Introduction

601.442/642 Modern Cryptography

20th January 2026

Course Staff

Instructors

Harry Eldridge
(heldrid2@jhmi.edu)

Aditya Hegde
(ahegde3@jhu.edu)

Course Staff

Instructors

Harry Eldridge
(heldrid2@jhmi.edu)

Aditya Hegde
(ahegde3@jhu.edu)

TA

Shruthi Prusty
(sprusty1@jhu.edu)

What is Cryptography?

A Brief History of Cryptography

A Brief History of Cryptography

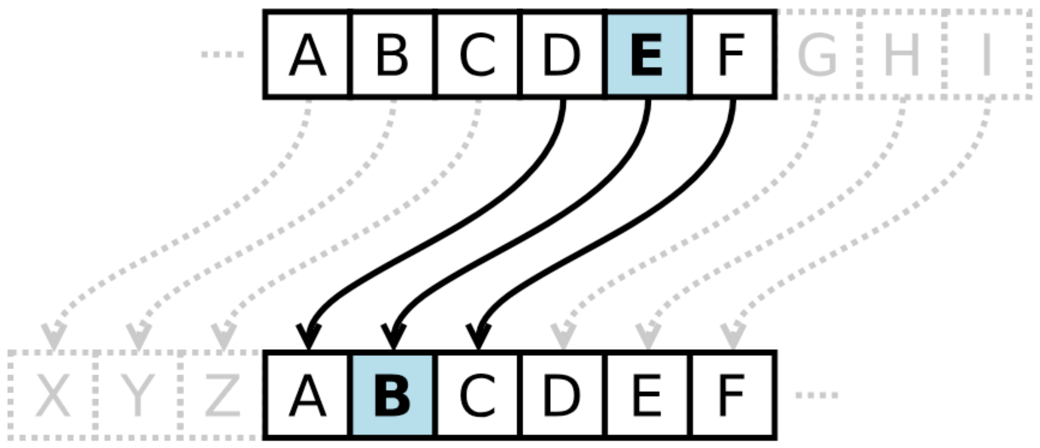
Classical Cryptography: The art of secret writing

Pre-1950

A Brief History of Cryptography

Classical Cryptography: The art of secret writing

Pre-1950

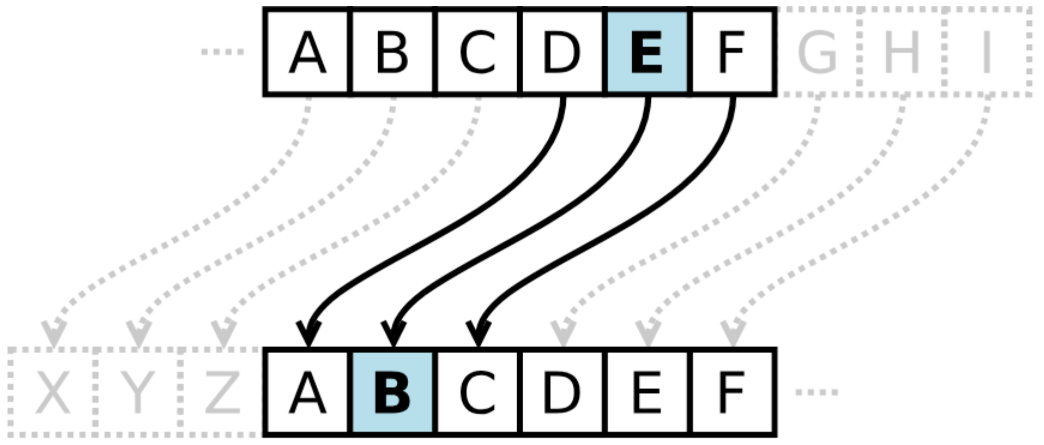


Caesar Cipher: Substitution cipher used by Julius Caesar for military correspondence in the last century BC.

A Brief History of Cryptography

Classical Cryptography: The art of secret writing

Pre-1950



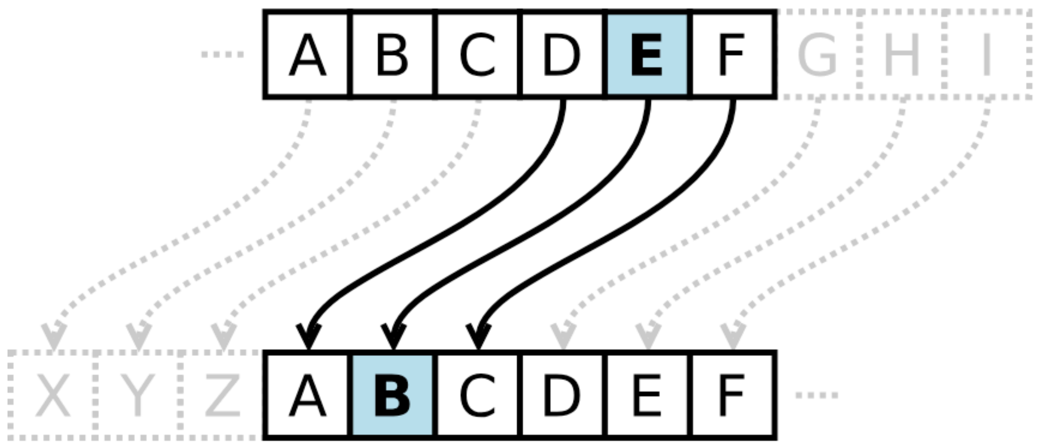
Caesar Cipher: Substitution cipher used by Julius Caesar for military correspondence in the last century BC.

Al-Kindi (9th century AD) provided a systematic way to break substitution ciphers.

A Brief History of Cryptography

Classical Cryptography: The art of secret writing

Pre-1950



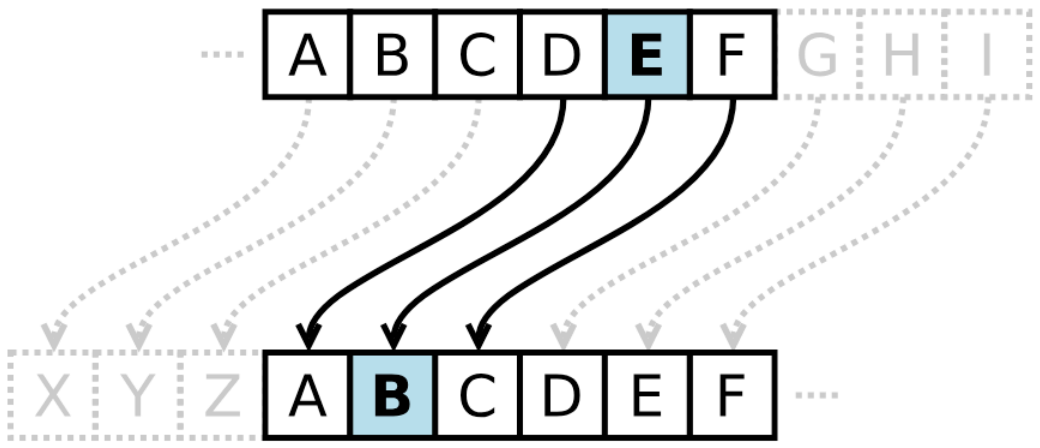
Caesar Cipher: Substitution cipher used by Julius Caesar for military correspondence in the last century BC.

Al-Kindi (9th century AD) provided a systematic way to break substitution ciphers.

A Brief History of Cryptography

Classical Cryptography: The art of secret writing

Pre-1950



Caesar Cipher: Substitution cipher used by Julius Caesar for military correspondence in the last century BC.

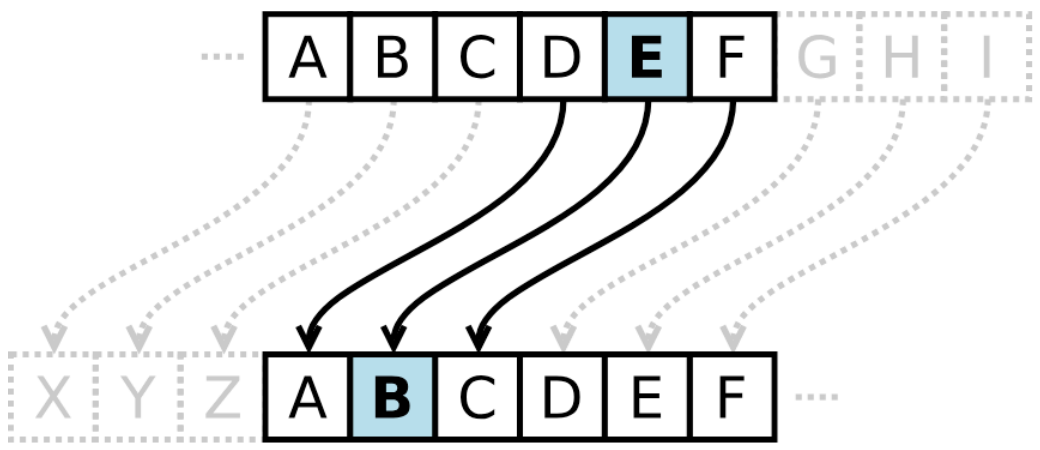
Al-Kindi (9th century AD) provided a systematic way to break substitution ciphers.

Enigma Machine: Cipher device used by Germany in World War II.

A Brief History of Cryptography

Classical Cryptography: The art of secret writing

Pre-1950



Caesar Cipher: Substitution cipher used by Julius Caesar for military correspondence in the last century BC.

Al-Kindi (9th century AD) provided a systematic way to break substitution ciphers.

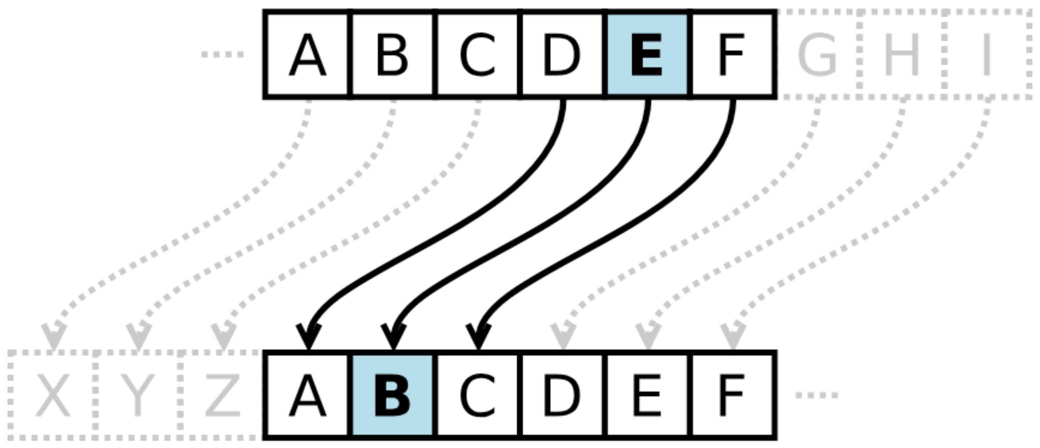
Enigma Machine: Cipher device used by Germany in World War II.

Finally broken by Alan Turing and his team at Bletchley Park.

A Brief History of Cryptography

Classical Cryptography: The art of secret writing

Pre-1950



Caesar Cipher: Substitution cipher used by Julius Caesar for military correspondence in the last century BC.

Al-Kindi (9th century AD) provided a systematic way to break substitution ciphers.

One of the first applications of computing

Enigma Machine: Cipher device used by Germany in World War II.

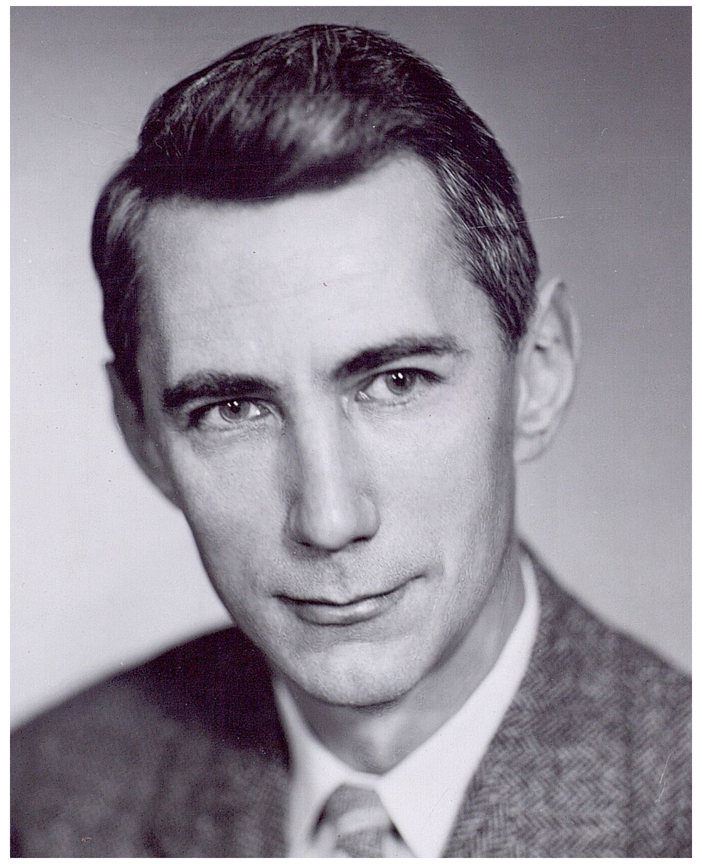
Finally broken by Alan Turing and his team at Bletchley Park.

A Brief History of Cryptography

The Origins of Modern Cryptography

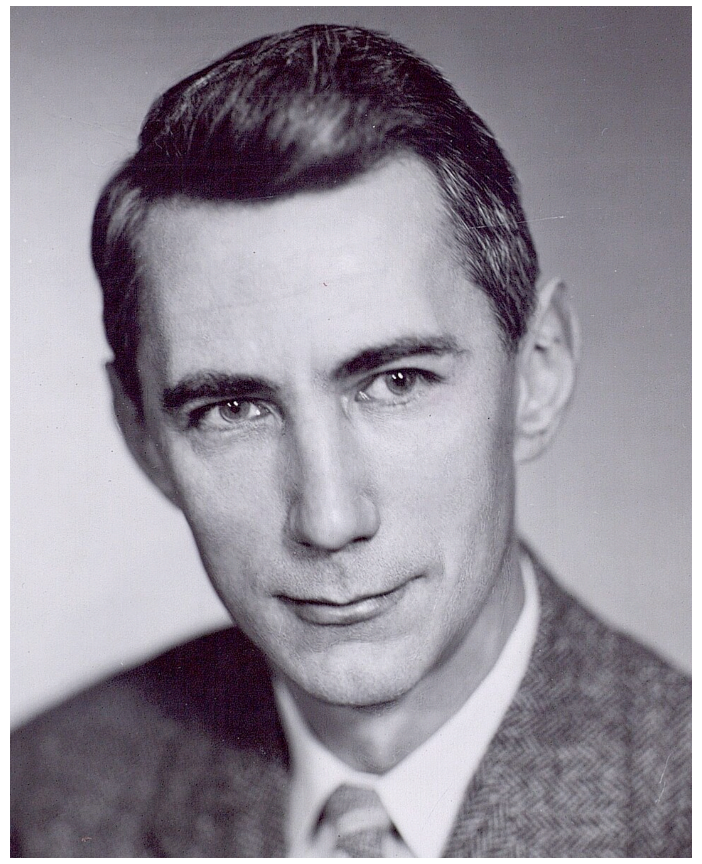
A Brief History of Cryptography

The Origins of Modern Cryptography



A Brief History of Cryptography

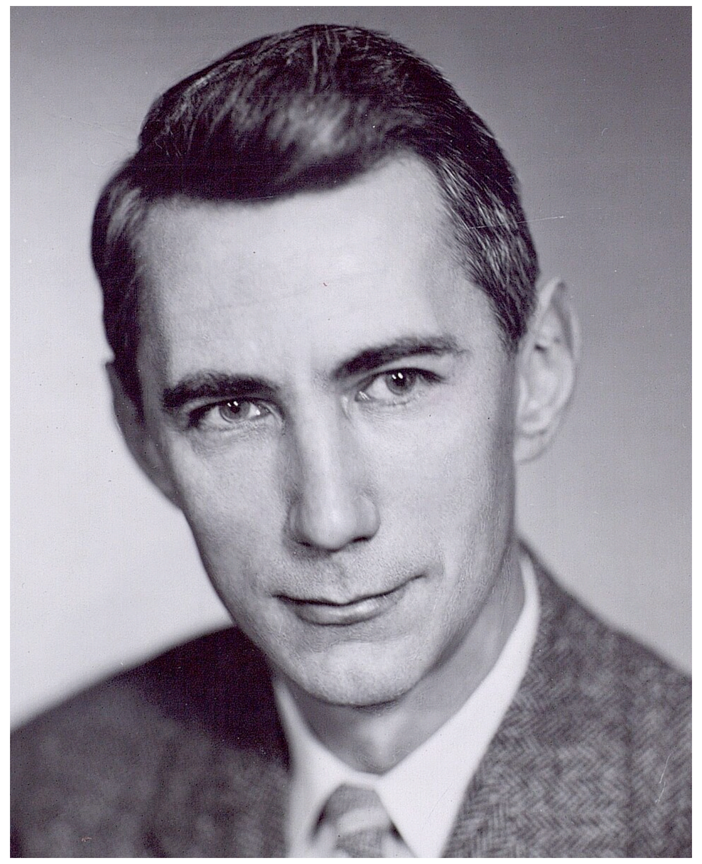
The Origins of Modern Cryptography



Claude Shannon

A Brief History of Cryptography

The Origins of Modern Cryptography



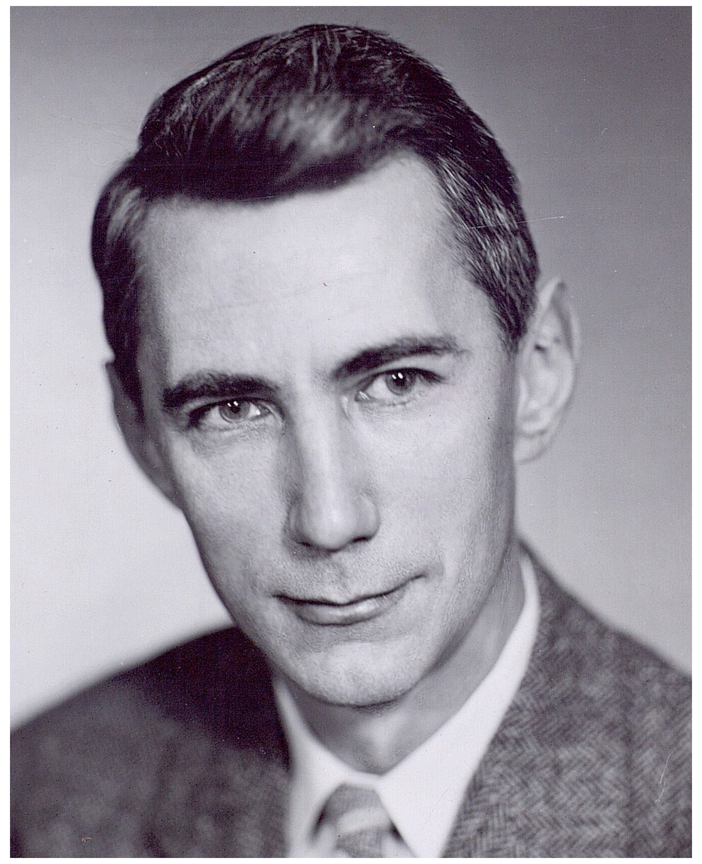
Claude Shannon

"Communication Theory of Secrecy Systems" (1949)

Mathematical foundations of secrecy

A Brief History of Cryptography

The Origins of Modern Cryptography


Claude Shannon

"Communication Theory of Secrecy Systems" (1949)

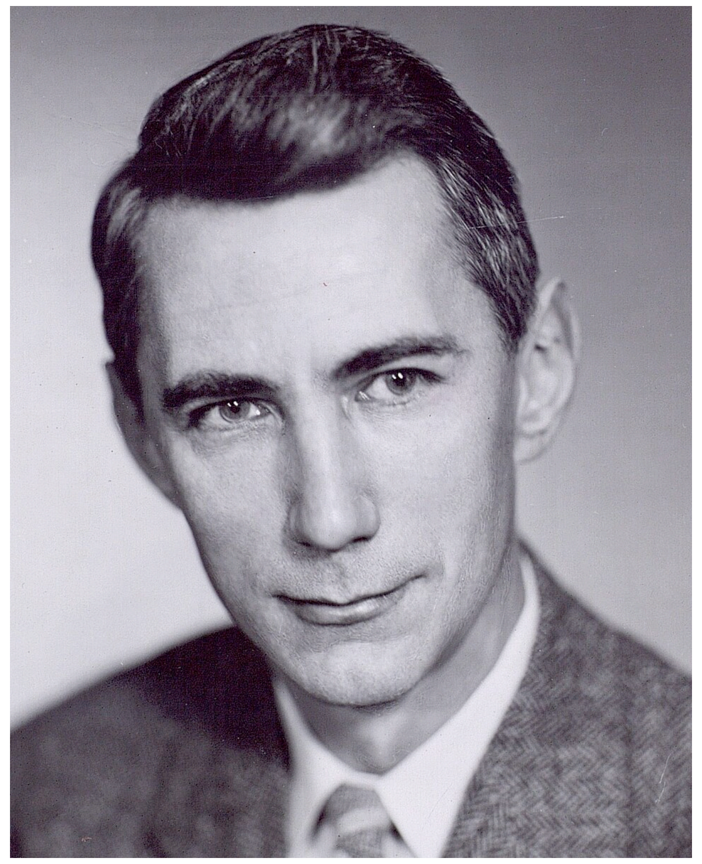
Mathematical foundations of secrecy

"A Mathematical Theory of Communication" (1948)

Founded information theory

A Brief History of Cryptography

The Origins of Modern Cryptography



Claude Shannon

"Communication Theory of Secrecy Systems" (1949)

Mathematical foundations of secrecy

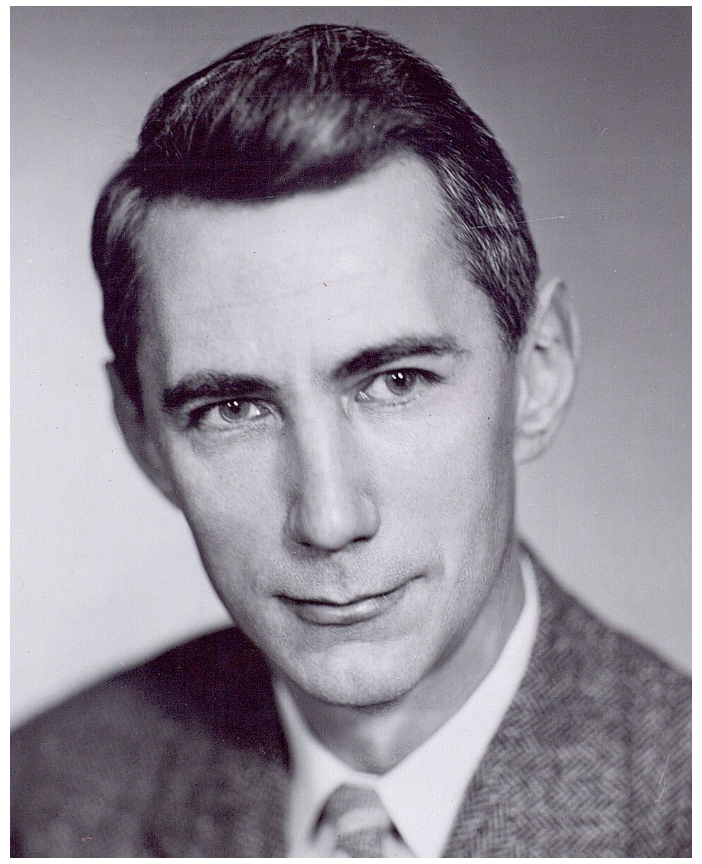
"A Mathematical Theory of Communication" (1948)

Founded information theory

From the 1970s onward, cryptography emerged as a rigorous, computational science.

