Algorithms for Decision Support

NP-Completeness (1/3)

Turing Machine, **P**, and **NP**

Overview

- Algorithms:Turing machine, Deterministic and non-deterministic
- Formal language framework: string and language
- Time complexity
  - Input size
  - Classes **P** and **NP**
    - Polynomial time verification

Overview

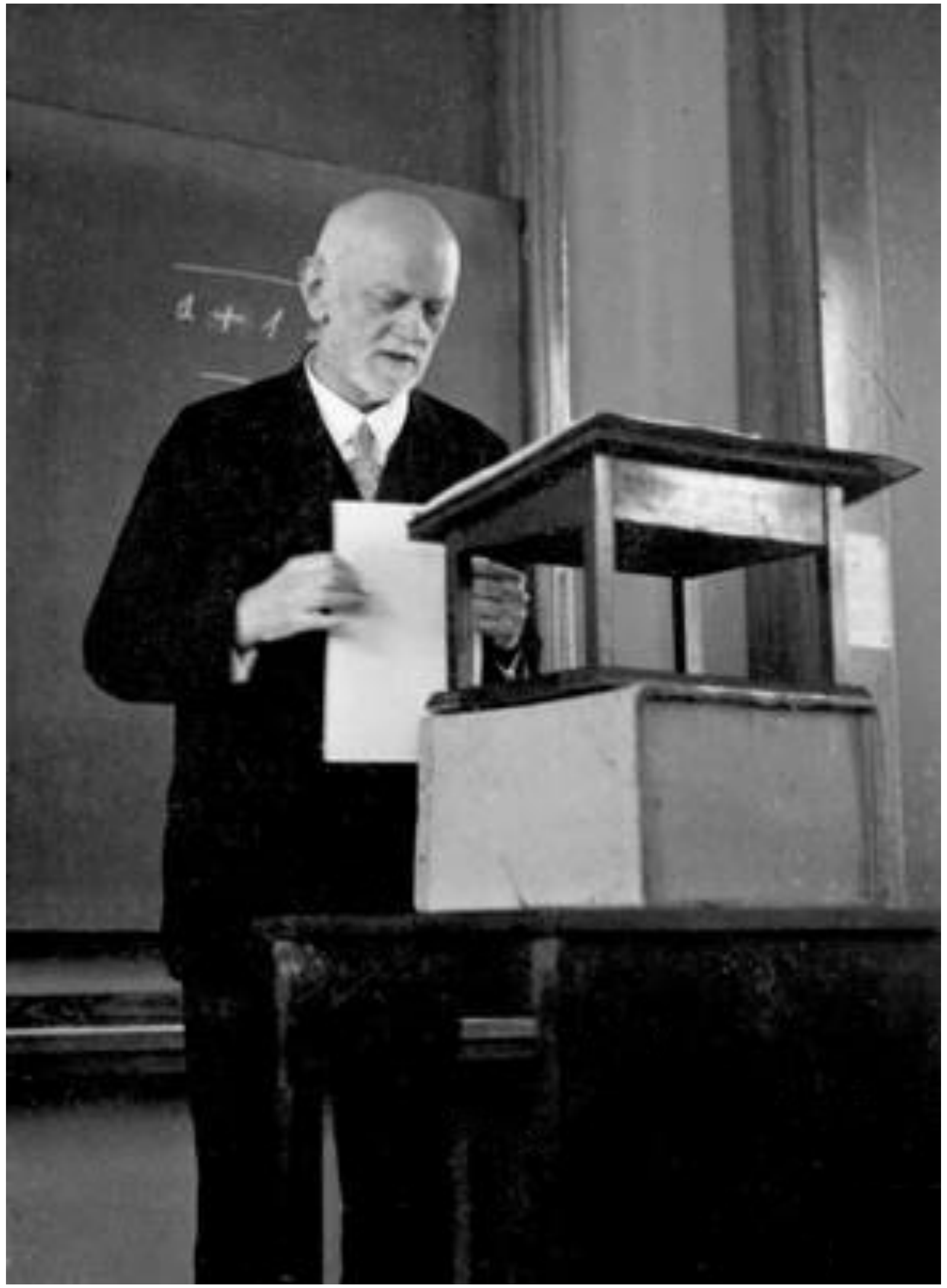
- Algorithms:Turing machine, Deterministic and non-deterministic

- Formal language framework: string and language

- Time complexity
  - Input size
  - Classes **P** and **NP**
    - Polynomial time verification

