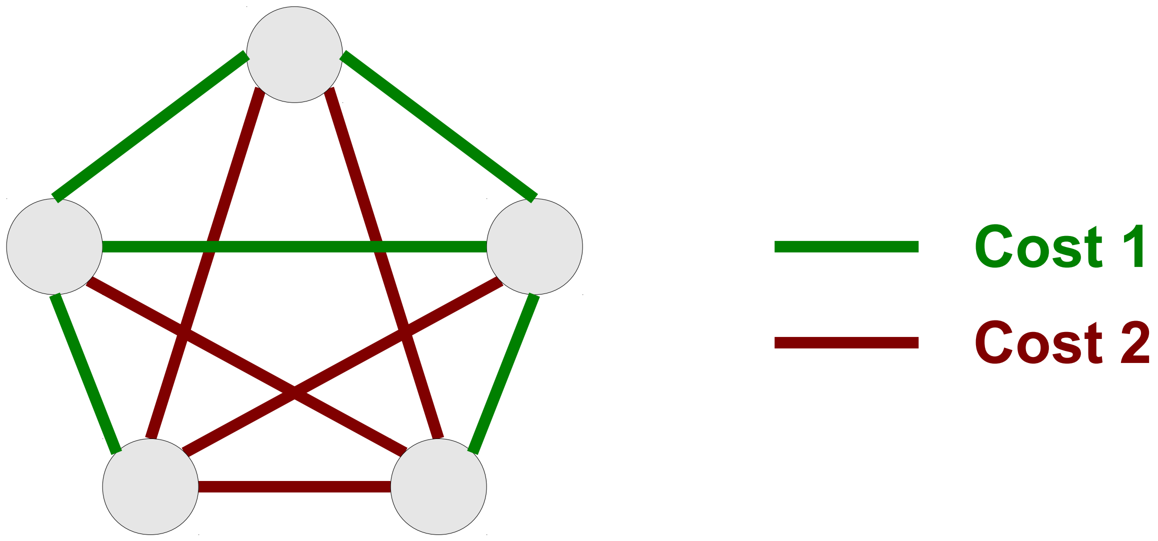
  the **TSP** is to find a Hamiltonian cycle in *G* with least total weight.
- Note that since *G* is complete, there has to be at least one Hamiltonian cycle. The challenge is finding the least-cost cycle.

TSP

- There are **many** variations on TSP:
  - Directed versus undirected graphs.
  - Complete versus incomplete graphs.
  - Finding a path versus finding a cycle.
  - Nonnegative weights versus arbitrary weights.
- All of these problems are known to be **NP**-hard.
- The best-known algorithms have horrible runtimes: $O(n^2 2^n)$.

TSP is **NP**-Hard

- To show that TSP is **NP**-hard, we reduce the undirected Hamiltonian cycle problem to TSP.



If there is a Hamiltonian cycle in the original graph, there is a TSP solution of cost n in the new graph (where n is the number of nodes)

If there is no Hamiltonian cycle, the TSP solution has cost at least n + 1.

Answering TSP

- TSP has great practical importance, and we want to be able to answer it.
    - Determining the shortest subway routes to link stations in a ring.
    - Determining the fastest way to move a mechanical drill so it can drill in the appropriate locations.
- Because TSP is **NP**-hard, obtaining an exact answer is impractically hard.
- Can we approximate it?

Approximating TSP

- An approximation algorithm to TSP is a k-approximation if it finds a Hamiltonian cycle whose cost is at most k times the optimal solution.
  - If the optimal solution has cost 10, a 2-approximation would return a Hamiltonian cycle of cost between 10 and 20, inclusive.
  - If the optimal solution has cost 10, a 3-approximation would return a Hamiltonian cycle of cost between 10 and 30, inclusive.
- For what values of k is there an efficient k-approximation to TSP?

Theorem: If $\mathbf{P} \neq \mathbf{NP}$, then there is no polynomial-time k-approximation to TSP for any natural number k.

Hardness of Approximation

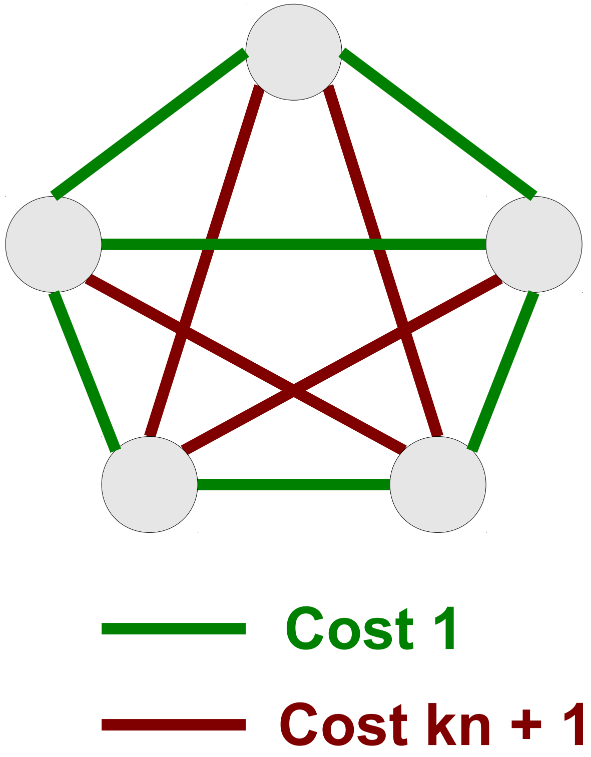
- The proof that TSP is hard to approximate is based on a beautiful construction:

If we could obtain a k-approximation to TSP in polynomial-time, we would have a polynomial-time algorithm for *UHAMCYCLE*.

- Since *UHAMCYCLE* is **NP**-complete, this is a contradiction if **P** ≠ **NP**.

The Construction

- **Proof Sketch**: Suppose, for the sake of contradiction, that there is a *k*-approximation to TSP for some *k*.



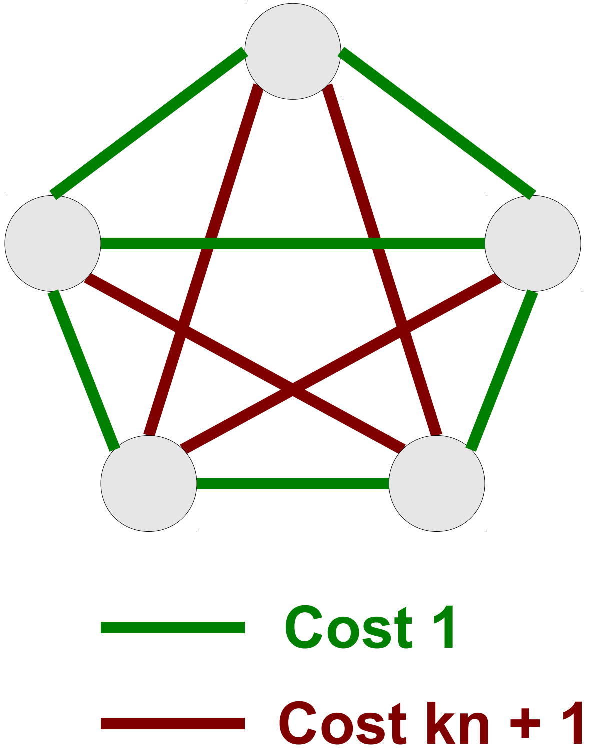
If the original graph of n nodes has a Hamiltonian cycle, the TSP solution has cost n.

The k-approximation algorithm thus must hand back a Hamiltonian cycle of cost at most kn.

What if we made the cost of the red edges so large that any solution using them must cost more than this?

The Construction

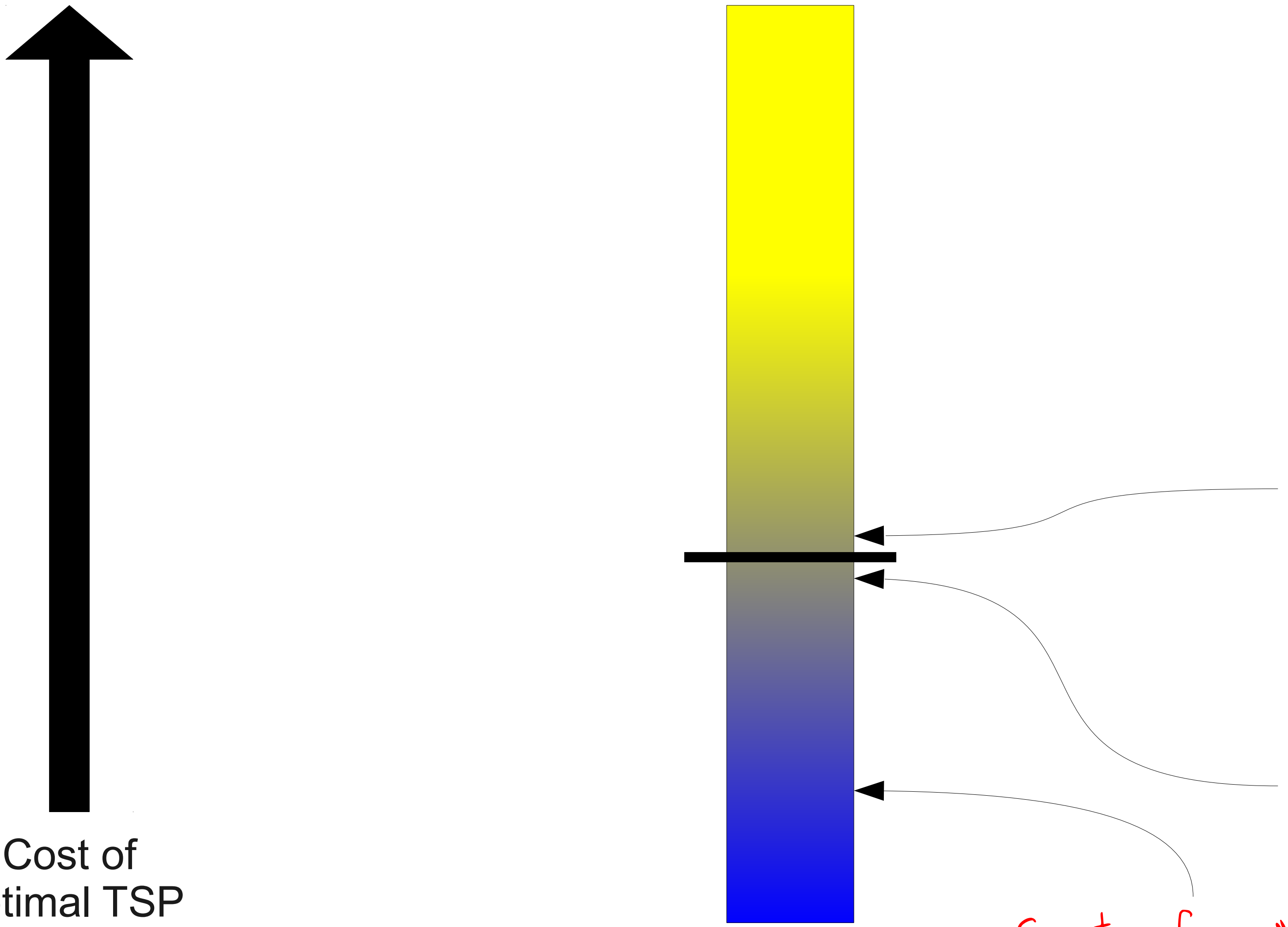
- **Proof Sketch**: Suppose, for the sake of contradiction, that there is a k-approximation to TSP for some k.



If the k-approximation hands back a solution of cost at most kn, the original graph has a Hamiltonian cycle.

If the k-approximation hands back a solution of cost at least kn + 1, the original graph has no Hamiltonian cycles.

What Just Happened?



Cost of optimal TSP solution

Cost of the best possible "no" answer.

k times the cost of a "yes" answer.

Cost of a "yes" answer.

What Just Happened?

- Create an enormous **gap** between the TSP answer for “yes” and “no” instances of the Hamiltonian cycle problem.
- Make the gap so large that the worst possible k-approximate answer to a “yes” instance is distinguishable from the best possible “no” instance.
- Decide whether there is a Hamiltonian cycle by measuring this difference.

The PCP Theorem

Approximating 3SAT

- 3SAT is a canonical **NP**-complete problems.
- What would it mean to approximate a 3SAT answer?

Approximating 3SAT

- The **MAX-3SAT** problem is

  Given a 3CNF formula φ, find a satisfying assignment the maximizes the number of satisfied clauses.

- **Idea**: If we can satisfy the entire formula, then MAX-3SAT finds a satisfying assignment. If not, MAX-3SAT finds the "best" assignment that it can.
- There is a known randomized 7/8-approximation algorithm for MAX-3SAT.
- Is it possible to do better?

The 3-CNF Value

- If φ is a 3-CNF formula, the **value of φ** (denoted **val(φ)**) is the fraction of the clauses of φ that can be simultaneously satisfied by any assignment.
- If φ is satisfiable, val(φ) = 1.
  - All of the clauses are satisfiable.
- If φ is unsatisfiable, val(φ) < 1.
  - Some (but not all) clauses can be satisfied.

The PCP Theorem

For any language $L \in \mathbf{NP}$,

There exists a poly-time reduction f from L to MAX-3SAT so

For any string w:

If $w \in L$, then $\text{val}(f(w)) = 1$.

If the original answer is "yes," the 3-CNF formula is satisfiable.

If $w \notin L$, then $\text{val}(f(w)) < 7/8$.

If the original answer is "no," fewer than 7/8 of the clauses can be satisfied.

What Does This Mean?

- Our proof that (unless **P = NP**) TSP cannot be efficiently approximated works by building up a gap between “yes” answers and “no” answers.
- The PCP theorem states that **any problem in NP** can be reduced to MAX-3SAT such that
  - All of the clauses are satisfiable if the original answer is “yes.”
  - Fewer than 7/8 of the clauses are satisfiable if the answer is “no.”

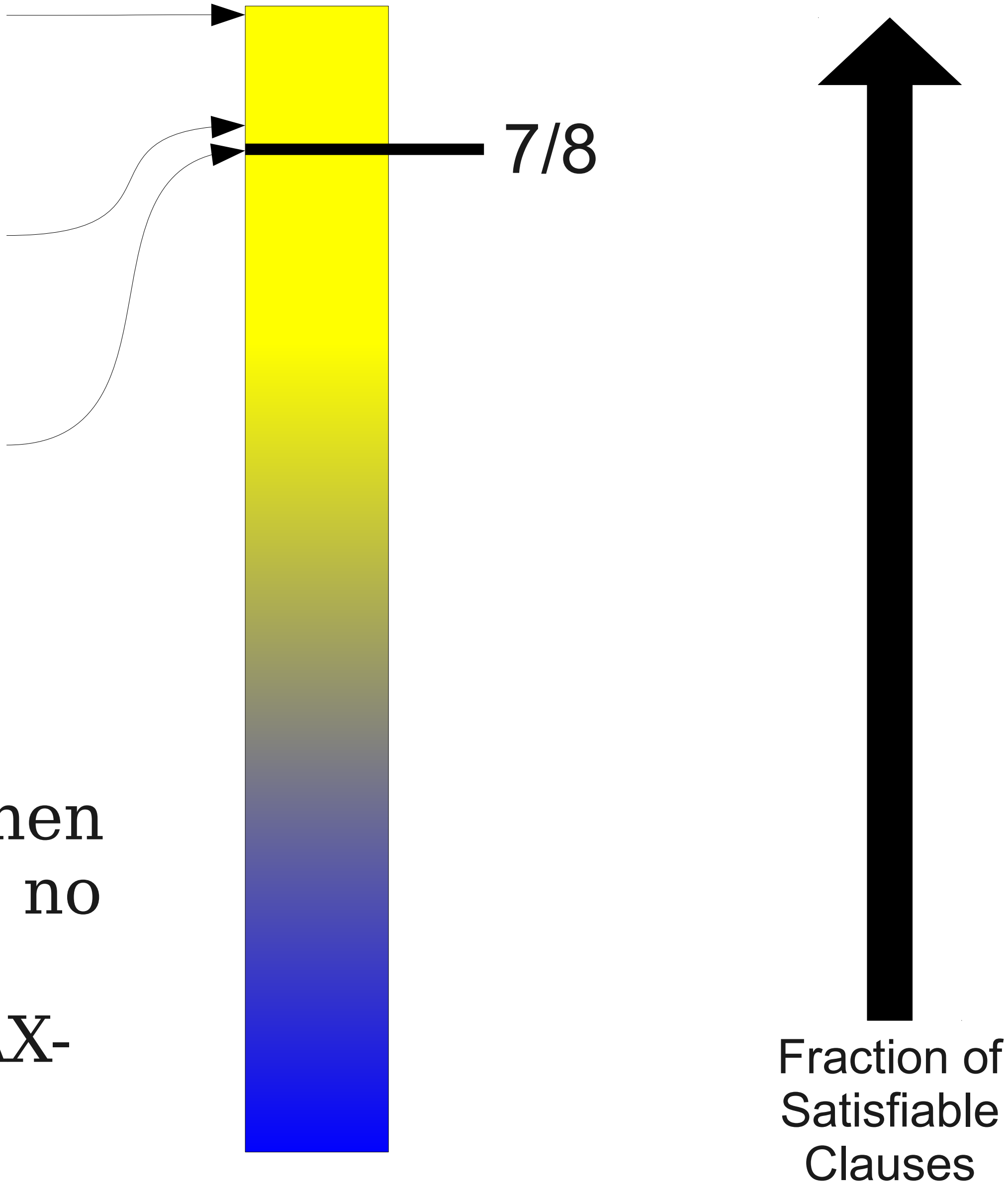
Theorem: If $\mathbf{P} \neq \mathbf{NP}$, then for any $r \geq 7/8$, there is no polynomial-time r-approximation to MAX-3SAT.

MAX-3SAT is Inapproximable

Fraction satisfiable in a "yes" answer

r times the fraction satisfiable in a "yes" answer

Fraction satisfiable in a "no" answer

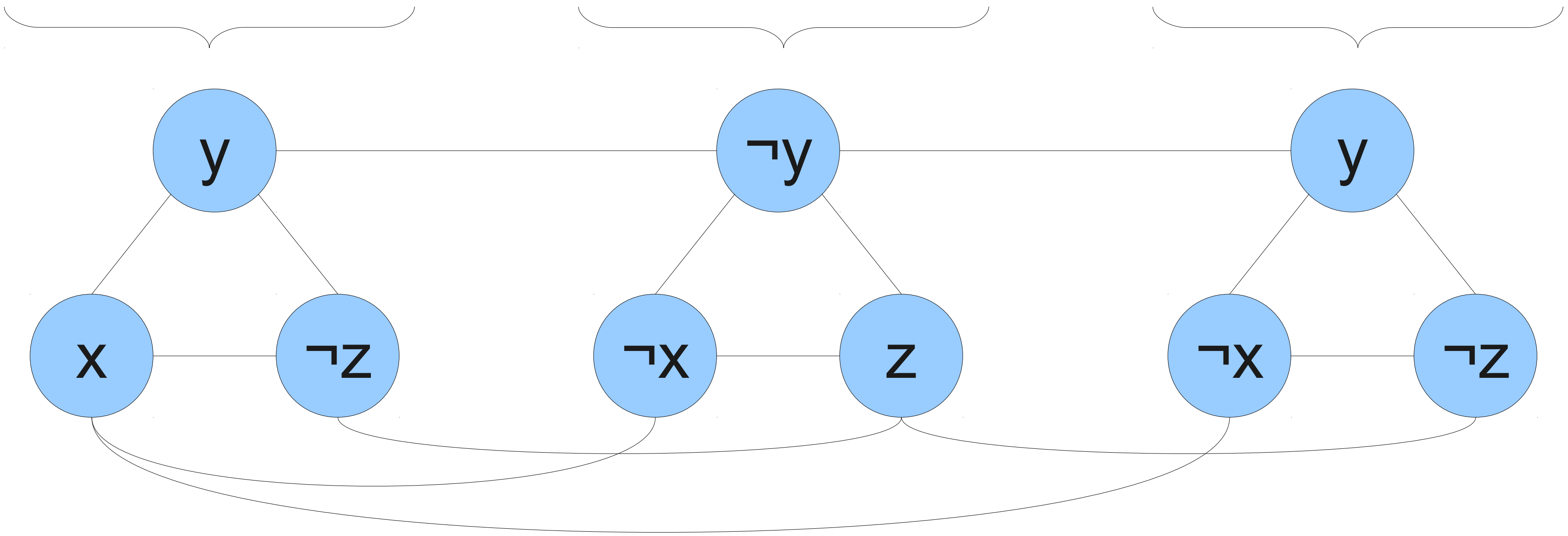


Theorem: If **P ≠ NP**, then for any $r \geq 7/8$, there is no polynomial-time r-approximation to MAX-3SAT.

Effects on Approximability

- Assuming **P ≠ NP**, there is a limit to how well we can approximate 3SAT.
- Look at our reduction from 3SAT to INDSET:

$$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$



Effects on Approximability

- Many reductions preserve the size of some difficult-to-approximate quantity.
- Assuming if **P ≠ NP**:
  - Because MAX-3SAT is not efficiently 7/8-approximable, MAX-INDSET is not efficiently 7/8-approximable either.
  - Because MAX-INDSET is not efficiently 7/8-approximable, MAX-SETPACK is not efficiently 7/8-approximable.
- Not all reductions have this property; some **NP**-hard problems can be efficiently approximated to very high precision.

Proving the PCP Theorem

Probabilistic Checking of Proofs and Hardness of Approximation Problems

Sanjeev Arora

CS-TR-476-94

(Revised version of a dissertation submitted at
CS Division, UC Berkeley, in August 1994)

Assume that $\sqrt{3\delta} < 10^{-3}$. Hence if plane C is not good for y then $I_C < 1 - \sqrt{3\delta}$. The Averaging Principle implies that

$$\Pr_{C \in \mathcal{C}}[\text{“}C \text{ is not good for } y\text{”}] = \mathrm{I}$$

Hence the probability that C fails the condi

$$\Pr_{C}[\text{“}C \text{ is not good for } b\text{”}] + \Pr_{C}[\text{“}C \text{ is not nic}$$

Assuming $1000\sqrt{3\delta} + \sqrt{3\delta} < 1$, the probabil aim is less than 1. Hence there exists a plane meeti

□

Summary of Approximability

- While many **NP**-hard problems can be efficiently approximated, certain **NP**-hard problems cannot unless **P = NP**.
- Efficiently approximating some **NP**-hard problems too well would solve **NP**-complete problems.
- The PCP theorem states that the gap between yes and no answers can be so large that approximating the solution would essentially solve the problem.

NP-Completeness and Cryptography

Cryptography

- **Cryptography** is the study of sending and receiving messages securely.
- Studies questions like
  - How can I send a message to you such that no one else can decode that message?
  - How can I convince you that I know something without revealing what that something is?
  - How can we exchange secret information while people are watching?
- Strong practical relevance and huge theoretical underpinnings.

Commitment Scheme

- A **commitment scheme** is a cryptographic protocol that allows someone to
  - **commit** to making some choice (without revealing what that choice is), and later
  - **reveal** what that choice was later on.
- Think about the "guess the number game."
- Lots of practical applications; we'll see one in a minute.

One-Way Functions

- A **one-way function** is a function $f : \Sigma^* \to \Sigma^*$ with the following properties:
  - $|f(x)| = |x|$ (the function always maps input strings to output strings of the same length).
  - Given $x \in \Sigma^*$, $f(x)$ can be computed in polynomial time.
  - Given $y \in \Sigma^*$, there is no polynomial-time algorithm for finding some $x \in \Sigma^*$ such that $f(x) = y$.
- In other words, it is easy to *evaluate* the function, but difficult to *invert* the function.

One-Way Functions and Commitments

- Given a one-way injective function *f*, it is easy to build a commitment scheme:
  - To **commit** to a choice *x*, compute *f*(*x*) and share it with everyone.
  - To **reveal** a commitment, reveal *x*. We can verify the commitment by computing *f*(*x*).
  - (There are some other details here I'm glossing over; take CS255 for a more thorough construction.)
- Given just *f*(*x*), there is no efficient algorithm for recovering *x*.

Do one-way functions exist?

Theorem: If a one-way function f exists, then $\mathbf{P} \neq \mathbf{NP}$.

One-Way Functions

- ***Theorem:*** If a one-way function f exists, then $\mathbf{P} \neq \mathbf{NP}$.
- To prove this, we can do the following:
  - Construct a language L based on the behavior of f.
  - Show $L \in \mathbf{NP}$.
  - Show $L \notin \mathbf{P}$.