A Brief History of Cryptography

The Origins of Modern Cryptography



Claude Shannon

"Communication Theory of Secrecy Systems" (1949)

Mathematical foundations of secrecy

"A Mathematical Theory of Communication" (1948)

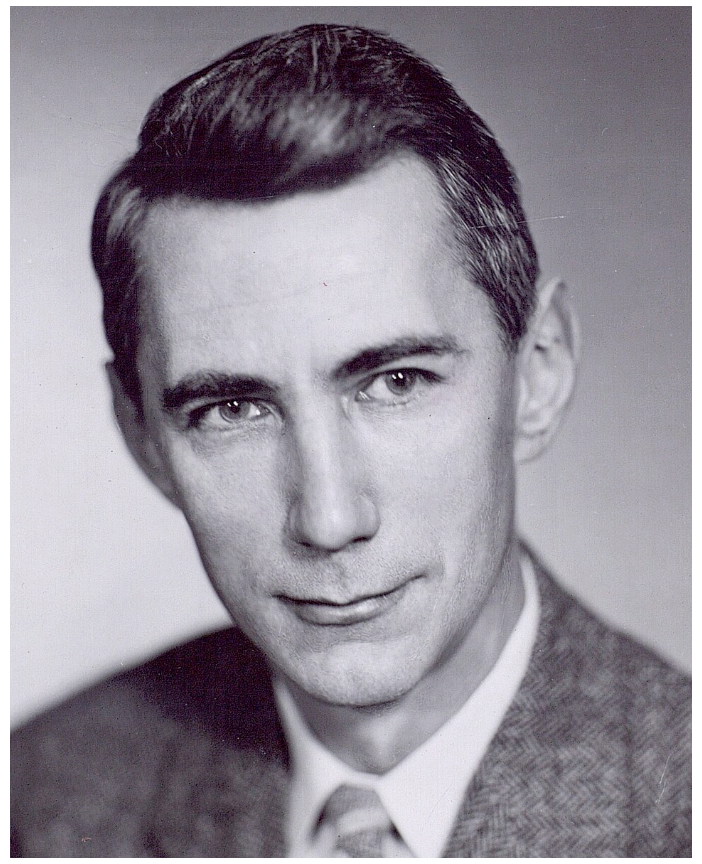
Founded information theory

From the 1970s onward, cryptography emerged as a rigorous, computational science.

The first annual international cryptography conference CRYPTO was held in 1981 with 102 attendees.

A Brief History of Cryptography

The Origins of Modern Cryptography



Claude Shannon

"Communication Theory of Secrecy Systems" (1949)

Mathematical foundations of secrecy

"A Mathematical Theory of Communication" (1948)

Founded information theory

From the 1970s onward, cryptography emerged as a rigorous, computational science.

The first annual international cryptography conference CRYPTO was held in 1981 with 102 attendees.

8 of them have since won the Turing Award (10 in total so far).

Modern Cryptography

Modern Cryptography

Design and analysis of systems that need to withstand malicious attempts to abuse it.

Modern Cryptography

Design and analysis of systems that need to withstand malicious attempts to abuse it.

Another Definition:

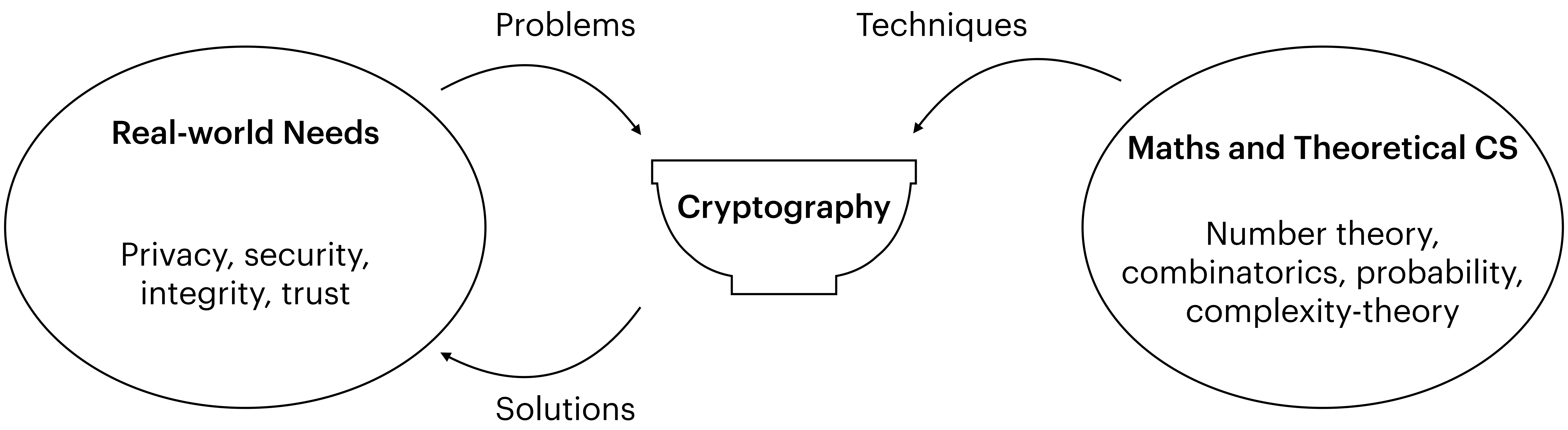
Algorithmic and mathematical foundations of secure communication and computation.

Modern Cryptography

Design and analysis of systems that need to withstand malicious attempts to abuse it.

Another Definition:

Algorithmic and mathematical foundations of secure communication and computation.



Modern Cryptography

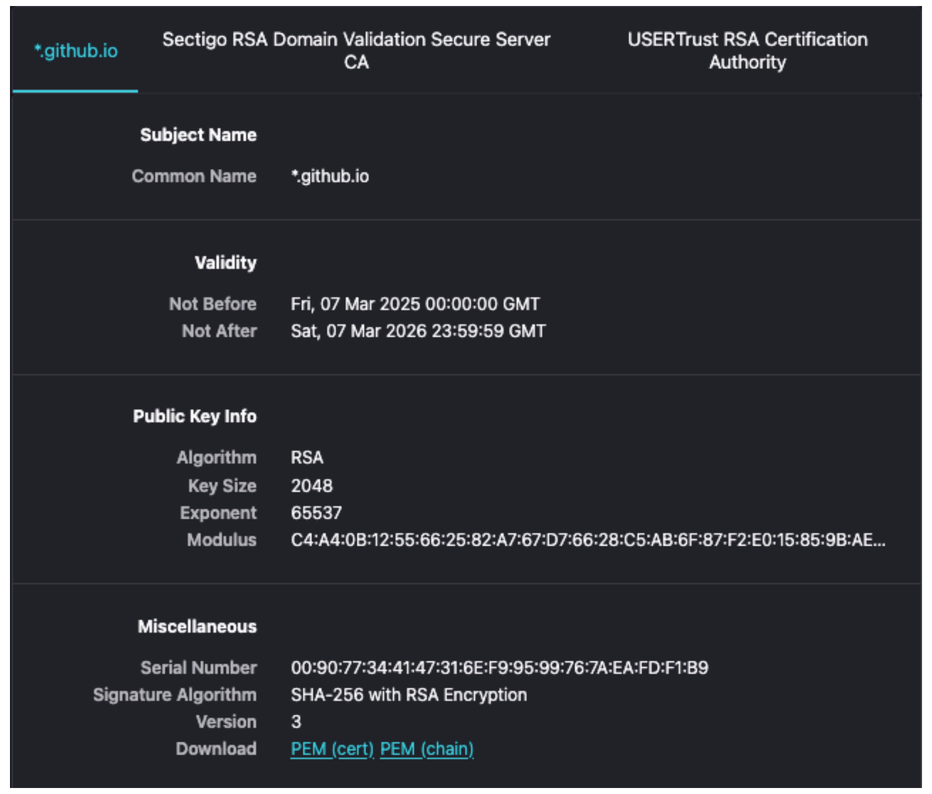
Design and analysis of systems that need to withstand malicious attempts to abuse it.

Modern Cryptography

Design and analysis of systems that need to withstand malicious attempts to abuse it.

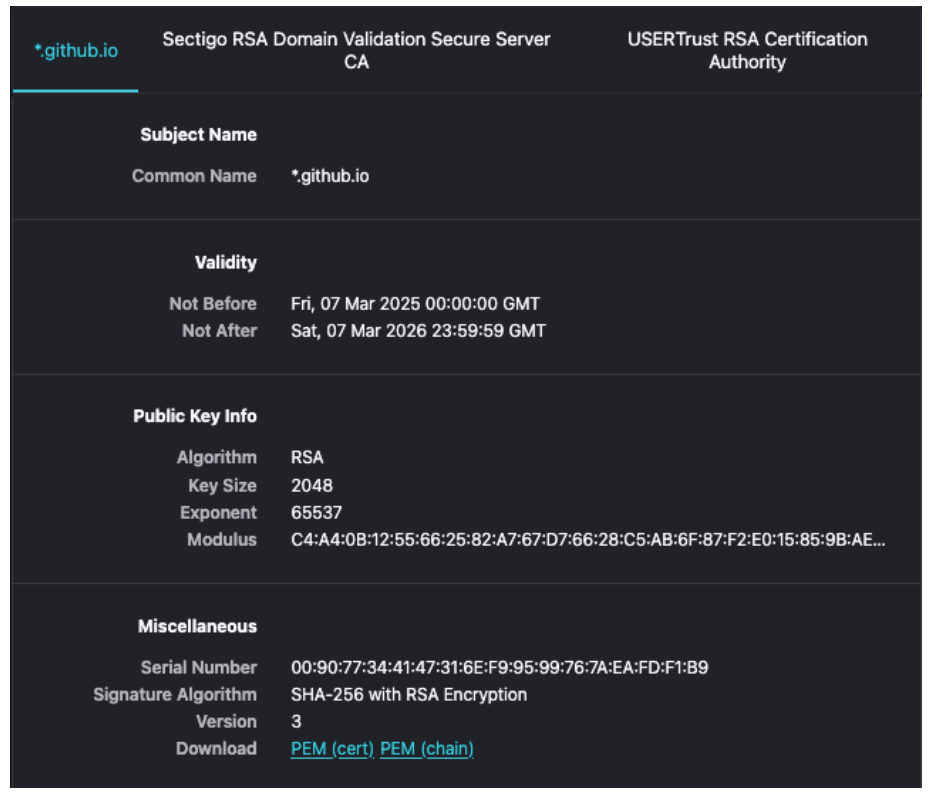
Modern Cryptography

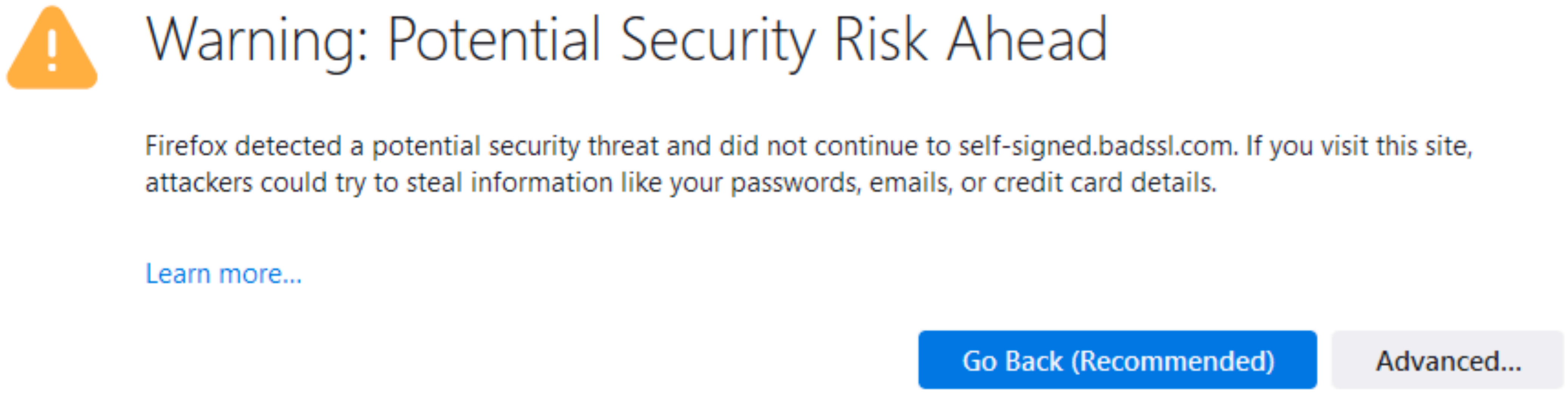
Design and analysis of systems that need to withstand malicious attempts to abuse it.



Modern Cryptography

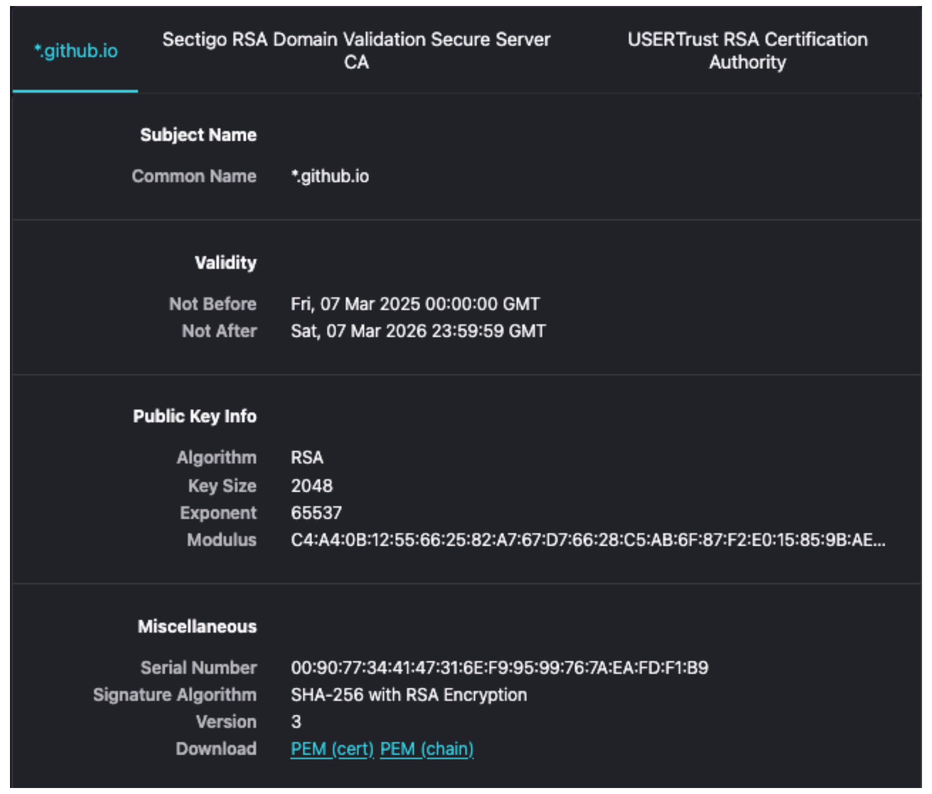
Design and analysis of systems that need to withstand malicious attempts to abuse it.





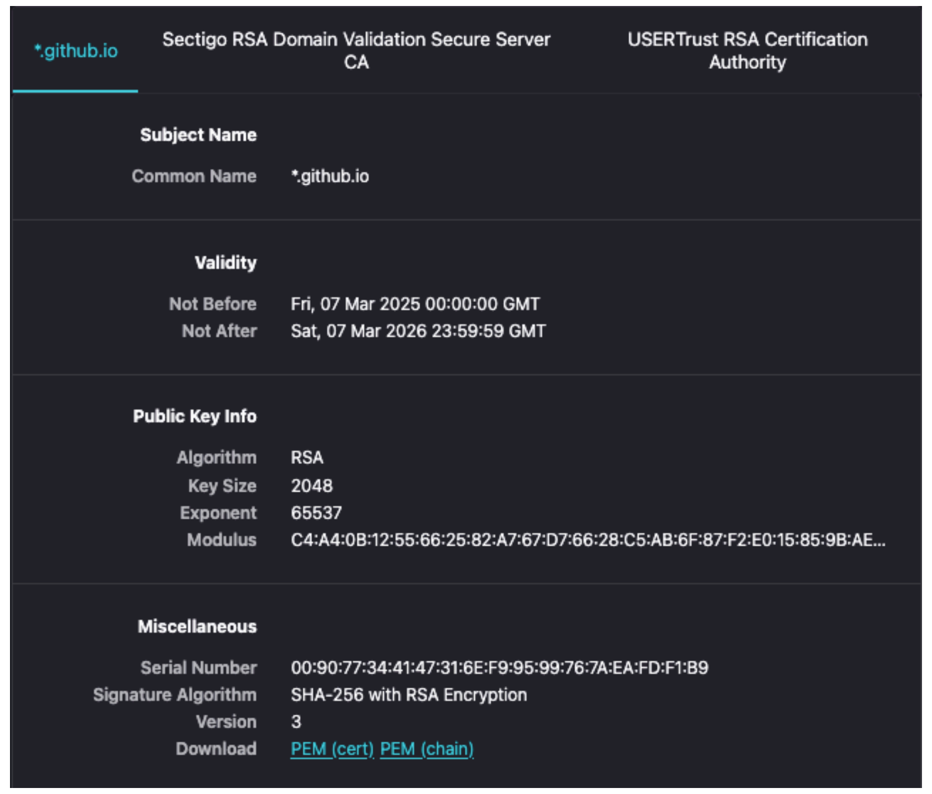
Modern Cryptography

Design and analysis of systems that need to withstand malicious attempts to abuse it.