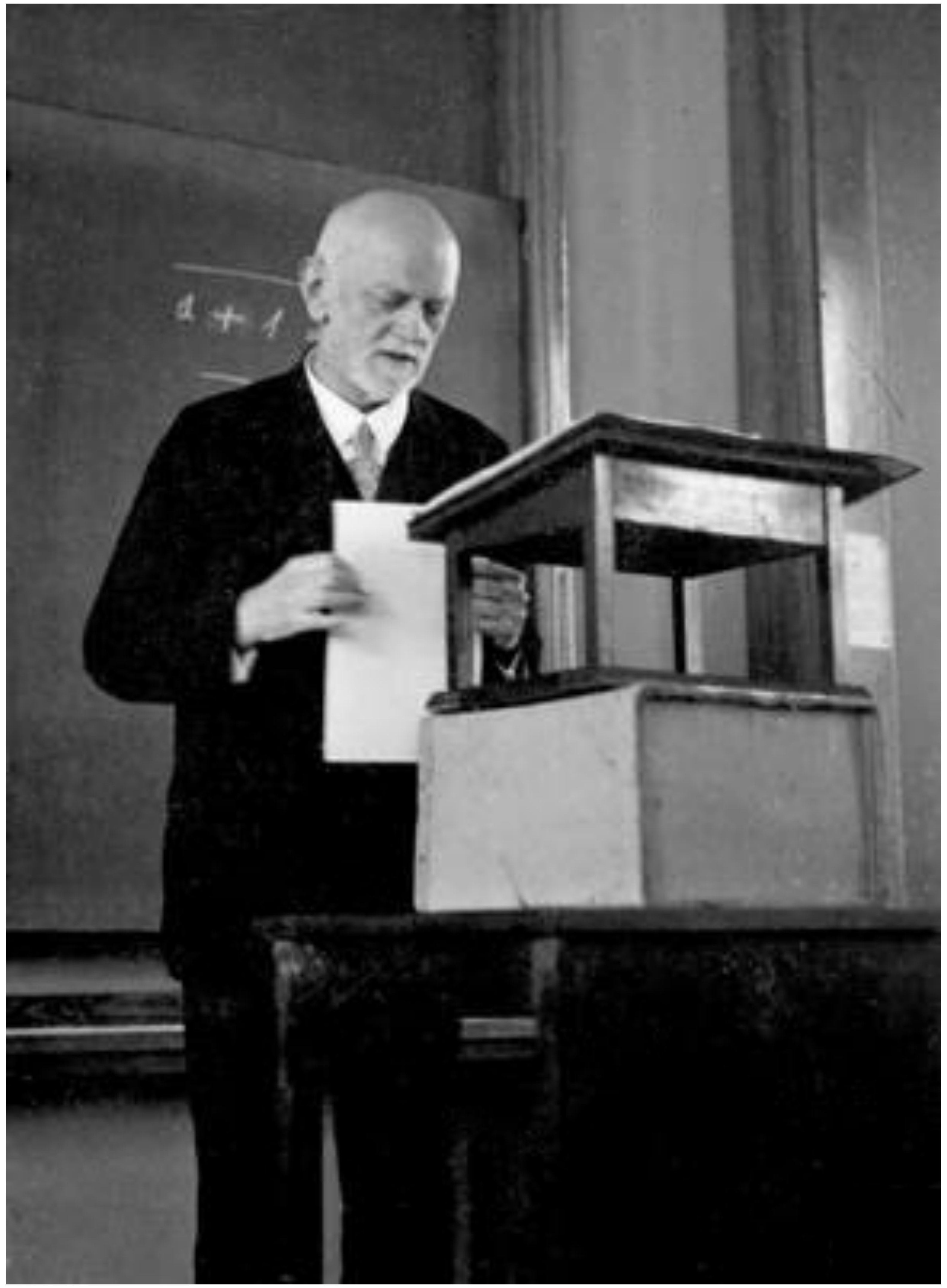
Hilbert's 10^{th} Problem

- In 1900, mathematician David Hilbert delivered a famous talk at the International Congress of Mathematicians in Paris



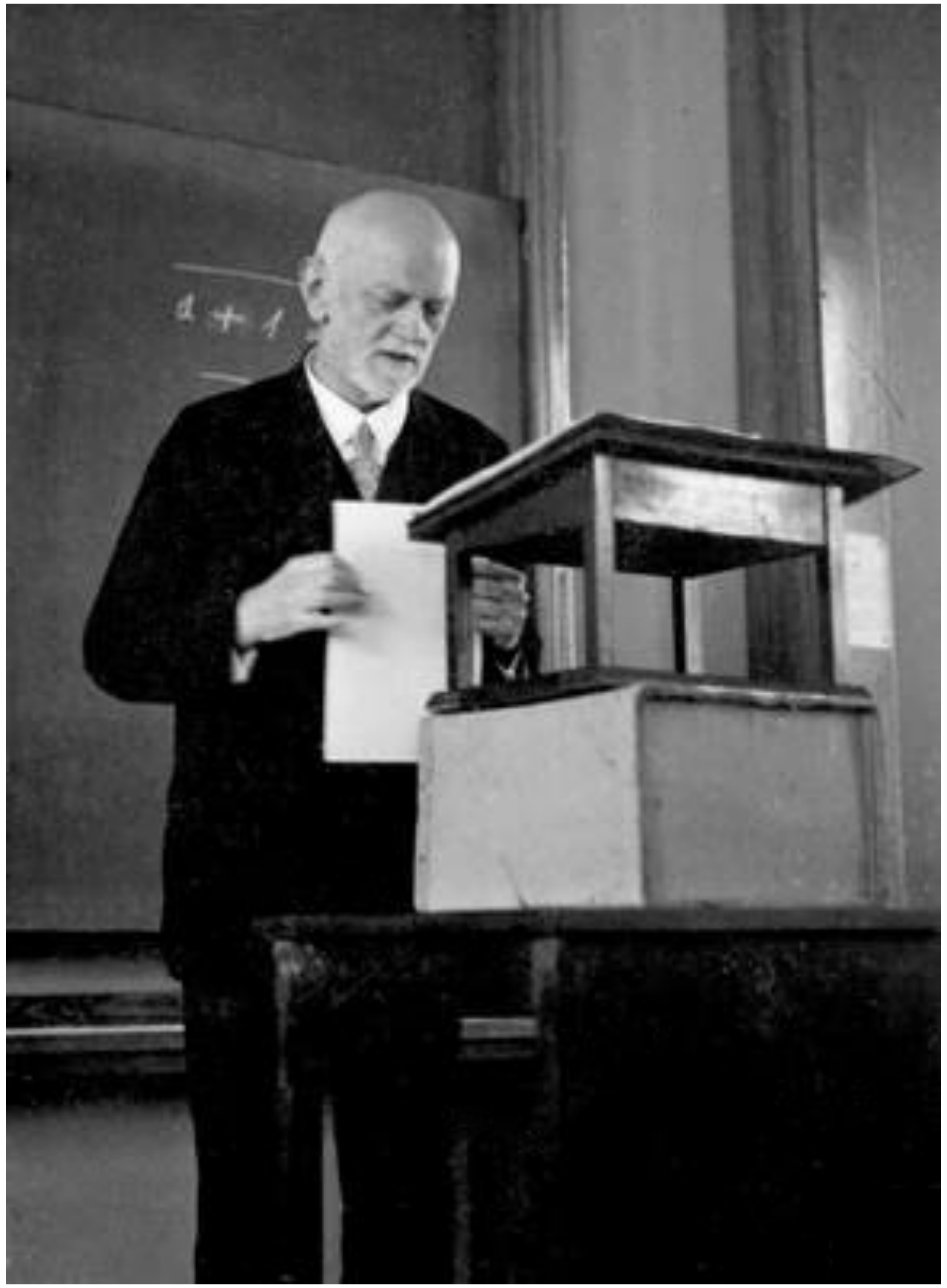
Hilbert's 10^{th} Problem

- In 1900, mathematician David Hilbert delivered a famous talk at the International Congress of Mathematicians in Paris
- He identified 23 math problems which he thinks are important in the coming century