One-Way Functions

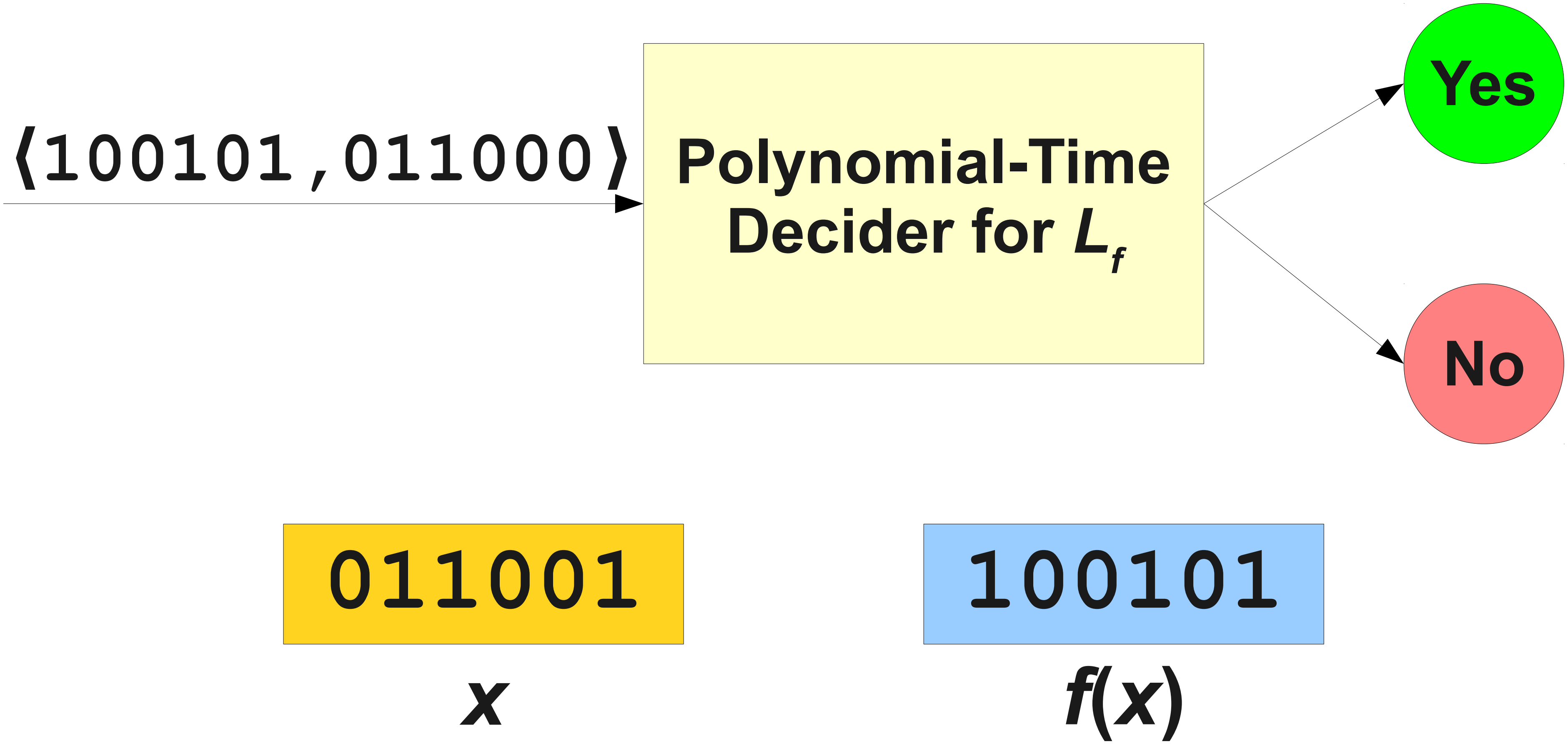
- Suppose $f : \Sigma^* \to \Sigma^*$ is a one-way function.
- Consider the language L_f:

$$\boldsymbol{L_f = \{ \langle y, w \rangle \mid \exists x \in \Sigma^*.\ f(wx) = y \}}$$

- In other words, w can be extended into a string that maps to y.
- $L_f \in$ **NP**, since we could build a polynomial-time verifier for it.
  - (How?)

$L_f = \{ \langle y, w \rangle \mid \exists x \in \Sigma^*.\ f(wx) = y \}$

- Suppose that $L_f \in \mathbf{P}$.
- Then given y, we can find an x where $f(x) = y$ in polynomial time.



⟨100101,011000⟩

Polynomial-Time Decider for L_f

Yes

No

011001

x

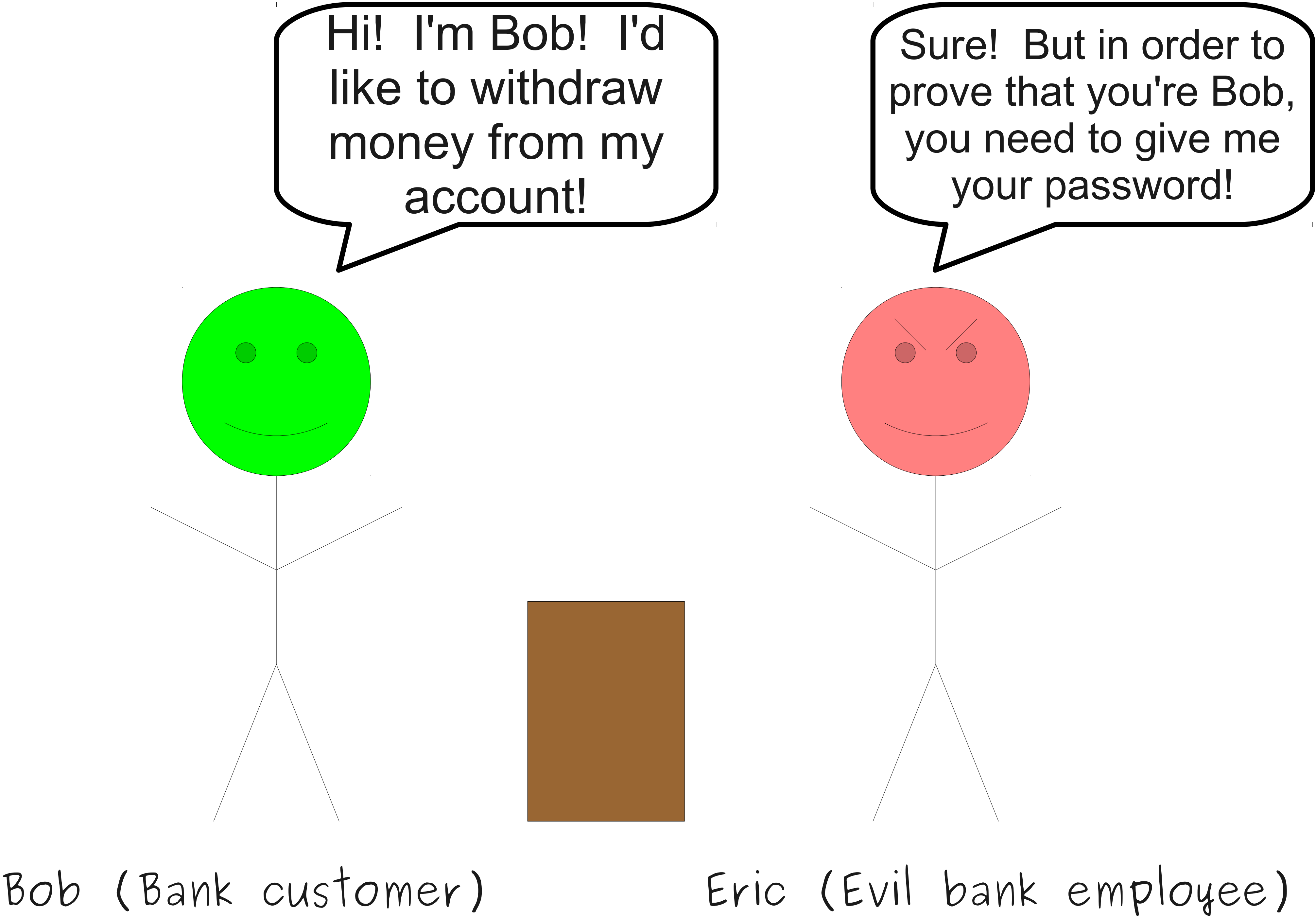
100101

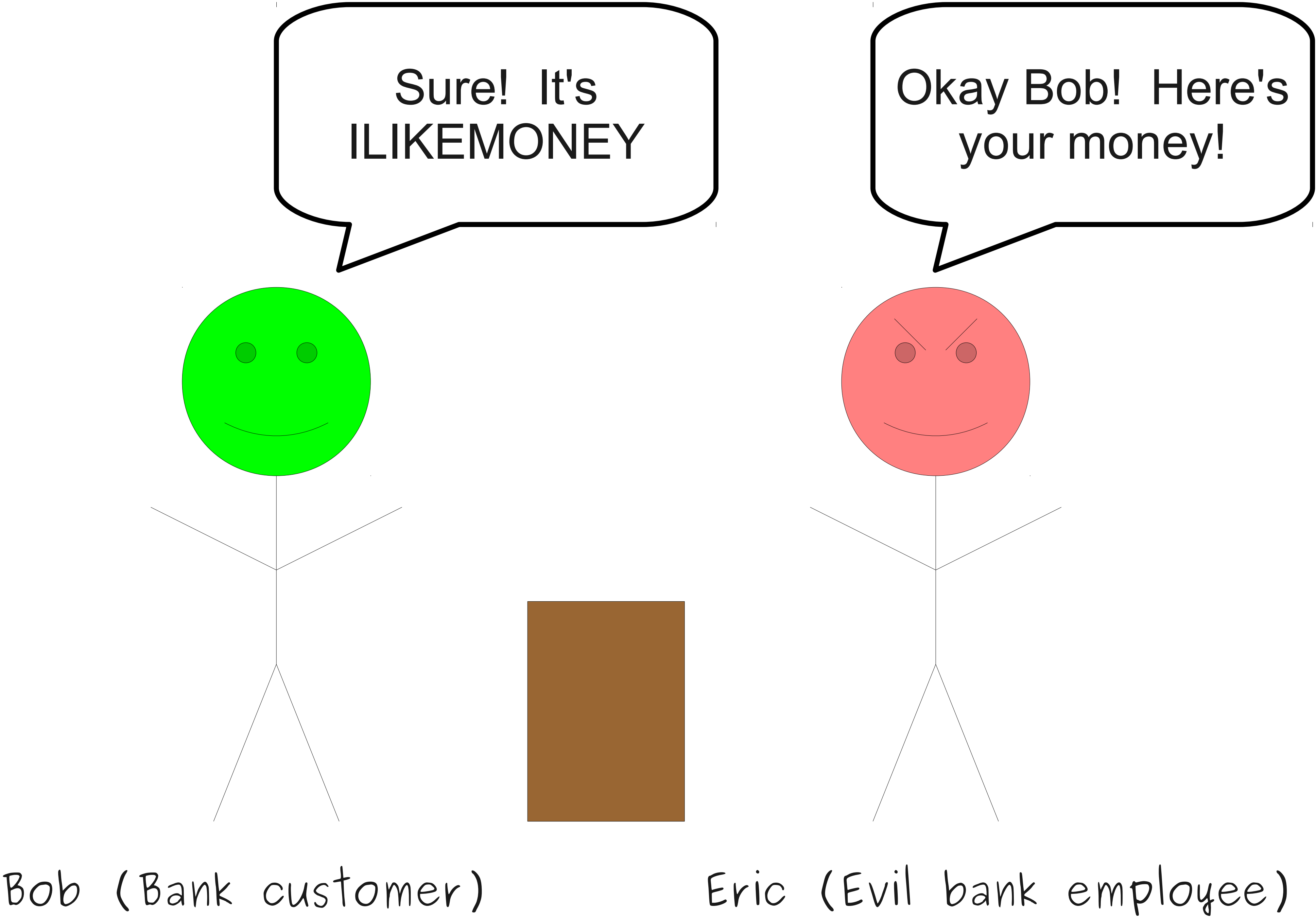
$f(x)$

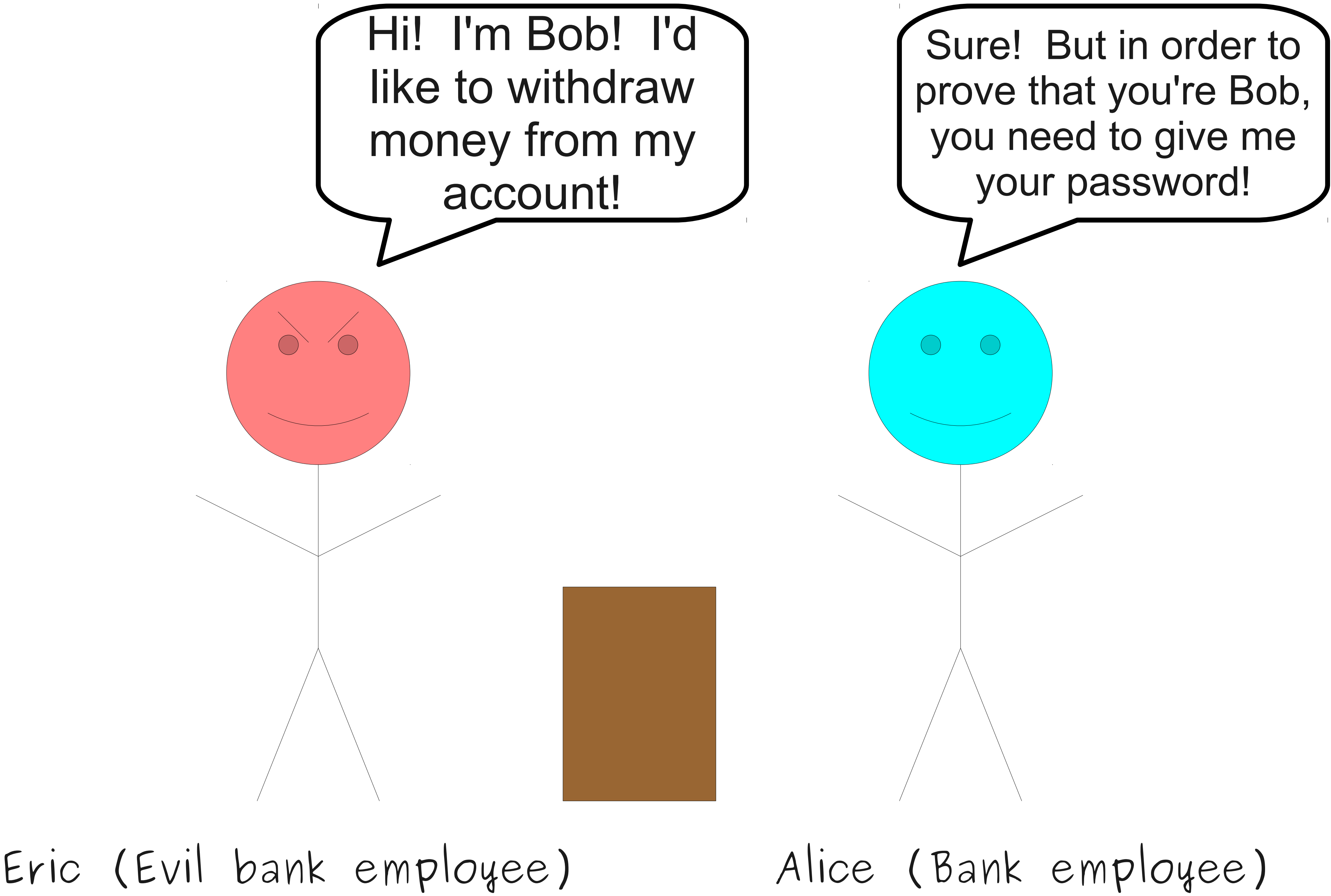
Why This Matters

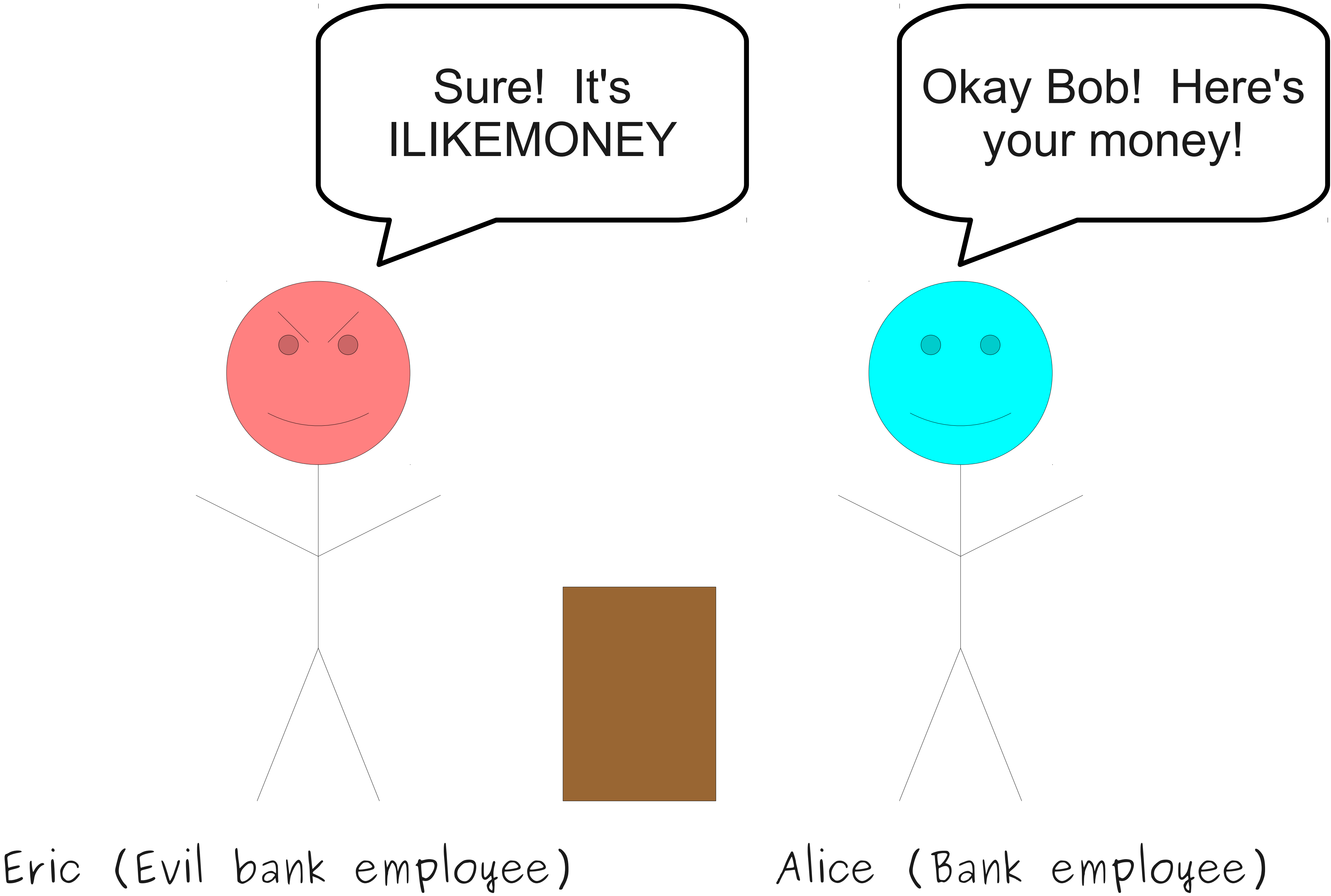
- Many building blocks in cryptography have not been proven to exist.
- Sometimes, their existence implies **P ≠ NP**.
- **These are not purely theoretical complexity classes!**

Zero-Knowledge Proofs


Hi! I'm Bob! I'd like to withdraw money from my account!
Sure! But in order to prove that you're Bob, you need to give me your password!
Bob (Bank customer)
Eric (Evil bank employee)


Sure! It's ILIKEMONEY
Okay Bob! Here's your money!
Bob (Bank customer)
Eric (Evil bank employee)


Hi! I'm Bob! I'd like to withdraw money from my account!
Sure! But in order to prove that you're Bob, you need to give me your password!
Eric (Evil bank employee)
Alice (Bank employee)


Sure! It's ILIKEMONEY
Okay Bob! Here's your money!
Eric (Evil bank employee)
Alice (Bank employee)

Authentication

- In many cases, someone needs to authenticate by proving that they are who they claim to be.
- Passwords are a common solution, but are seriously flawed.
- Is there a better way to do this?

Zero-Knowledge Proofs

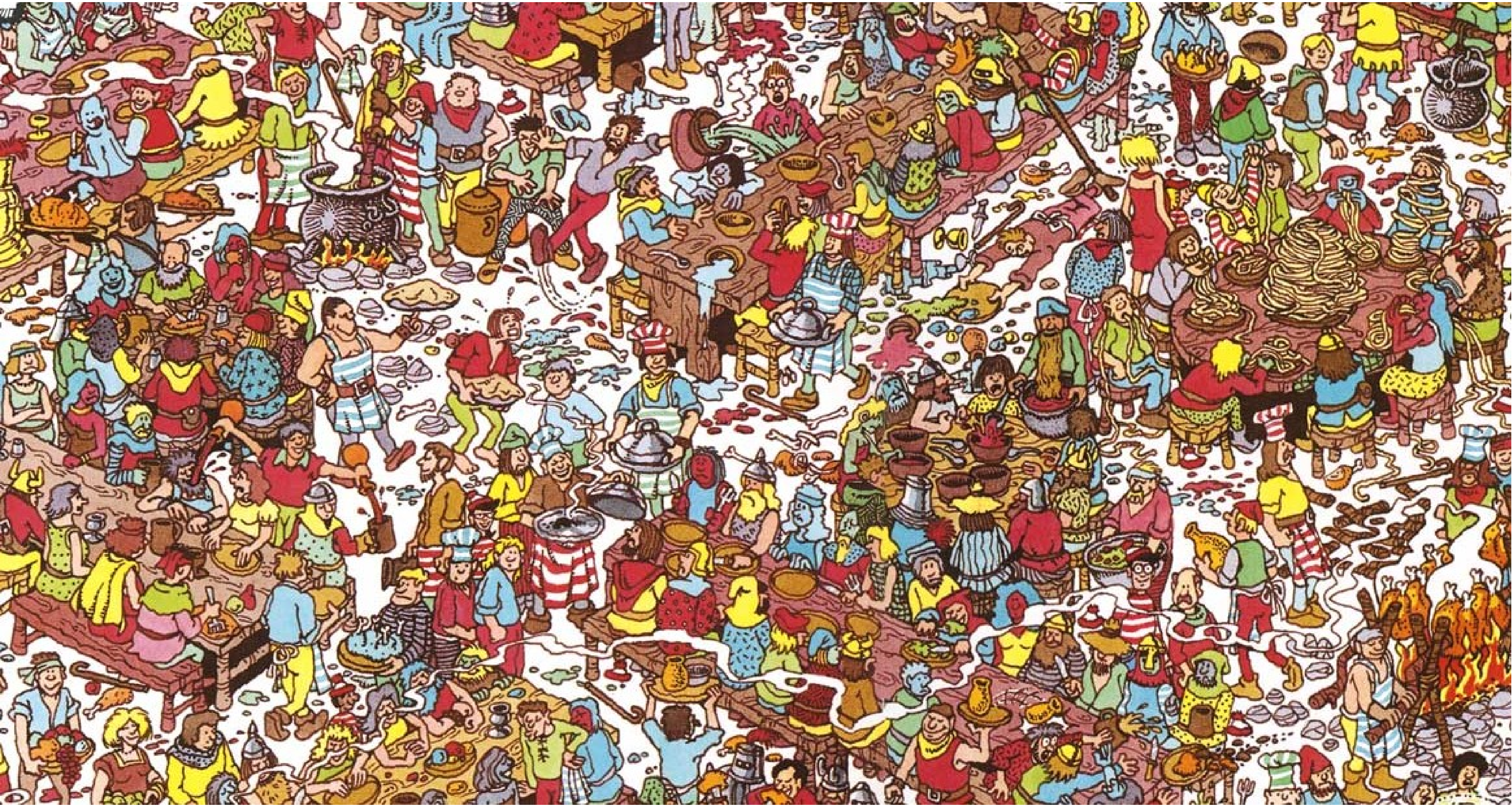
- A **zero-knowledge proof** is a system in which one party (the **prover**) can convince a second party (the **verifier**) that the prover has some knowledge without the verifier ever learning that knowledge.
- In other words:
  - The **verifier** wants to check that the **prover** knows something.
  - The **prover** convinces the **verifier** beyond a reasonable doubt that they do indeed know something.
  - At the end, the **verifier** has not learned any information.