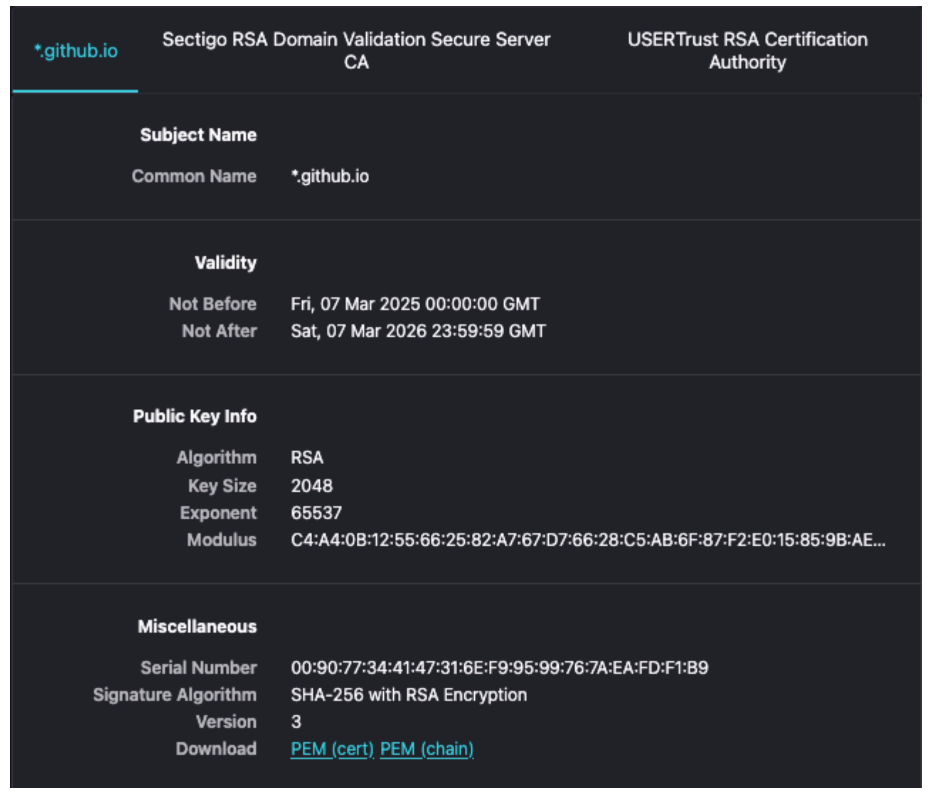
Modern Cryptography

Design and analysis of systems that need to withstand malicious attempts to abuse it.



Modern Cryptography

Design and analysis of systems that need to withstand malicious attempts to abuse it.



This course is about the foundations of Cryptography.

- Pseudorandomness
- Public-key Encryption
- Digital Signatures
- Hash Functions
- Zero-knowledge Proofs
- Secure Computation

The Pillars of Modern Cryptography

Definitions

Hardness Assumptions

Proofs

The Pillars of Modern Cryptography

Definitions

Hardness Assumptions

Proofs

If you cannot **define** something, you cannot achieve it.

The Pillars of Modern Cryptography

Definitions

Hardness Assumptions

Proofs

If you cannot **define** something, you cannot achieve it.

Model Worst-case Adversary:
- What they know
- What they can do
- What are their goals

The Pillars of Modern Cryptography

Definitions

Hardness Assumptions

Proofs

Use hard problems to **constrain** the adversary.

The Pillars of Modern Cryptography

Definitions

Hardness Assumptions

Proofs

Use hard problems to **constrain** the adversary.

Source of hard problems: number theory,

The Pillars of Modern Cryptography

Definitions

Hardness Assumptions

Proofs

Use hard problems to **constrain** the adversary.

Source of hard problems: number theory, geometry, coding theory, algebra.

The Pillars of Modern Cryptography

Definitions

Hardness Assumptions

Proofs

Use hard problems to **constrain** the adversary.

Source of hard problems: number theory, geometry, coding theory, algebra.

Cryptography is the science of useful hardness.

The Pillars of Modern Cryptography

Definitions

Hardness Assumptions

Proofs

Formally argue why a system satisfies the definition.

The Pillars of Modern Cryptography

Definitions

Hardness Assumptions

Proofs

Formally argue why a system satisfies the definition.

Reductions: If an adversary breaks system S w.r.t. definition D

then

there is an adversary that breaks the hardness assumption.

The Pillars of Modern Cryptography

Definitions

Hardness Assumptions

Proofs

Formally argue why a system satisfies the definition.

Reductions: If an adversary breaks system S w.r.t. definition D

then

there is an adversary that breaks the hardness assumption.

Either ensure security or solve a hard problem!

Course Objectives

Course Objectives

- Learn the modern, provable security-based approach to cryptography.

Course Objectives

- Learn the modern, provable security-based approach to cryptography.
    - Learn the mathematical language used to express cryptographic concepts.

Course Objectives

- Learn the modern, provable security-based approach to cryptography.
  - Learn the mathematical language used to express cryptographic concepts.
  - **Think intuitively but write rigorous proofs.**

Course Objectives

- Learn the modern, provable security-based approach to cryptography.
    - Learn the mathematical language used to express cryptographic concepts.
    - **Think intuitively but write rigorous proofs.**
- When you encounter crypto

Course Objectives

- Learn the modern, provable security-based approach to cryptography.
  - Learn the mathematical language used to express cryptographic concepts.
  - **Think intuitively but write rigorous proofs.**
- When you encounter crypto
  - Understand key terms

Course Objectives

- Learn the modern, provable security-based approach to cryptography.
  - Learn the mathematical language used to express cryptographic concepts.
  - **Think intuitively but write rigorous proofs.**
- When you encounter crypto
  - Understand key terms
  - Framework to reason about security guarantees

Course Objectives

- Learn the modern, provable security-based approach to cryptography.
  - Learn the mathematical language used to express cryptographic concepts.
  - **Think intuitively but write rigorous proofs.**
- When you encounter crypto
  - Understand key terms
  - Framework to reason about security guarantees
  - Understand what goes on “under the hood”

Course Objectives

- Learn the modern, provable security-based approach to cryptography.
  - Learn the mathematical language used to express cryptographic concepts.
  - **Think intuitively but write rigorous proofs.**
- When you encounter crypto
  - Understand key terms
  - Framework to reason about security guarantees
  - Understand what goes on “under the hood”
- Develop “crypto mindset”

Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation

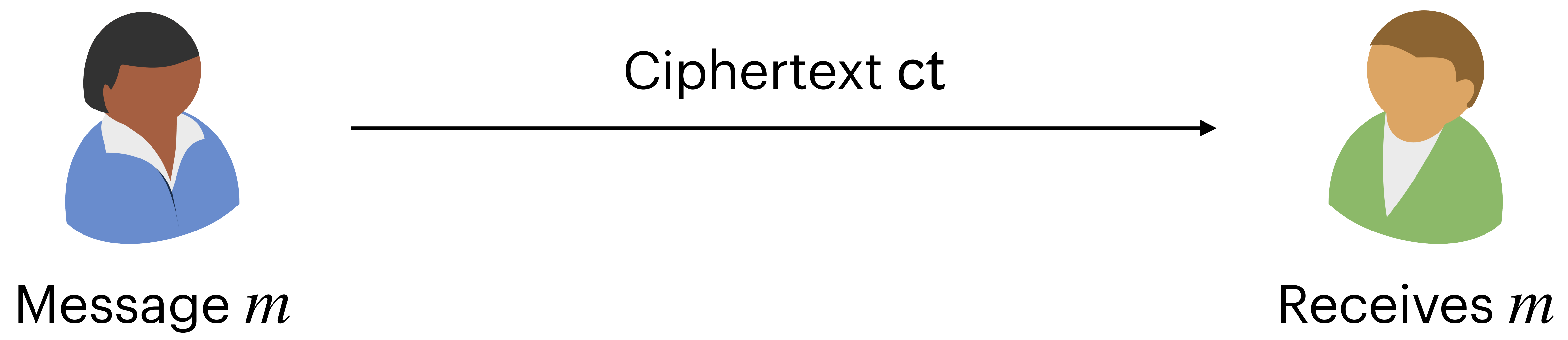
Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation

Foundations of provable security

Topics

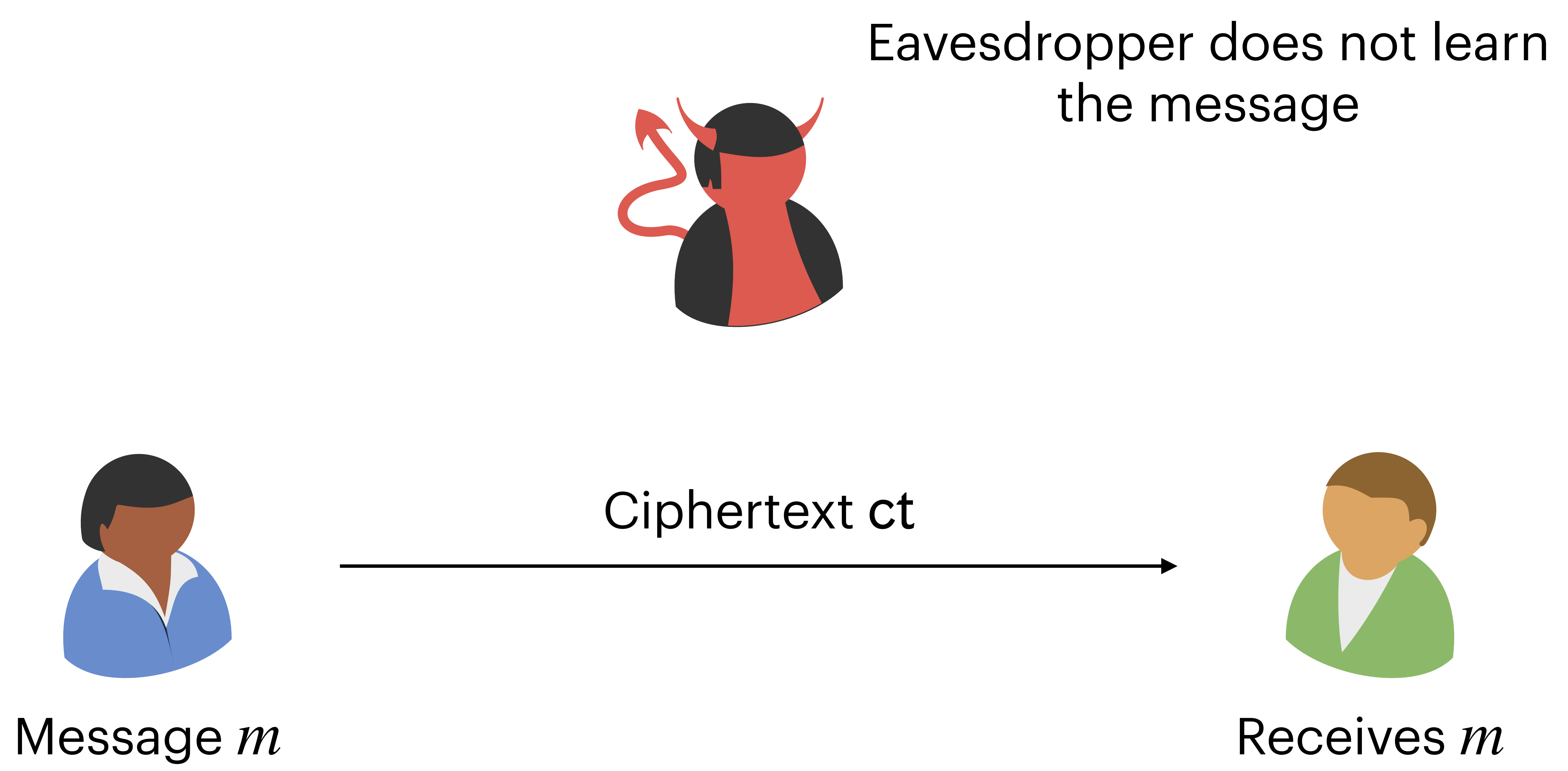
- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation



Encryption Schemes

Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation



Encryption Schemes

Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation

Authentication and Integrity

Topics

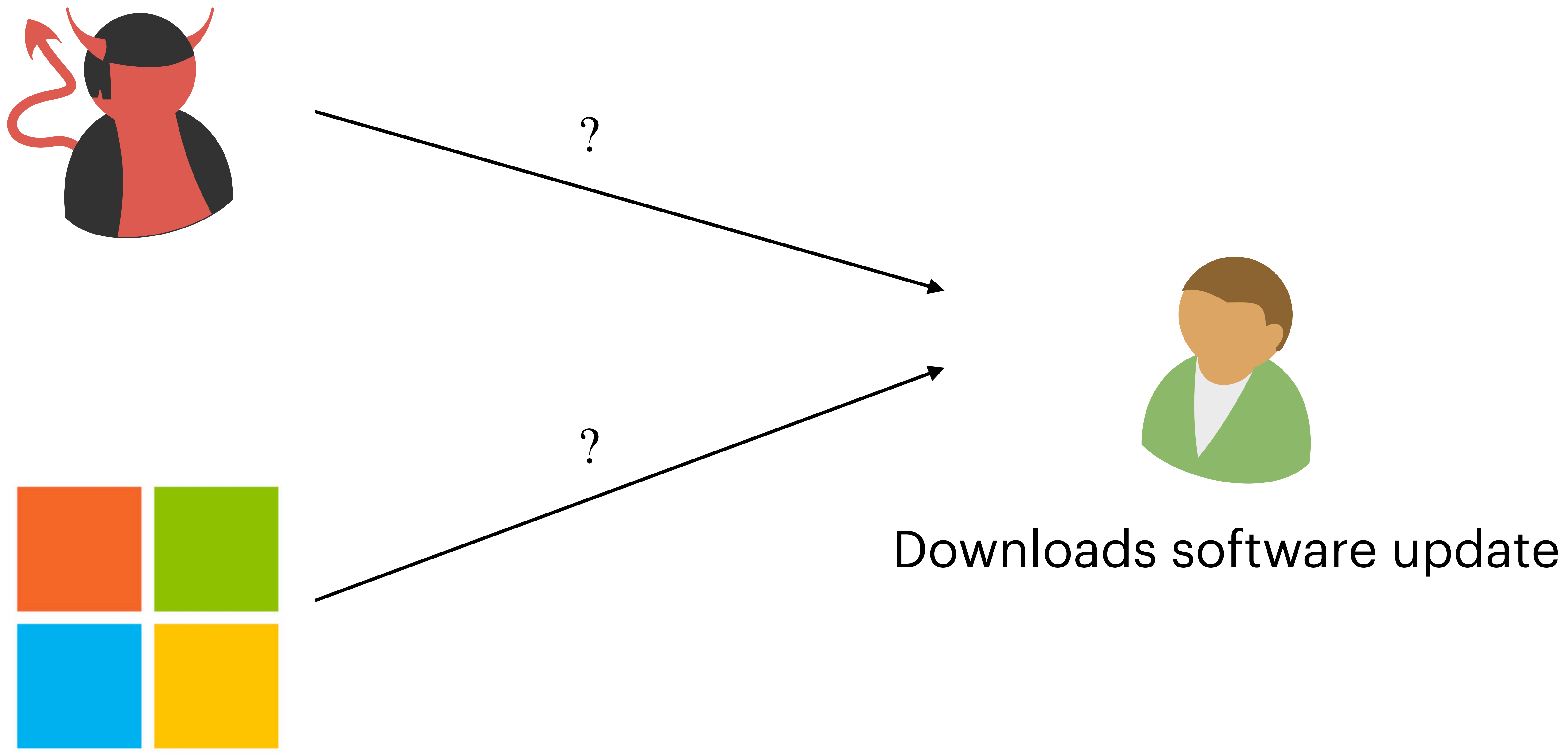
- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation

Downloads software update

Authentication and Integrity

Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation



Authentication and Integrity

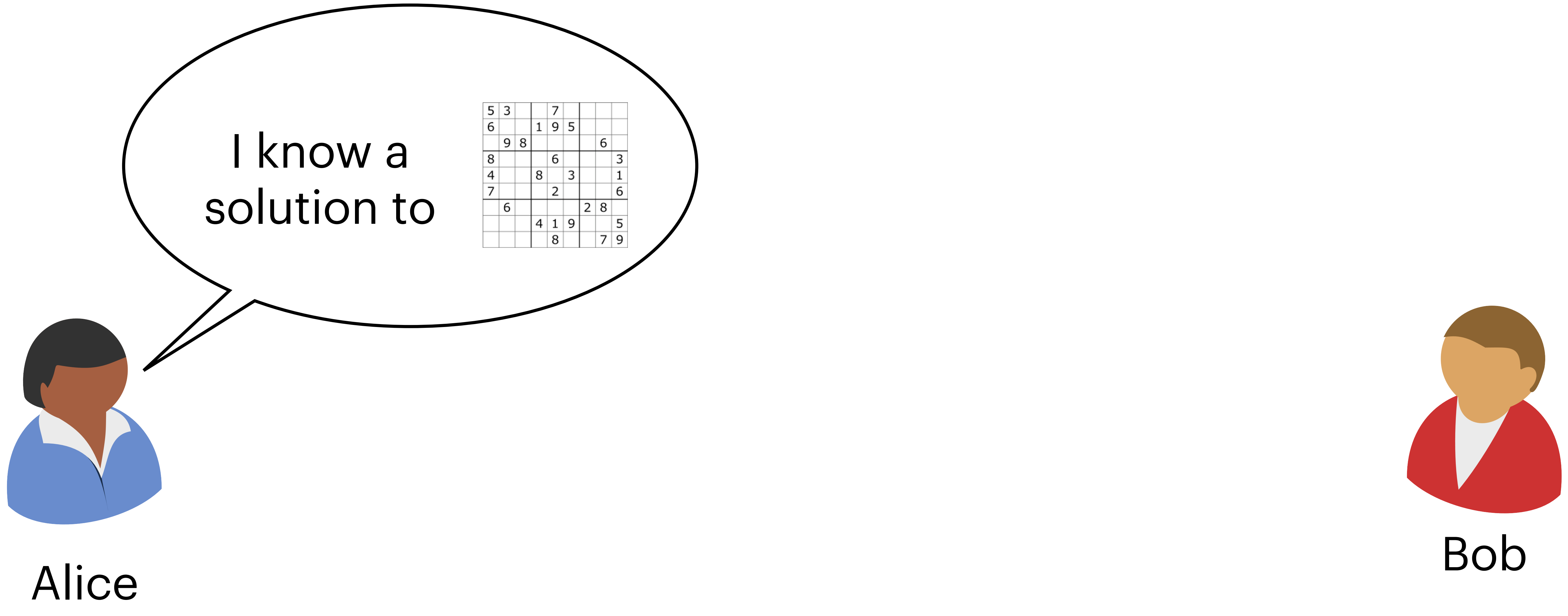
Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation

Prove that a statement is true without conveying any additional knowledge.

Topics

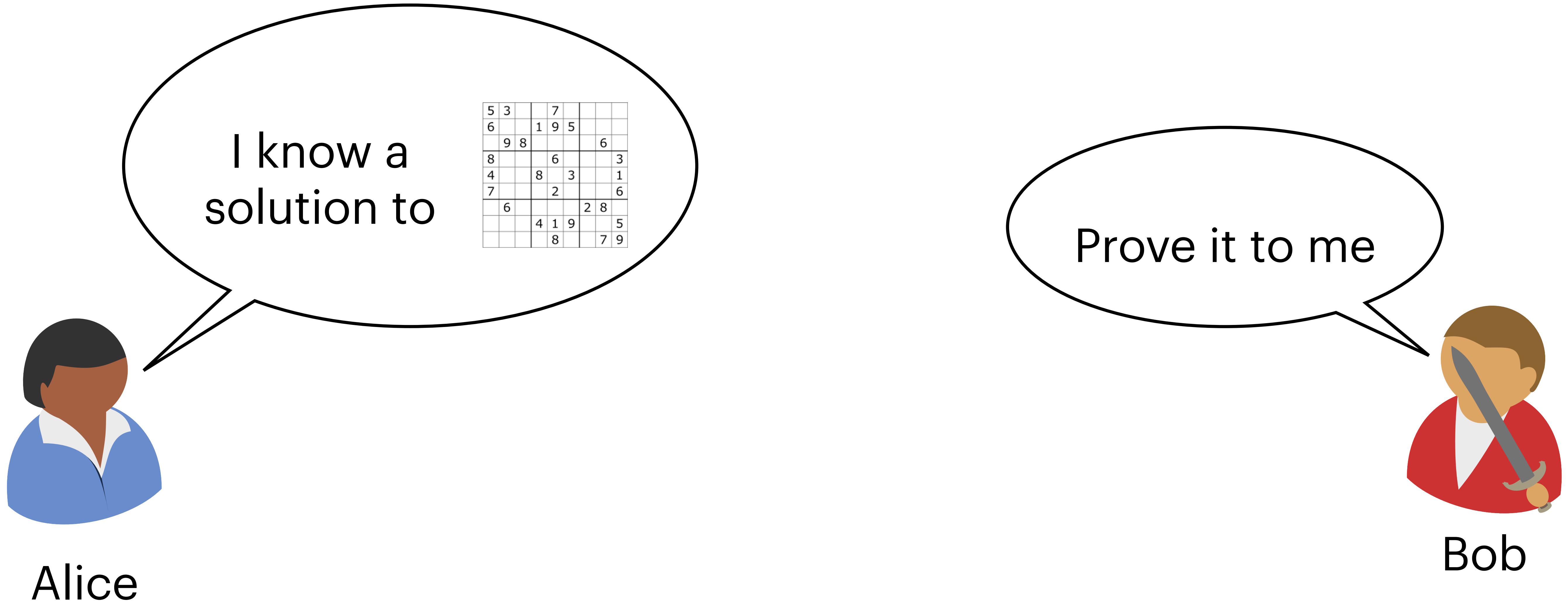
- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation



Prove that a statement is true without conveying any additional knowledge.