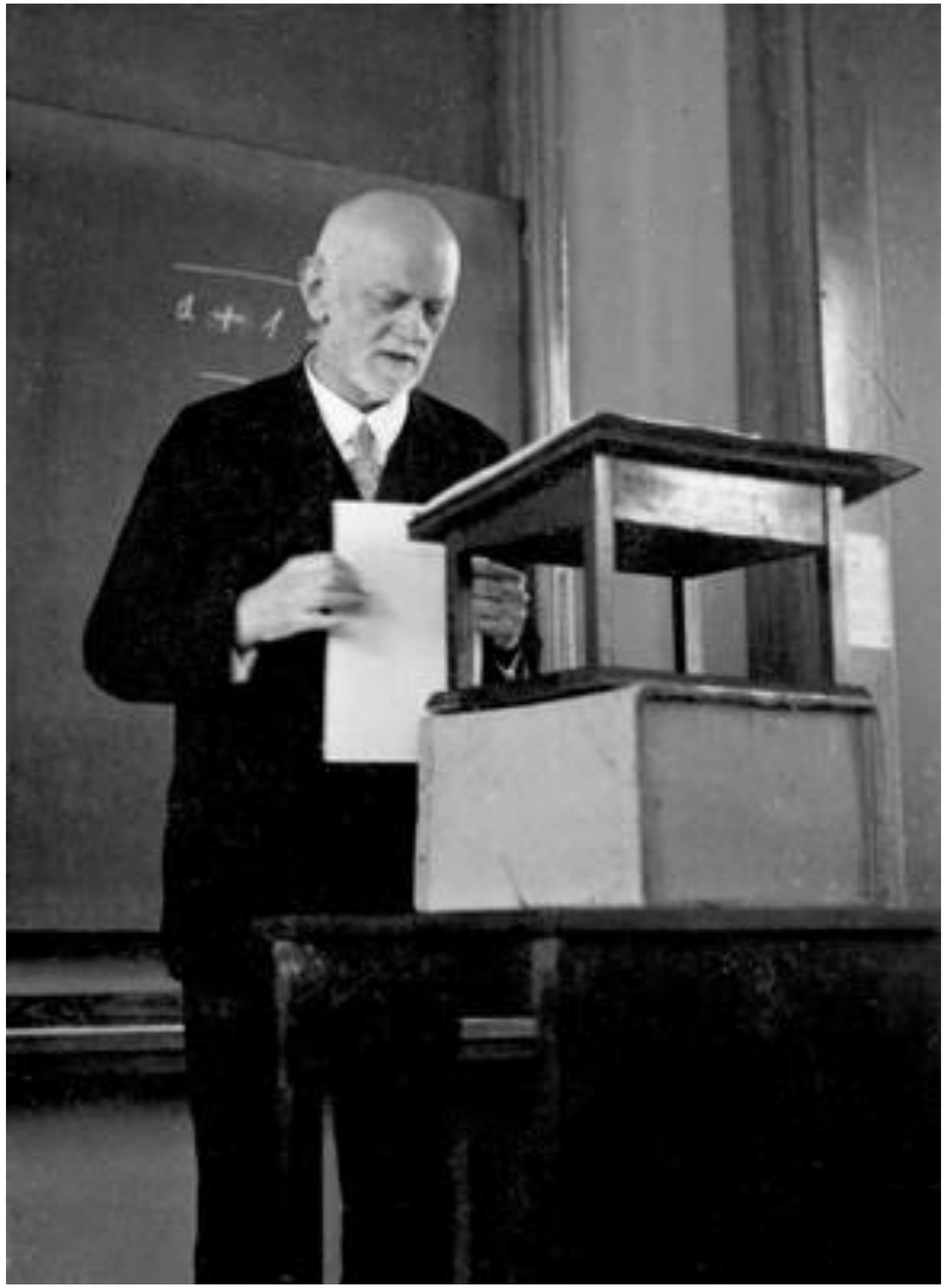
Hilbert's 10^{th} Problem

- In 1900, mathematician David Hilbert delivered a famous talk at the International Congress of Mathematicians in Paris
- He identified 23 math problems which he thinks are important in the coming century
- Hilbert's 10th problem conceded algorithms:
  Given a multi-variable polynomial F with integral coefficients. To devise a process according to which it can be determined in a finite number of operations whether F has integral roots.
  - $x^2 + 2xy + y^2 - 1 = 0$