Zero-Knowledge Proofs and Passwords

- Zero-Knowledge proofs can be used as a nifty replacement for passwords.
- To log in to a system, you (the prover) can convince that system (the verifier) that you are who you claim to be.
- In doing so, the verifier could not impersonate you later on.

How can you possibly build a zero-knowledge proof system?

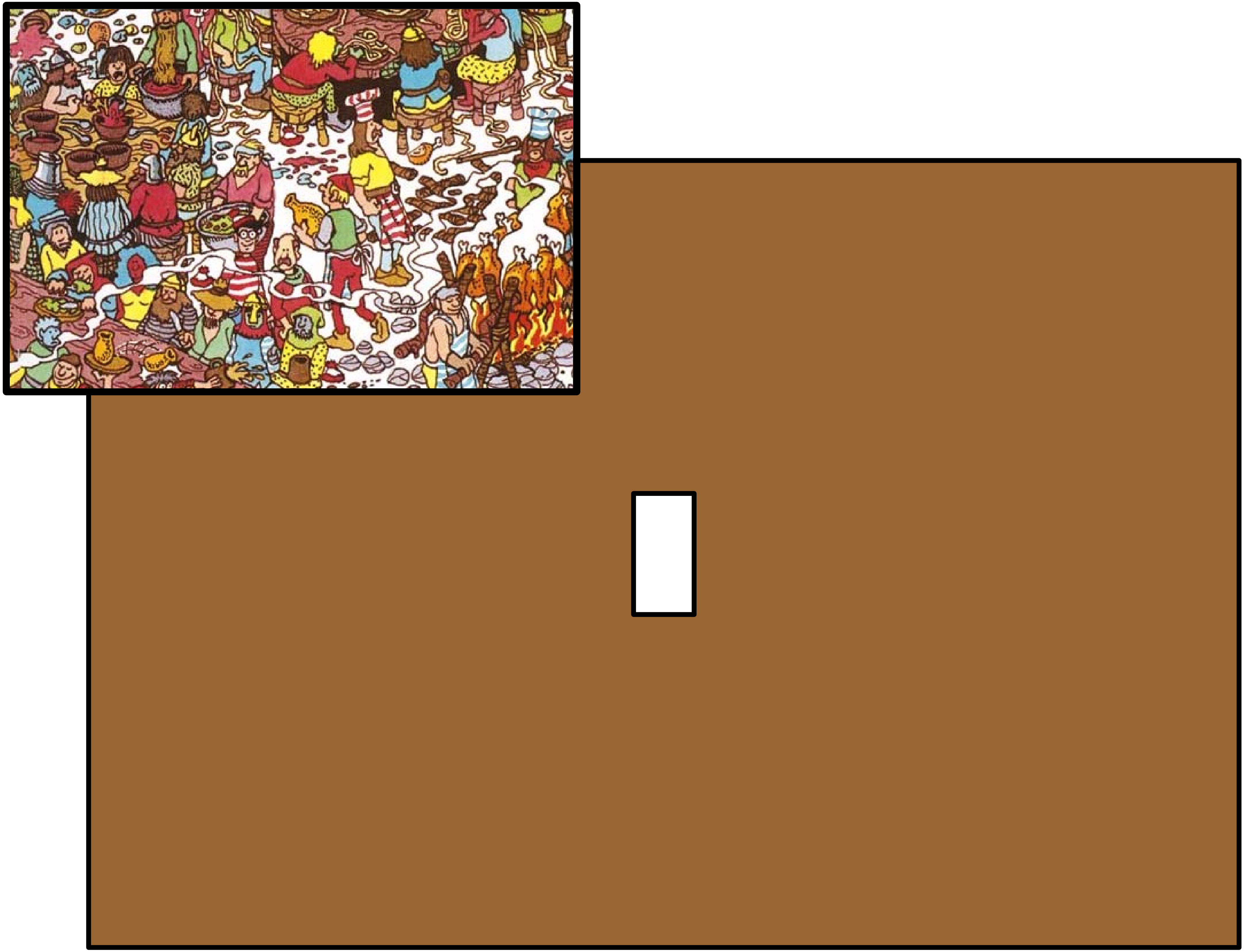
Where's Waldo?

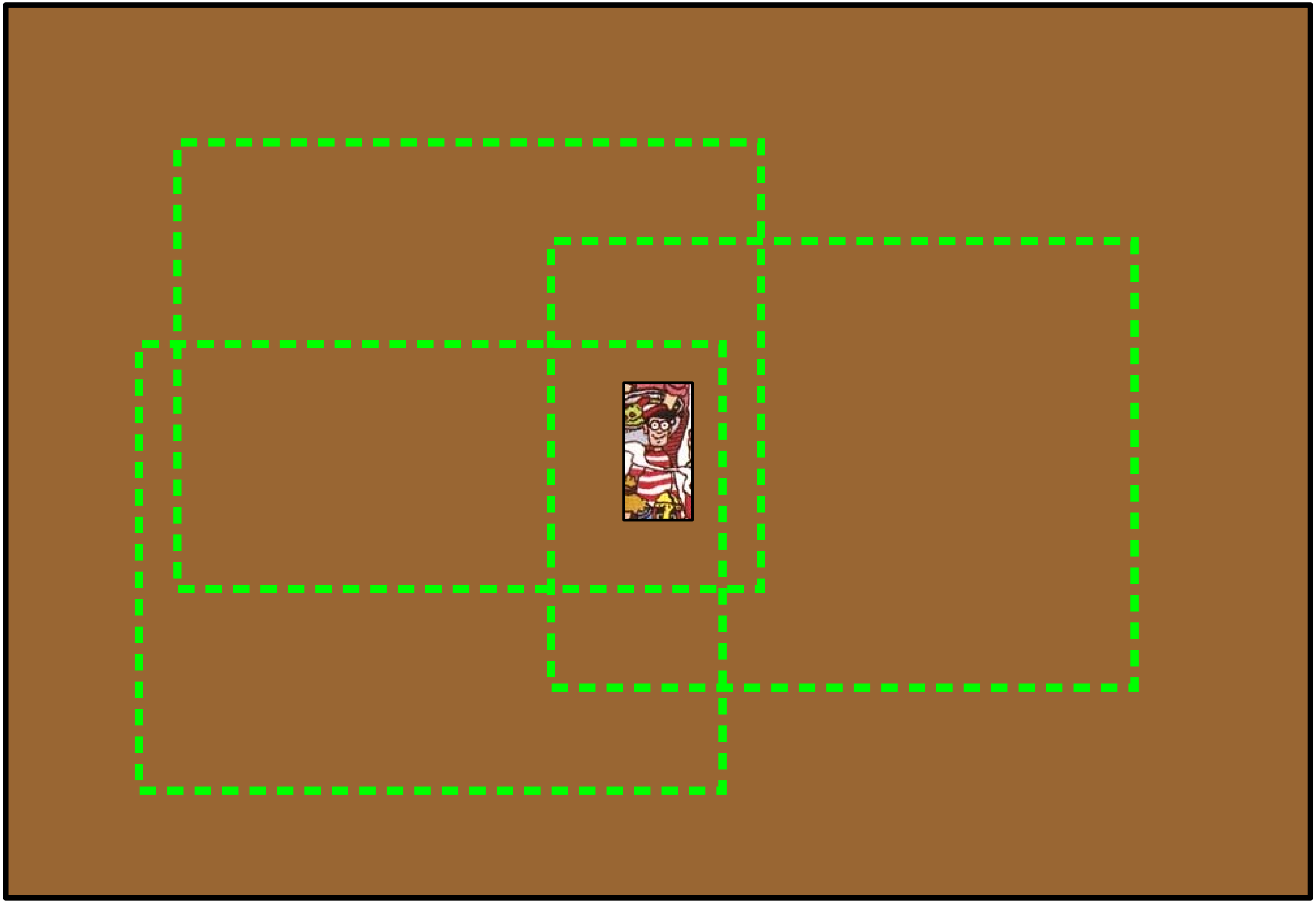


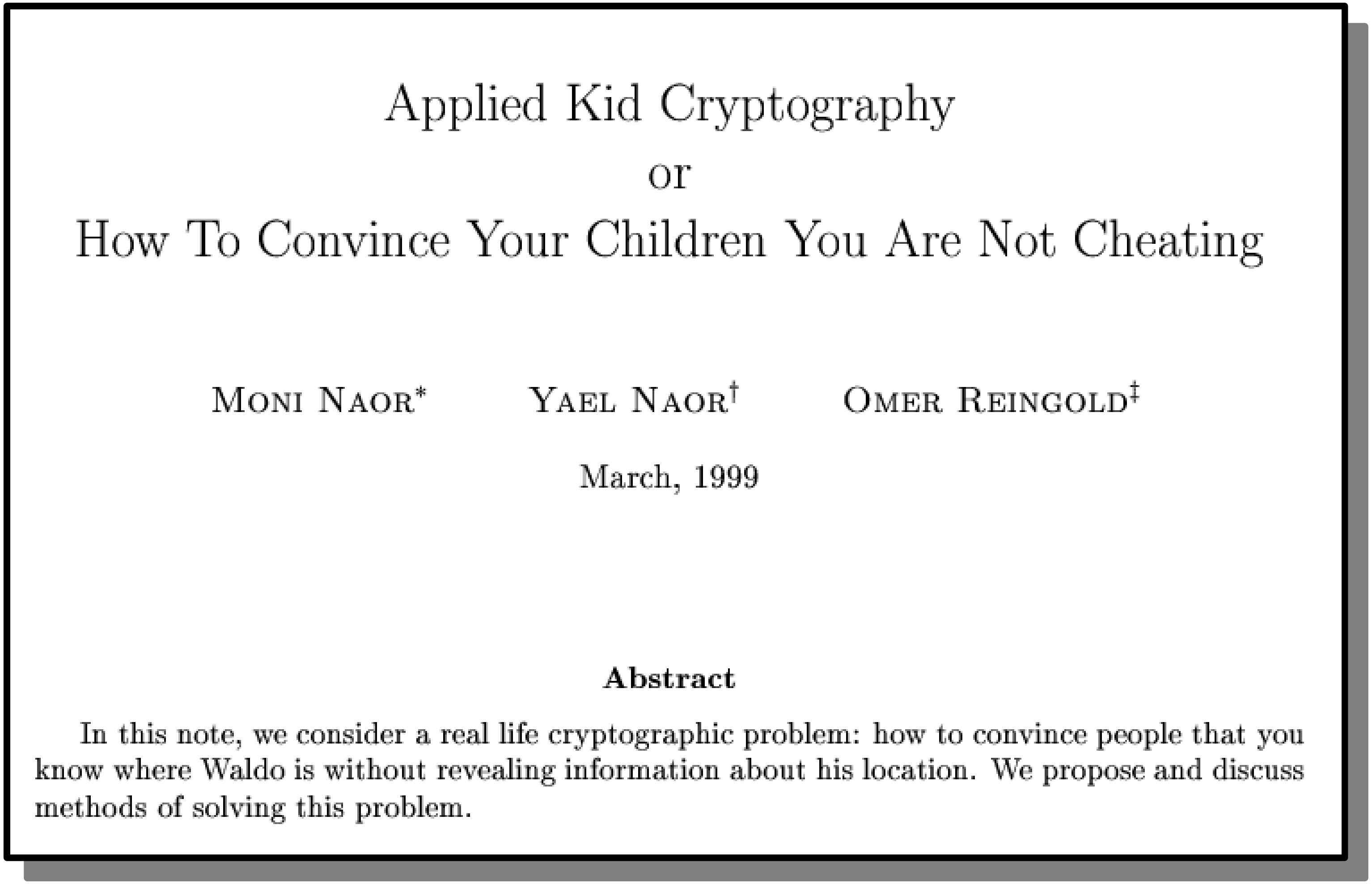
Source: http://www.findwaldo.com/maps/gluttons/gluttons_small.jpg

Zero-Knowledge Proofs of Waldo

- Suppose that I know with certainty where Waldo is.
- I want to convince you that I know his location without revealing his position.
- This is a zero-knowledge proof for “Where's Waldo?”
- How might I do this?







Applied Kid Cryptography
or
How To Convince Your Children You Are Not Cheating

Moni Naor[*] Yael Naor[†] Omer Reingold[‡]

March, 1999

Abstract

In this note, we consider a real life cryptographic problem: how to convince people that you know where Waldo is without revealing information about his location. We propose and discuss methods of solving this problem.

Zero-Knowledge Proofs of Waldo

- If I know where Waldo is, I can position the book under the cardboard so that you can see Waldo, but not where he is relative to the rest of the picture.
- You only learn that I know where Waldo is, but you already knew that!
- If I **don't** know where Waldo is, you will discover this very quickly!

A More Elaborate Zero-Knowledge Proof

3-Colorability

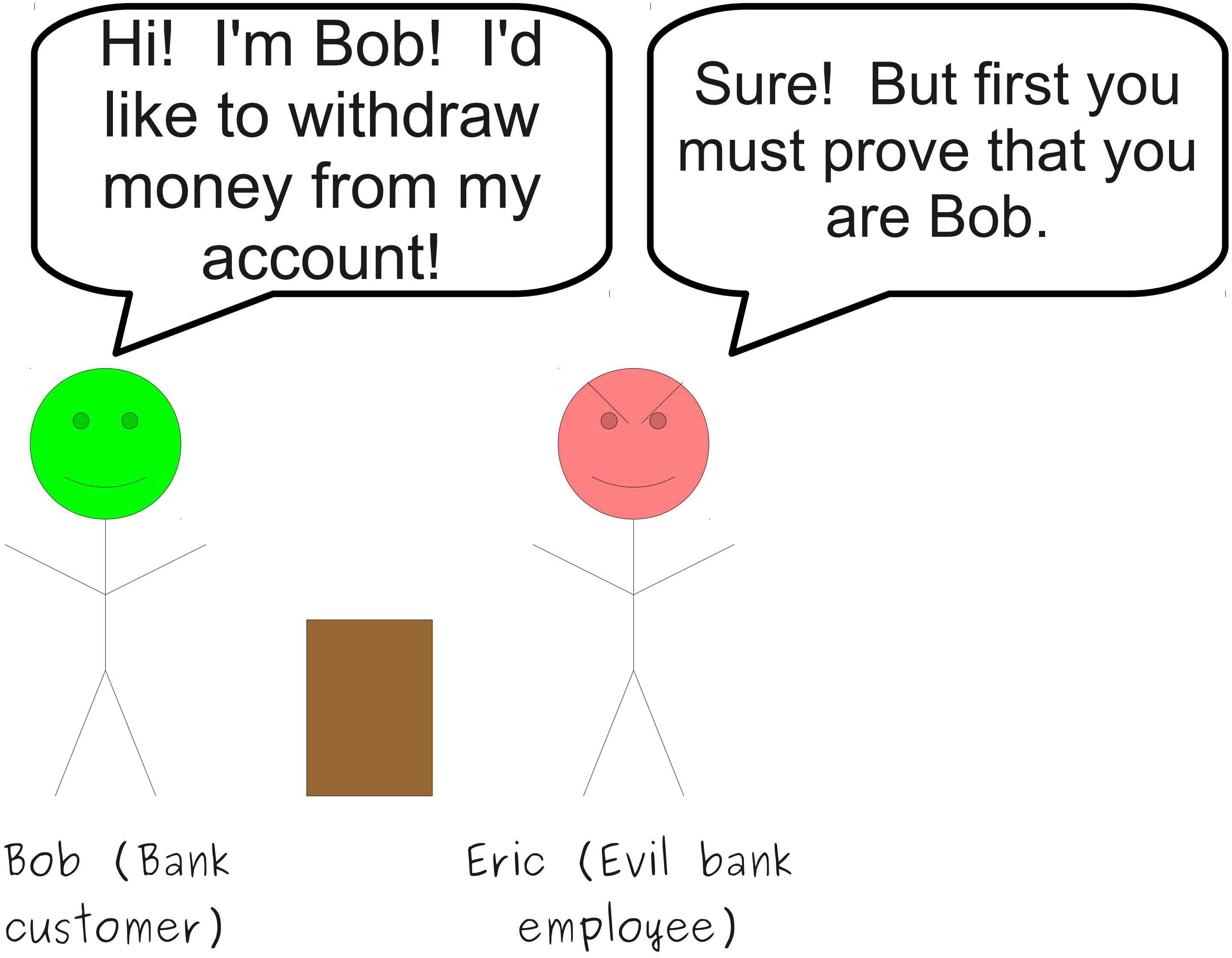
- Recall: An undirected graph *G* is called **3-colorable** iff each of its nodes can be colored one of three colors such that no two nodes of the same color are connected by an edge.
- *3COLOR* is the language of all 3-colorable graphs.
- Recall: *3COLOR* is **NP**-complete.

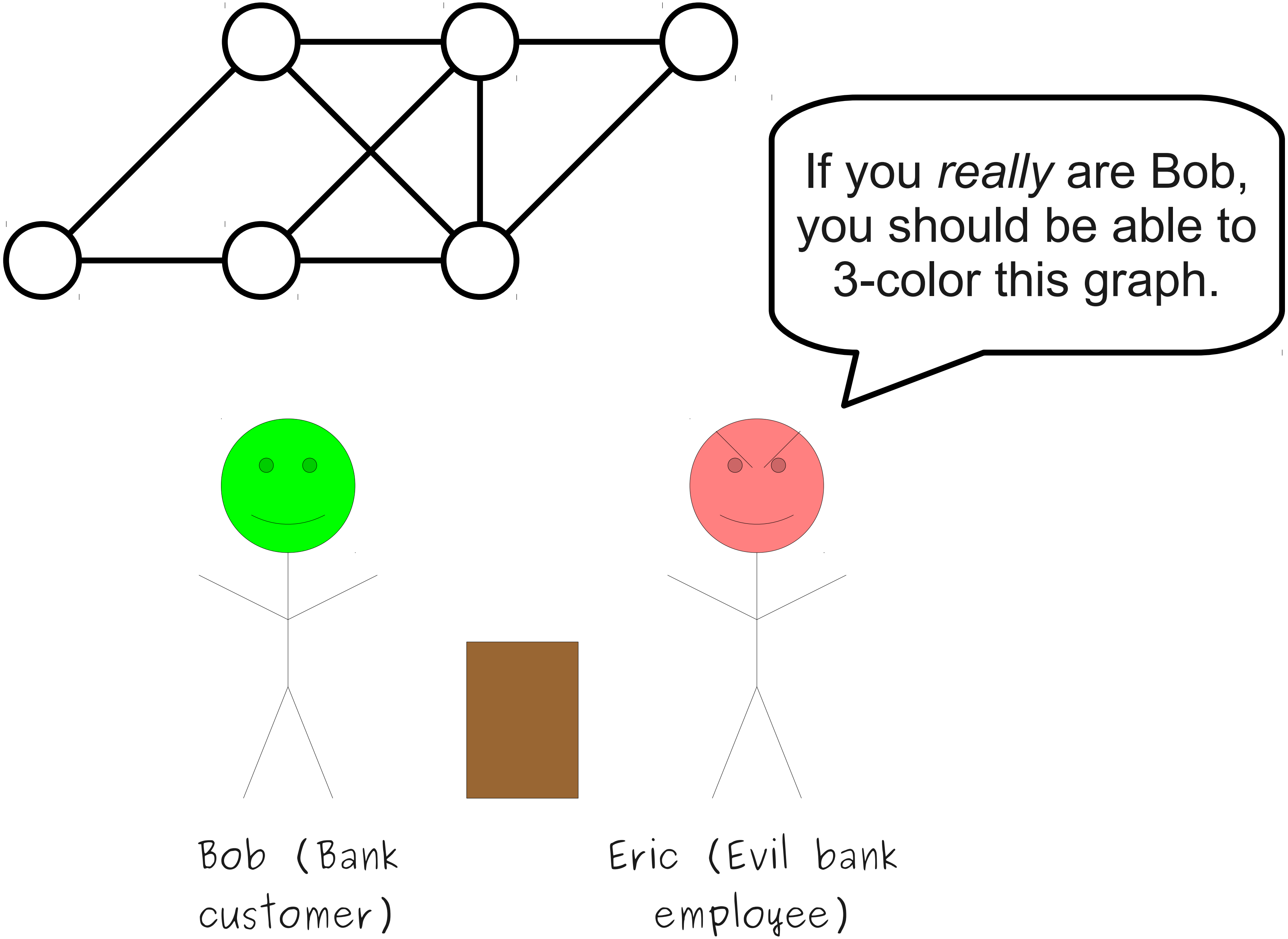
3-Colorability

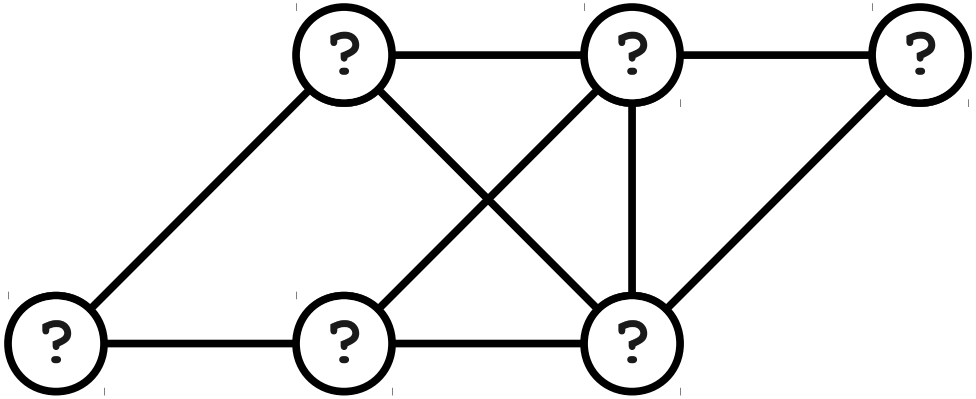
- Since *3COLOR* is **NP**-complete, it is believed that there is no efficient algorithm for finding a 3-coloring for an arbitrary graph.
- However, it is easy to *verify* that a 3-colorable graph is indeed 3-colorable if you already know the coloring.
- Could we build a zero-knowledge proof from this?

Zero-Knowledge and *3COLOR*

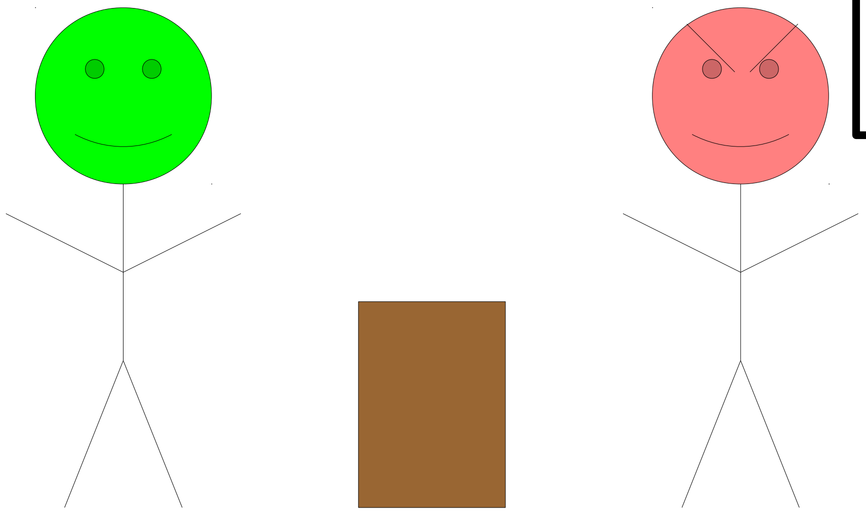
- **Idea:** Assign each person a random graph that is known to be 3-colorable.
- Tell each person how to 3-color that graph.
- To authenticate, the prover convinces the verifier that they can 3-color the graph, but does so without revealing the coloring.
  - Details in a minute.