Topics

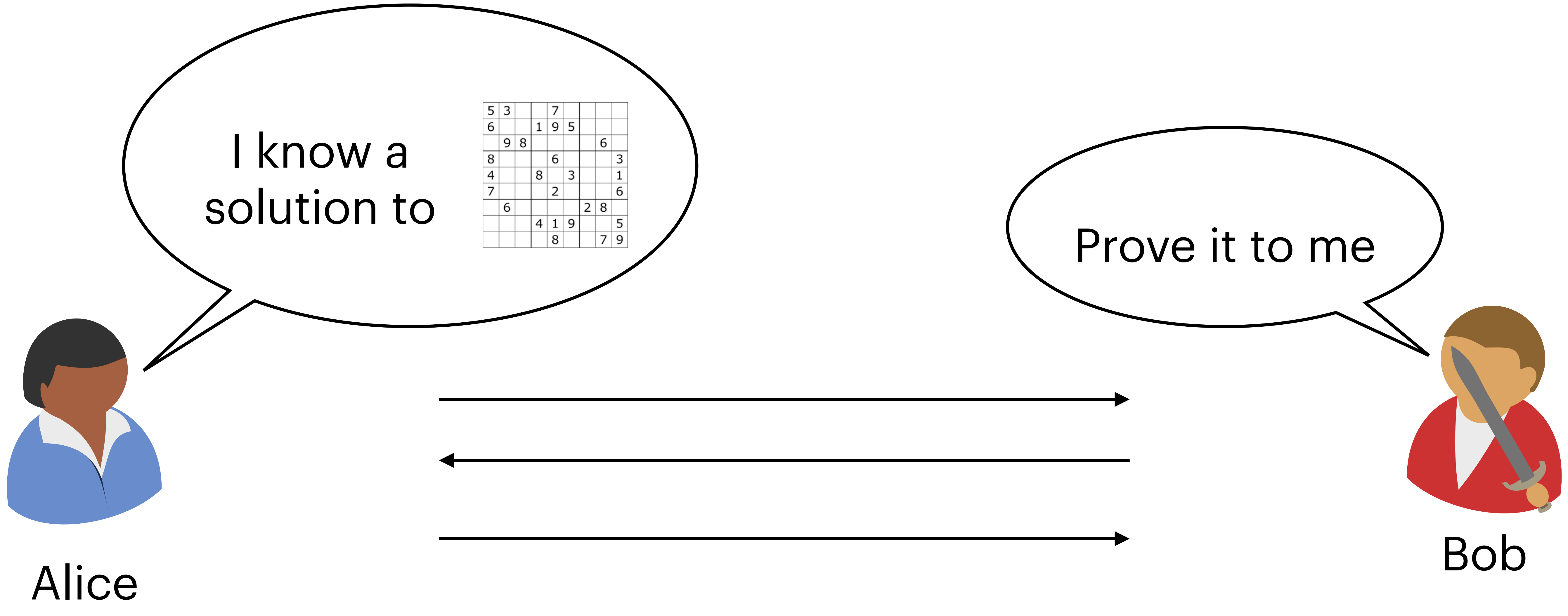
- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation



Prove that a statement is true without conveying any additional knowledge.

Topics

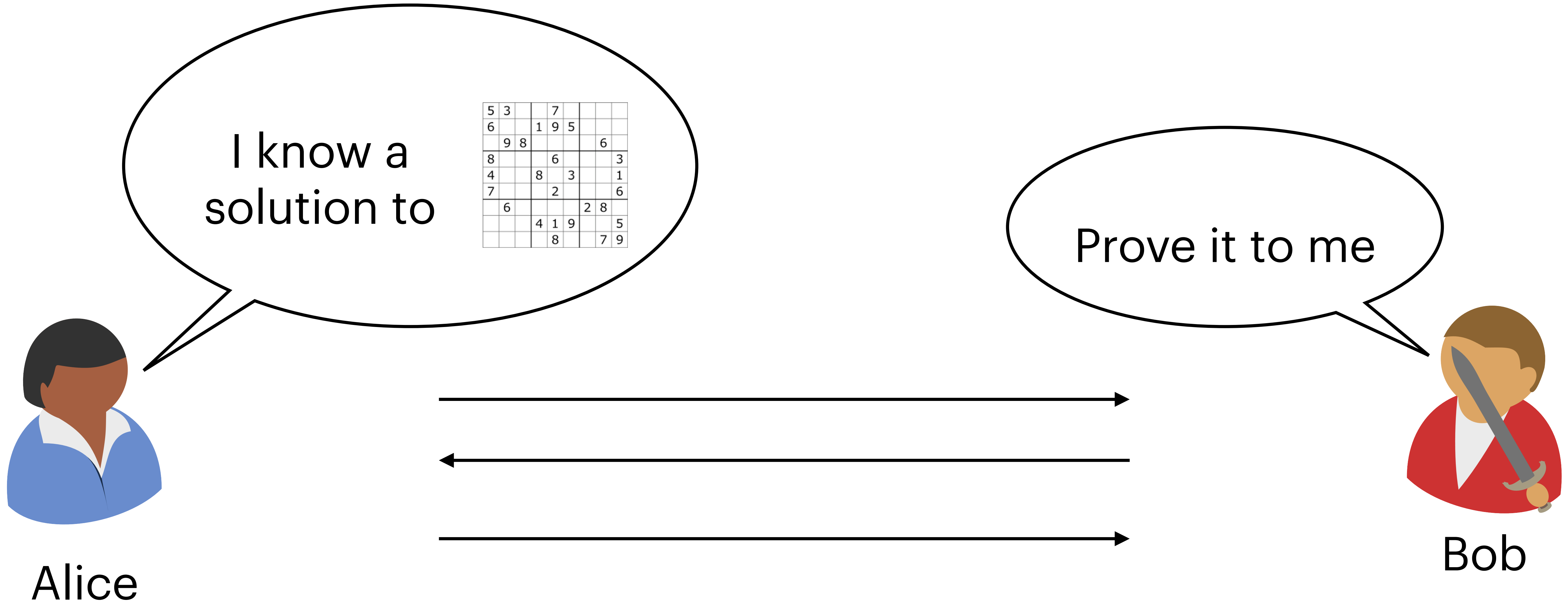
- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation



Prove that a statement is true without conveying any additional knowledge.

Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation

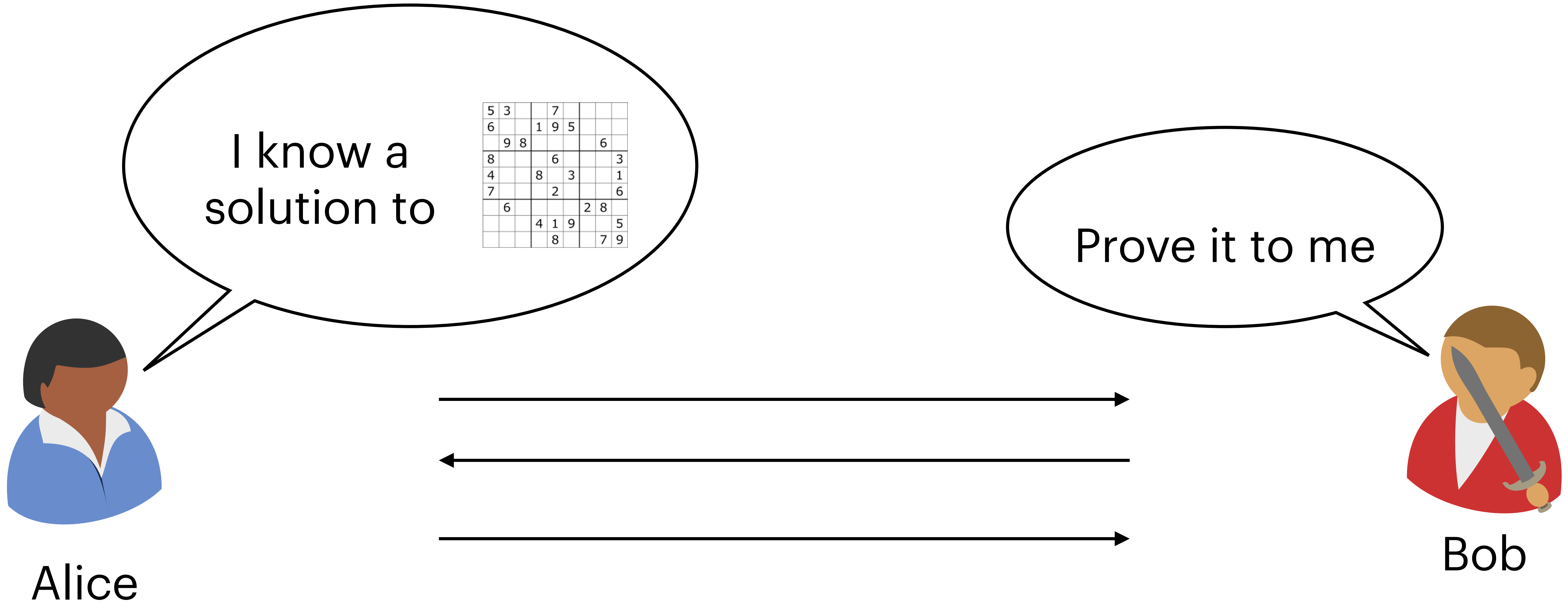


Bob is convinced that Alice has a solution

Prove that a statement is true without conveying any additional knowledge.

Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation



Bob is convinced that Alice has a solution

Bob learns nothing about the solution

Prove that a statement is true without conveying any additional knowledge.

Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation

Compute on private inputs to only learn the output.

Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation

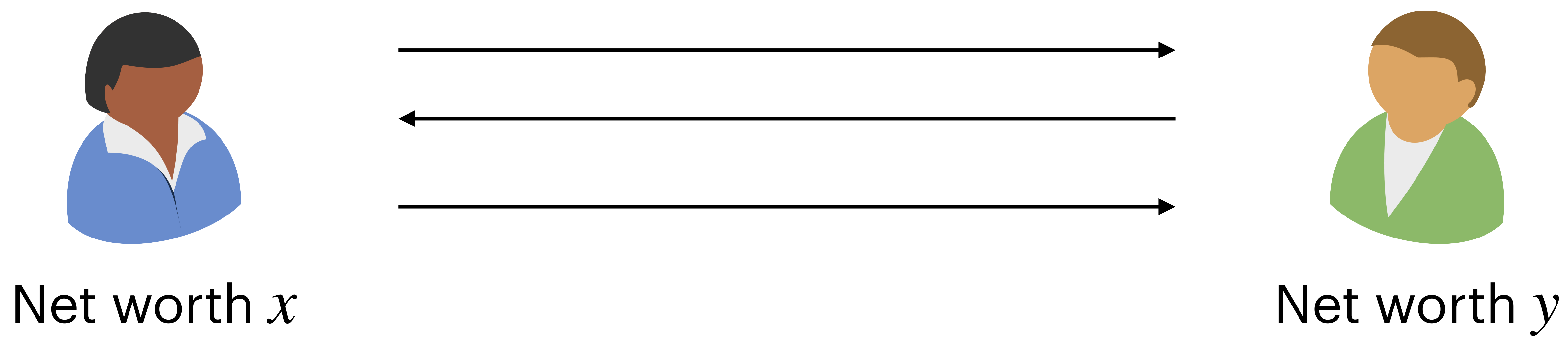
Net worth x

Net worth y

Compute on private inputs to only learn the output.

Topics

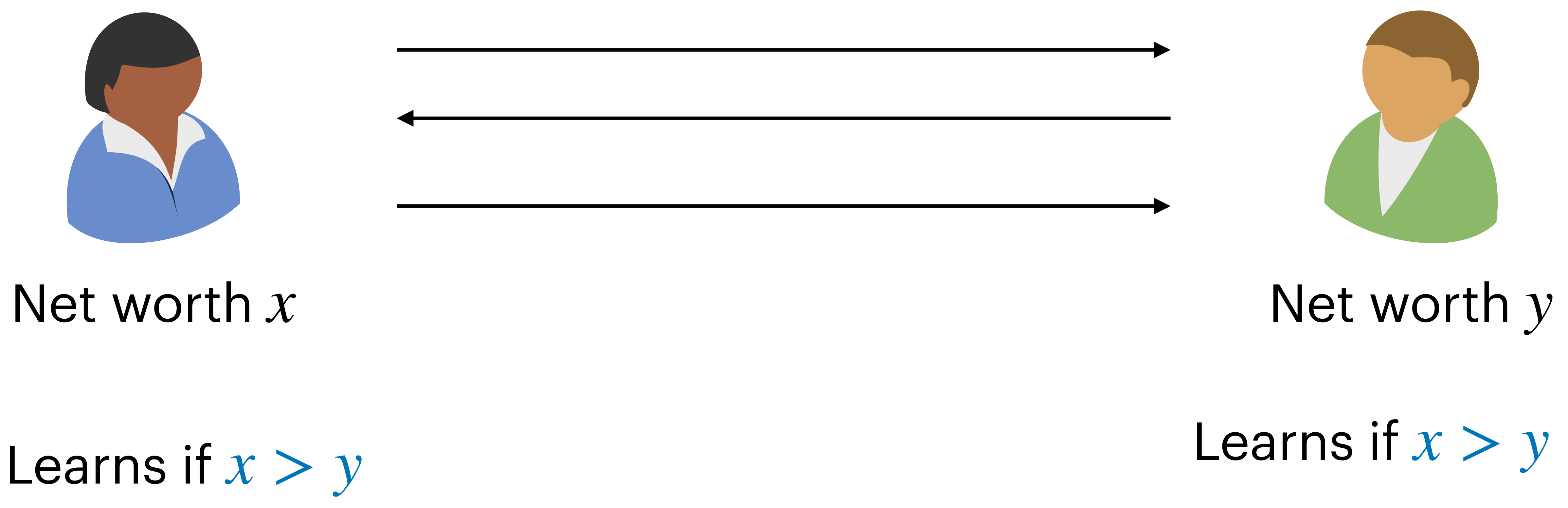
- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation



Compute on private inputs to only learn the output.

Topics

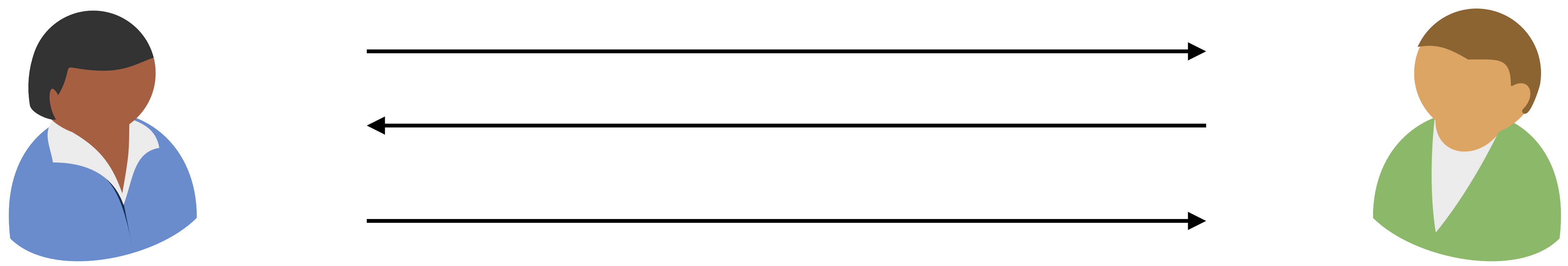
- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation



Compute on private inputs to only learn the output.

Topics

- Perfect Security
- Computational Security
- One-way Functions
- Pseudorandomness
- Symmetric-key Encryption
- Key Agreement
- Public-key Encryption
- Message Authentication Codes
- Hash Functions
- Digital Signatures
- Zero-knowledge Proofs
- Secure Computation



Net worth x

Net worth y

Learns if $x > y$

Learns nothing about y

Learns if $x > y$

Learns nothing about x

Compute on private inputs to only learn the output.

Logistics

Course Logistics

- Course website: https://adishegde.github.io/modern_crypto_sp26/
- In person classes, no Zoom or recordings
- Use Canvas for homework submission, discussion board, and announcements
- Grading:
  - 25% Homework
  - 15% Midterm 1
  - 25% Midterm 2
  - 30% Final
  - 5% Class participation

Homework

- Weekly assignments
- Submit via Canvas
- Must be typeset (use LaTeX or Typst)
- 48 “late hours”
- Okay to collaborate, list your collaborators
- No using AI on homeworks

Textbook and References

- No official textbook
- Free textbook *A Graduate Course in Applied Cryptography* is a great reference: https://toc.cryptobook.us/
- Syllabus, lecture notes, and slides will be available on the course website

Prerequisite / Background

Required reading before next class: pre-req lecture notes

https://adishegde.github.io/modern_crypto_sp26/notes/prerequisite_notes.pdf

Logic

Logic

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

Logic

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

Logic

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Logic

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

$$0 \oplus 1 = 1$$

Logic

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

$$0 \oplus 1 = 1$$

$$\begin{array}{r} 01010 \\ \oplus\ 11011 \\ \hline 10001 \end{array}$$

Logic

$x \wedge y$

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

$0 \oplus 1 = 1$

$$\begin{array}{r} 01010 \\ \oplus\ 11011 \\ \hline 10001 \end{array}$$

Logic

$x \wedge y$

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

$x \vee y$

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

$0 \oplus 1 = 1$

$$\begin{array}{r} 01010 \\ \oplus\ 11011 \\ \hline 10001 \end{array}$$

Logic

$x \wedge y$

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

$x \vee y$

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

$0 \oplus 1 = 1$

$$\neg(x \wedge y) = \neg x \vee \neg y$$

$$\begin{array}{r} 01010 \\ \oplus\ 11011 \\ \hline 10001 \end{array}$$

Logic

$x \wedge y$

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

$x \vee y$

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

$0 \oplus 1 = 1$

$$\neg(x \wedge y) = \neg x \vee \neg y$$

$$\neg(x \vee y) = \neg x \wedge \neg y$$

$$\begin{array}{r} 01010 \\ \oplus\ 11011 \\ \hline 10001 \end{array}$$

Logic: Implication

Logic: Implication

$$P \Rightarrow Q$$

Logic: Implication

$P \Rightarrow Q$ “If x = 19, then x is prime”

Logic: Implication

$$P \Rightarrow Q$$

"If x = 19, then x is prime"

Contrapositive:

$$\neg Q \Rightarrow \neg P$$

Logic: Implication

$P \Rightarrow Q$ “If *x* = 19, then *x* is prime”

Contrapositive:

$\neg Q \Rightarrow \neg P$ “If *x* is *not* prime, then *x* != 19”

Logic: Implication

$P \Rightarrow Q$ "If x = 19, then x is prime"

Contrapositive:

$\neg Q \Rightarrow \neg P$ "If x is *not* prime, then x != 19"

A statement and its contrapositive are *logically equivalent*. Often when we want to prove a statement we will prove its contrapositive.

Logic: Quantifiers

Logic: Quantifiers

Universal Quantifier

$$\forall x \in A, P(x)$$

Logic: Quantifiers

Universal Quantifier

$\forall x \in A$

$$\forall x \in A, P(x)$$

"For all integers x

Logic: Quantifiers

Universal Quantifier

$\forall x \in A$ $P(x)$

$$\forall x \in A, P(x)$$

"For all integers x, $x > 0$"

Logic: Quantifiers

Universal Quantifier

$\forall x \in A$ $P(x)$

$$\forall x \in A, P(x)$$

"For all integers x, $x > 0$"

Existential Quantifier

$$\exists x \in A, P(x)$$

Logic: Quantifiers

Universal Quantifier

$\forall x \in A$ $P(x)$

$$\forall x \in A, P(x)$$

"For all integers x, $x > 0$"

Existential Quantifier

$\exists x \in A$

$$\exists x \in A, P(x)$$

"There exists an integer x such that

Logic: Quantifiers

Universal Quantifier

$\forall x \in A$ $P(x)$

$$\forall x \in A, P(x)$$

"For all integers x, $x > 0$"

Existential Quantifier

$\exists x \in A$ $P(x)$

$$\exists x \in A, P(x)$$

"There exists an integer x such that $x > 0$"

Logic: Nesting Quantifiers

Logic: Nesting Quantifiers

$\forall x \in A, \exists y \in A, P(x, y)$

Logic: Nesting Quantifiers

$\forall x \in A$

$$\forall x \in A, \exists y \in A, P(x, y)$$

"For all integers *x*

Logic: Nesting Quantifiers

$\forall x \in A$ $\exists y \in A$

$$\forall x \in A, \exists y \in A, P(x, y)$$

"For all integers x , there exists an integer y such that

Logic: Nesting Quantifiers

$\forall x \in A$ $\exists y \in A$

$\forall x \in A, \exists y \in A, P(x, y)$ "For all integers x, there exists an integer y such that $x + y = 0$" $P(x, y)$

Logic: Nesting Quantifiers

$\forall x \in A$

$\exists y \in A$

$$\forall x \in A, \exists y \in A, P(x, y)$$

"For all integers *x*, there exists an integer *y* such that *x* + *y* = 0"

$P(x, y)$

$$\exists y \in A, \forall x \in A, P(x, y)$$

Logic: Nesting Quantifiers

$\forall x \in A$ $\exists y \in A$

$\forall x \in A, \exists y \in A, P(x, y)$ "For all integers *x*, there exists an integer *y* such that

x + *y* = 0" $P(x, y)$

$\exists y \in A$

$\exists y \in A, \forall x \in A, P(x, y)$ "There exists an integer *y* such that

Logic: Nesting Quantifiers

$\forall x \in A$ $\exists y \in A$

$\forall x \in A, \exists y \in A, P(x, y)$ "For all integers x, there exists an integer y such that $x + y = 0$" $P(x, y)$

$\exists y \in A$ $\forall x \in A$

$\exists y \in A, \forall x \in A, P(x, y)$ "There exists an integer y such that for all integers x

Logic: Nesting Quantifiers

$\forall x \in A$ $\exists y \in A$

$\forall x \in A, \exists y \in A, P(x, y)$ "For all integers *x*, there exists an integer *y* such that *x* + *y* = 0" $P(x, y)$

$\exists y \in A$ $\forall x \in A$

$\exists y \in A, \forall x \in A, P(x, y)$ "There exists an integer *y* such that for all integers *x* *x* + *y* = 0" $P(x, y)$

Logic: Nesting Quantifiers

$\forall x \in A$ $\exists y \in A$

$$\forall x \in A, \exists y \in A, P(x, y)$$

“For all integers *x*, there exists an integer *y* such that *x* + *y* = 0” $P(x, y)$

Order of quantifiers really matters!