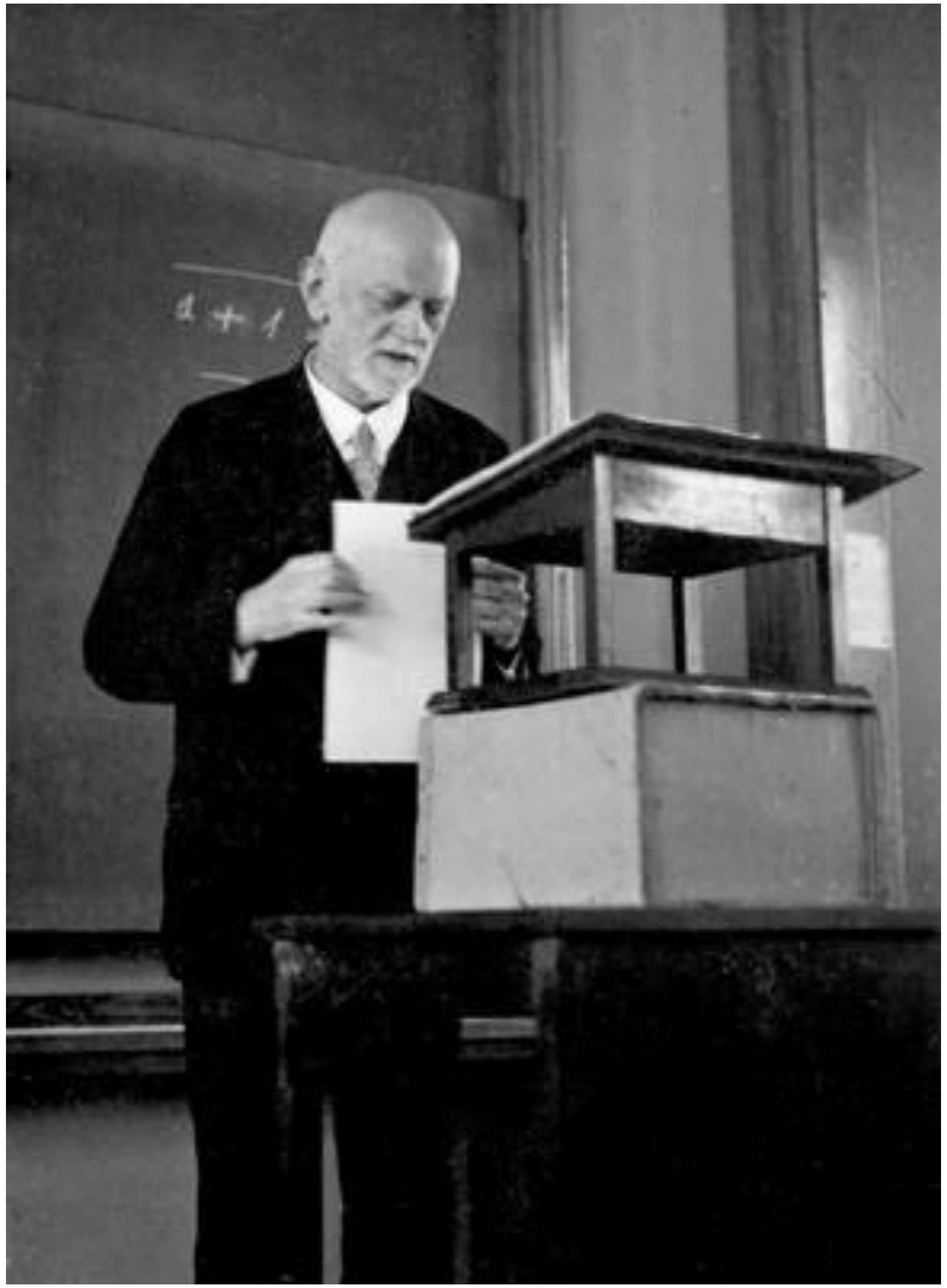
Hilbert's 10th Problem

- In 1900, mathematician David Hilbert delivered a famous talk at the International Congress of Mathematicians in Paris
- He identified 23 math problems which he thinks are important in the coming century
- Hilbert's 10th problem conceded algorithms:
  Given a multi-variable polynomial F with integral coefficients. To devise a process according to which it can be determined in a finite number of operations whether F has integral roots.
  - $x^2 + 2xy + y^2 - 1 = 0$ $(x = 2, y = -3)$



Hilbert's 10^{th} Problem

- In 1900, mathematician David Hilbert delivered a famous talk at the International Congress of Mathematicians in Paris
- He identified 23 math problems which he thinks are important in the coming century
- Hilbert's 10th problem conceded algorithms:
  Given a multi-variable polynomial F with integral coefficients. To devise a process according to which it can be determined in a finite number of operations whether F has integral roots.
  - $x^2 + 2xy + y^2 - 1 = 0$ $(x = 2, y = -3)$
  - $x^2 + y^2 - 3 = 0$



What is an algorithm/computer?

Turing Machine

- Proposed by Alan Turing in 1936

Alan Turing 1912~1954

Turing Machine

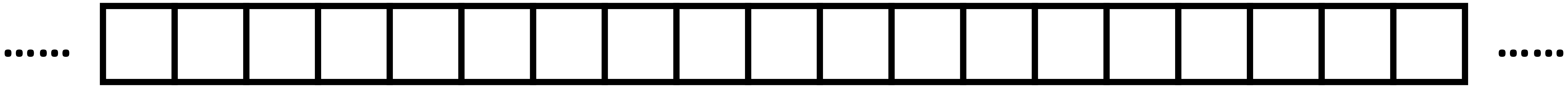
- Proposed by Alan Turing in 1936
- A model of a general purpose computer:
  a Turing machine can do every thing that a real computer can do

Alan Turing 1912~1954

Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory

Alan Turing 1912~1954



Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else

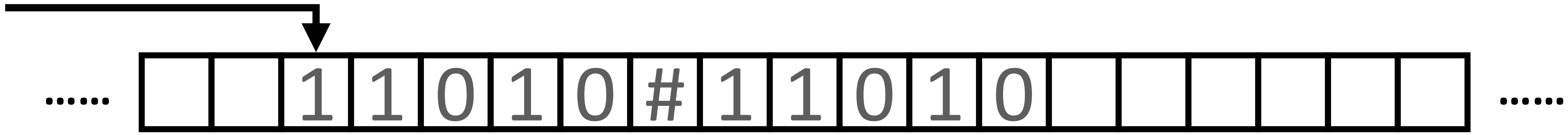
Alan Turing 1912~1954

……			1	1	0	1	0	#	1	1	0	1	0							……

Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape

Alan Turing 1912~1954



Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape

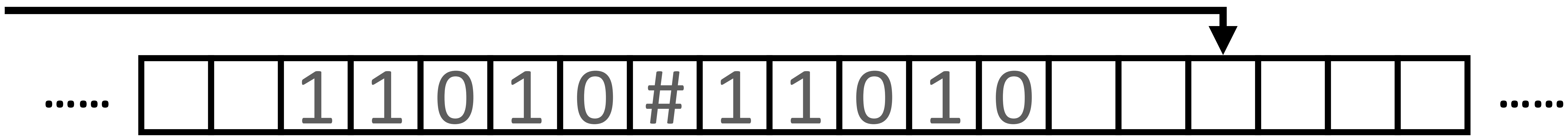
Alan Turing 1912~1954

......			1	1	0	1	0	#	1	1	0	1	0							......

Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape

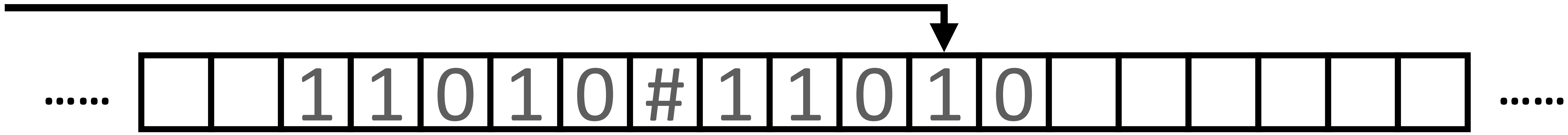
Alan Turing 1912~1954



Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape

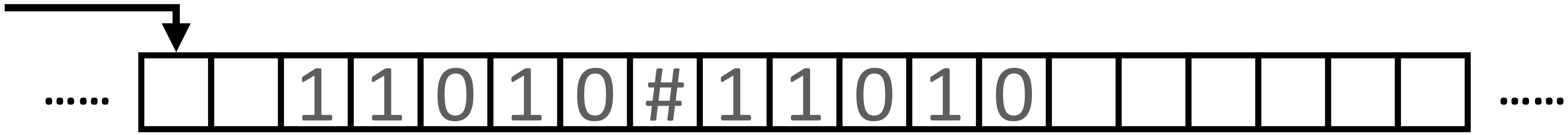
Alan Turing 1912~1954



Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape

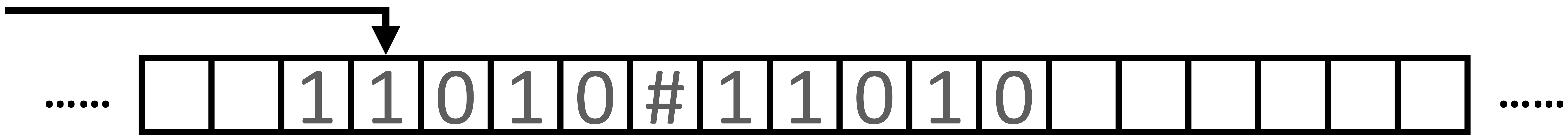
Alan Turing 1912~1954



Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape

Alan Turing 1912~1954



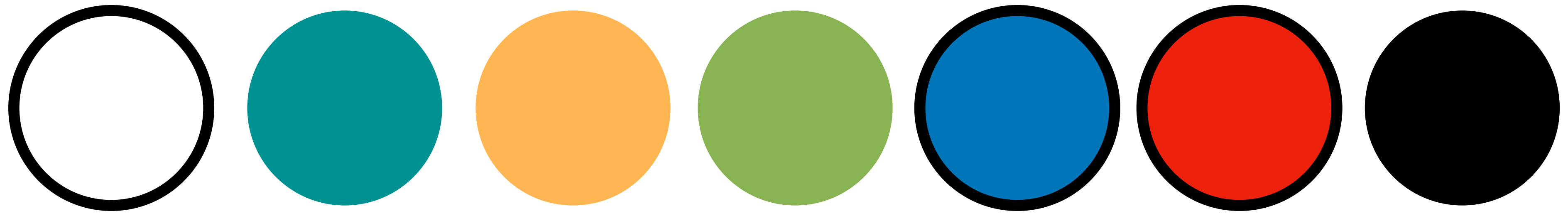
Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape
  - control

Alan Turing 1912~1954

Turing Machine

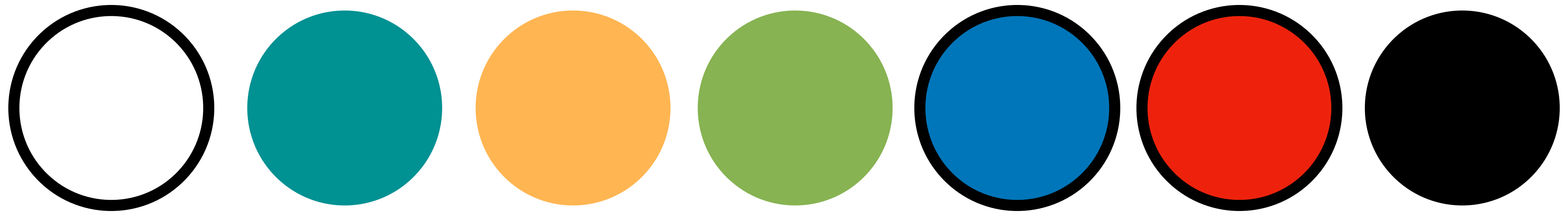
- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape
  - Finite-**state** control



Alan Turing 1912~1954

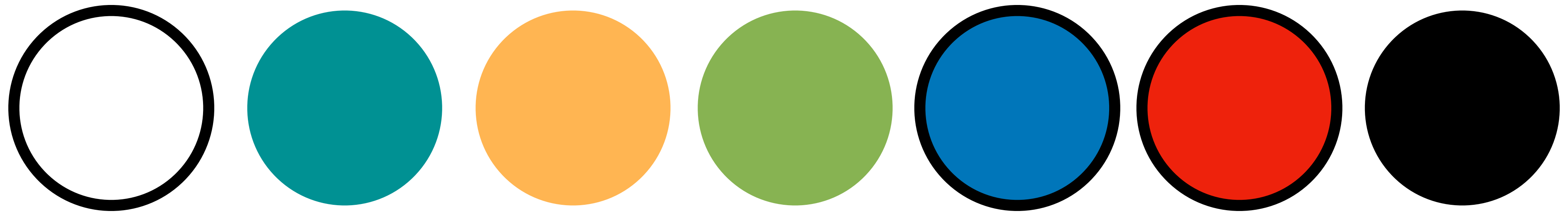
Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape
  - Finite-state control



Alan Turing 1912~1954

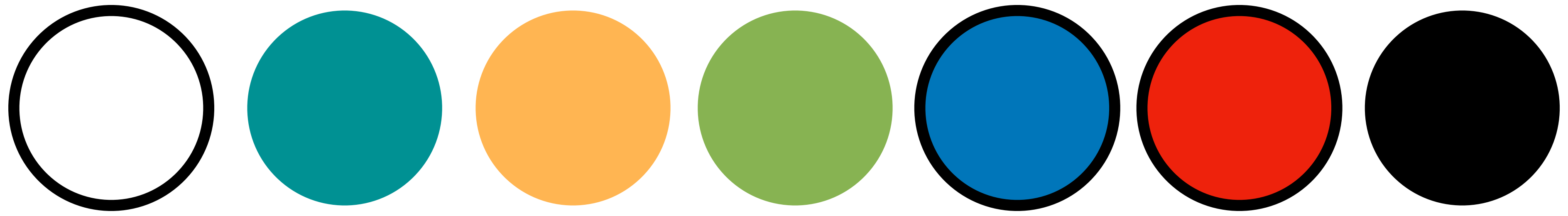
Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape
  - Finite-state control 


Alan Turing 1912~1954

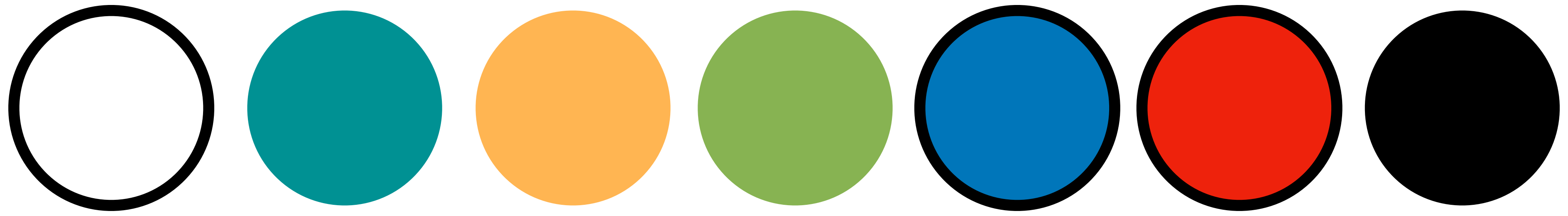
Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape
  - Finite-state control