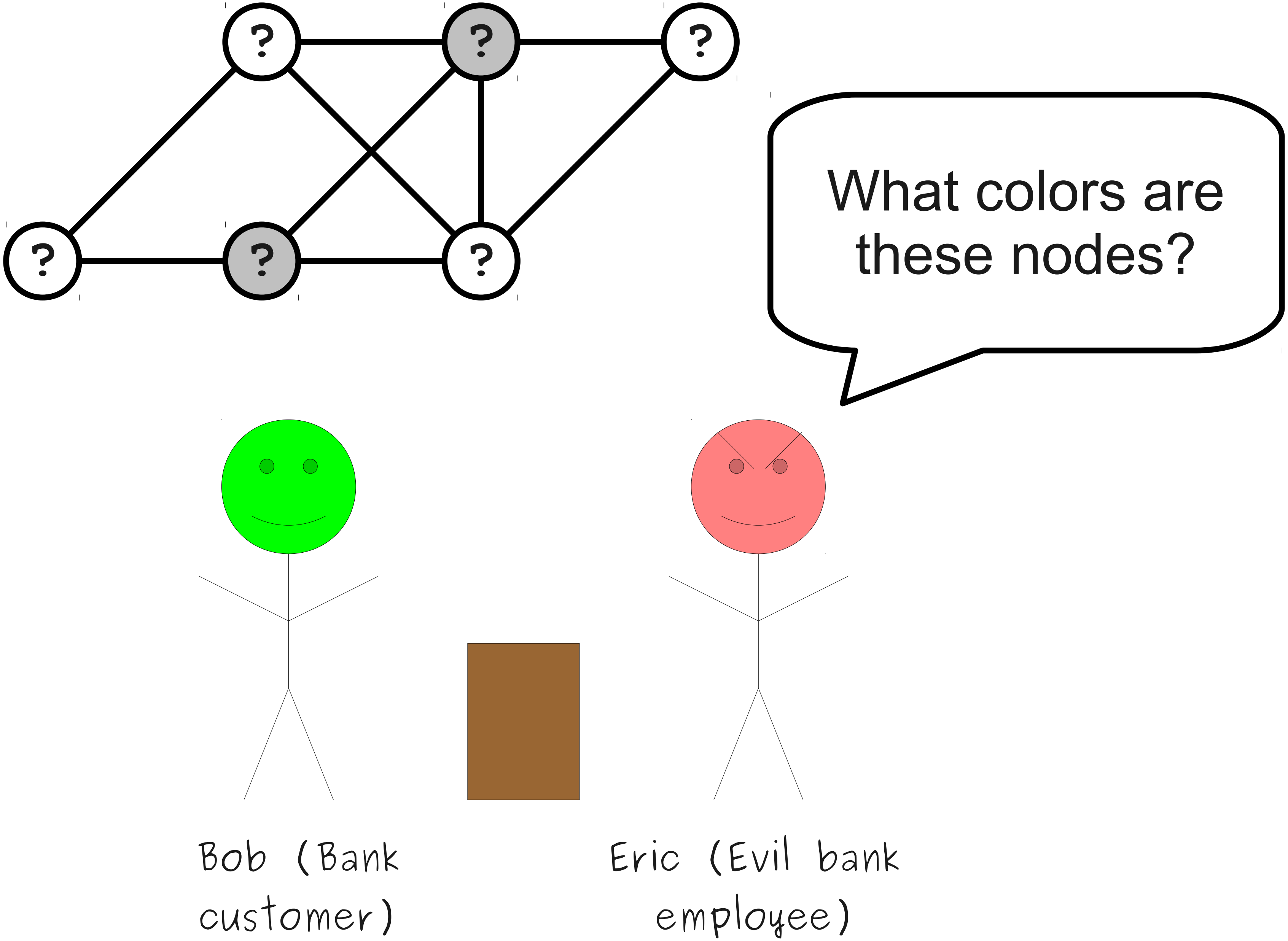

Hi! I'm Bob! I'd like to withdraw money from my account!
Sure! But first you must prove that you are Bob.
Bob (Bank customer)
Eric (Evil bank employee)

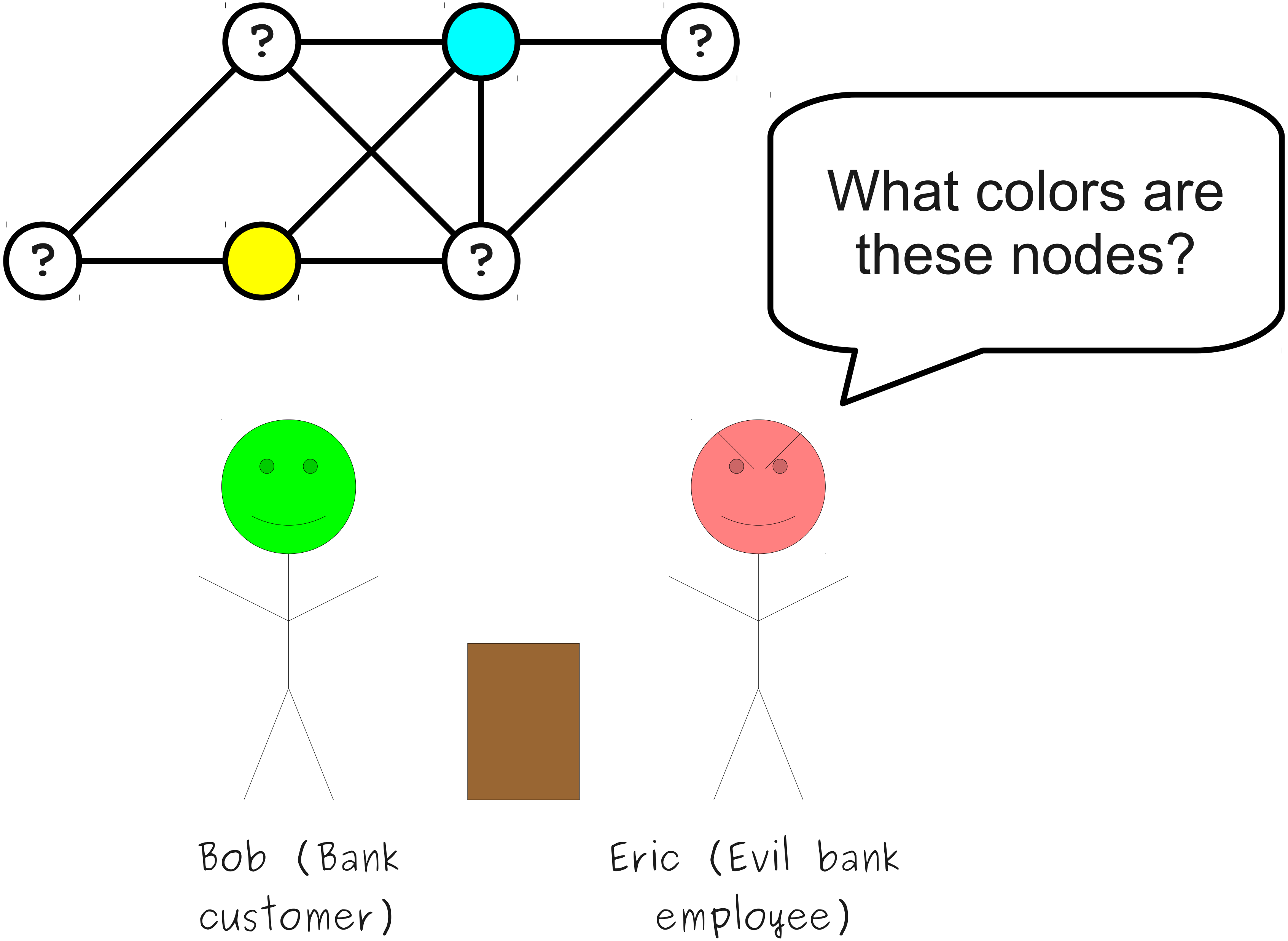

If you *really* are Bob, you should be able to 3-color this graph.
Bob (Bank customer)
Eric (Evil bank employee)

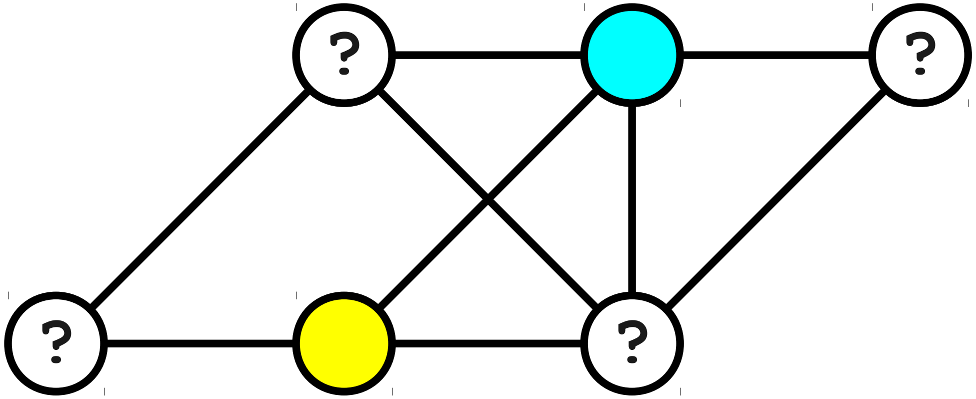


Key insight one: Bob can commit to the colors of the nodes without actually revealing those colors.



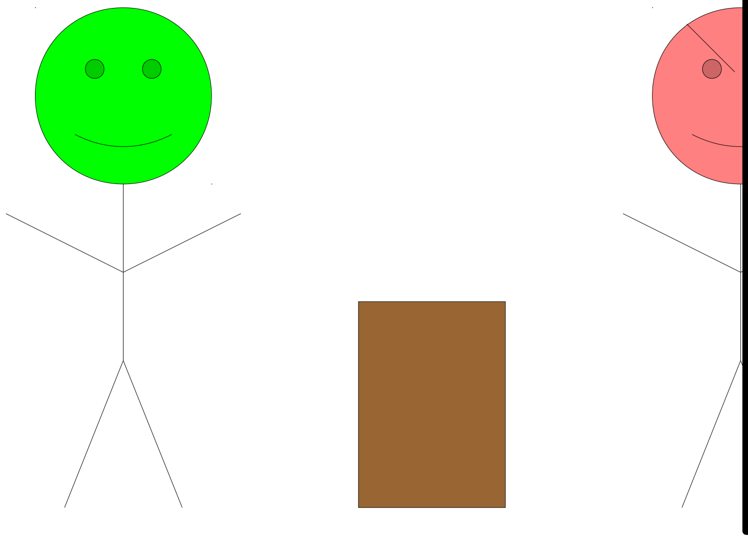

?
?
?
?
?
?
What colors are these nodes?
Bob (Bank customer)
Eric (Evil bank employee)


?
?
?
?
What colors are these nodes?
Bob (Bank customer)
Eric (Evil bank employee)



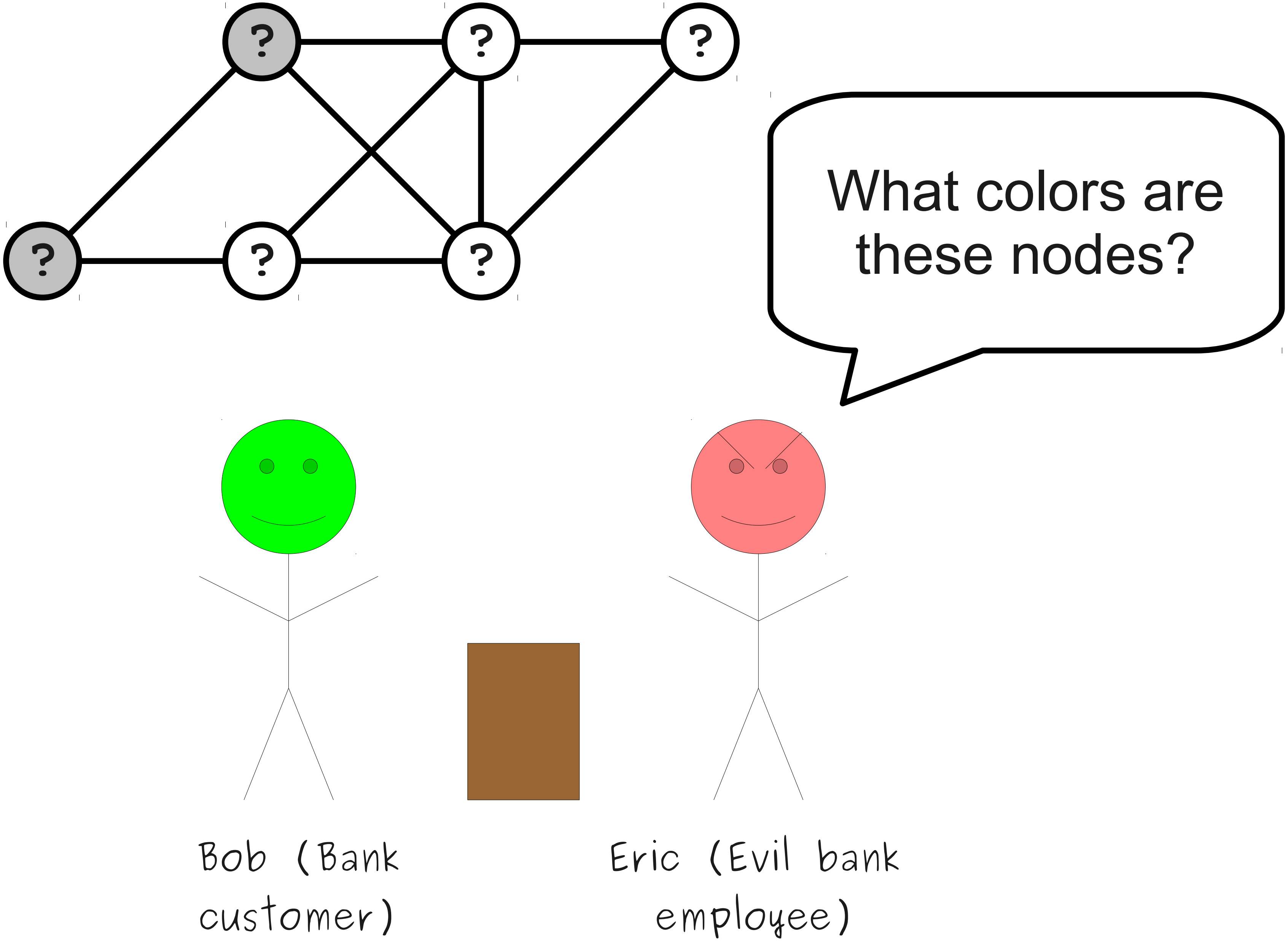
If Bob doesn't know how to 3-color the graph, there's a chance that these nodes will be the same color. In that case, Eric knows Bob is lying.

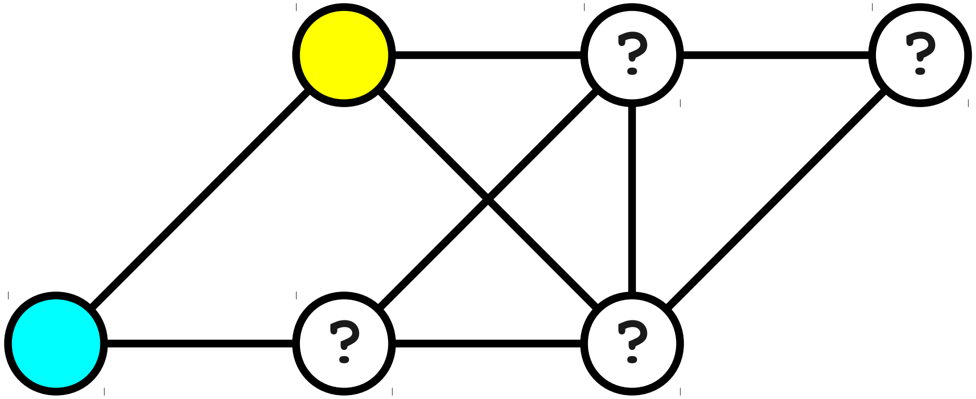
If Eric repeats this a few times and the colors are always different, Eric can be confident that Bob really knows the coloring.

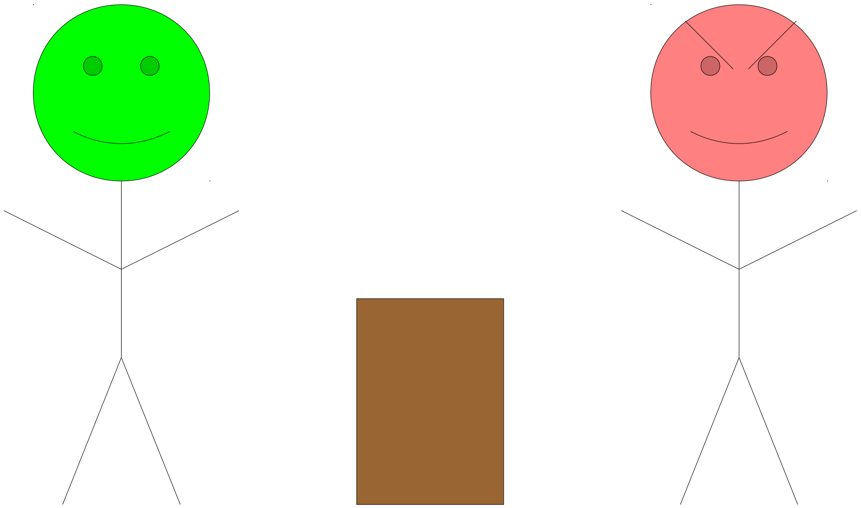


Bob (Bank customer)

Eric (Evil bank employee)

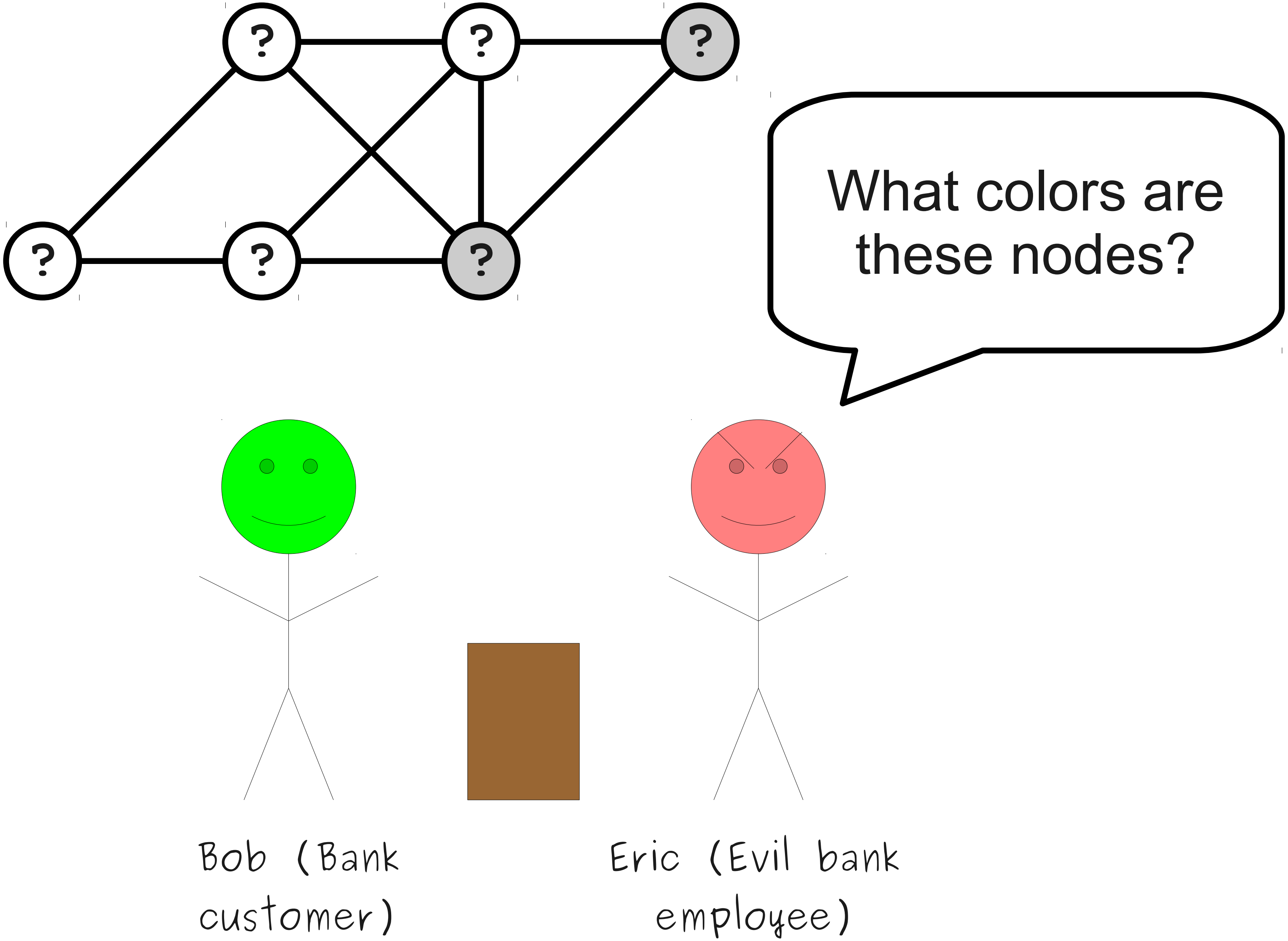

?
?
?
?
?
?
What colors are these nodes?
Bob (Bank customer)
Eric (Evil bank employee)

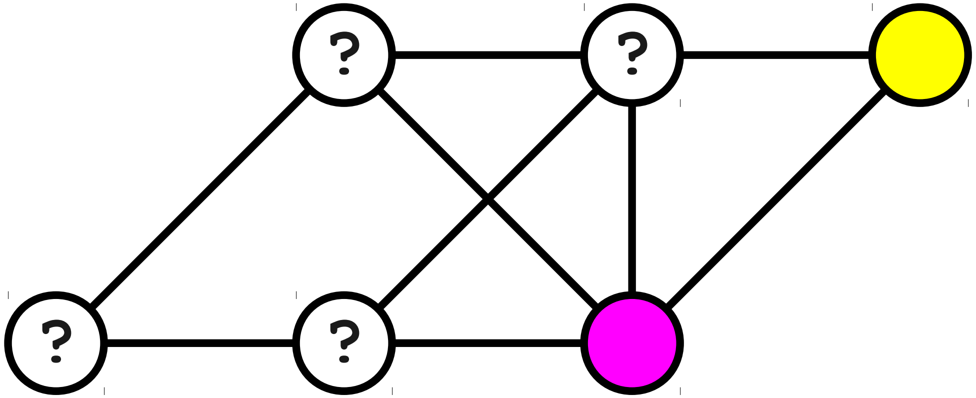


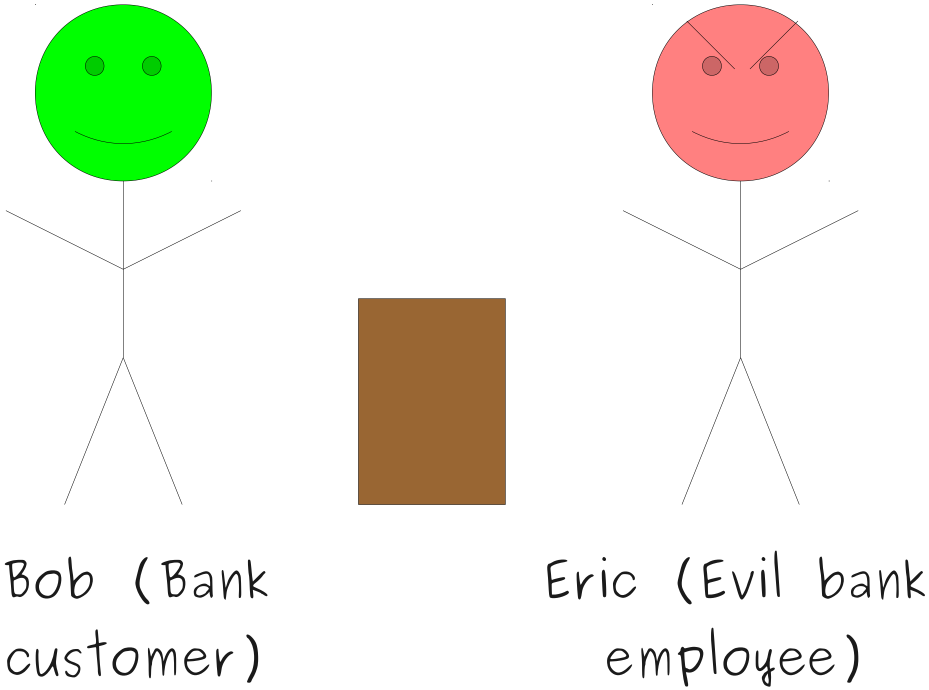


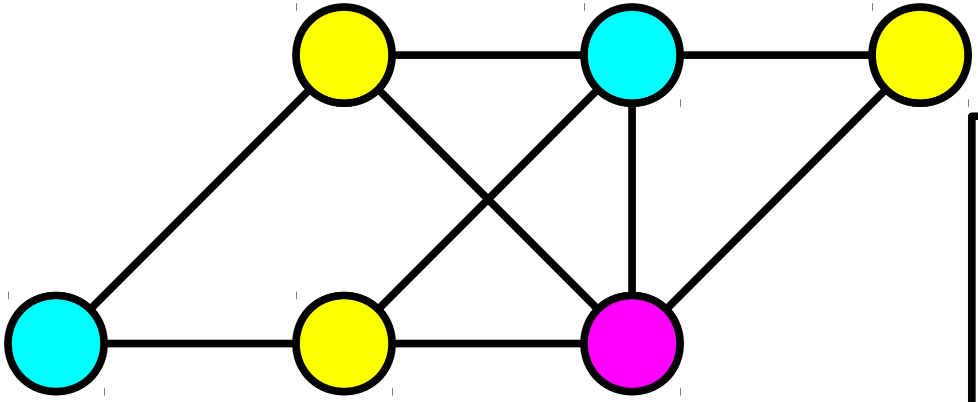
Bob (Bank customer)

Eric (Evil bank employee)


?
?
?
?
?
?
What colors are these nodes?
Bob (Bank customer)
Eric (Evil bank employee)

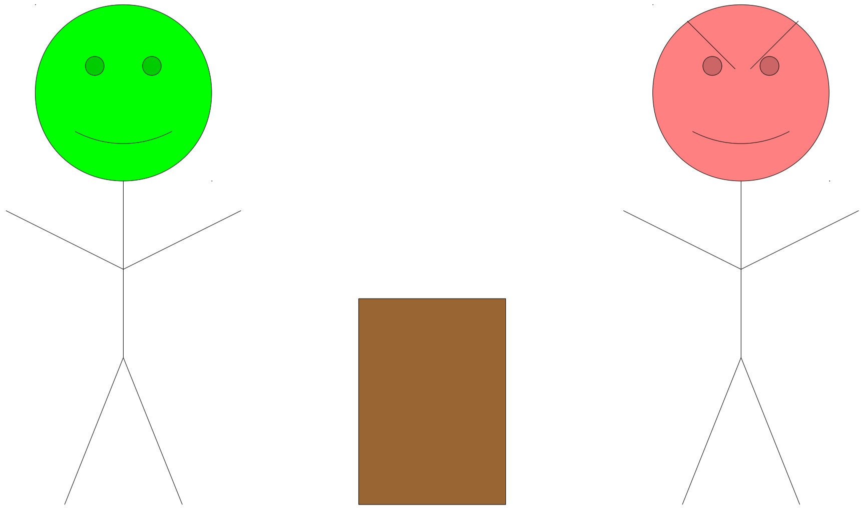

?
?
?
?
?