$\exists y \in A$ $\forall x \in A$

$$\exists y \in A, \forall x \in A, P(x, y)$$

“There exists an integer *y* such that for all integers *x* *x* + *y* = 0” $P(x, y)$

Logic: Negating Quantifiers

Logic: Negating Quantifiers

$\forall x \in A$	$P(x)$
"For all integers x	$x > 0$"

$$\forall x \in A, P(x)$$

Logic: Negating Quantifiers

$\forall x \in A$	$P(x)$
"For all integers x	$x > 0$"

$$\forall x \in A, P(x)$$

$$\neg(\forall x \in A, P(x))$$

Logic: Negating Quantifiers

$\forall x \in A$	$P(x)$
"For all integers x	$x > 0$"

$$\forall x \in A, P(x)$$

$$\neg(\forall x \in A, P(x))$$

$$\exists x \in A, \neg P(x)$$

Logic: Negating Quantifiers

$\forall x \in A$ $P(x)$

$\forall x \in A, P(x)$ "For all integers x $x > 0$"

$\neg(\forall x \in A, P(x))$

$\exists x \in A$

$\exists x \in A, \neg P(x)$ "There exists an integer x such that

Logic: Negating Quantifiers

$\forall x \in A$ $P(x)$

$$\forall x \in A, P(x)$$

"For all integers x $x > 0$"

$$\neg(\forall x \in A, P(x))$$

$\exists x \in A$ $\neg P(x)$

$$\exists x \in A, \neg P(x)$$

"There exists an integer x such that $x < 0$

Logic: Negating Nested Quantifiers

Logic: Negating Nested Quantifiers

$$\forall x \in A, \exists y \in B, \exists z \in C, P(x)$$

Logic: Negating Nested Quantifiers

$$\forall x \in A, \exists y \in B, \exists z \in C, P(x)$$

$$\neg(\forall x \in A, \exists y \in B, \exists z \in C, P(x))$$

Logic: Negating Nested Quantifiers

$$\forall x \in A, \exists y \in B, \exists z \in C, P(x)$$

$$\neg(\forall x \in A, \exists y \in B, \exists z \in C, P(x))$$

Negate each quantifier in turn

Logic: Negating Nested Quantifiers

$$\forall x \in A, \exists y \in B, \exists z \in C, P(x)$$

$$\neg(\forall x \in A, \exists y \in B, \exists z \in C, P(x))$$

Negate each quantifier in turn

$$\exists x \in A, \forall y \in B, \forall z \in C, \neg P(x)$$

Logic: Putting it all Together

Logic: Putting it all Together

$$\forall x, P(x) \wedge \forall y, P(y) \Rightarrow \forall z, Q(Z)$$

Logic: Putting it all Together

$$\forall x, P(x) \land \forall y, P(y) \Rightarrow \forall z, Q(Z)$$

$$\exists z, \neg Q(z) \Rightarrow \exists x, \neg P(X) \lor \exists y, \neg P(y)$$

Probability: Distributions

Probability: Distributions

Sample Space: the possible outcomes of a probabilistic experiment

Probability: Distributions

Sample Space: the possible outcomes of a probabilistic experiment

Example: All binary strings of length 3 (000, 001, 010,…)

Probability: Distributions

Sample Space: the possible outcomes of a probabilistic experiment

Example: All binary strings of length 3 (000, 001, 010,…)

Distribution: A *distribution* over a sample space assigns a probability to every element of the space such that the sum of the probabilities is *1*.

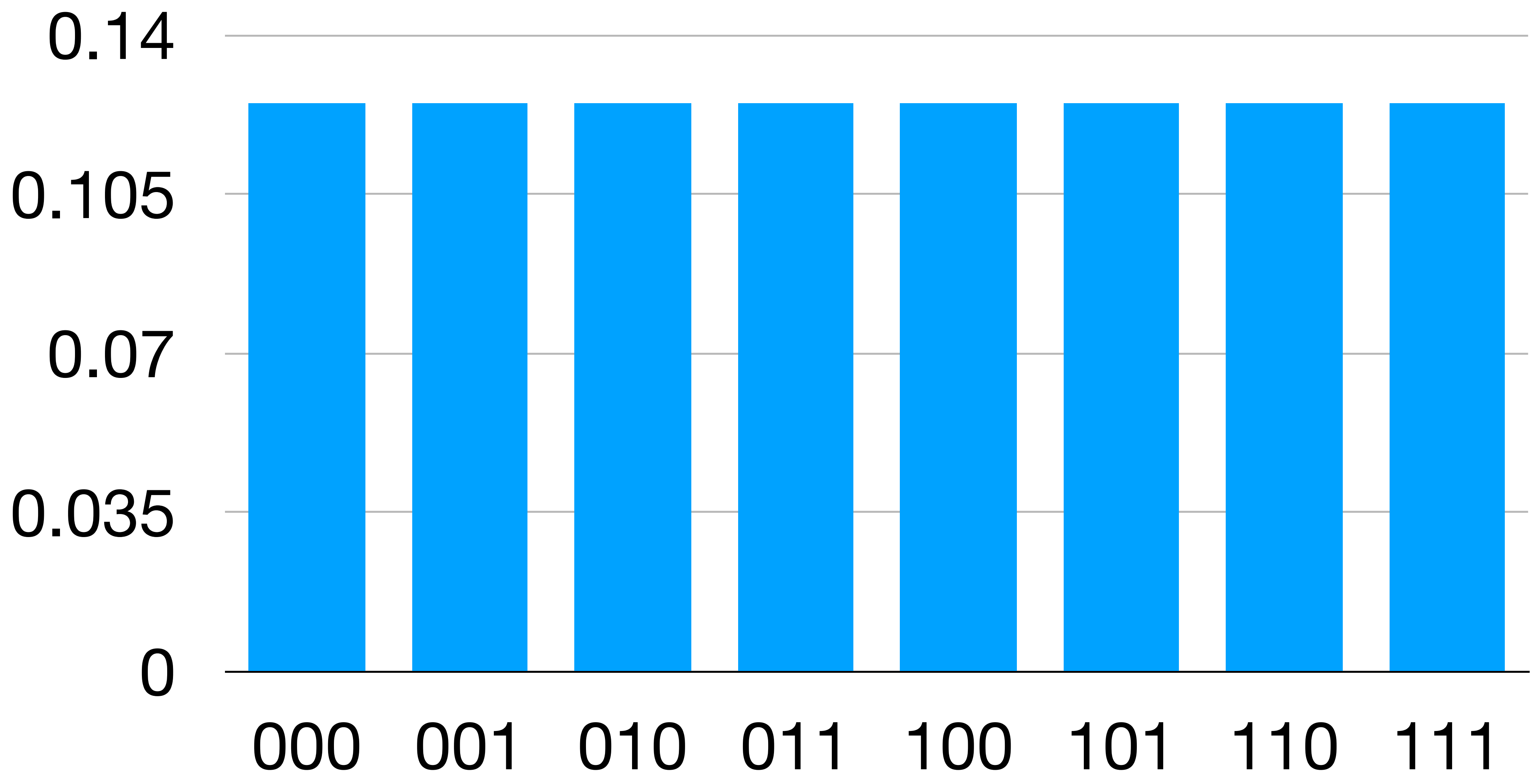
Probability: Distributions

Sample Space: the possible outcomes of a probabilistic experiment

Example: All binary strings of length 3 (000, 001, 010,…)

Distribution: A *distribution* over a sample space assigns a probability to every element of the space such that the sum of the probabilities is *1*.

Example: the uniform distribution (all probabilities equal)



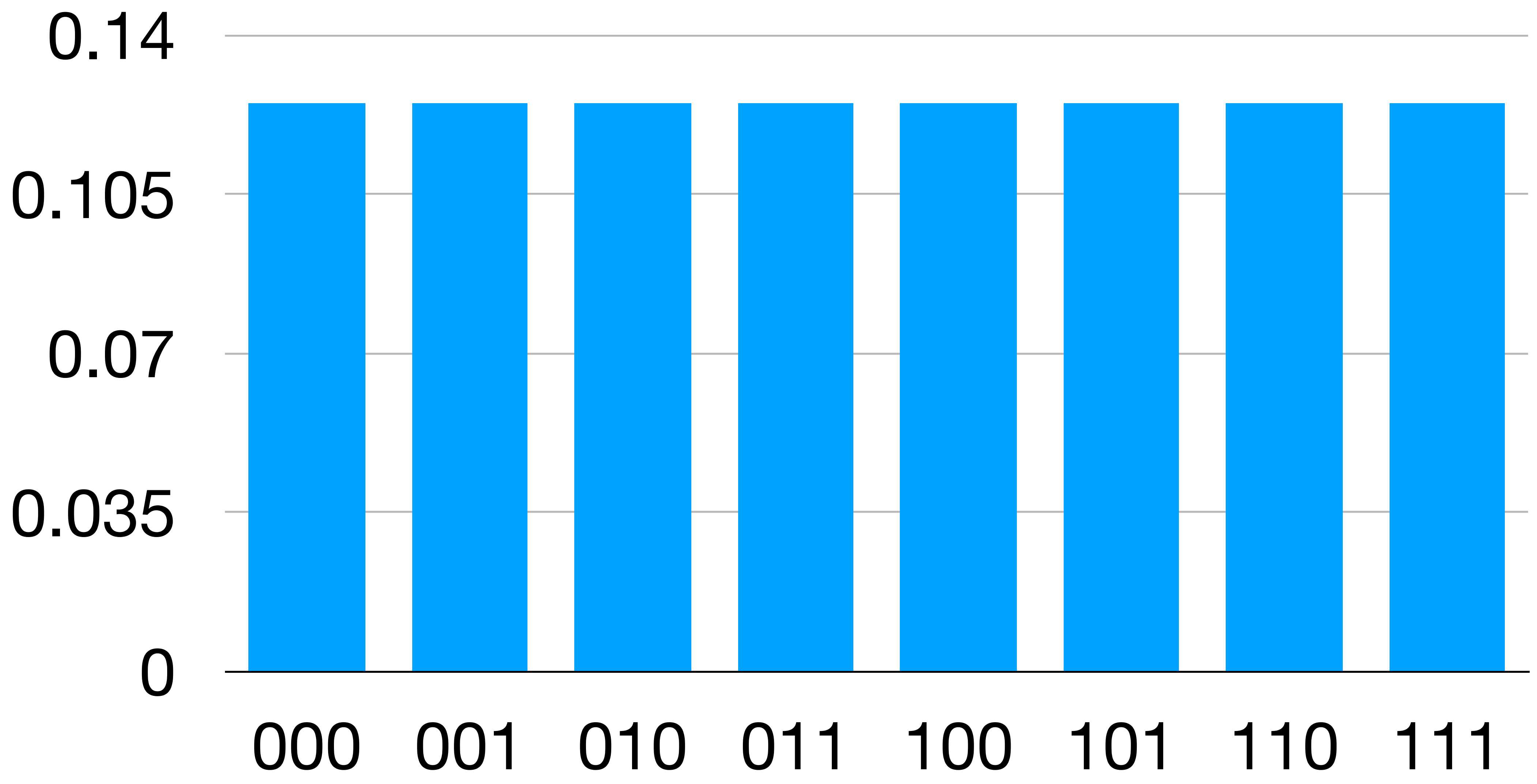
Probability: Distributions

Sample Space: the possible outcomes of a probabilistic experiment

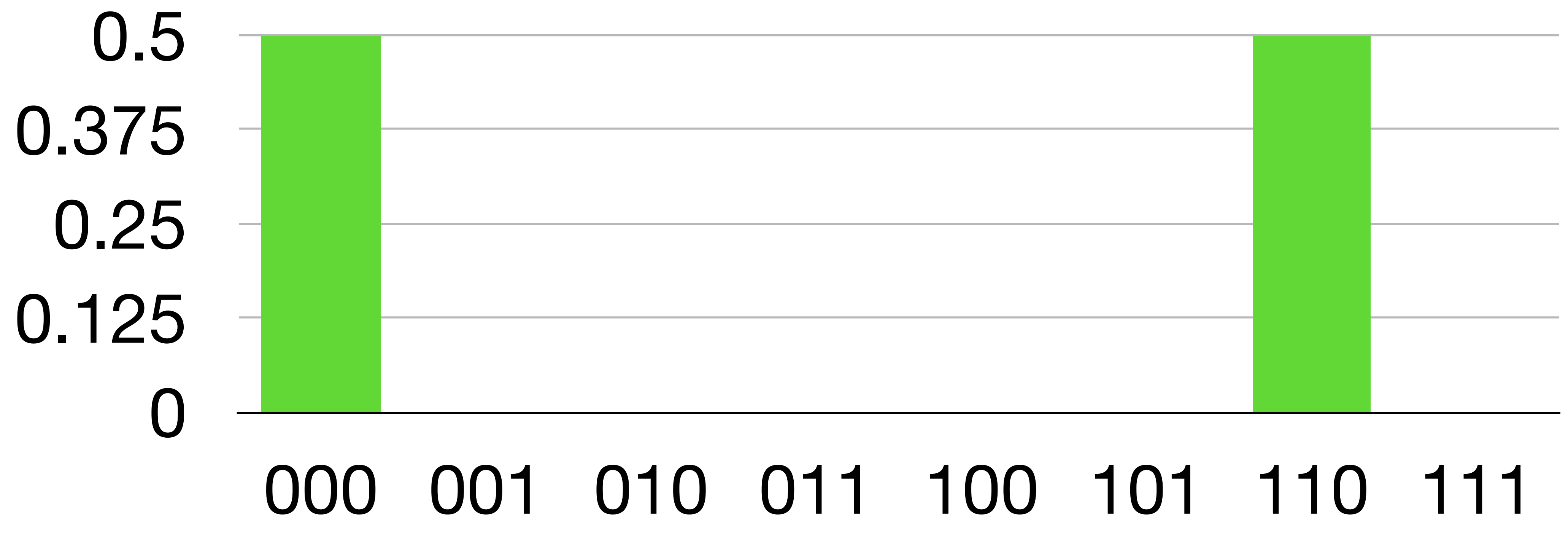
Example: All binary strings of length 3 (000, 001, 010,…)

Distribution: A *distribution* over a sample space assigns a probability to every element of the space such that the sum of the probabilities is *1*.

Example: the uniform distribution (all probabilities equal)



Example: some other distribution



Probability: Distributions

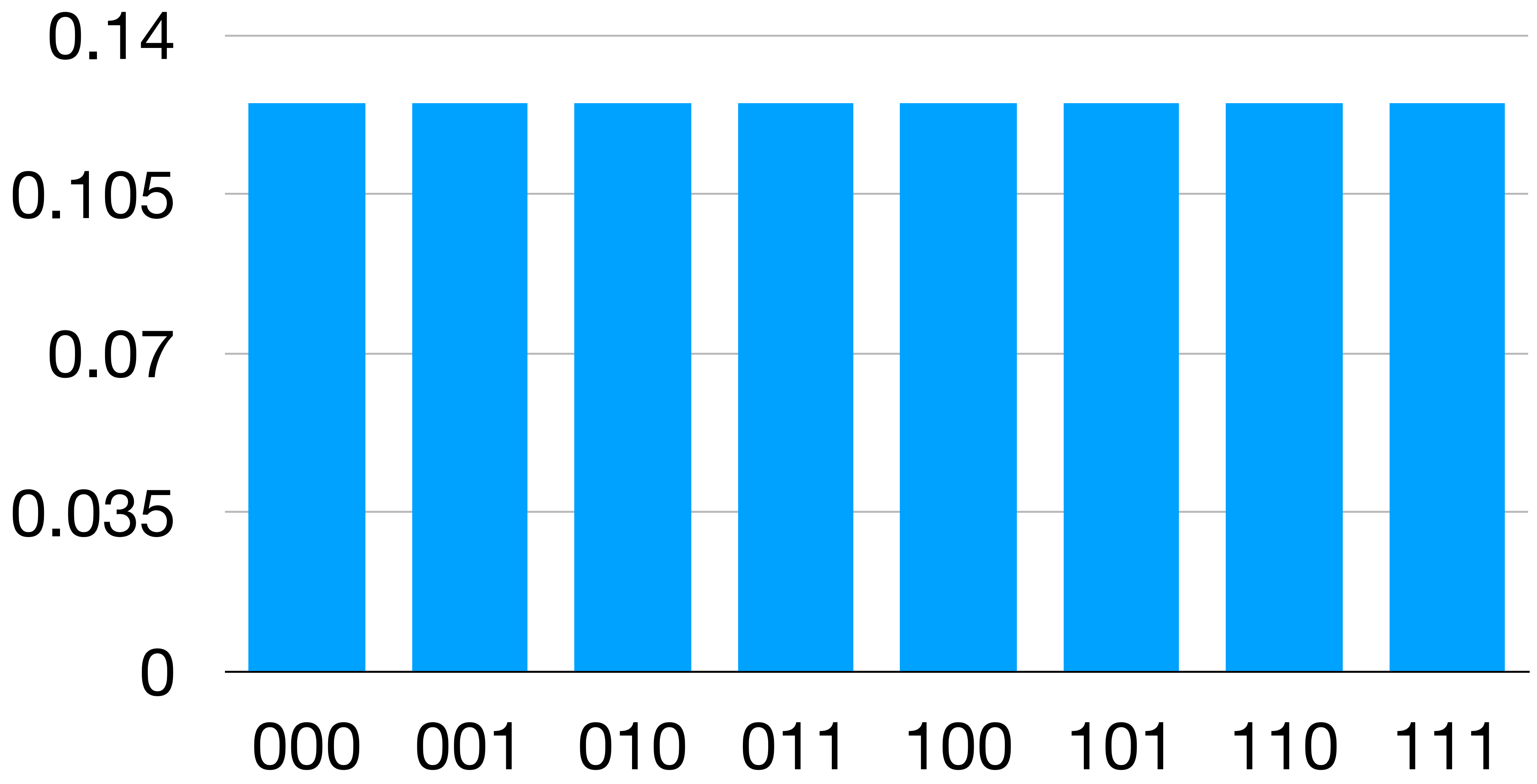
Sampling from a distribution means selecting an element in accordance with the assigned probabilities

Sample Space: the possible outcomes of a probabilistic experiment

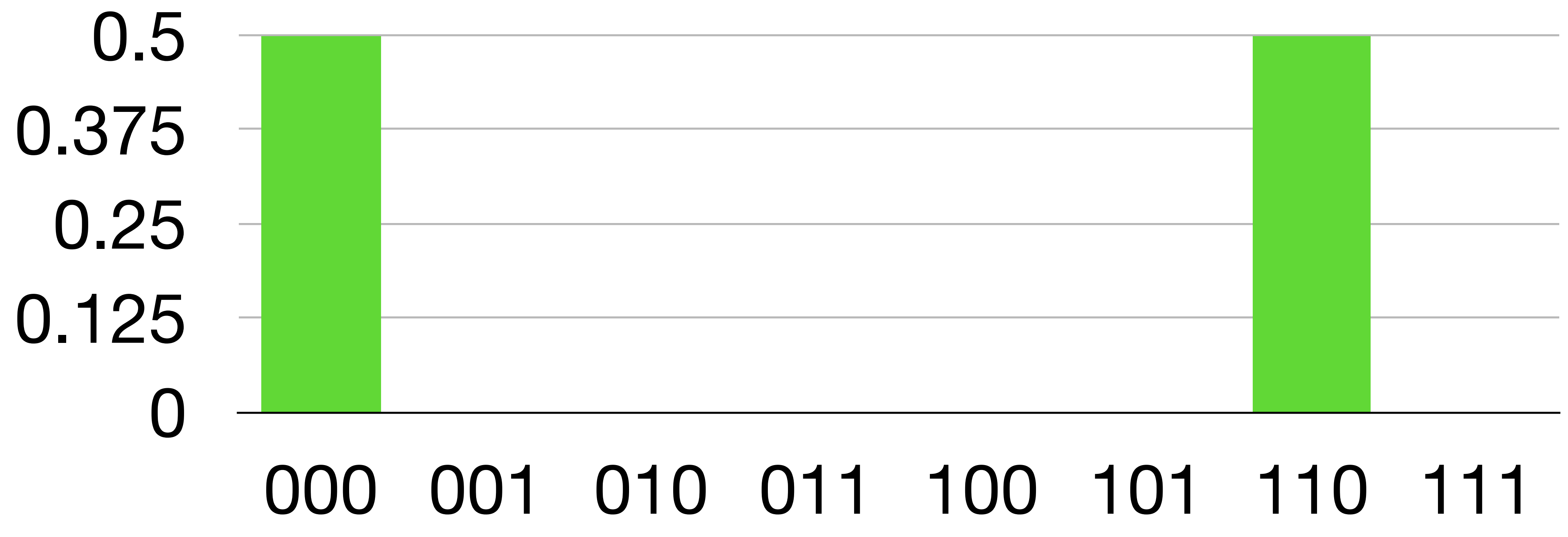
Example: All binary strings of length 3 (000, 001, 010,…)

Distribution: A *distribution* over a sample space assigns a probability to every element of the space such that the sum of the probabilities is *1*.