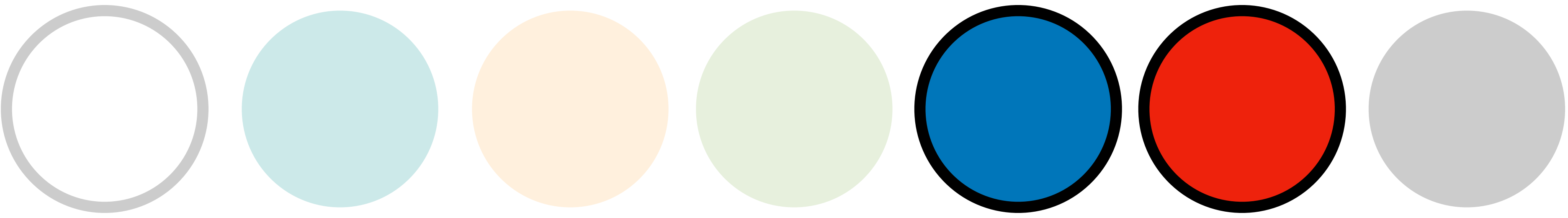
Alan Turing 1912~1954

Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape
  - Finite-state control 

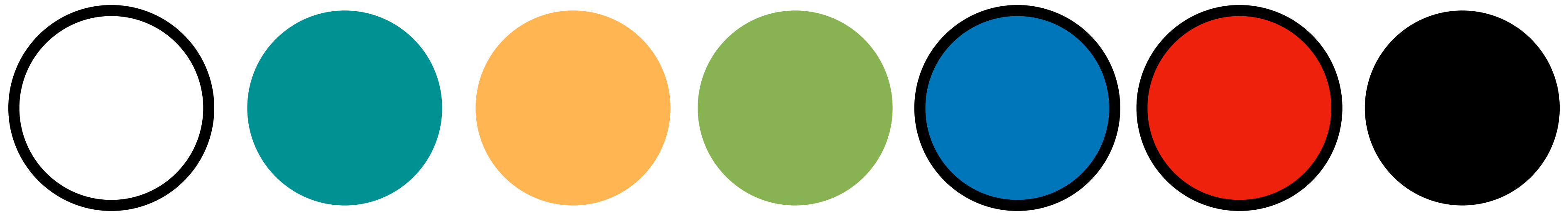
Alan Turing 1912~1954

Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape
  - Finite-state control 
    - Two special ***halting*** states: *accept* and *reject*

Alan Turing 1912~1954

Turing Machine

- Proposed by Alan Turing in 1936
  - An infinitely long tape/memory
    - Initially contains the (finite) input sequence and is blank everywhere else
  - A tape head that can read and write symbols and move around on the tape
  - Finite-state control 
    - Two special ***halting*** states: *accept* and *reject*

Alan Turing 1912~1954

Turing Machine

- A Turing machine's *configuration*:

Turing Machine

- A Turing machine's *configuration*:
  - Its current state
  - what the read-write head reads

Turing Machine

- A Turing machine's *configuration*:
  - Its current state
  - what the read-write head reads
- By its current configuration, a Turing machine
  - decides the tape symbol to write on the tape,
  - switches to the next state, and
  - moves the tape head to its left or right

Turing Machine

- A Turing machine's *configuration*:
    - Its current state
    - what the read-write head reads
- By its current configuration, a Turing machine
    - **decides the tape symbol to write on the tape,**
    - switches to the next state, and
    - moves the tape head to its left or right

Turing Machine

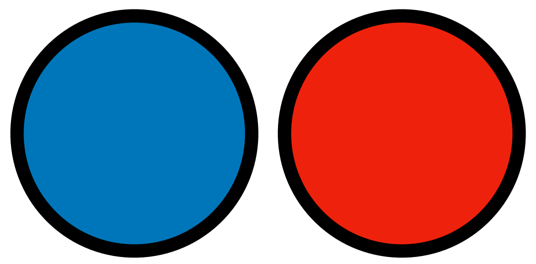
- A Turing machine's *configuration*:
  - Its current state
  - what the read-write head reads
- By its current configuration, a Turing machine
  - decides the tape symbol to write on the tape,
  - **switches to the next state**, and
  - moves the tape head to its left or right

Turing Machine

- A Turing machine's *configuration*:
  - Its current state
  - what the read-write head reads
- By its current configuration, a Turing machine
  - decides the tape symbol to write on the tape,
  - switches to the next state, and
  - **moves the tape head to its left or right**

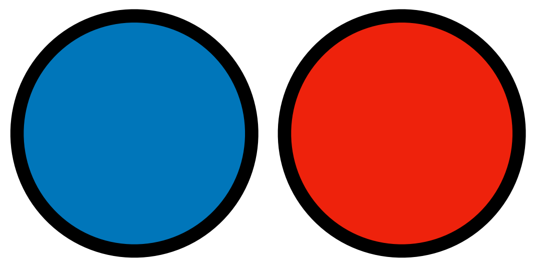
Turing Machine

- Output: ***accept*** or ***reject*** (both **halting** configurations)



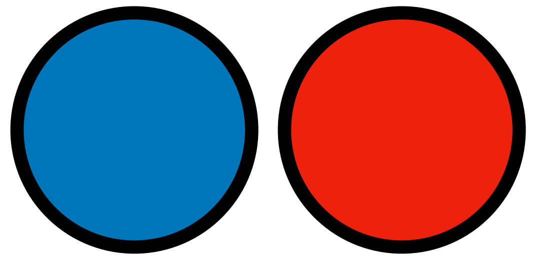
Turing Machine

- Output: ***accept*** or ***reject*** (both **halting** configurations)
  - Obtained by entering designated accepting and rejecting states

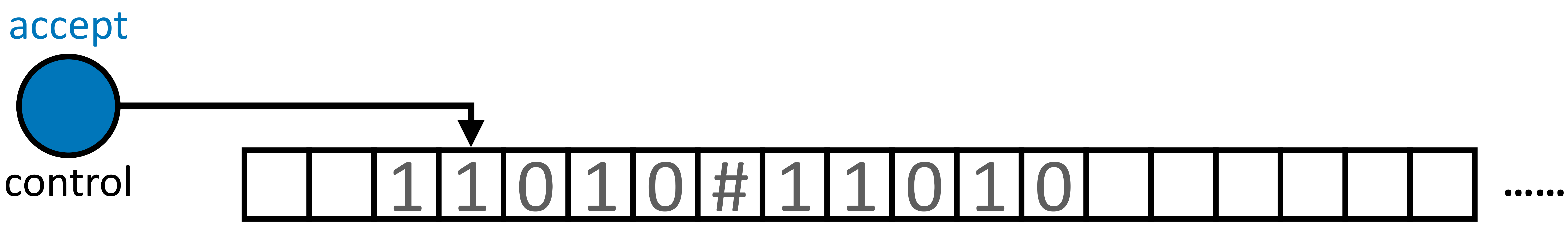


Turing Machine

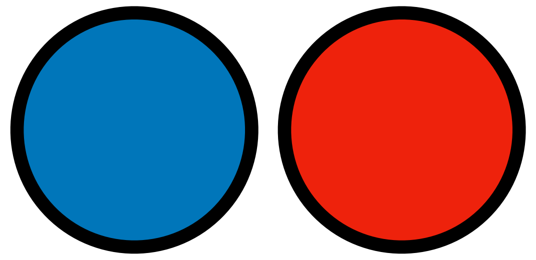
- Output: ***accept*** or ***reject*** (both **halting** configurations)