Bob (Bank customer)
Eric (Evil bank employee)

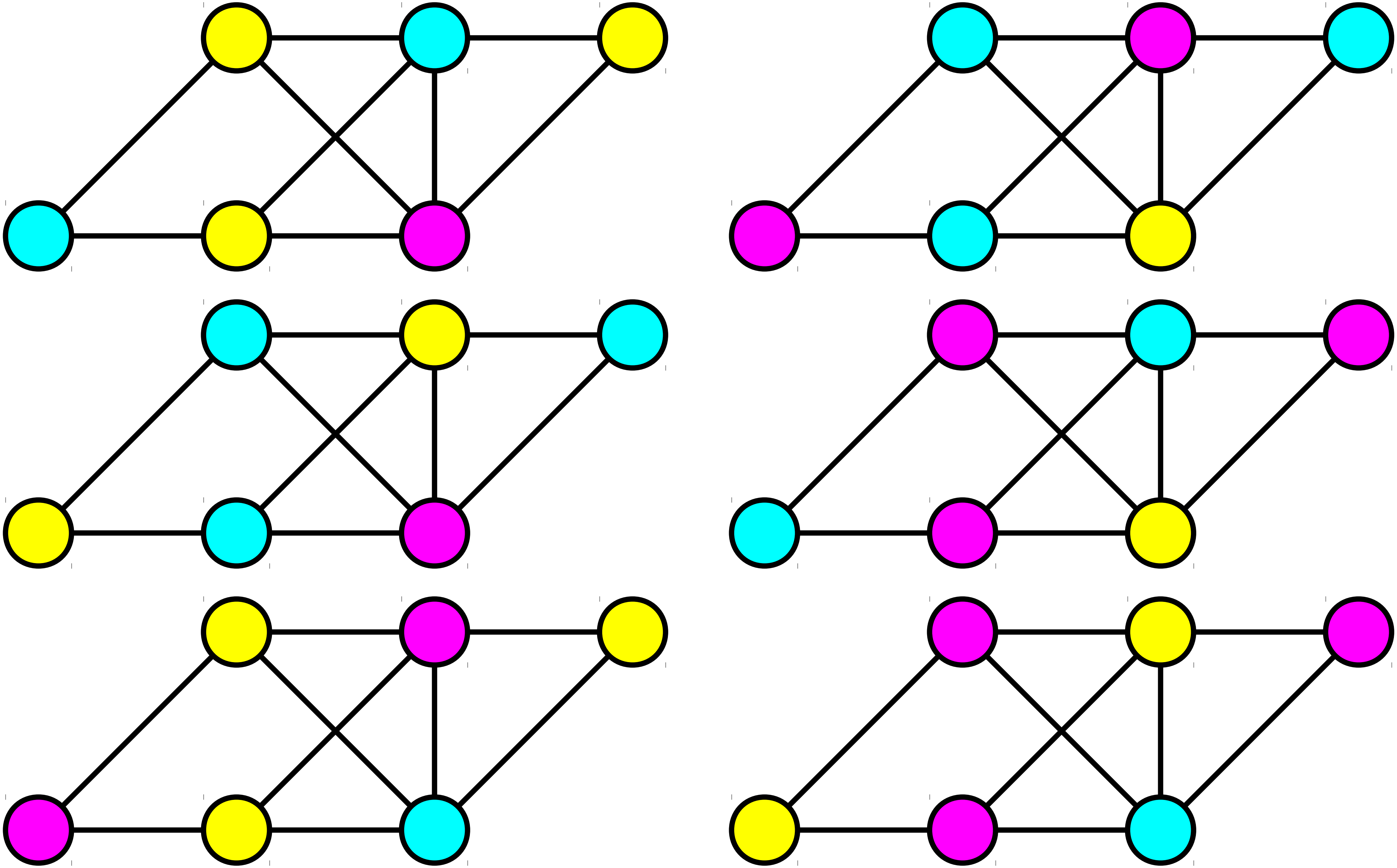


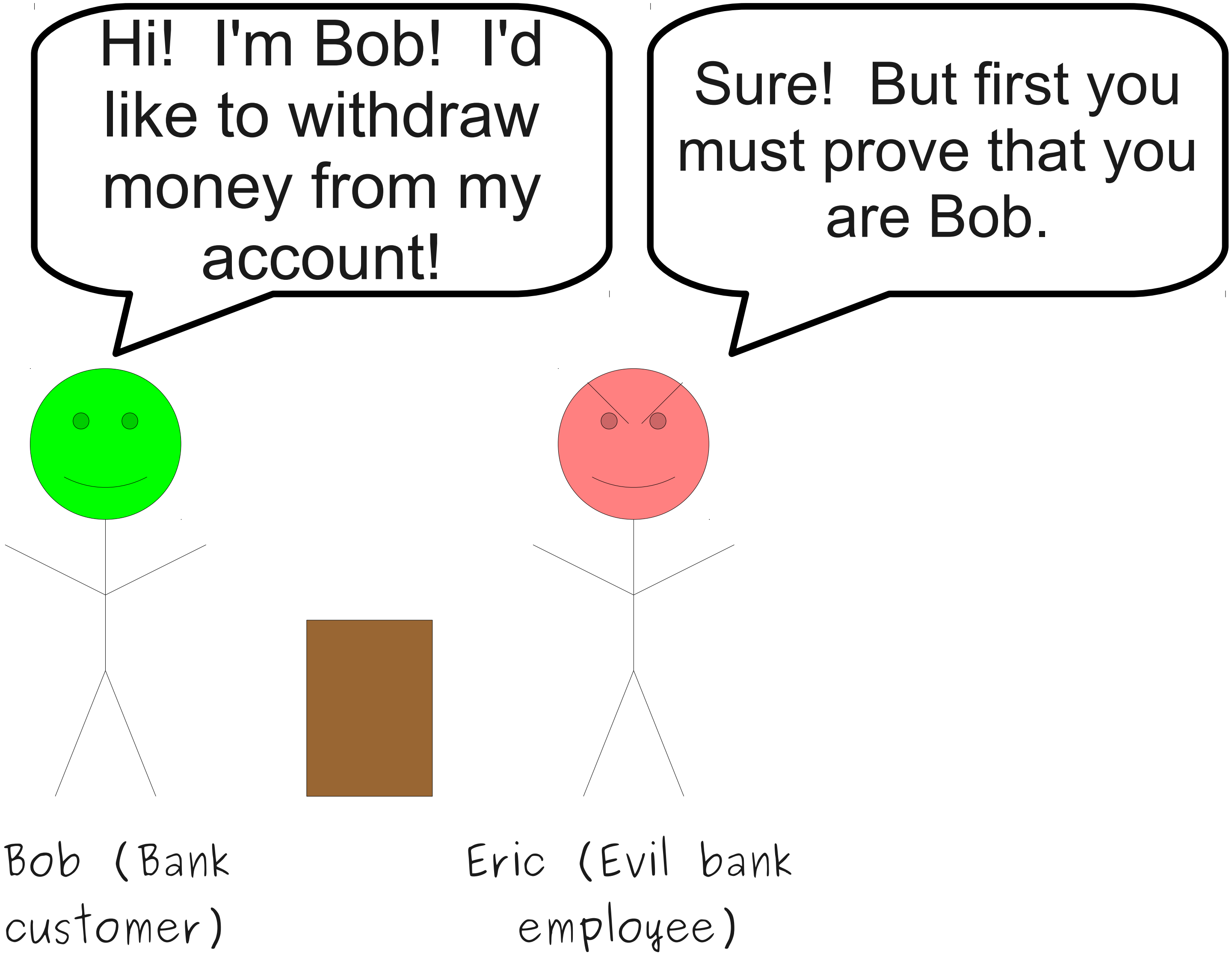
Uh oh! Eric now knows the complete coloring!

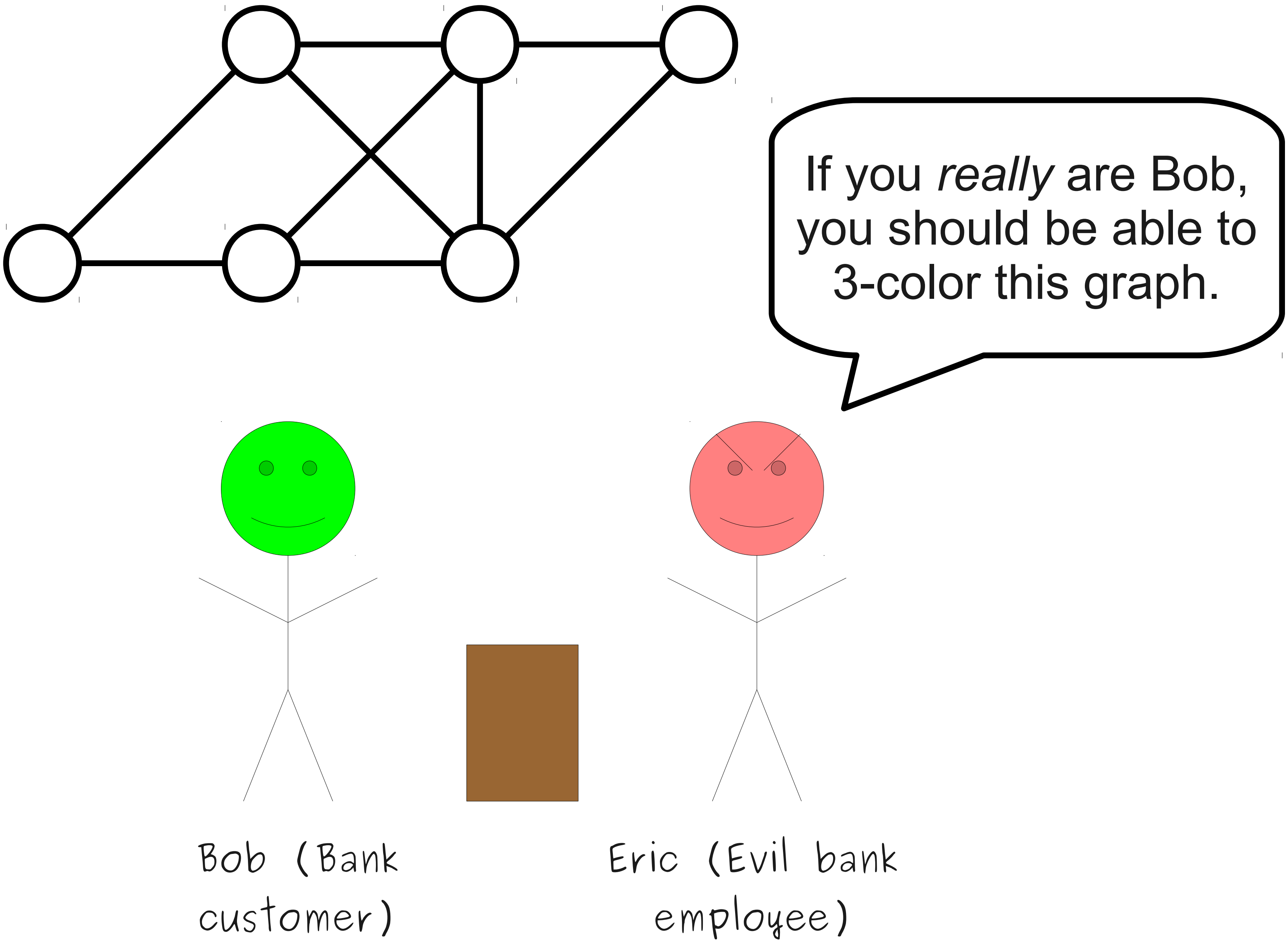
How can we address this?

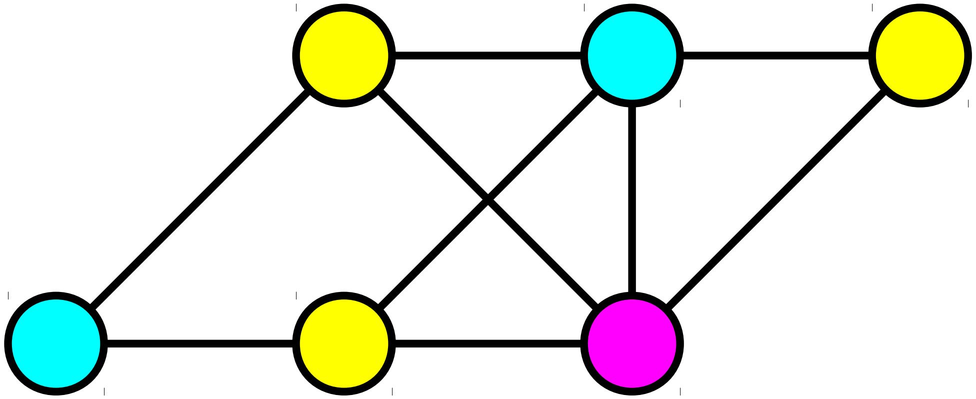


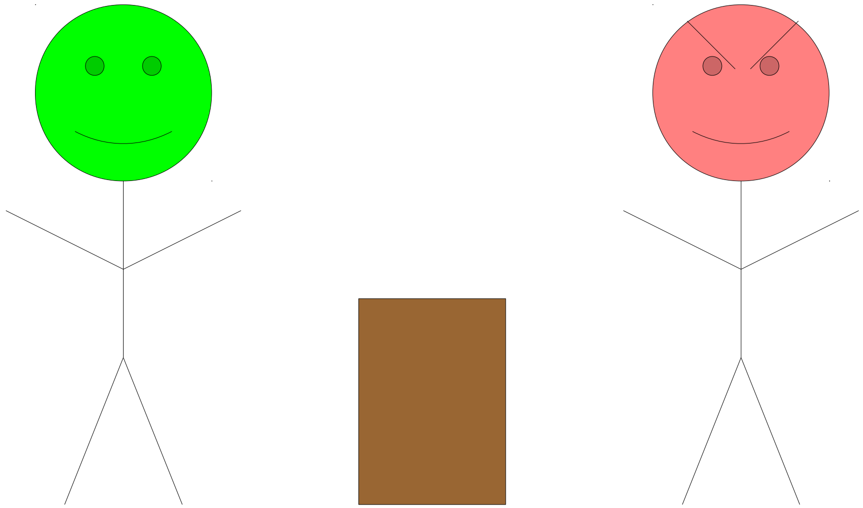
Legal 3-Colorings




Hi! I'm Bob! I'd like to withdraw money from my account!
Sure! But first you must prove that you are Bob.
Bob (Bank customer)
Eric (Evil bank employee)

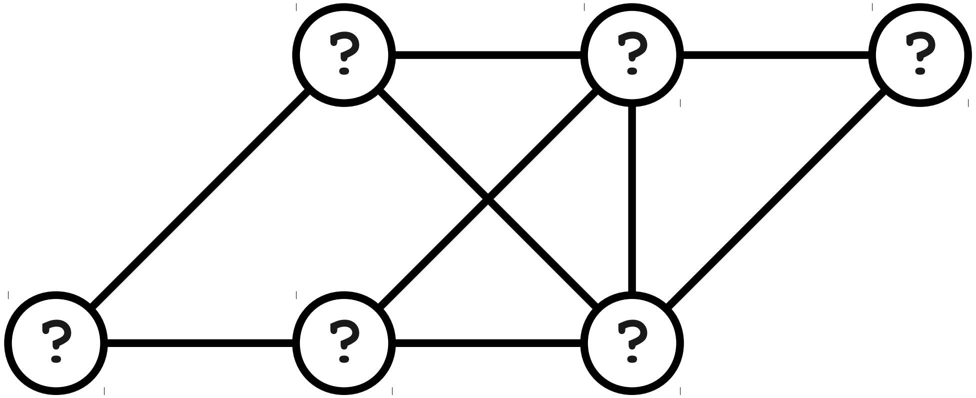

If you *really* are Bob, you should be able to 3-color this graph.
Bob (Bank customer)
Eric (Evil bank employee)

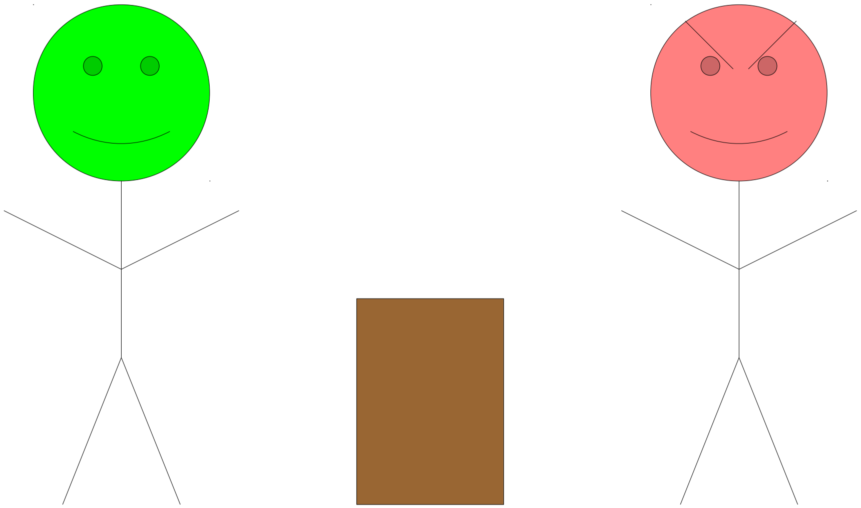




Bob (Bank customer)

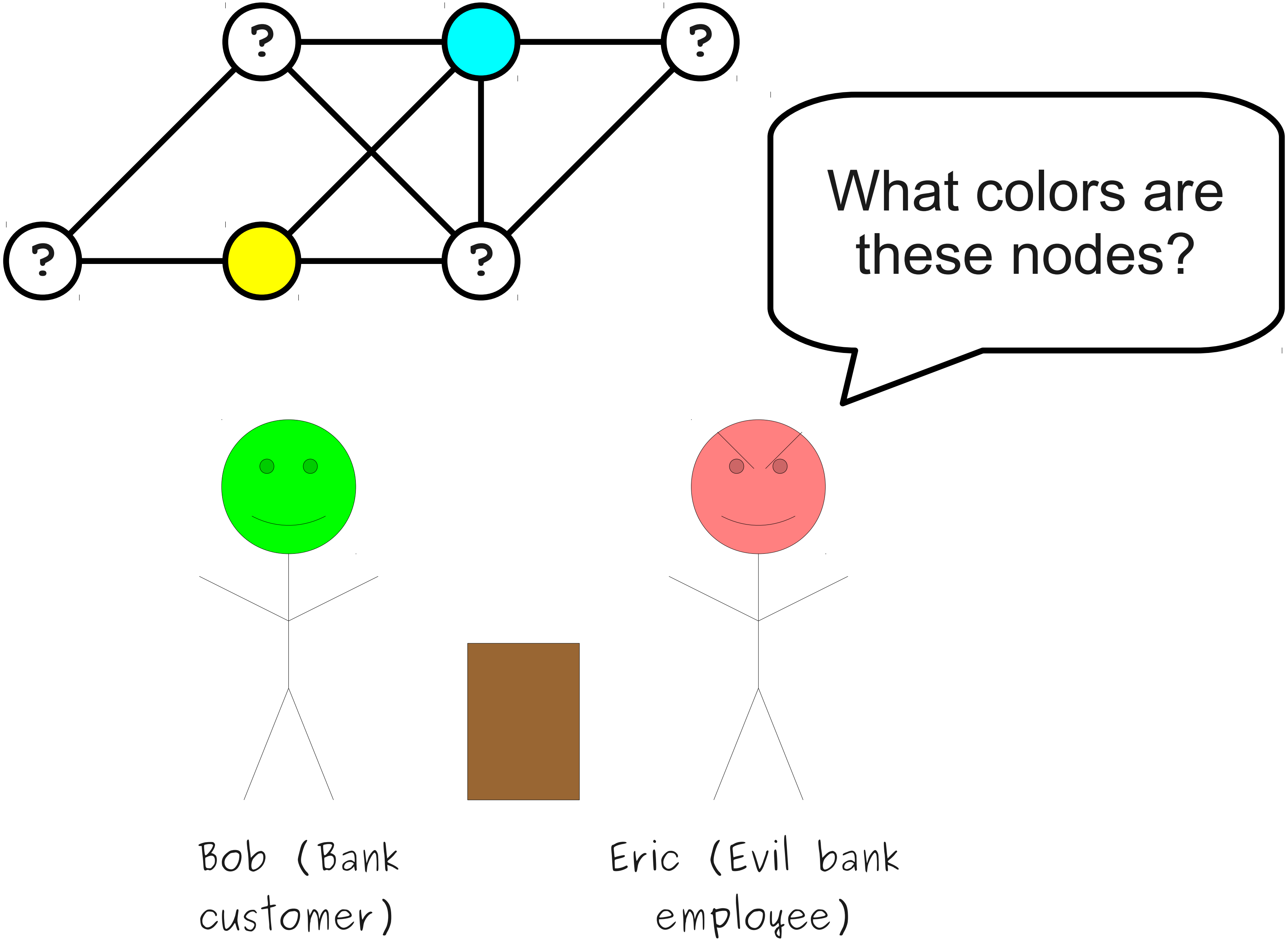
Eric (Evil bank employee)