Example: the uniform distribution (all probabilities equal)



Example: some other distribution



Turing Machines

Turing Machines

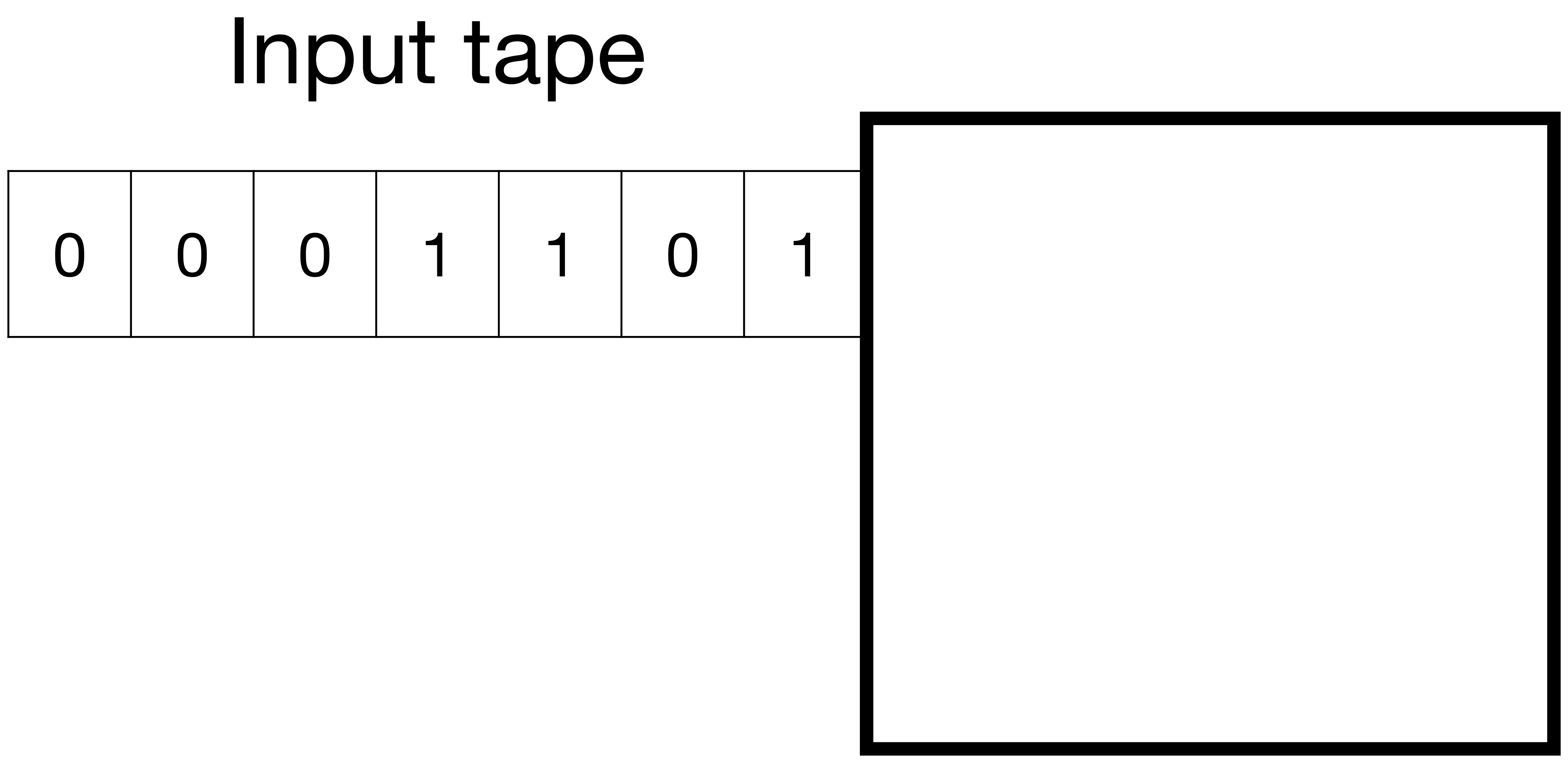
To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.

Turing Machines

To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.

Turing Machines

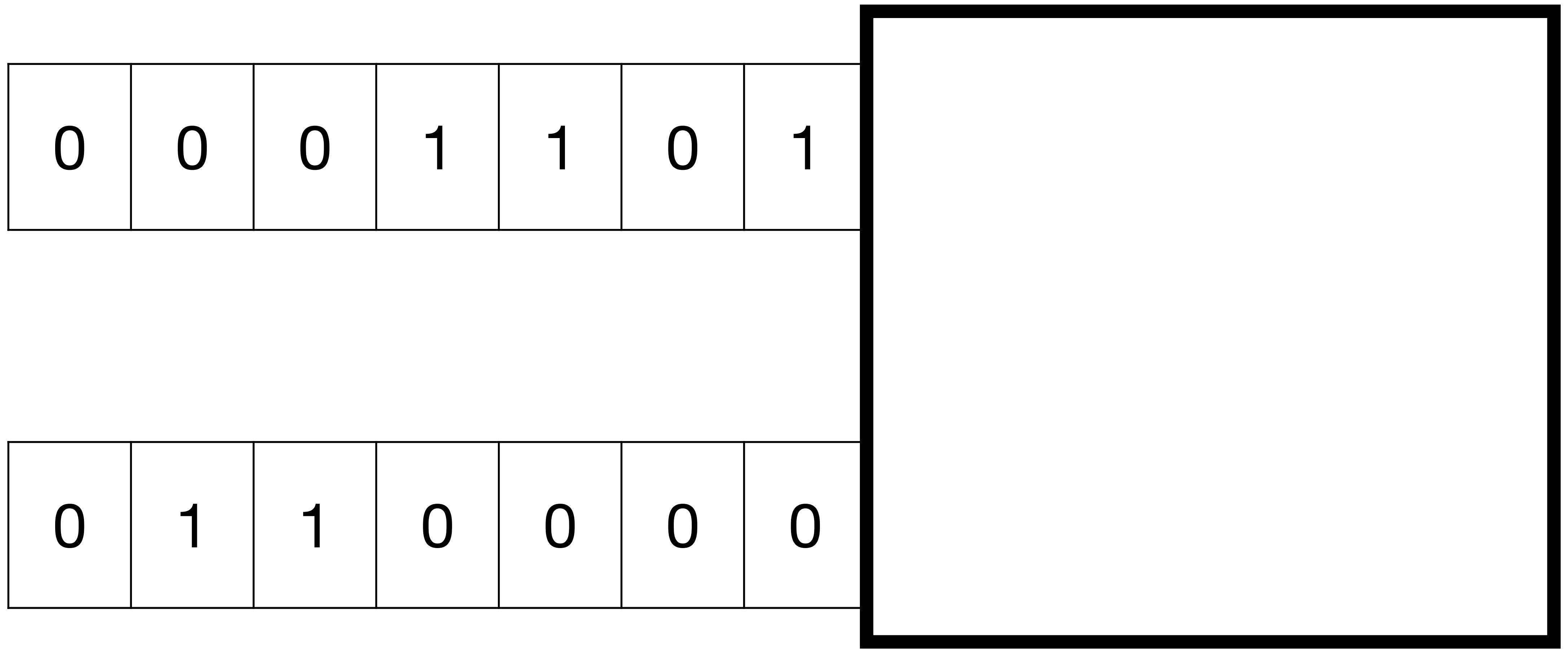
To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.



Turing Machines

To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.

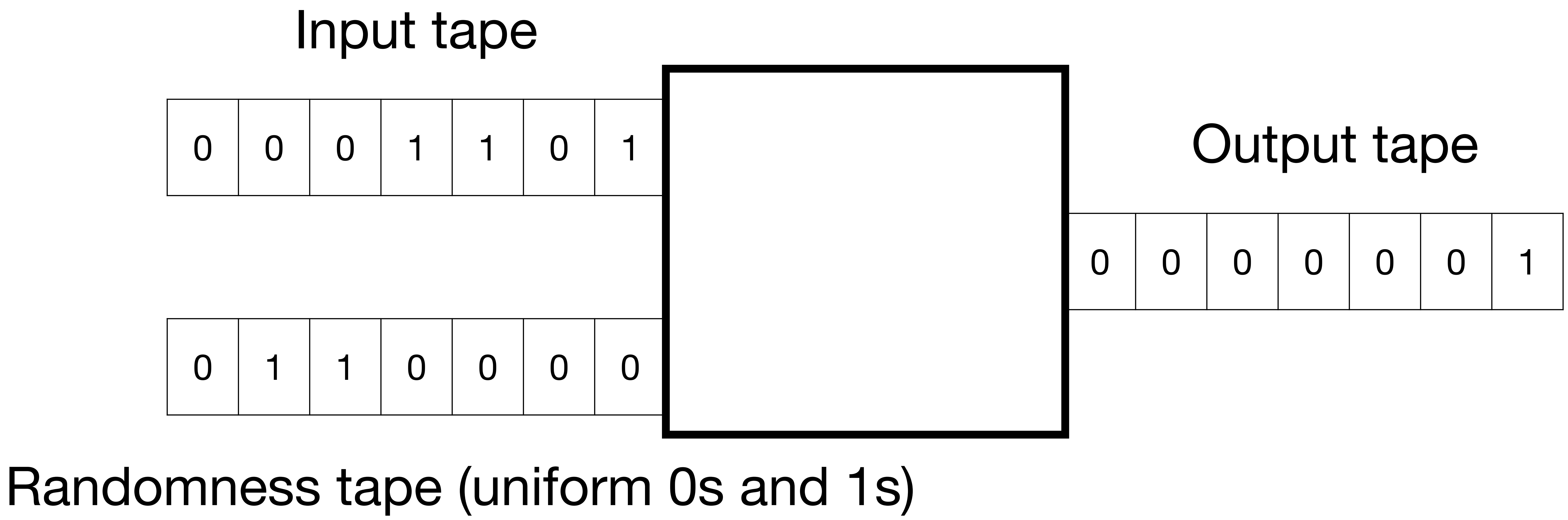
Input tape



Randomness tape (uniform 0s and 1s)

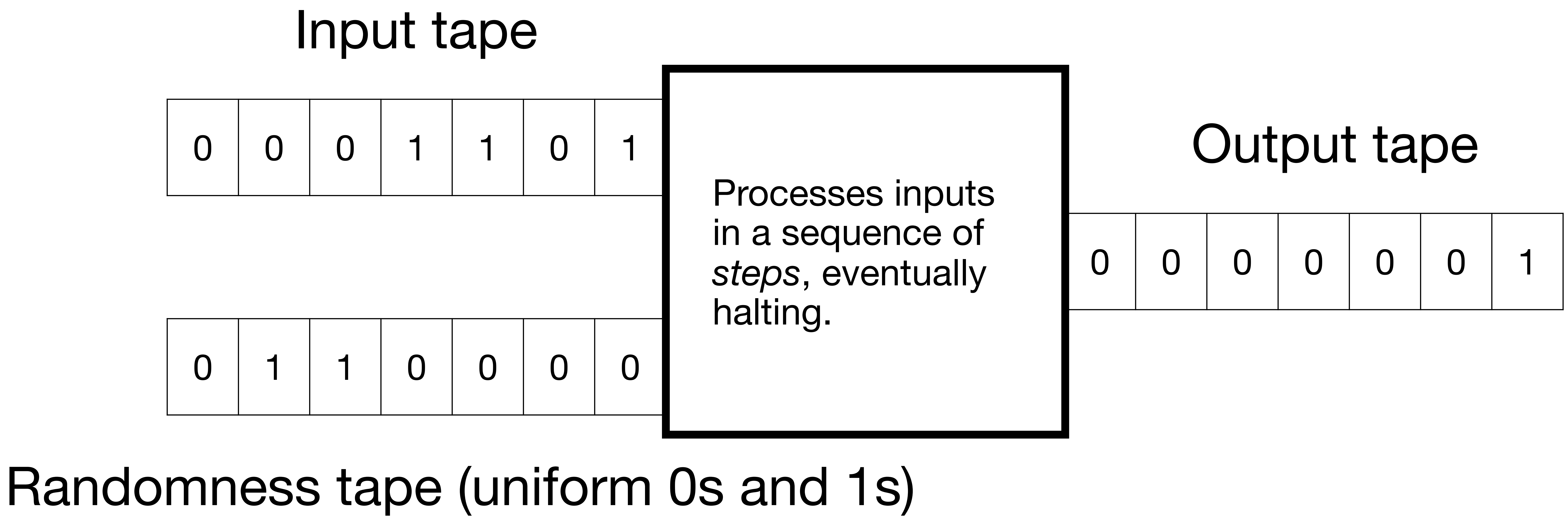
Turing Machines

To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.



Turing Machines

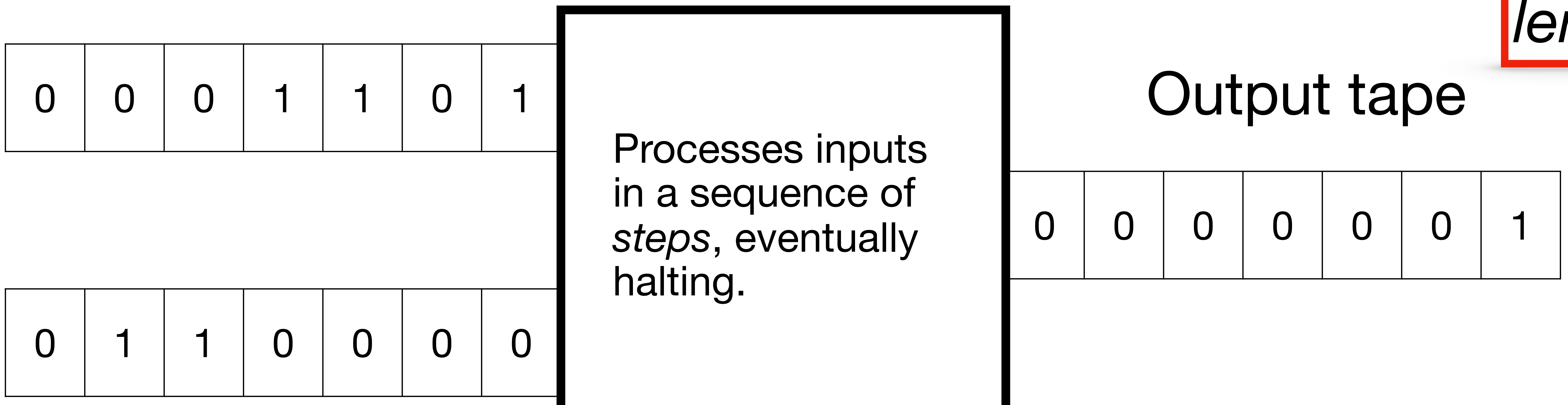
To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.



Turing Machines

To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.

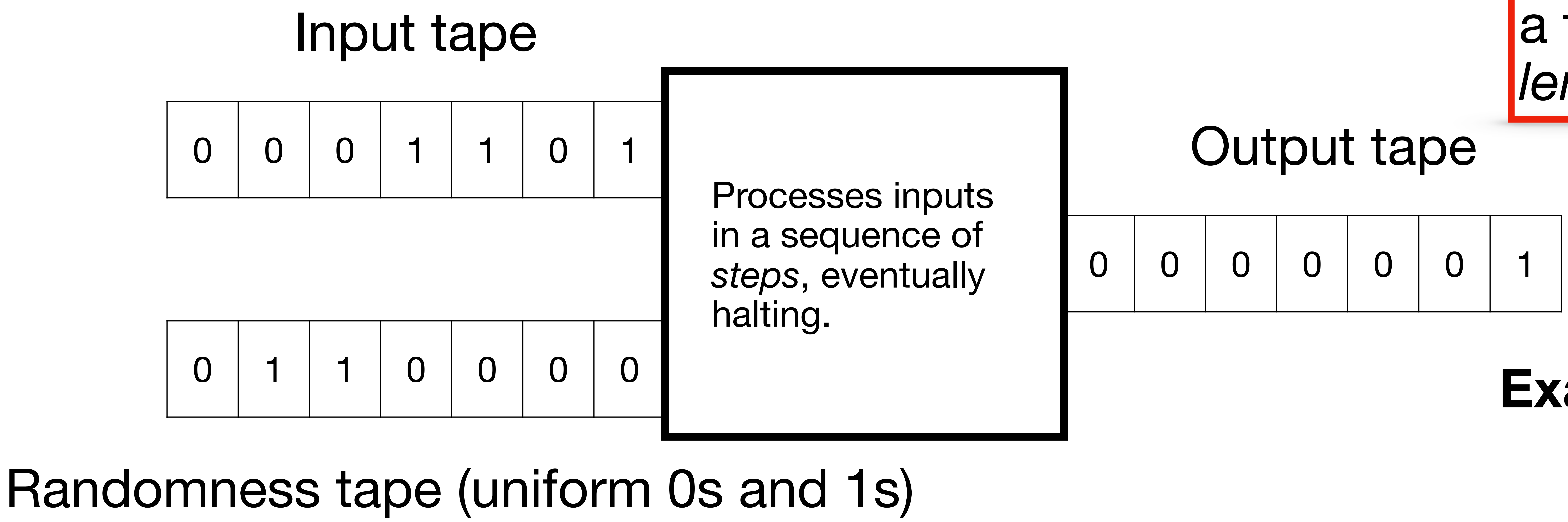
What we care about: The maximum number of *steps* the machine takes before halting as a function of the *input length.*



Turing Machines

To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.

What we care about: The maximum number of *steps* the machine takes before halting as a function of the *input length.*

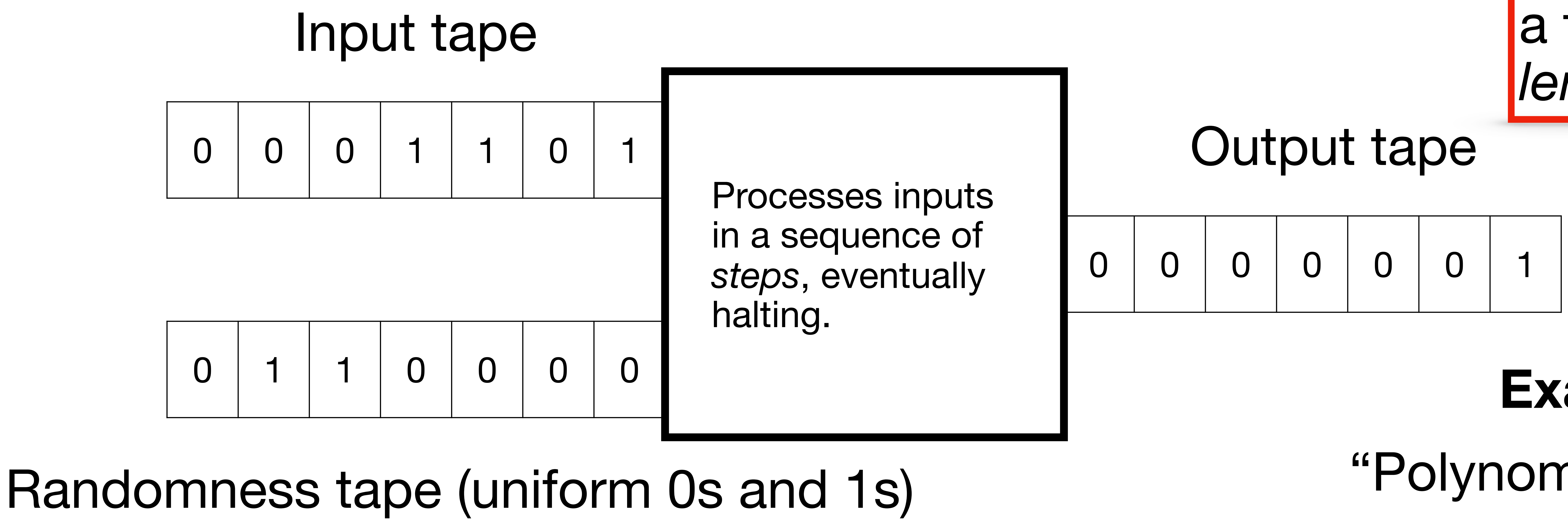


Example: $T(x) = x^5$

Turing Machines

To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.