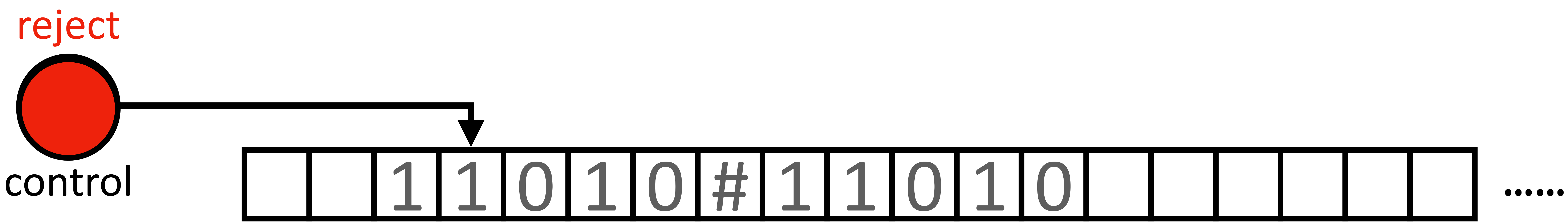


  - Obtained by entering designated accepting and rejecting states

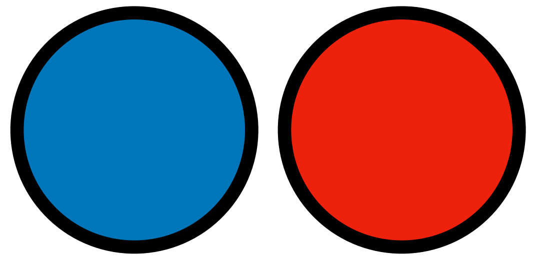


Turing Machine

- Output: ***accept*** or ***reject*** (both **halting** configurations) 
  - Obtained by entering designated accepting and rejecting states

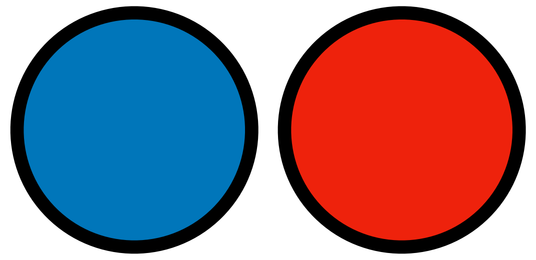


Turing Machine

- Output: ***accept*** or ***reject*** (both **halting** configurations) 
  - Obtained by entering designated accepting and rejecting states
  - When a Turing machine enters the *accept* state, it accepts the input immediately; when a Turing machine enters the *reject* state, it rejects the input immediately

Turing Machine

- Output: ***accept*** or ***reject*** (both **halting** configurations)



  - Obtained by entering designated accepting and rejecting states
  - When a Turing machine enters the *accept* state, it accepts the input immediately; when a Turing machine enters the *reject* state, it rejects the input immediately
  - If it does not enter the accept or reject states, TM will run forever (***loop***), and never halt

Turing Machine

- We can use Turing machines to solve problems!

Turing Machine

- We can use Turing machines to solve problems!
  - Does 1100010000000010 contains 2^k 0s for some integer k?

Turing Machine

- We can use Turing machines to solve problems!
  - Does 1100010000000010 contains 2^k 0s for some integer k?
  - Is 15 a prime number?

Turing Machine

- We can use Turing machines to solve problems!
  - Does 1100010000000010 contains 2^k 0s for some integer k?
  - Is 15 a prime number?
  - Is a graph G connected?

Turing Machine Example

- There is a Turing machine that decides if an input string has 2^k 0's:

Turing Machine Example

- There is a Turing machine that decides if an input string has 2^k 0's:

M = "On input string w:

1. Sweep left to right across the tape, crossing off every other 0.

2. If in stage 1 the tape contained a single 0, *accept*.

3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, *reject*.

4. Return the head to the left-hand end of the tape.

5. Go to stage 1."

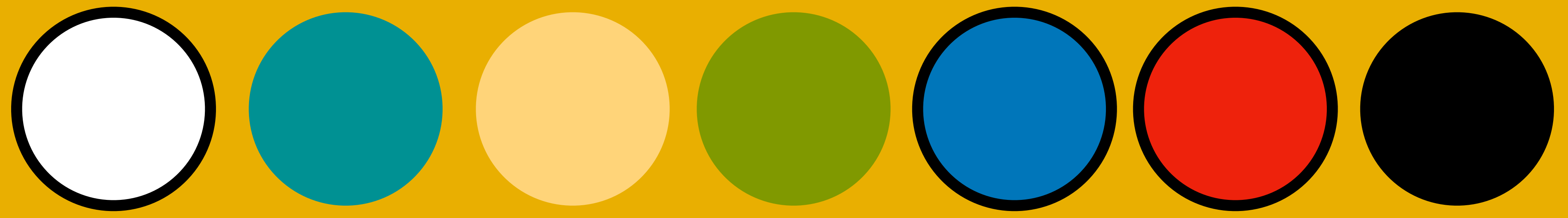
Turing Machine Example

- There is a Turing machine that decides if an input string has 2^k 0's:

M = "On input string w:

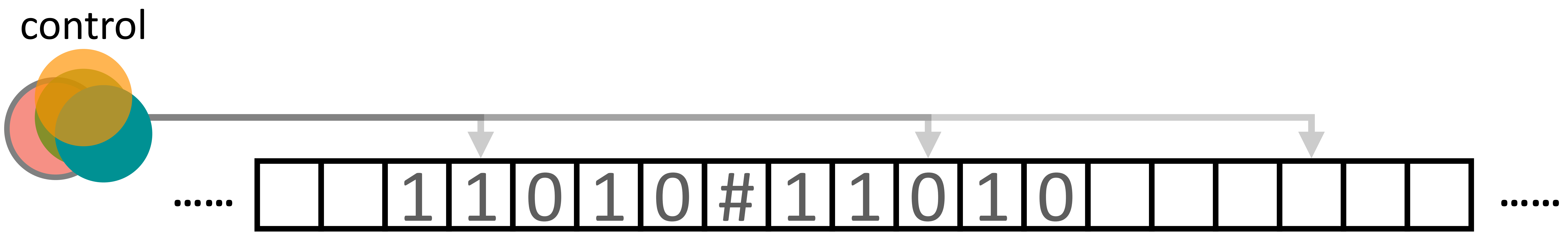
1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, *accept*.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, *reject*.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1."