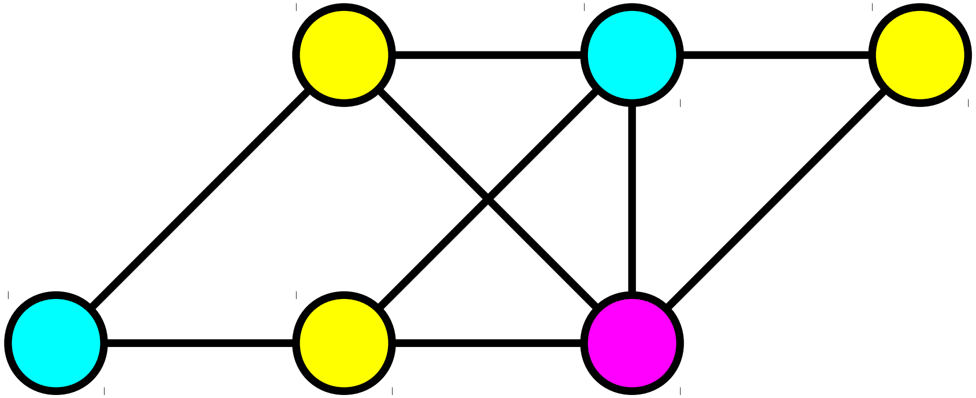


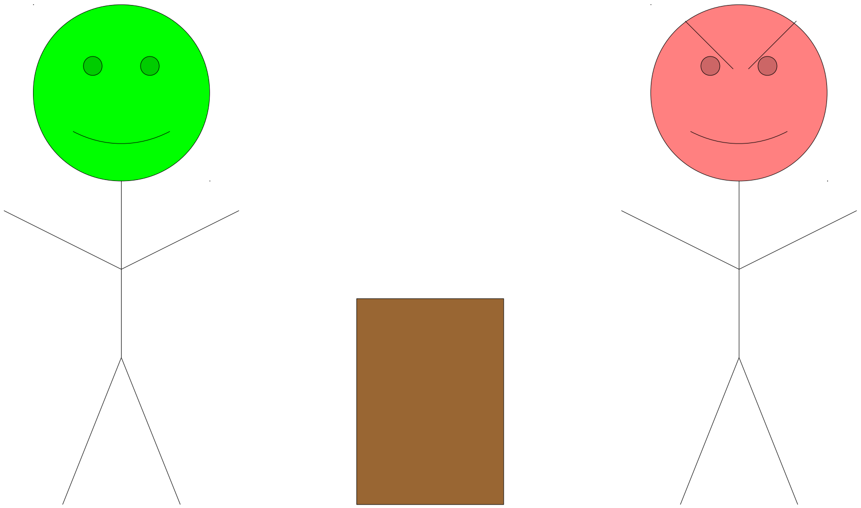


Bob (Bank
customer)

Eric (Evil bank
employee)

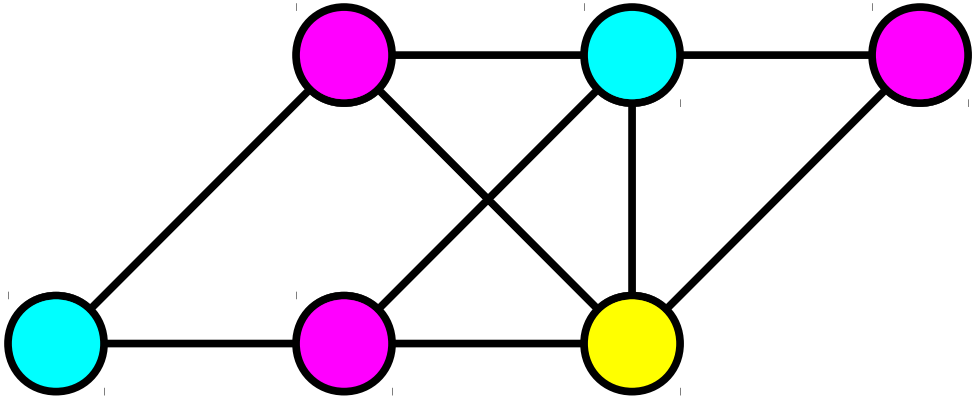

?
?
?
?
What colors are these nodes?
Bob (Bank customer)
Eric (Evil bank employee)

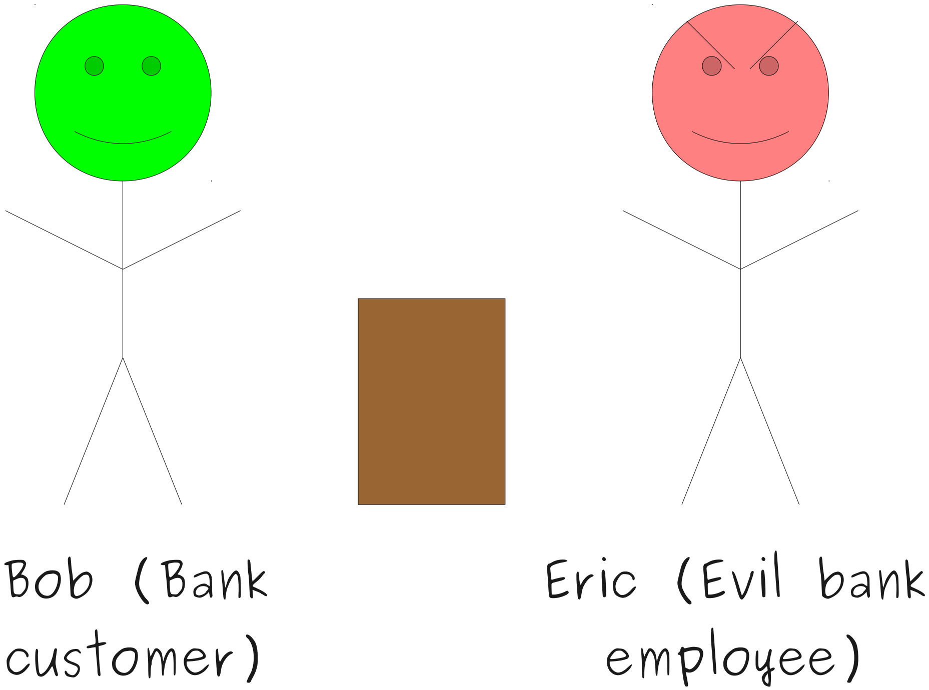


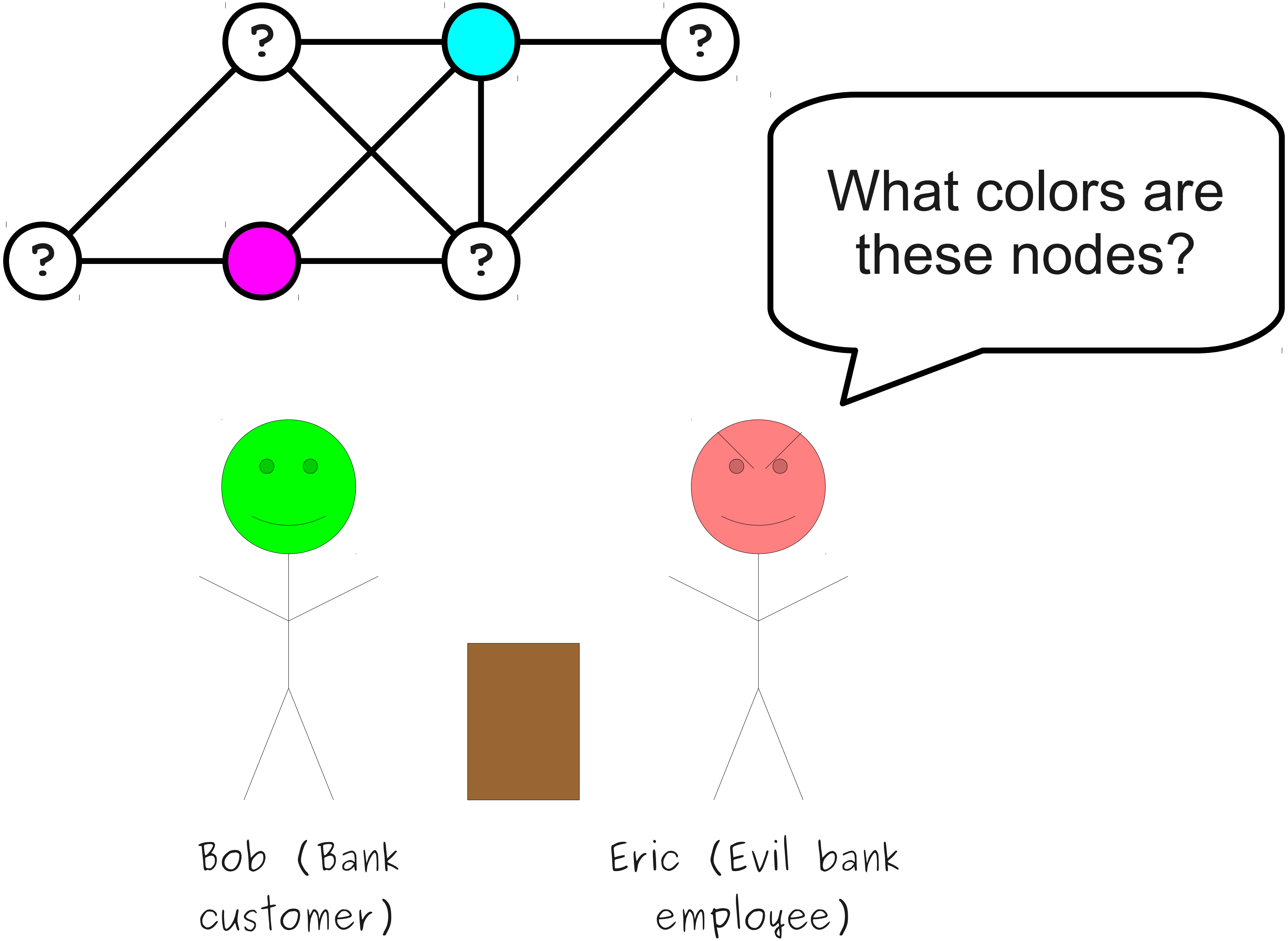


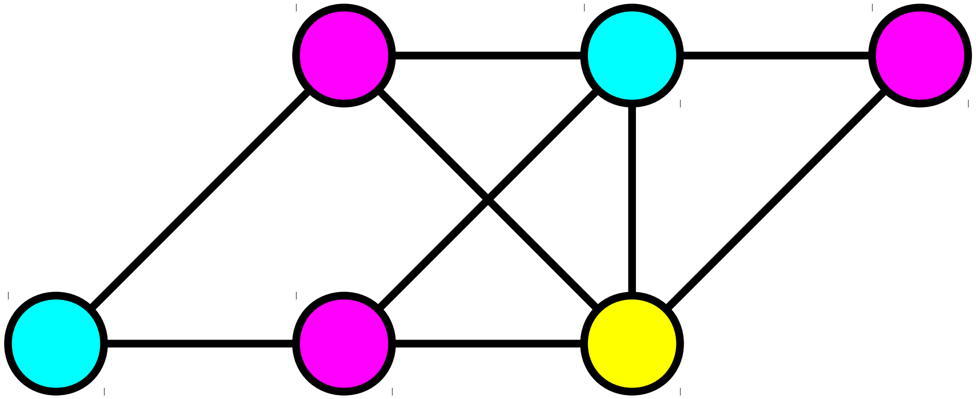
Bob (Bank customer)

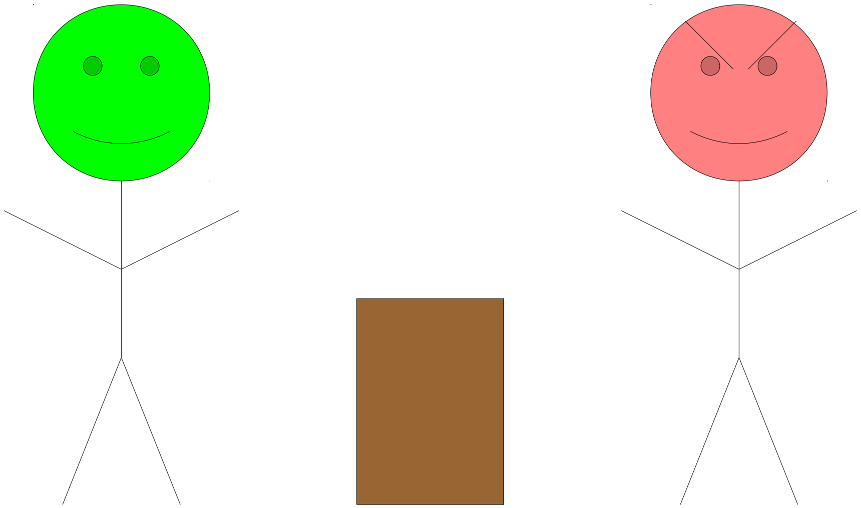
Eric (Evil bank employee)




Bob (Bank customer)
Eric (Evil bank employee)

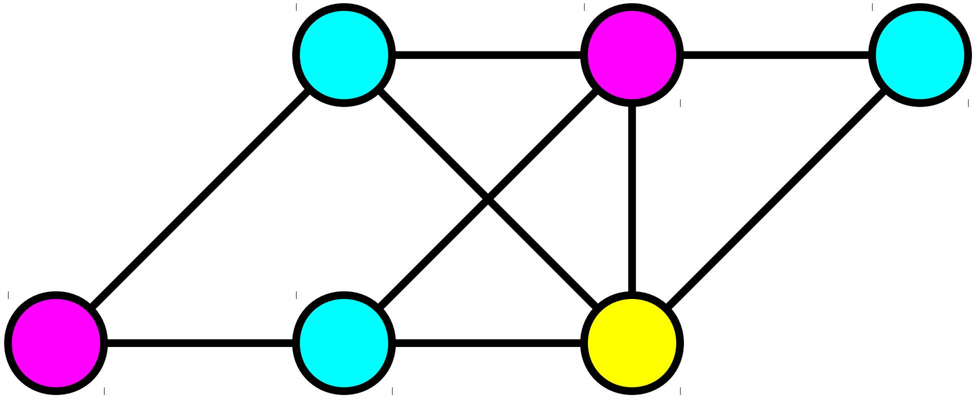

?
?
?
?
What colors are these nodes?
Bob (Bank customer)
Eric (Evil bank employee)

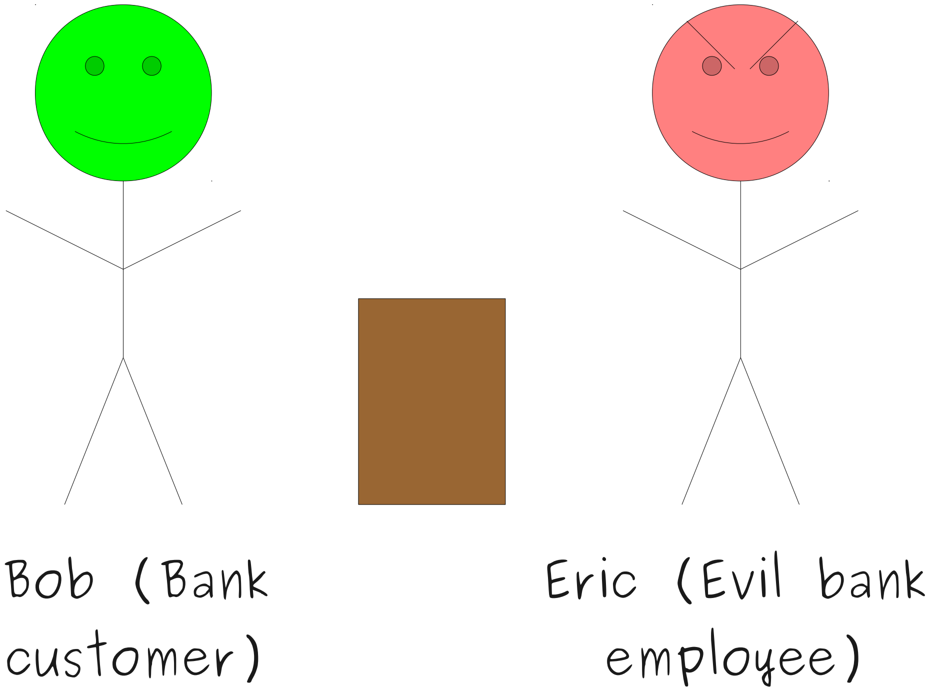




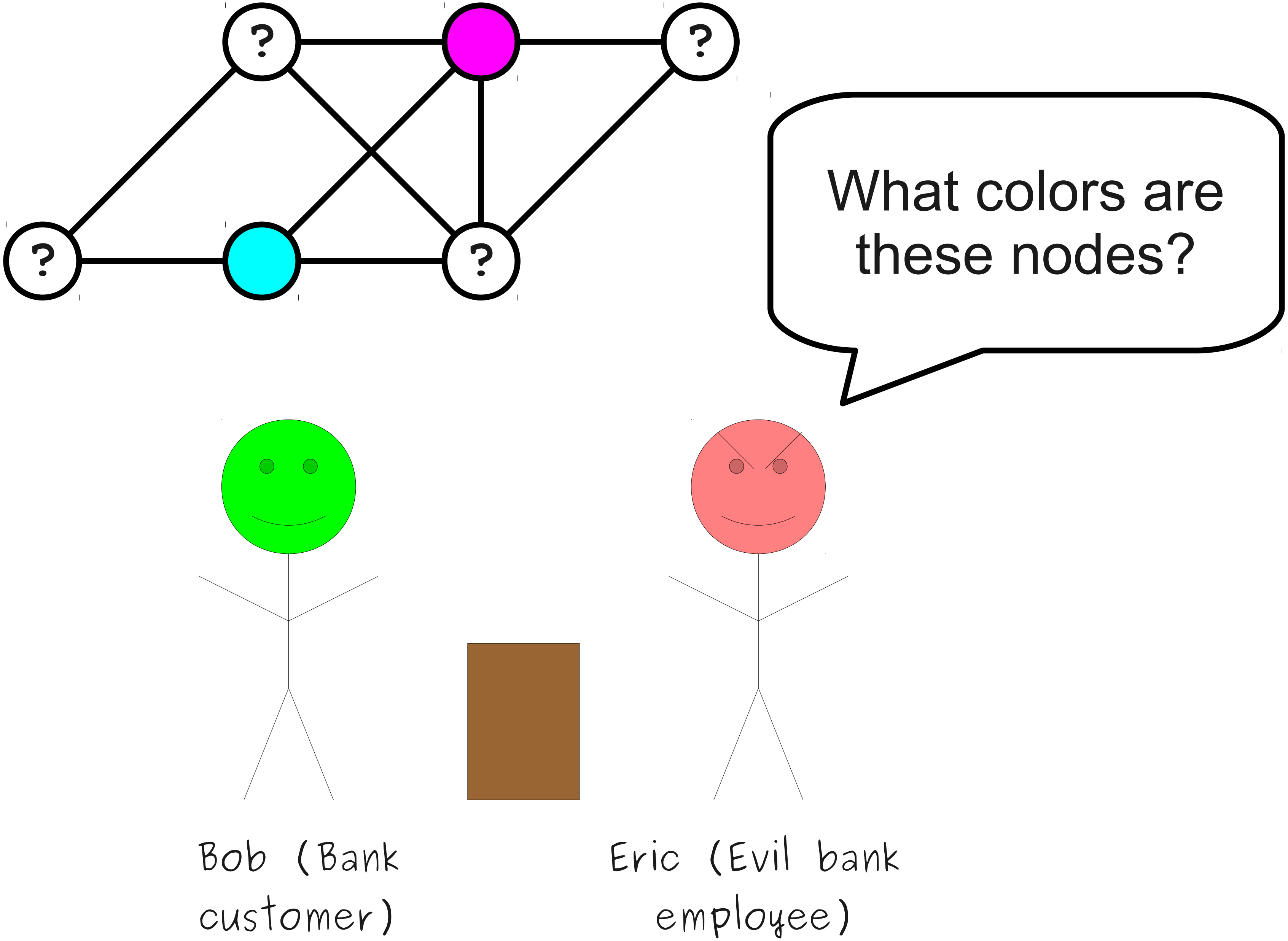
Bob (Bank customer)

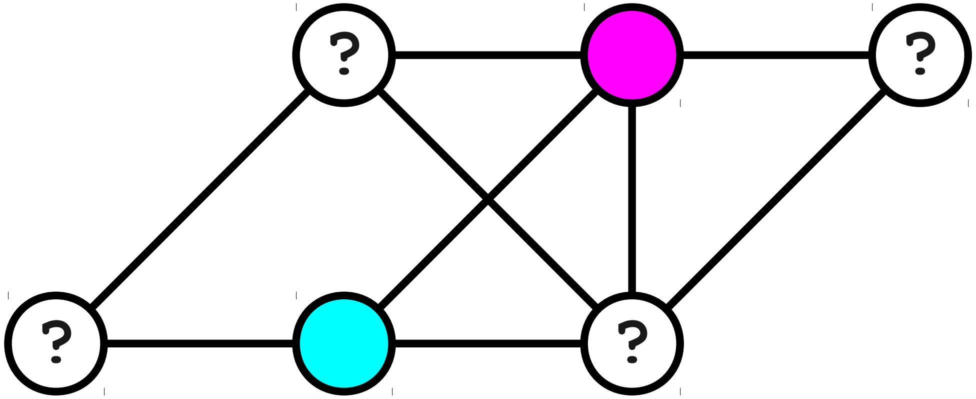
Eric (Evil bank employee)

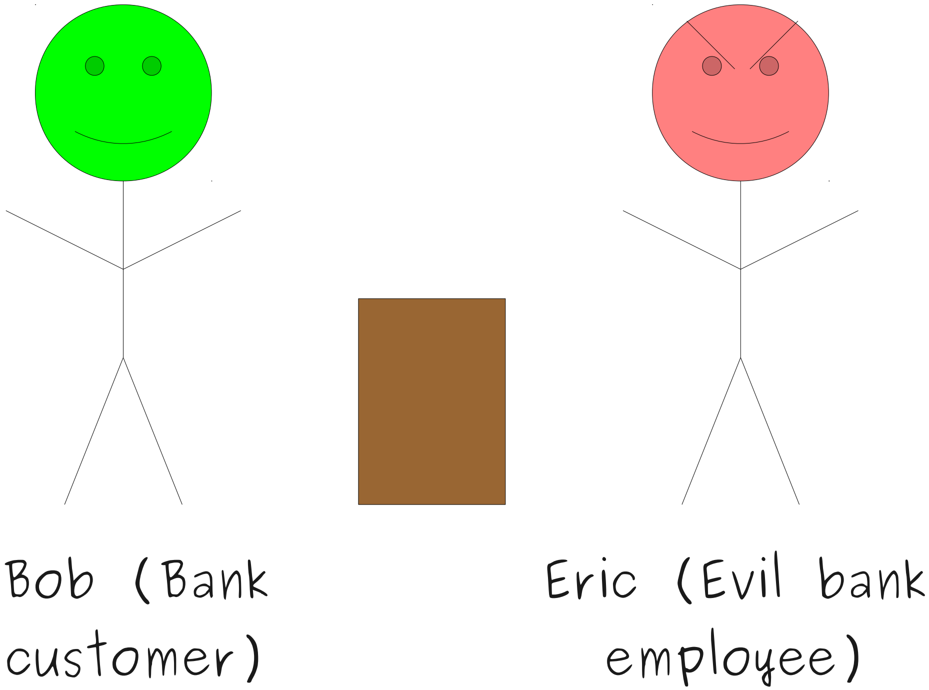


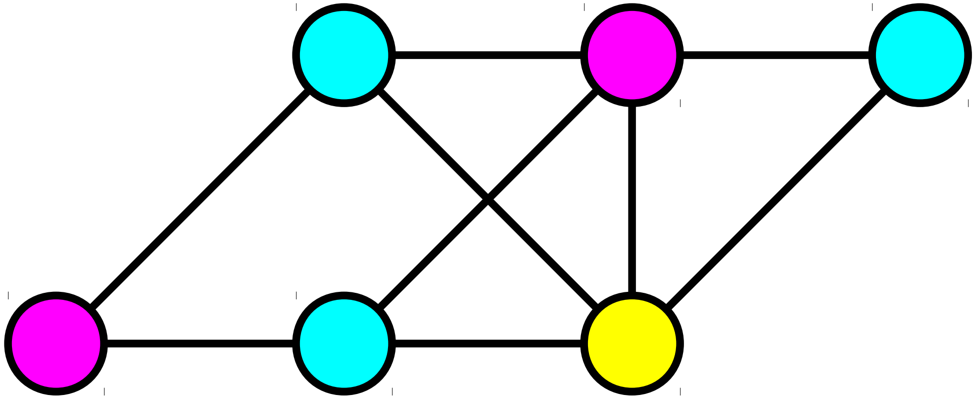


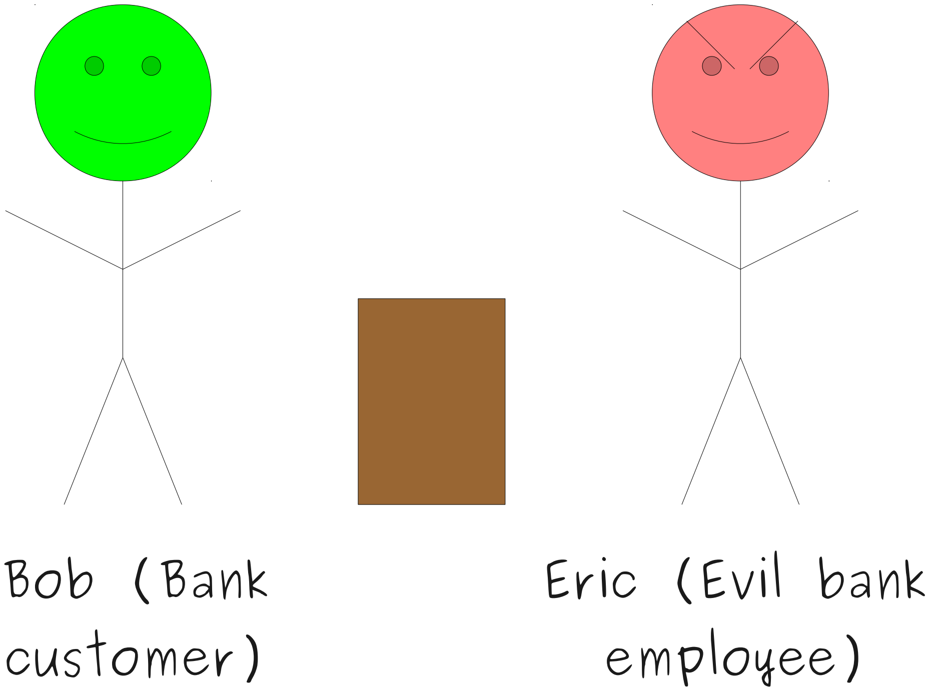
Bob (Bank customer)
Eric (Evil bank employee)