What we care about: The maximum number of *steps* the machine takes before halting as a function of the *input length.*



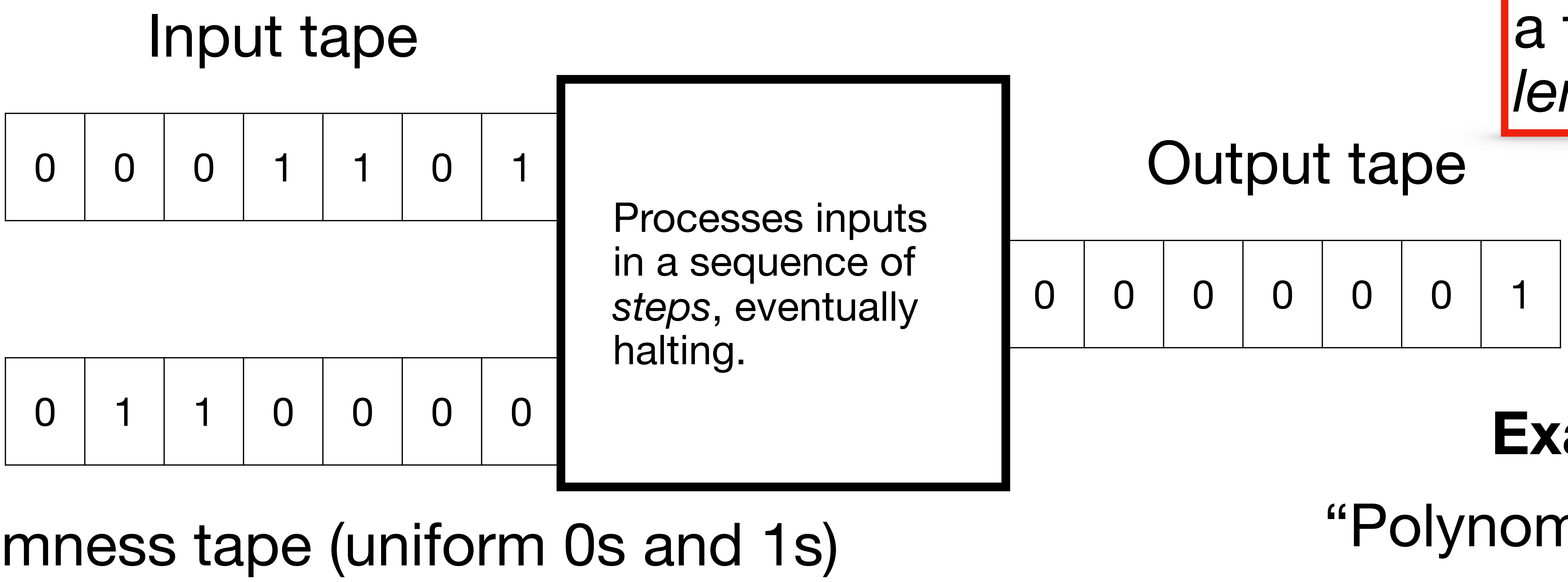
Example: $T(x) = x^5$

"Polynomial runtime": efficient!

Turing Machines

To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.

What we care about: The maximum number of *steps* the machine takes before halting as a function of the *input length.*



Example: $T(x) = x^5$

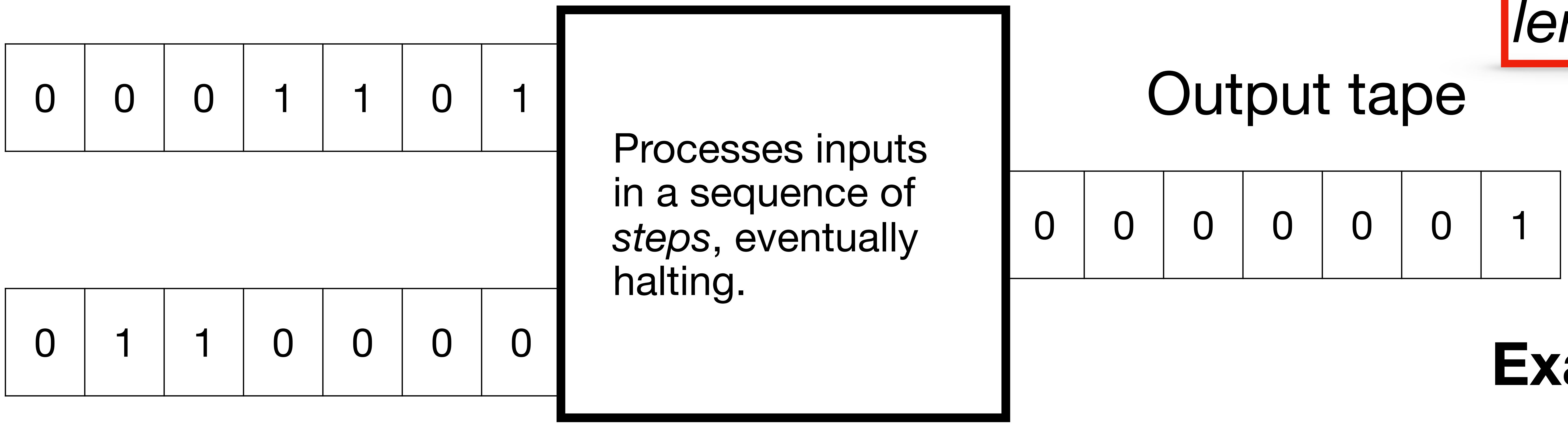
"Polynomial runtime": efficient!

Example: $T(x) = 2^x$

Turing Machines

To reason formally about computation we need to have a formal definition of it. We will use the Probabilistic Turing Machine model.

What we care about: The maximum number of *steps* the machine takes before halting as a function of the *input length.*



Example: $T(x) = x^5$

“Polynomial runtime”: efficient!

PPT = “Probabilistic Polynomial Time”

Example: $T(x) = 2^x$

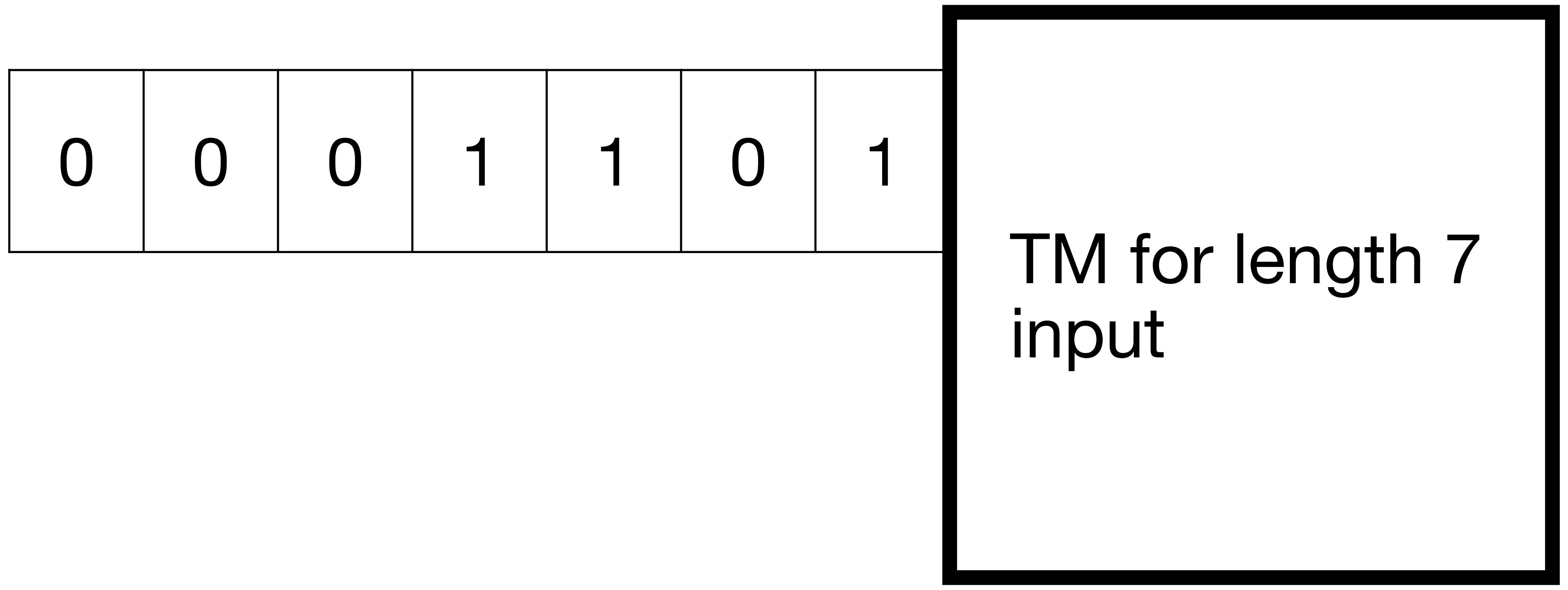
Non-uniform Turing Machines

Non-uniform Turing Machines

We can make an algorithm “more powerful” by letting it be *completely different* for every input length

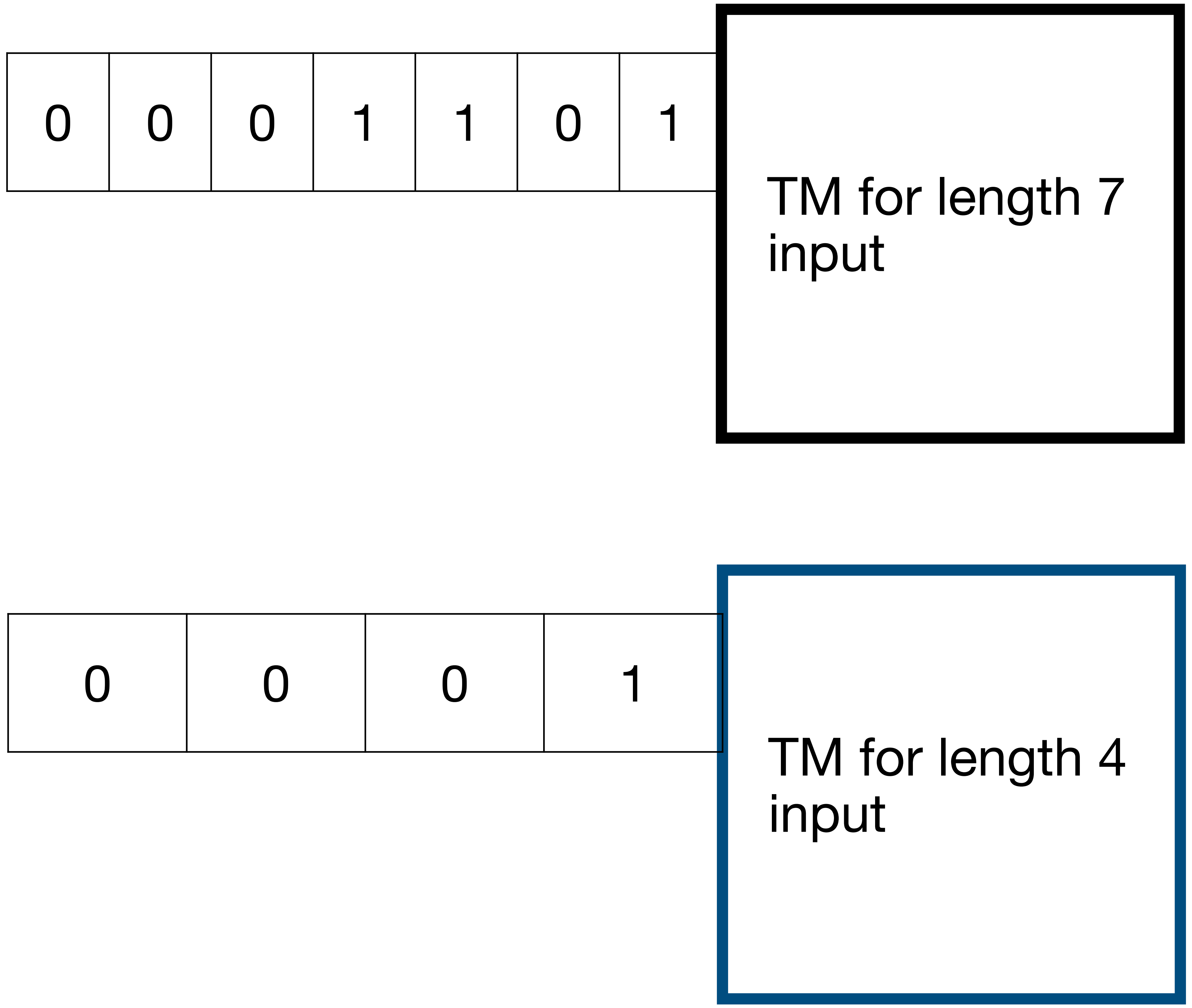
Non-uniform Turing Machines

We can make an algorithm “more powerful” by letting it be *completely different* for every input length



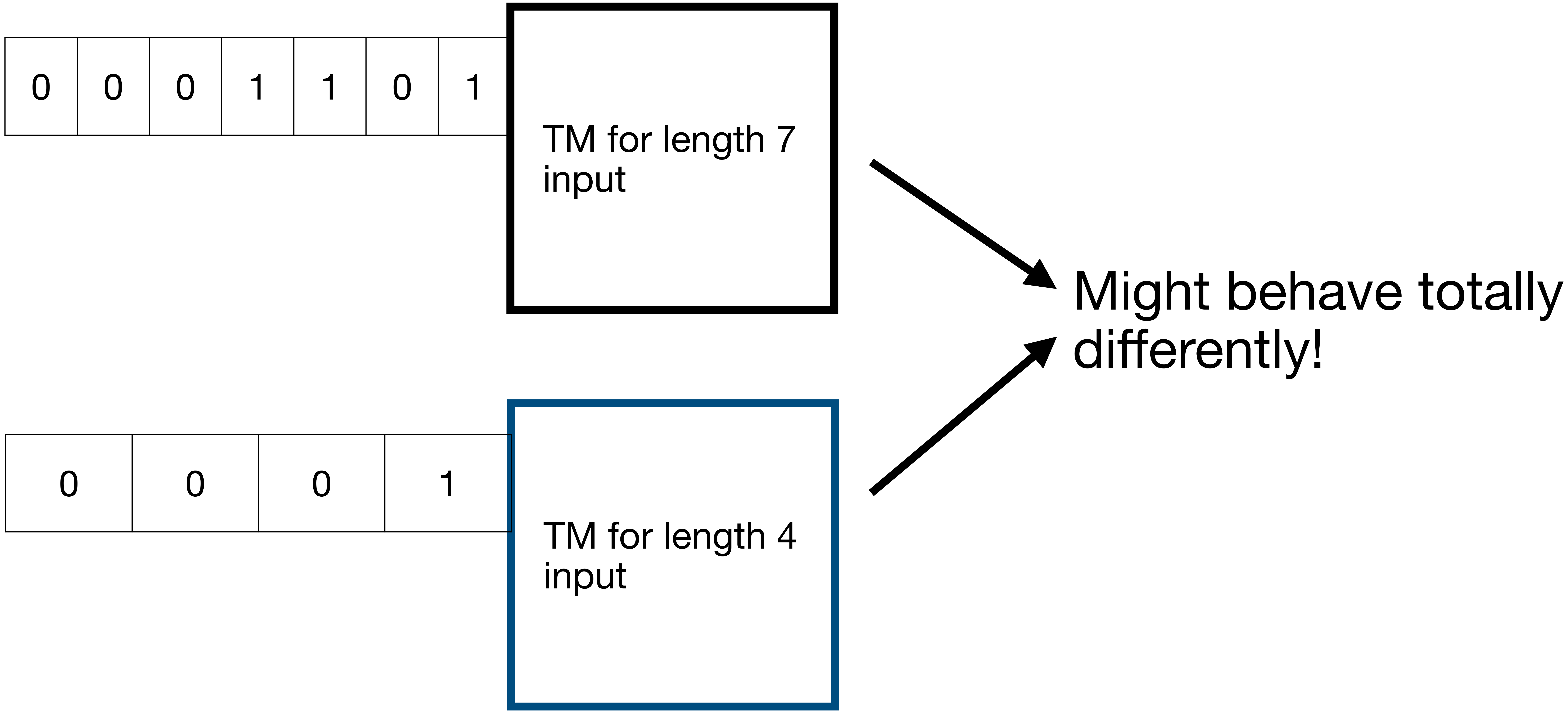
Non-uniform Turing Machines

We can make an algorithm "more powerful" by letting it be *completely different* for every input length



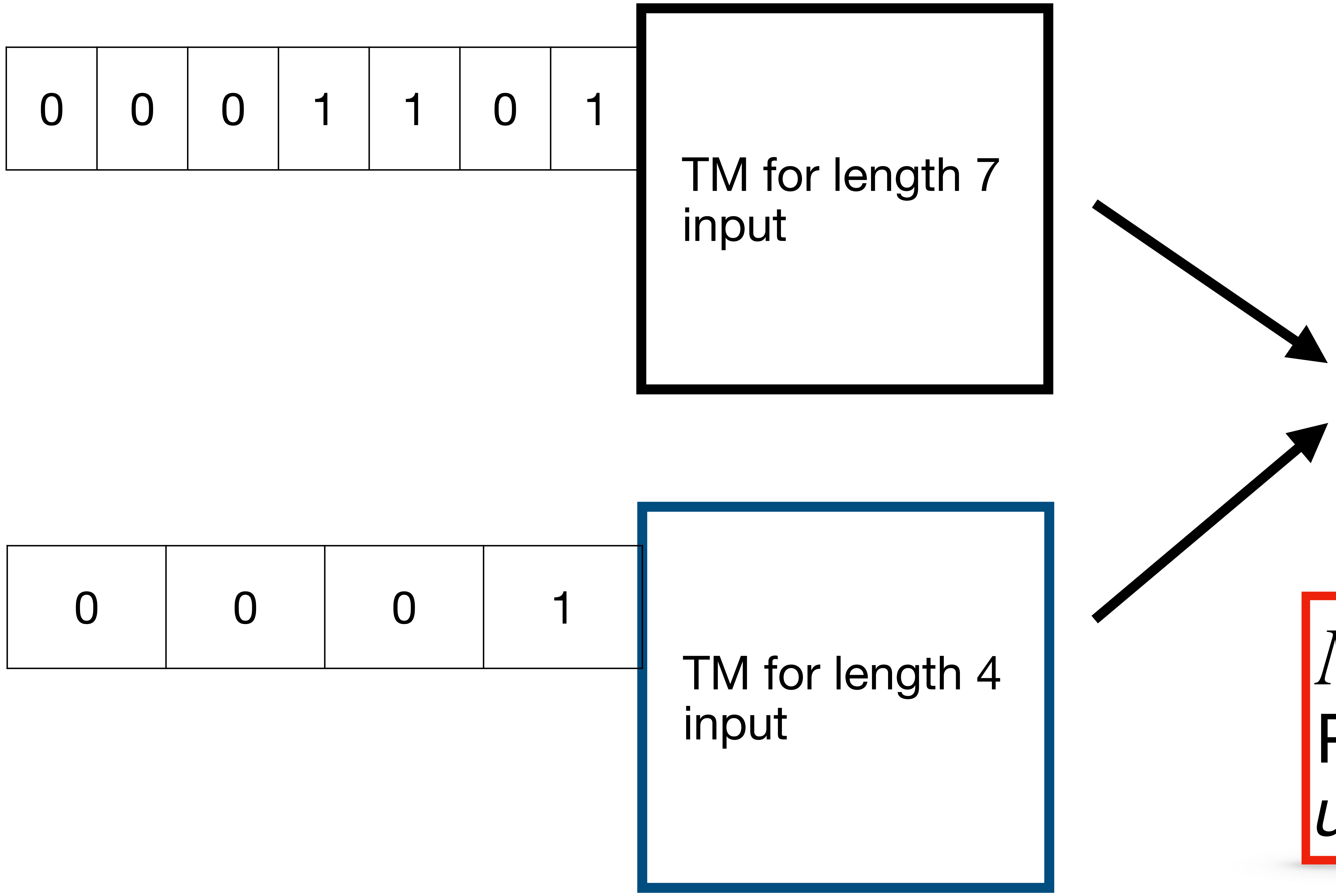
Non-uniform Turing Machines

We can make an algorithm "more powerful" by letting it be *completely different* for every input length



Non-uniform Turing Machines

We can make an algorithm "more powerful" by letting it be *completely different* for every input length



Might behave totally differently!

$M = \{M_1, M_2, \dots\}$ where each M_1 is a PPT Turing Machine is called a *Non-uniform PPT Turing Machine* (NUPPT)

Asymptotic Notation

Asymptotic Notation

- Helps us answer the question: “how efficient is your algorithm”

Asymptotic Notation

- Helps us answer the question: “how efficient is your algorithm”
- We care about how the *runtime* (number of steps) scales as a function of the *input length*

Asymptotic Notation

- Helps us answer the question: “how efficient is your algorithm”
- We care about how the *runtime* (number of steps) scales as a function of the *input length*
- Specifically, we care about the *limit* of this function: what does it trend towards?

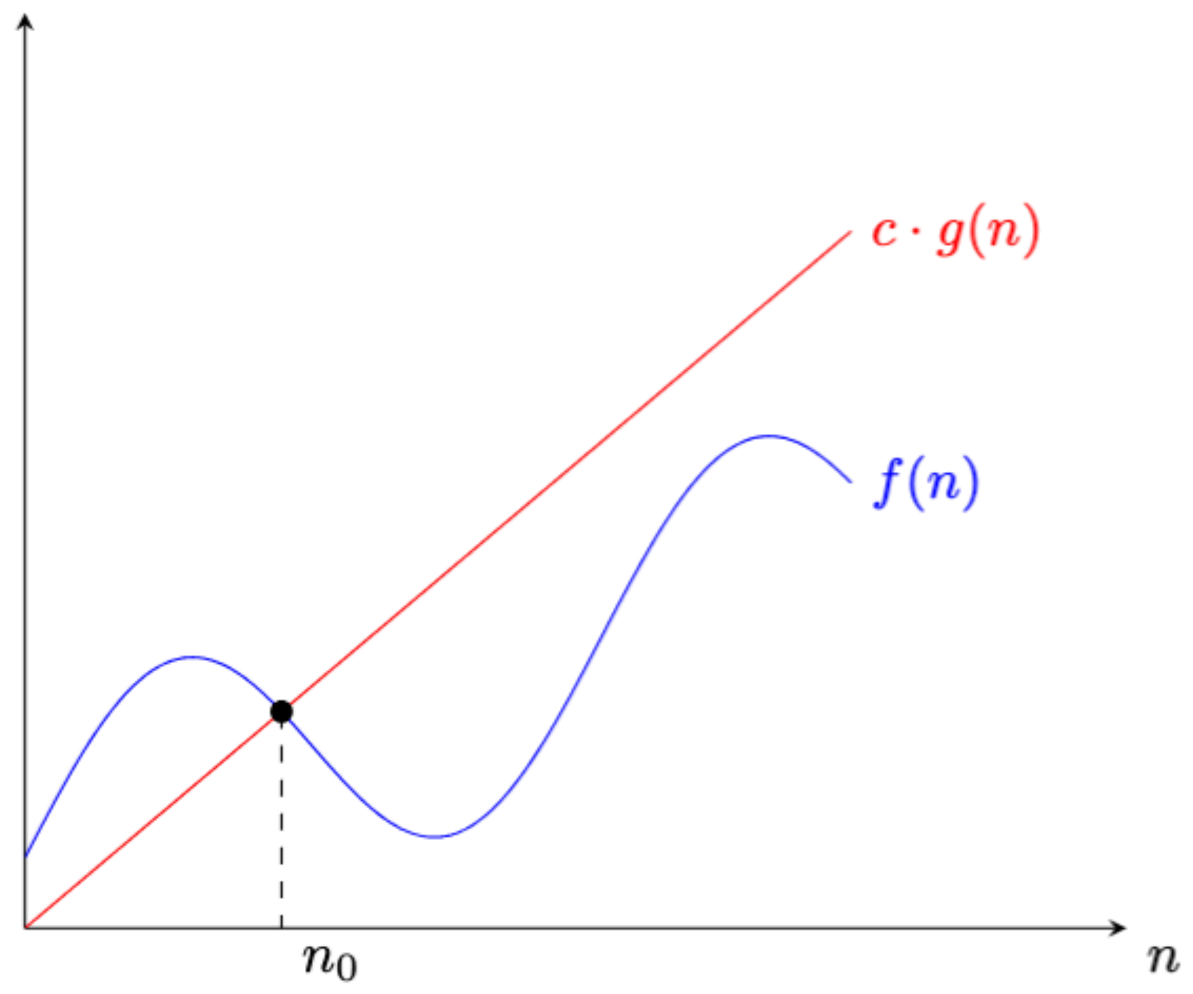
Asymptotic Notation

Asymptotic Notation

Big O: $f(x) \in O(g(x))$ if

$$\exists c, n_0 \in \{1,2,3,\ldots\} \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)$$

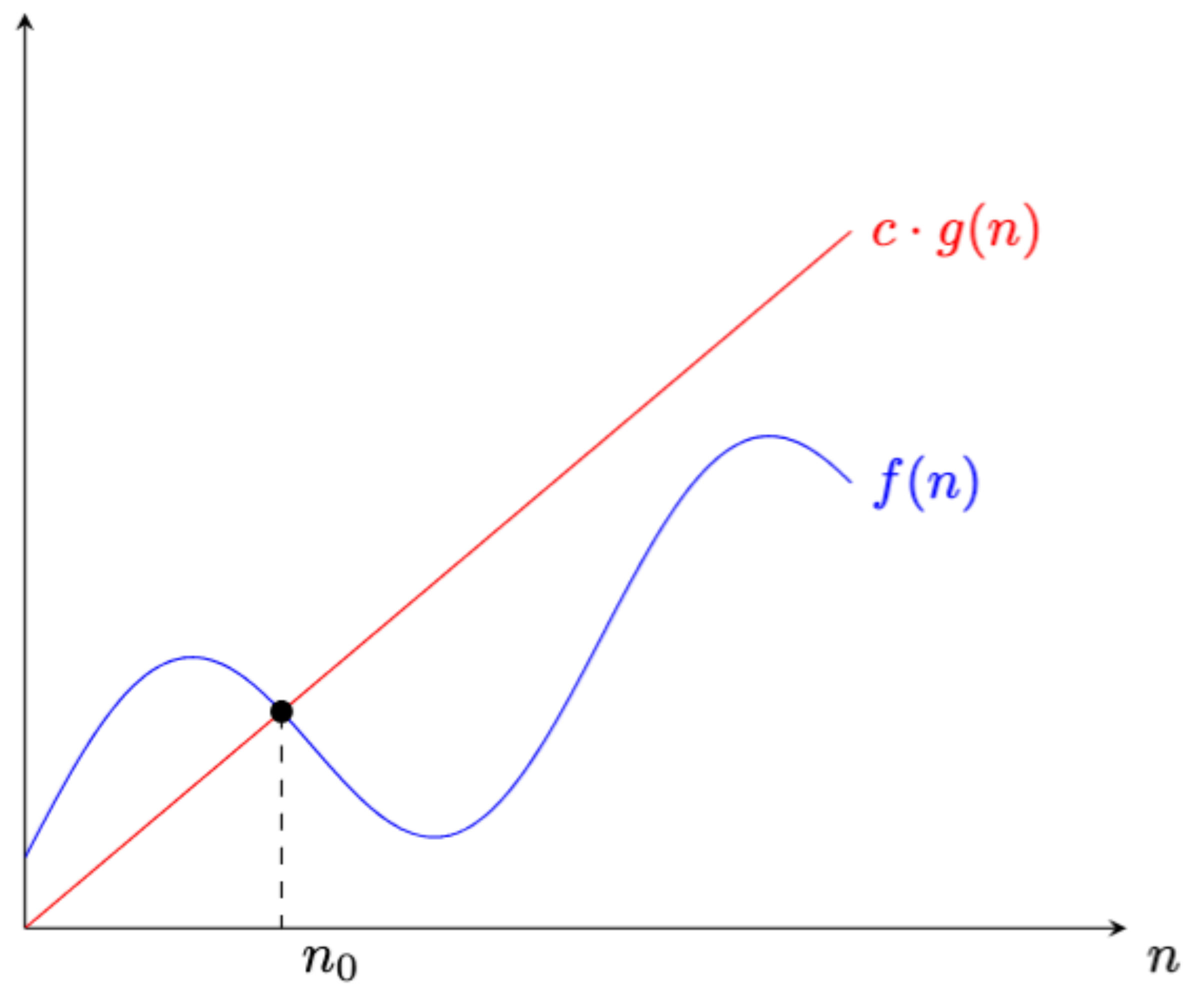
Asymptotic Notation



Big O: $f(x) \in O(g(x))$ if

$$\exists c, n_0 \in \{1,2,3,\ldots\} \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)$$

Asymptotic Notation

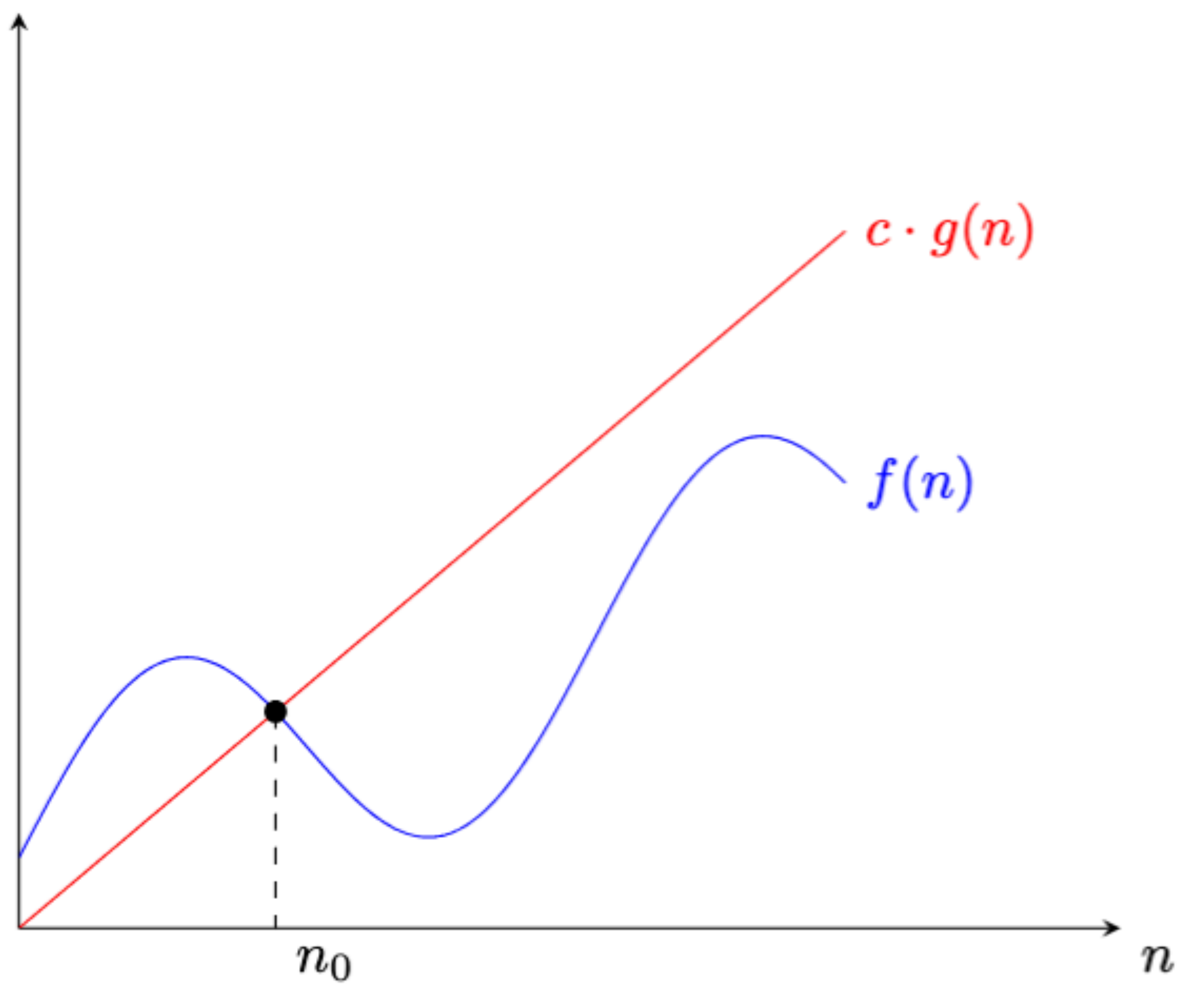


Big O: $f(x) \in O(g(x))$ if

$$\exists c, n_0 \in \{1,2,3,\ldots\} \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)$$

Another way of saying that an algorithm is "efficient" is to say that its input-length to runtime function $T(x)$ is in $O(x^d)$ for some constant d.

Asymptotic Notation



Big O: $f(x) \in O(g(x))$ if

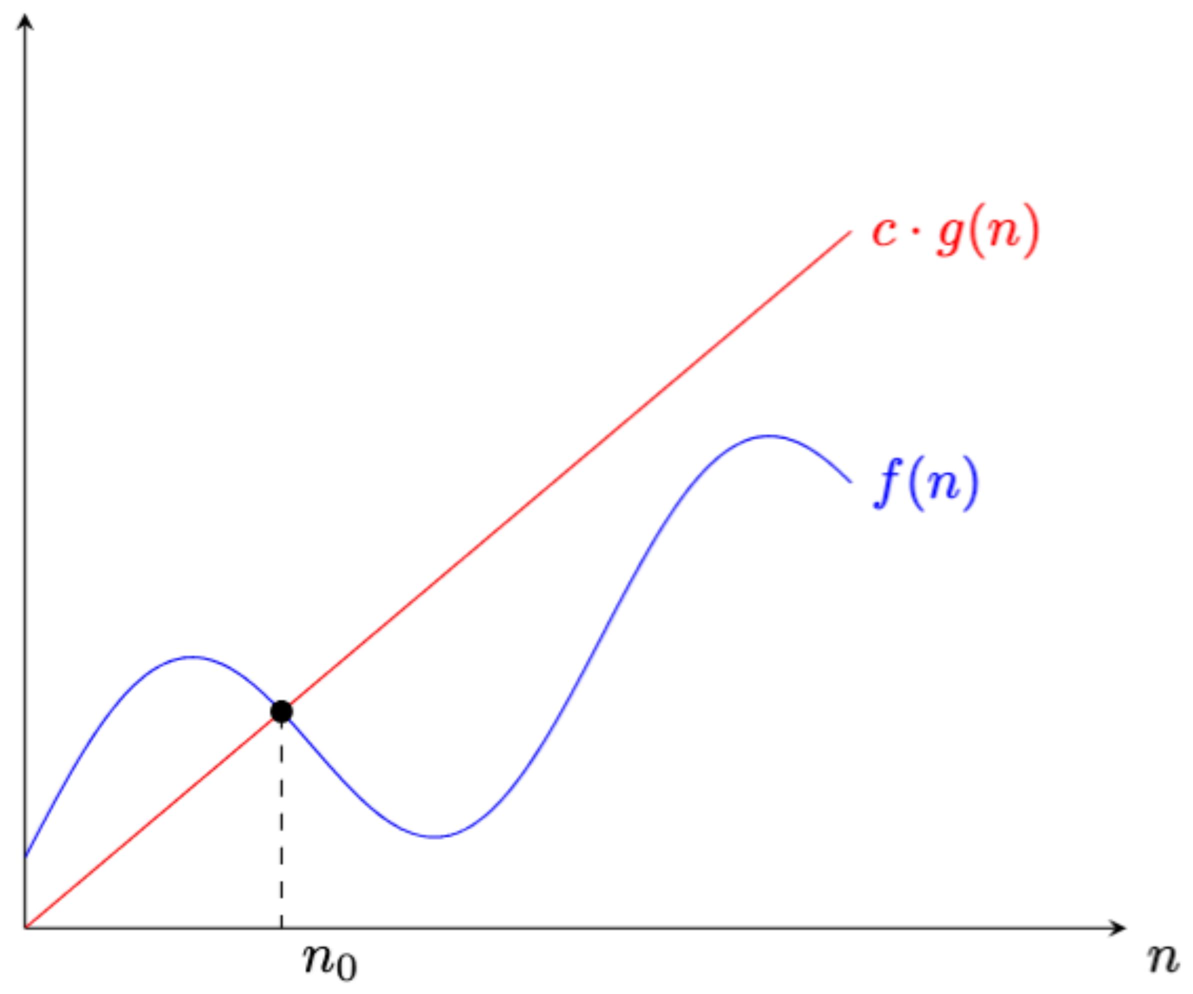
$$\exists c, n_0 \in \{1,2,3,\ldots\} \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)$$

Another way of saying that an algorithm is "efficient" is to say that its input-length to runtime function $T(x)$ is in $O(x^d)$ for some constant d.

Little Omega: $f(x) \in \omega(g(x))$ if

$$\exists c, n_0 \in \{1,2,3,\ldots\} \text{ such that } \forall n \geq n_0,\ 0 \leq g(n) < c \cdot f(n)$$

Asymptotic Notation



Big O: $f(x) \in O(g(x))$ if

$$\exists c, n_0 \in \{1,2,3,\ldots\} \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)$$

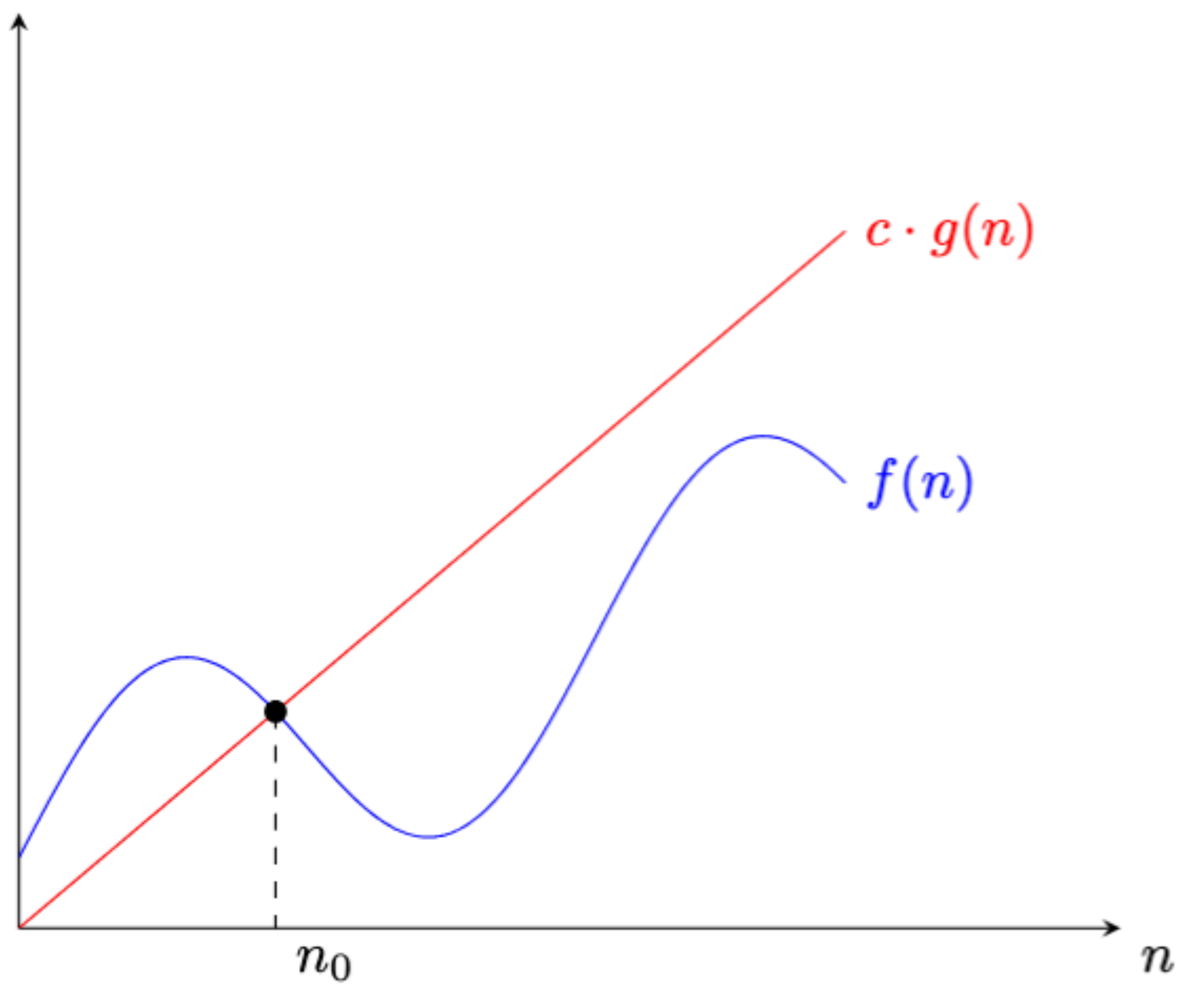
Another way of saying that an algorithm is "efficient" is to say that its input-length to runtime function $T(x)$ is in $O(x^d)$ for some constant d.

Little Omega: $f(x) \in \omega(g(x))$ if

$$\exists c, n_0 \in \{1,2,3,\ldots\} \text{ such that } \forall n \geq n_0,\ 0 \leq g(n) < c \cdot f(n)$$

Little omega means that $f(x)$ grows *strictly faster* than $g(x)$

Asymptotic Notation



Big O: $f(x) \in O(g(x))$ if

$$\exists c, n_0 \in \{1,2,3,\ldots\} \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)$$

Another way of saying that an algorithm is "efficient" is to say that its input-length to runtime function $T(x)$ is in $O(x^d)$ for some constant d.

Little Omega: $f(x) \in \omega(g(x))$ if

$$\exists c, n_0 \in \{1,2,3,\ldots\} \text{ such that } \forall n \geq n_0,\ 0 \leq g(n) < c \cdot f(n)$$

Little omega means that $f(x)$ grows *strictly faster* than $g(x)$

We may say that a $f(x)$ is "super-polynomial" to mean that $f(x) \in \omega(x^d)$ for any constant d