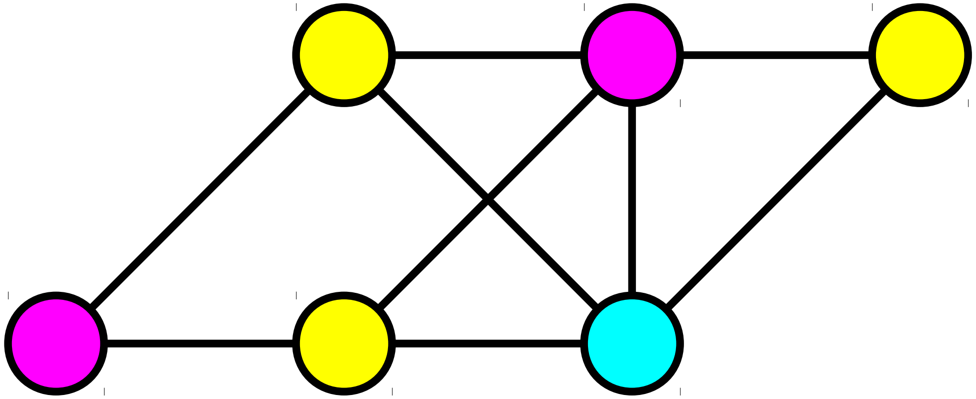

?
?
?
?
What colors are these nodes?
Bob (Bank customer)
Eric (Evil bank employee)

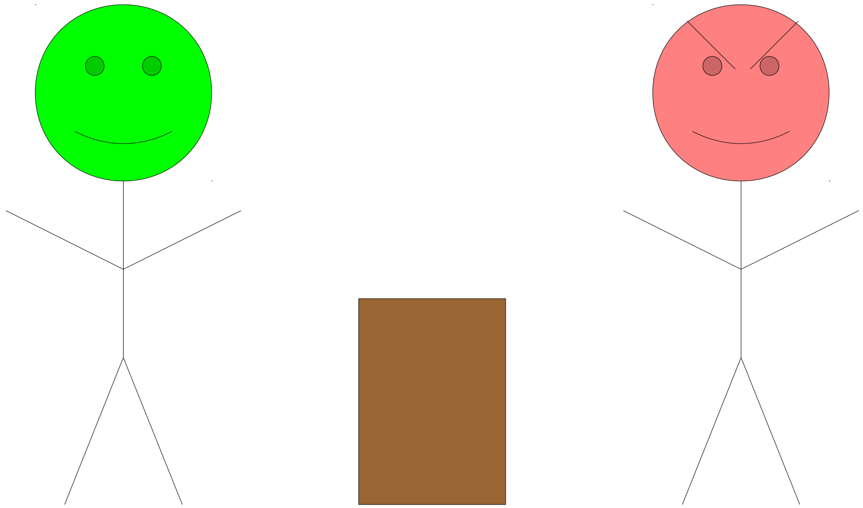

?
?
?
?


Bob (Bank customer)
Eric (Evil bank employee)



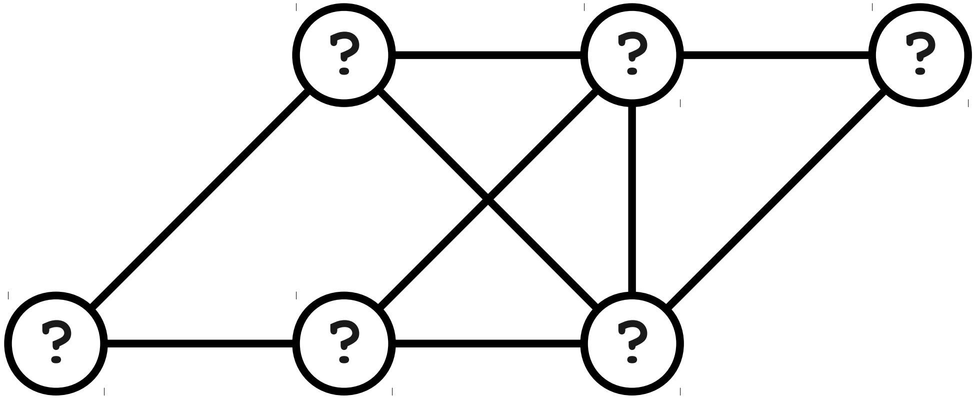

Bob (Bank customer)
Eric (Evil bank employee)

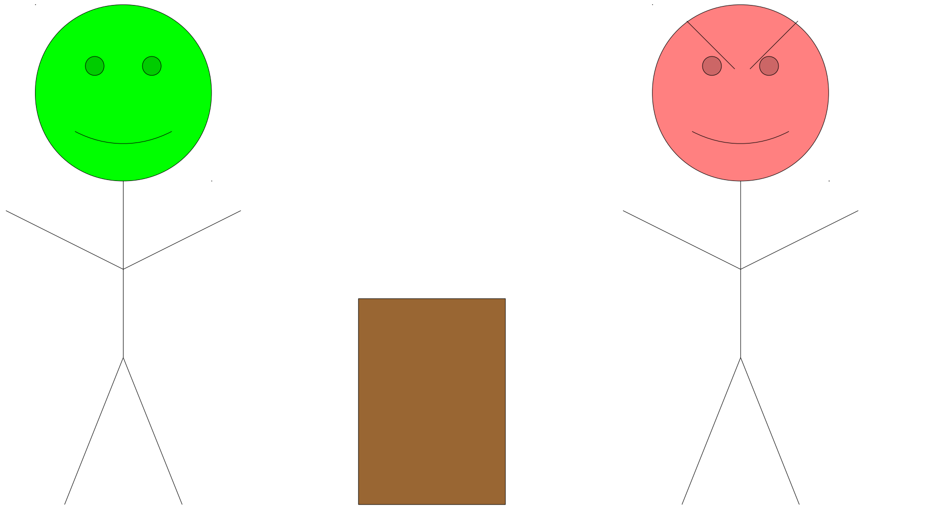


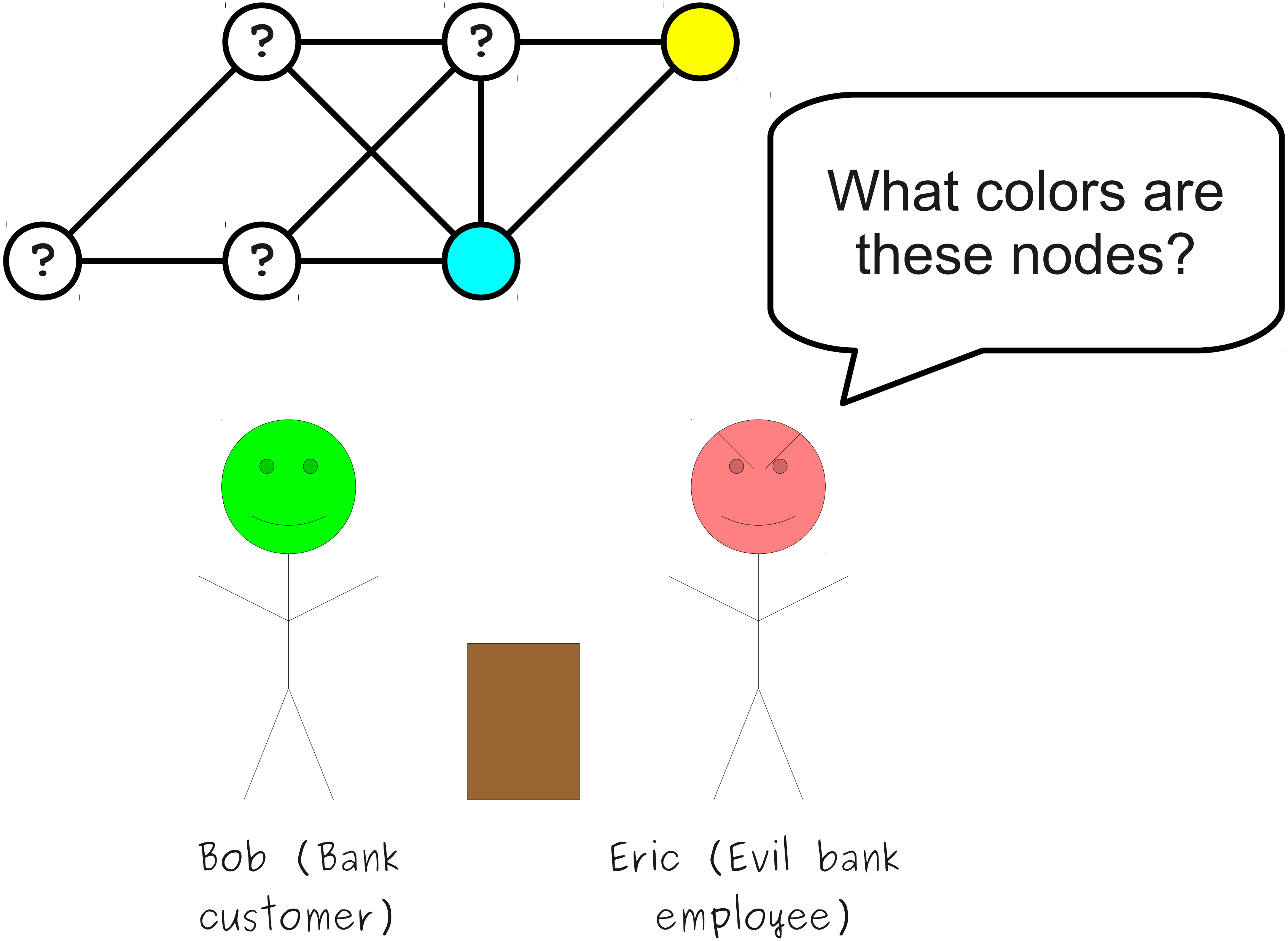


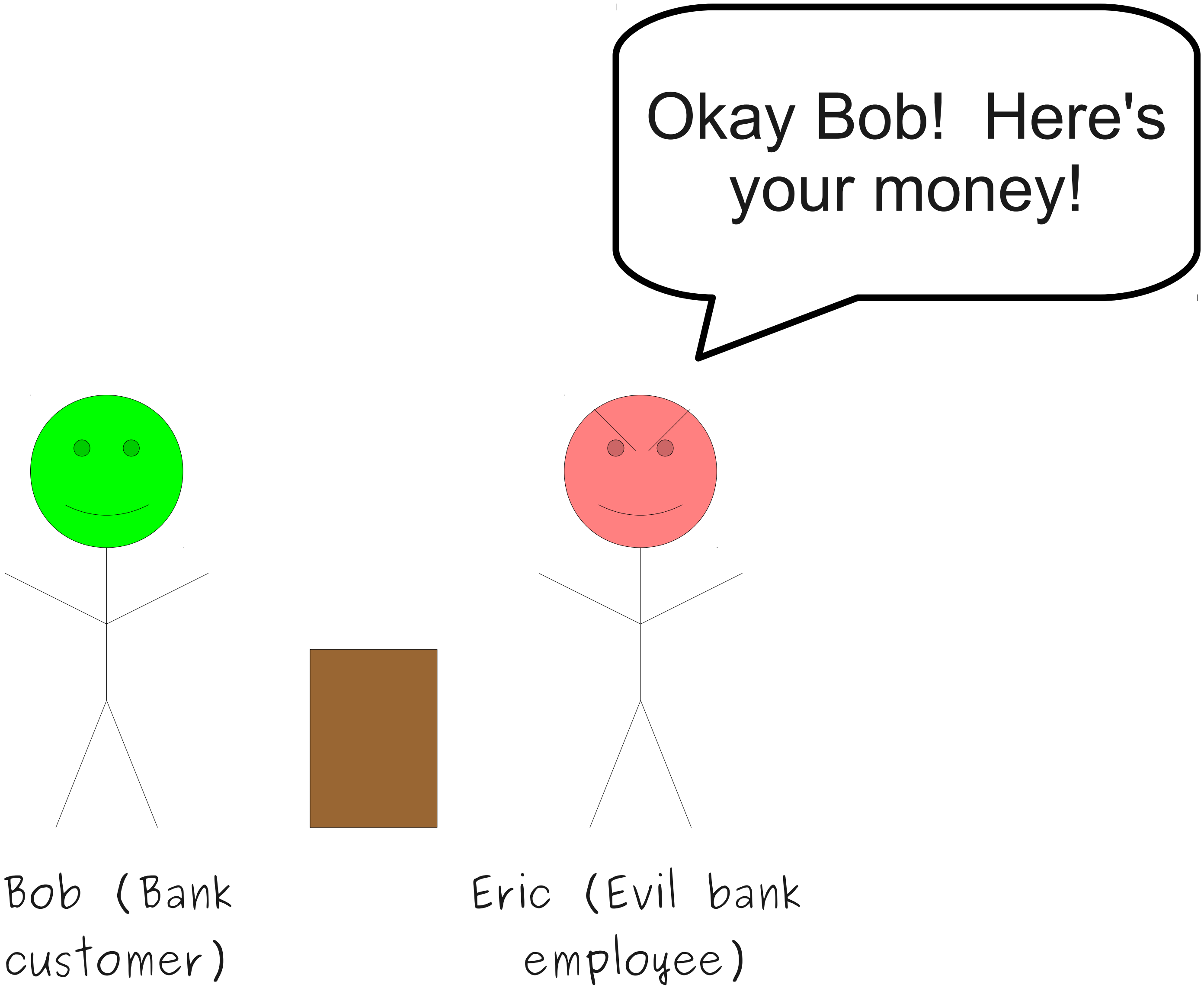
Bob (Bank customer)

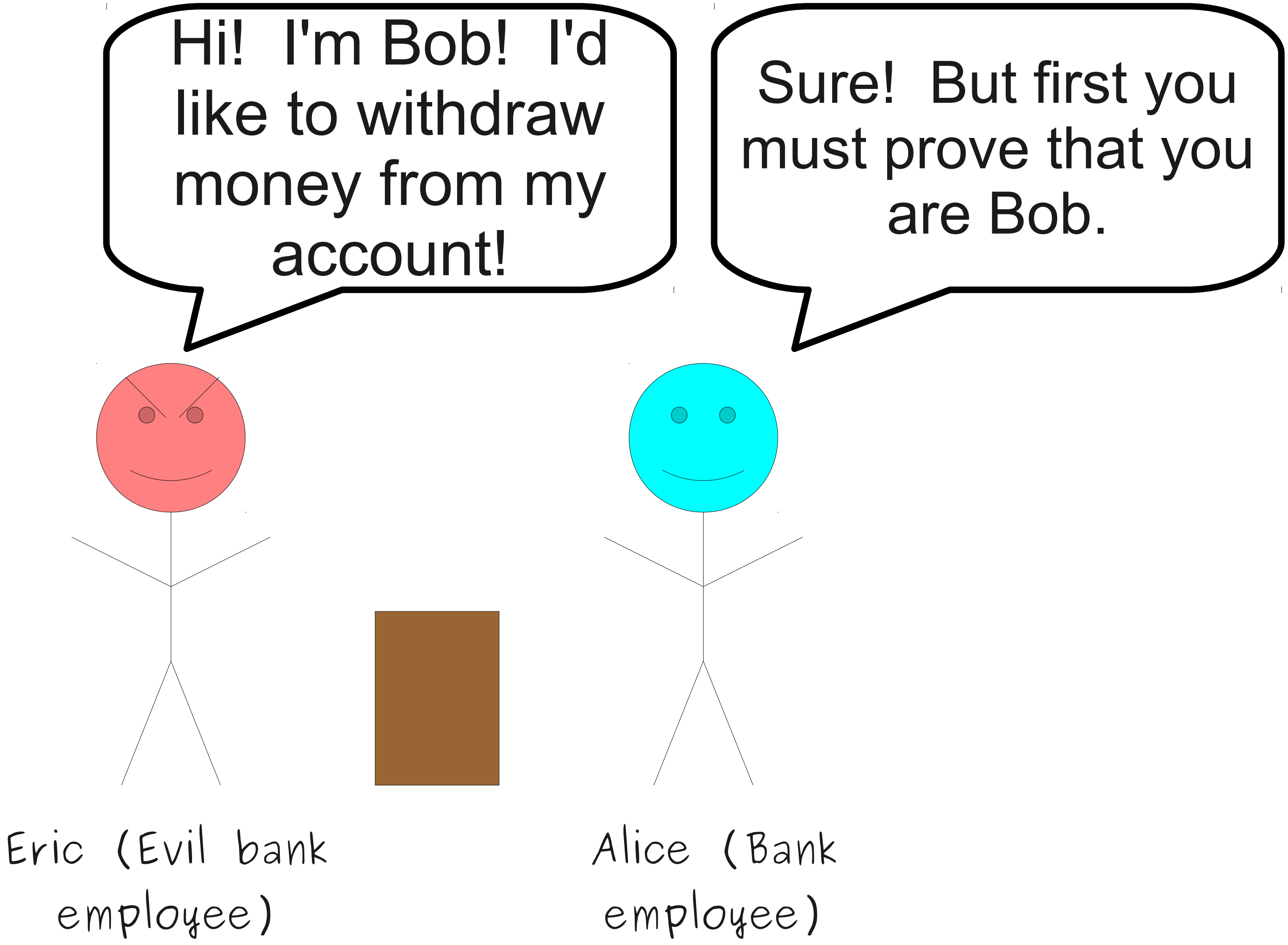
Eric (Evil bank employee)

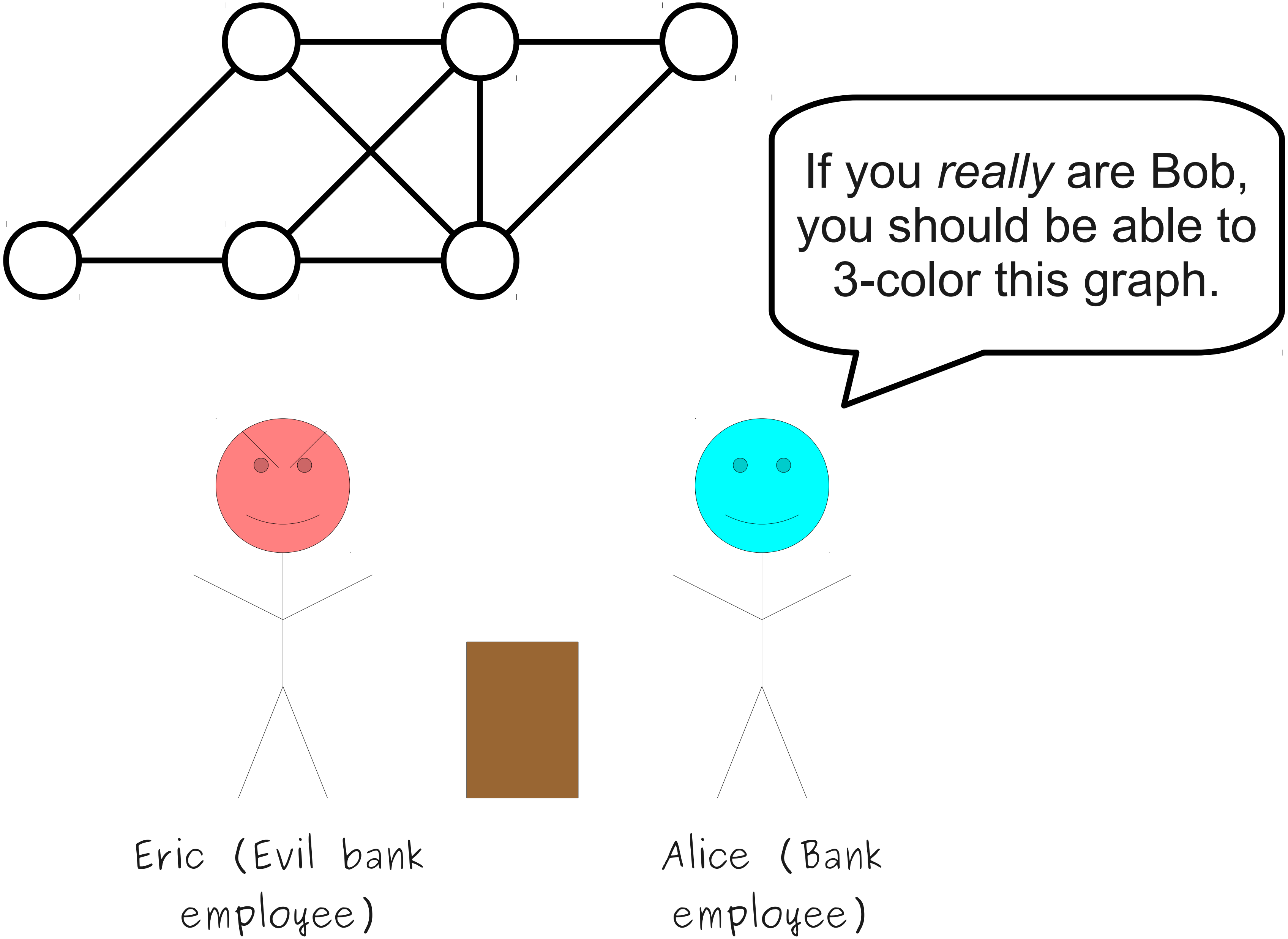

?
?
?
?
?
?

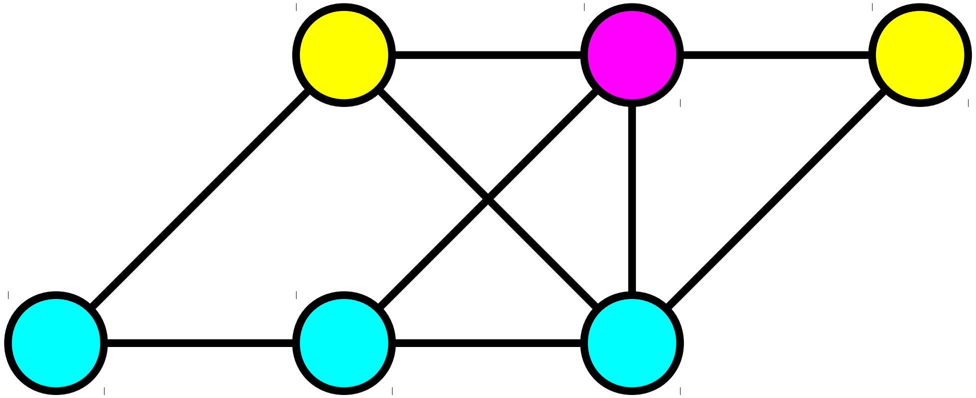

Bob (Bank customer)
Eric (Evil bank employee)

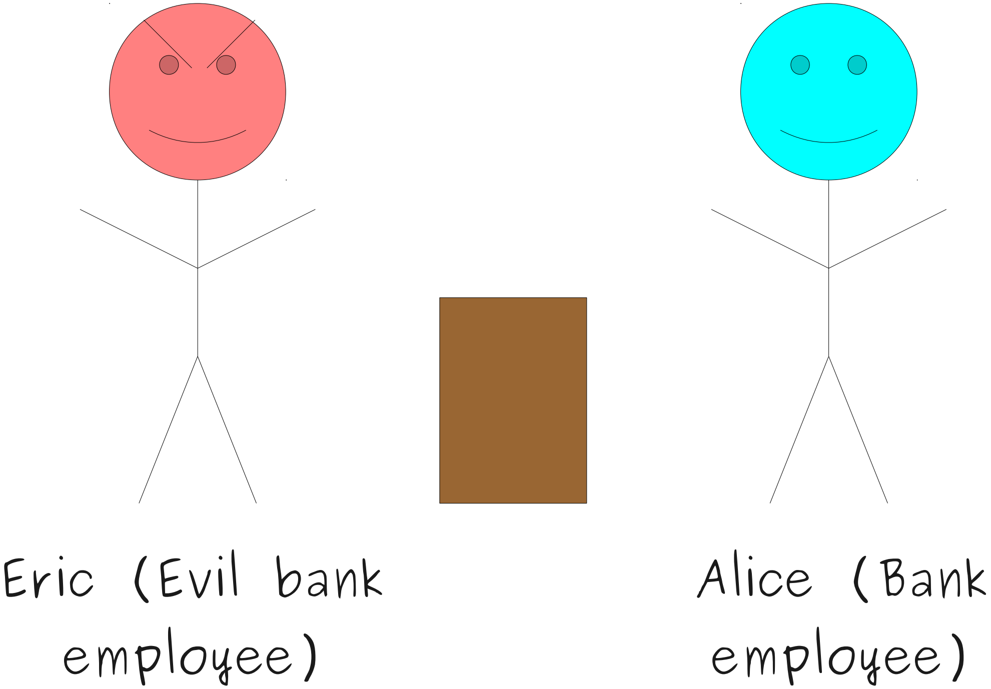

?
?
?
?
?
What colors are these nodes?
Bob (Bank customer)
Eric (Evil bank employee)

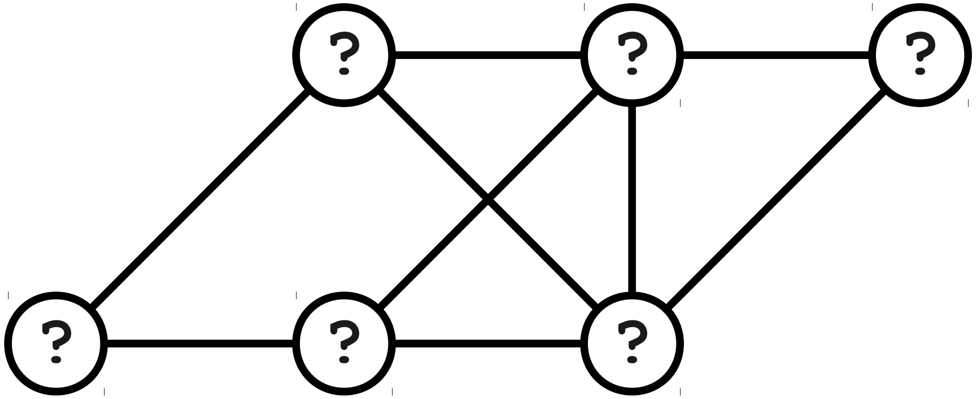

Okay Bob! Here's your money!
Bob (Bank customer)
Eric (Evil bank employee)

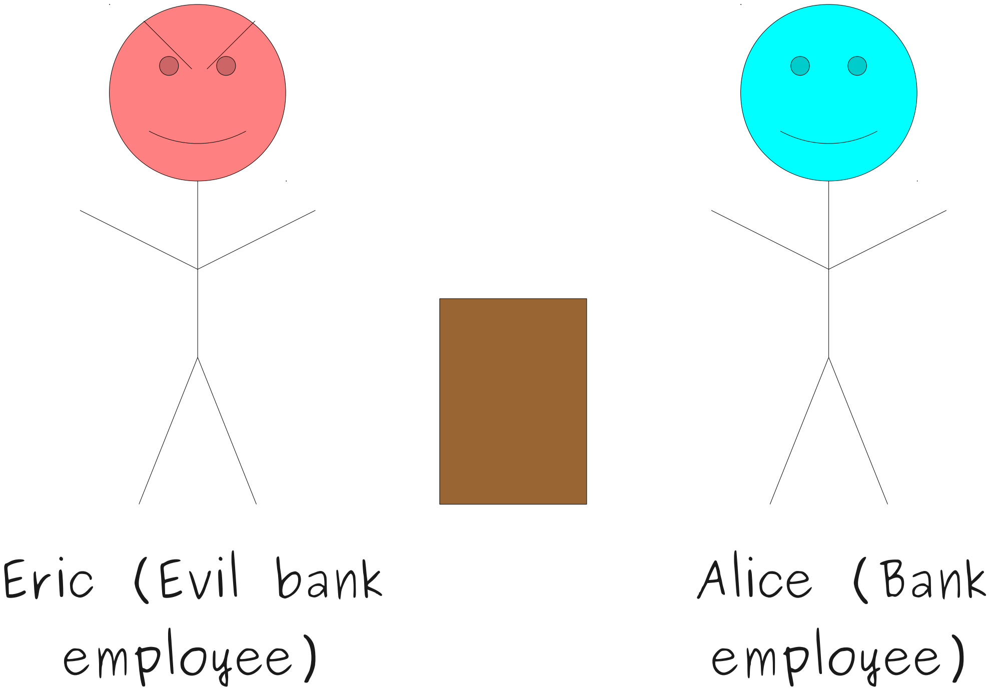

Hi! I'm Bob! I'd like to withdraw money from my account!
Sure! But first you must prove that you are Bob.
Eric (Evil bank employee)
Alice (Bank employee)

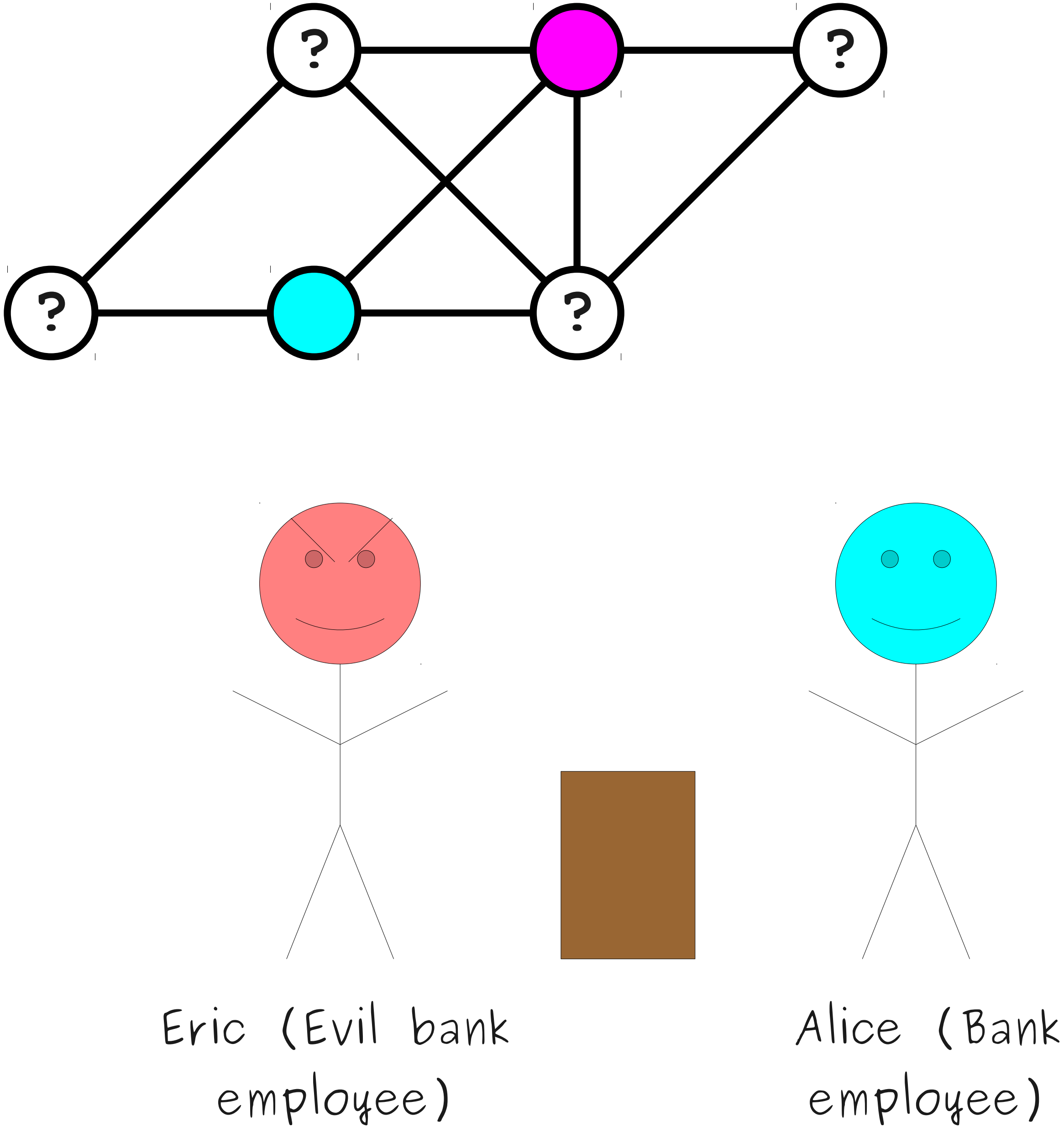

If you *really* are Bob, you should be able to 3-color this graph.
Eric (Evil bank employee)
Alice (Bank employee)

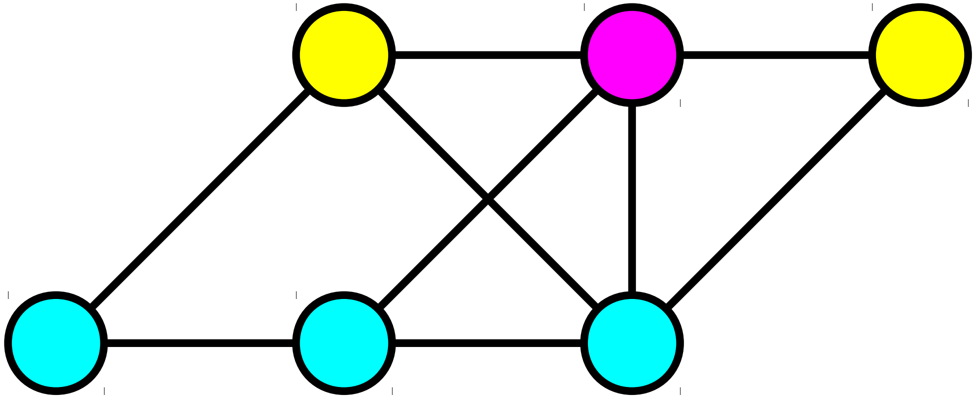


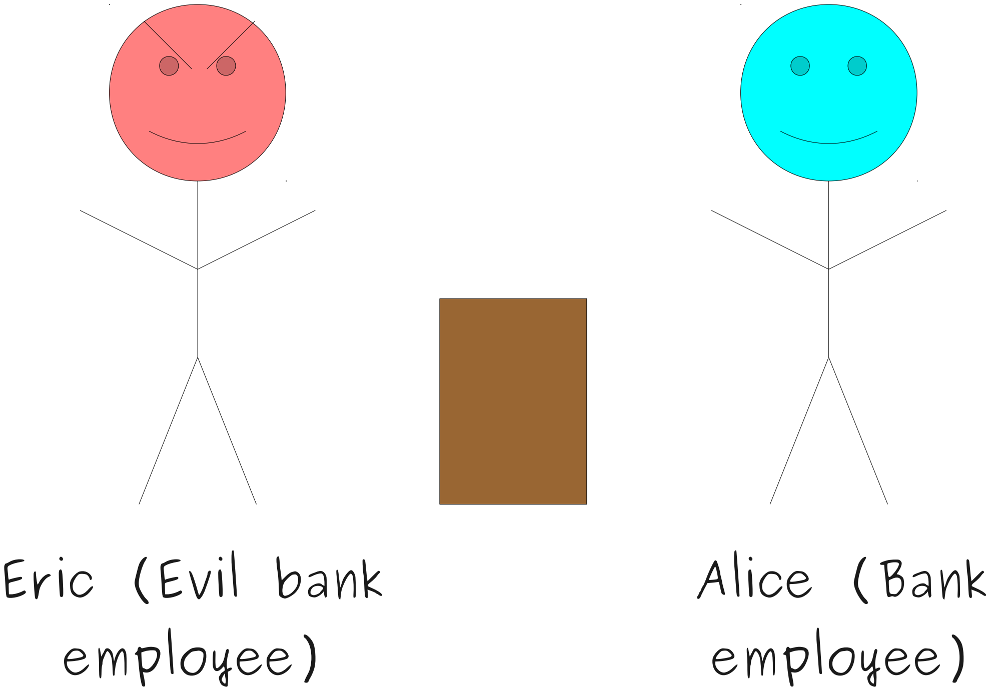

Eric (Evil bank employee)
Alice (Bank employee)

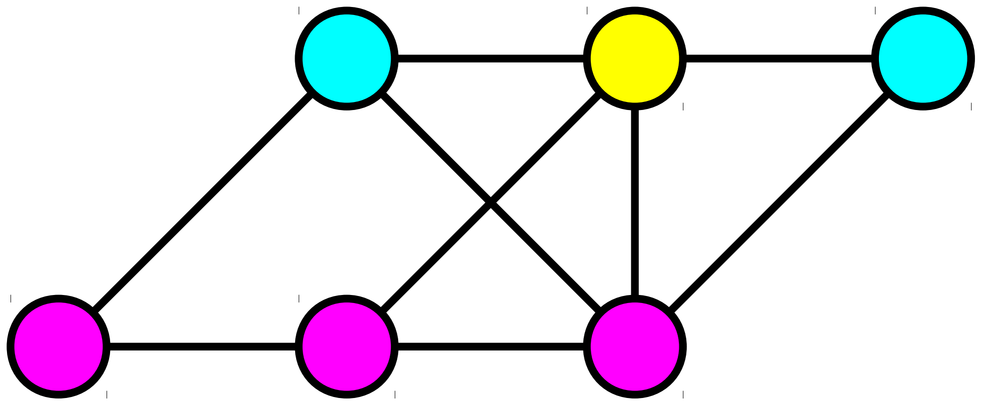

?
?
?
?
?
?

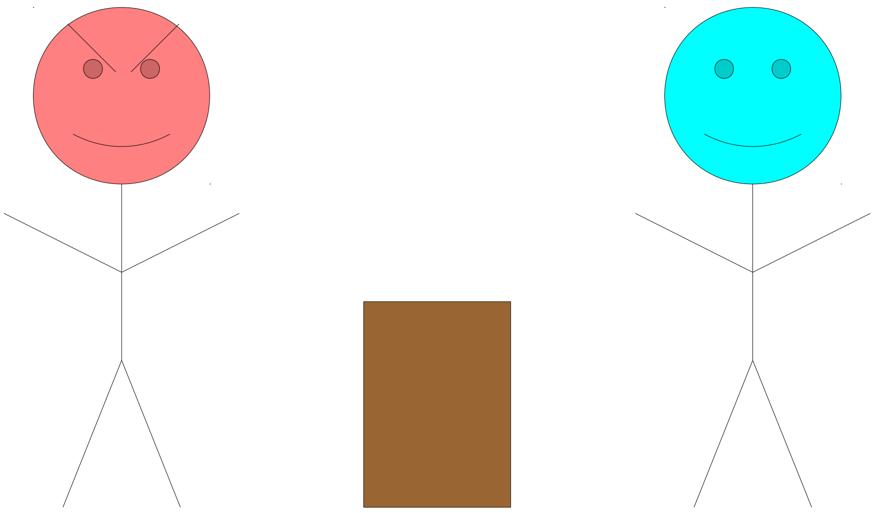

Eric (Evil bank employee)
Alice (Bank employee)

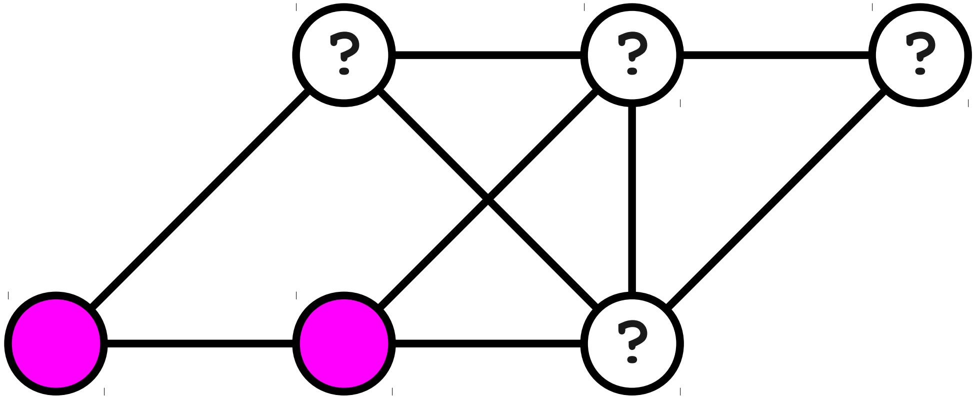

?
?
?
?
?
Eric (Evil bank employee)
Alice (Bank employee)

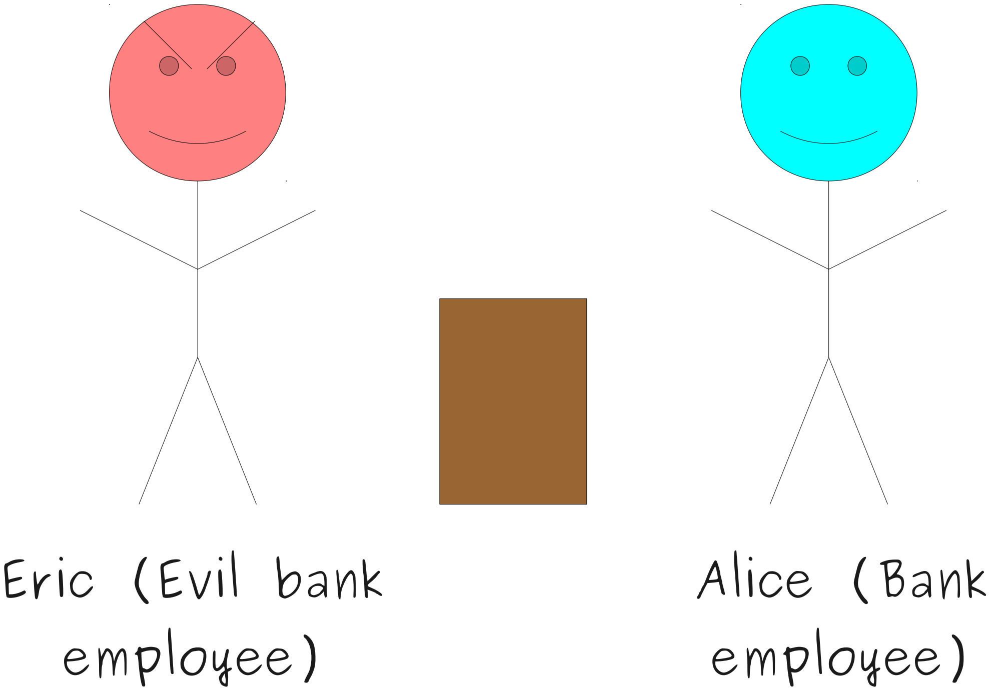


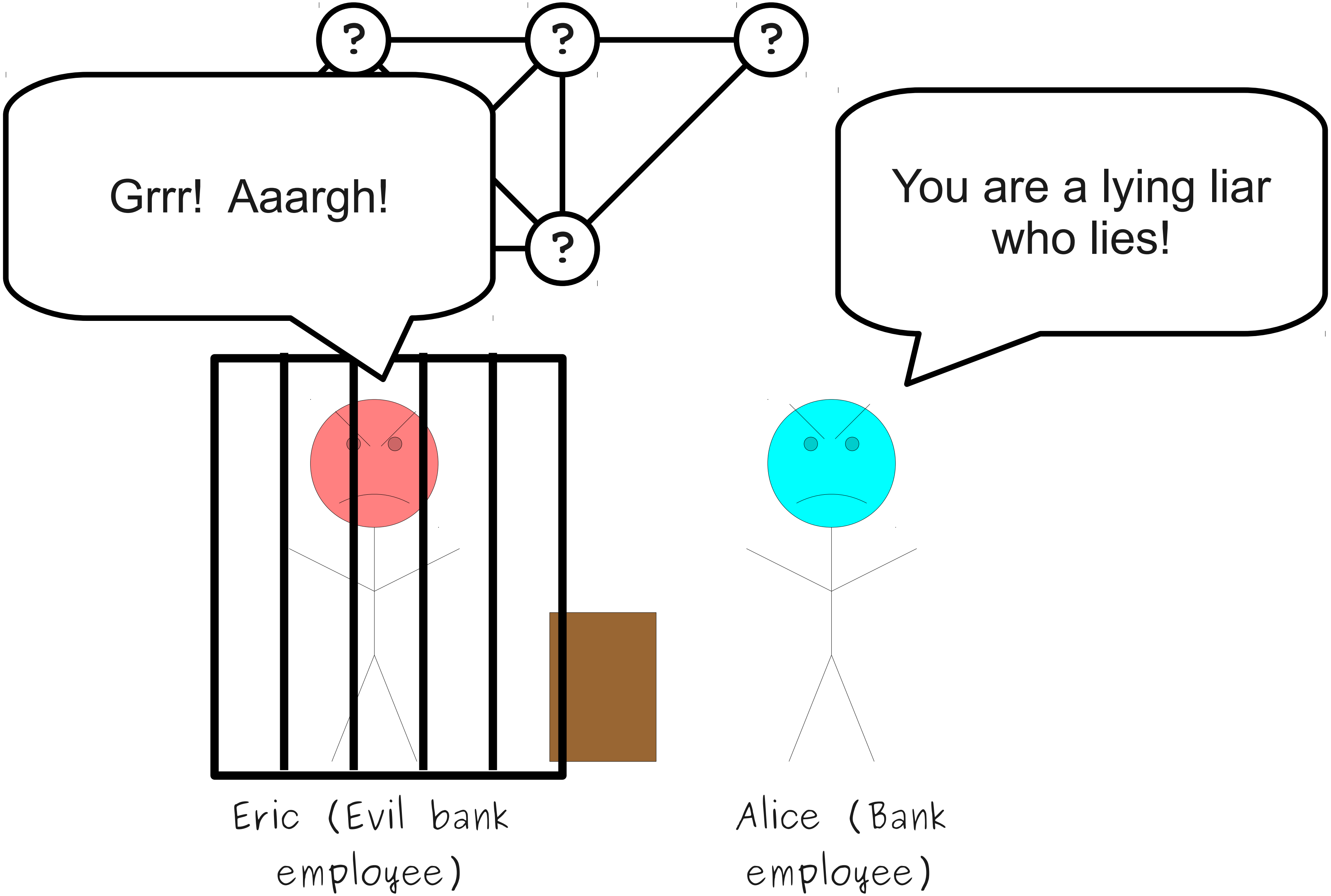

Eric (Evil bank employee)
Alice (Bank employee)




Eric (Evil bank employee)
Alice (Bank employee)


?
?
?
?


Eric (Evil bank employee)
Alice (Bank employee)


?
?
?
?
Grrr! Aaargh!
You are a lying liar who lies!
Eric (Evil bank employee)
Alice (Bank employee)

Why This Works

- At each step, Eric (the verifier) will see one of two things:
  - Two nodes of the same color connected by an edge (so Bob (the prover) is definitely lying).
  - Two nodes of different colors connected by an edge. Eric already knew that this would have to happen in a 3-coloring, and can't use the specific colors to reconstruct a 3-coloring of the graph.
- After a “large number” of rounds, Eric can conclude, with extremely high probability, that Bob is not lying.
- No matter how long Eric does this, he will *never* learn Bob's coloring.

Why This Matters

- This whole scheme is wrapped up in **P** and **NP**.
  - Searching for a commitment scheme led us to one-way functions. If a one-way function exists, then **P** ≠ **NP**.
  - Building a zero-knowledge proof required us to use a **NP**-complete problem:
    - **NP**-completeness *probably* means “impossible to solve efficiently.”
    - Membership in **NP** means “efficiently verifiable.”

Summary

- Some **NP**-complete problems cannot even be *approximated* by a polynomial-time algorithm (unless **P** = **NP**).
- The **PCP theorem** shows that many fundamental **NP**-complete problems are hard to approximate (assuming **P** ≠ **NP**).
- Cryptography makes extensive use of hard problems as building blocks for secure communication.
- **Zero-knowledge proofs** and **commitment schemes** are intimately connected with **P** and **NP**, but use hard problems to build secure systems.

Next Time

- **The Big Picture**
  - How does everything fit together?
- **Where to Go from Here**
  - What's next in theoretical computer science?