Turing machine

- An infinitely long tape/memory
  - Initially contains the (finite) input sequence and is blank everywhere else
- A tape head that can read and write symbols and move around on the tape
- Finite-state control 
  - The Turing machine may end up with an *accept* state or *reject* state
  - It *accepts* the input or *rejects* the input

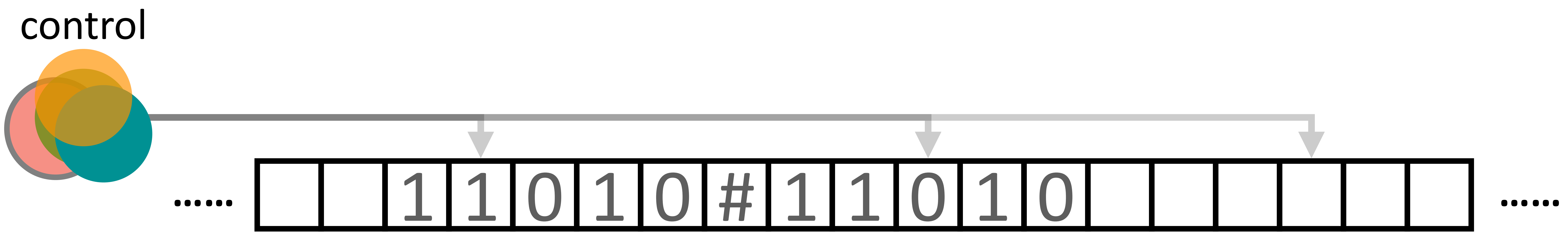
Nondeterministic Turing Machine

Nondeterministic Turing Machine



- It is like a (deterministic) Turing machine, but with non-deterministic control

Nondeterministic Turing Machine

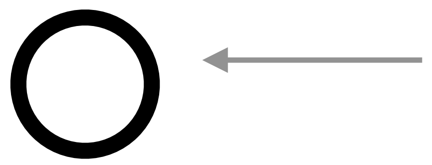


- It is like a (deterministic) Turing machine, but with non-deterministic control

- For an input w, we can describe all possible computations of nondeterministic Turing machine by a *computation tree*

Nondeterministic Turing Machine

Deterministic



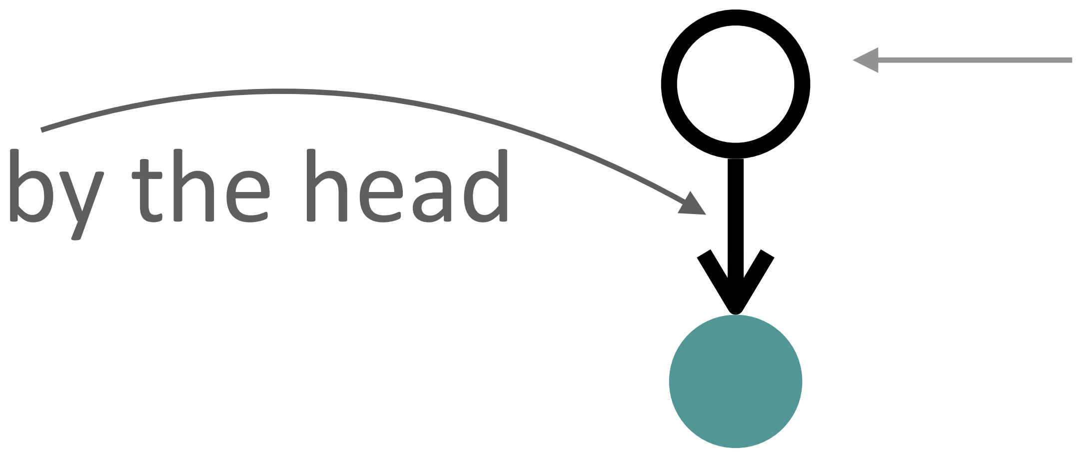
Root: initial configuration

: Configuration

(The current control state and
what the read-write head reads

Nondeterministic Turing Machine

Deterministic

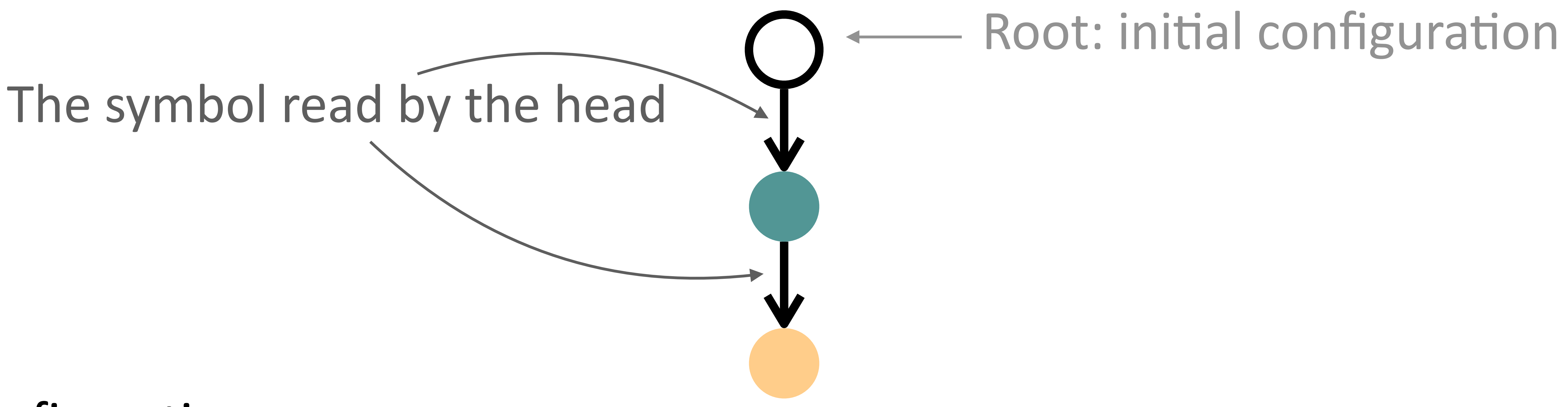


: Configuration

(The current control state and
what the read-write head reads

Nondeterministic Turing Machine

Deterministic

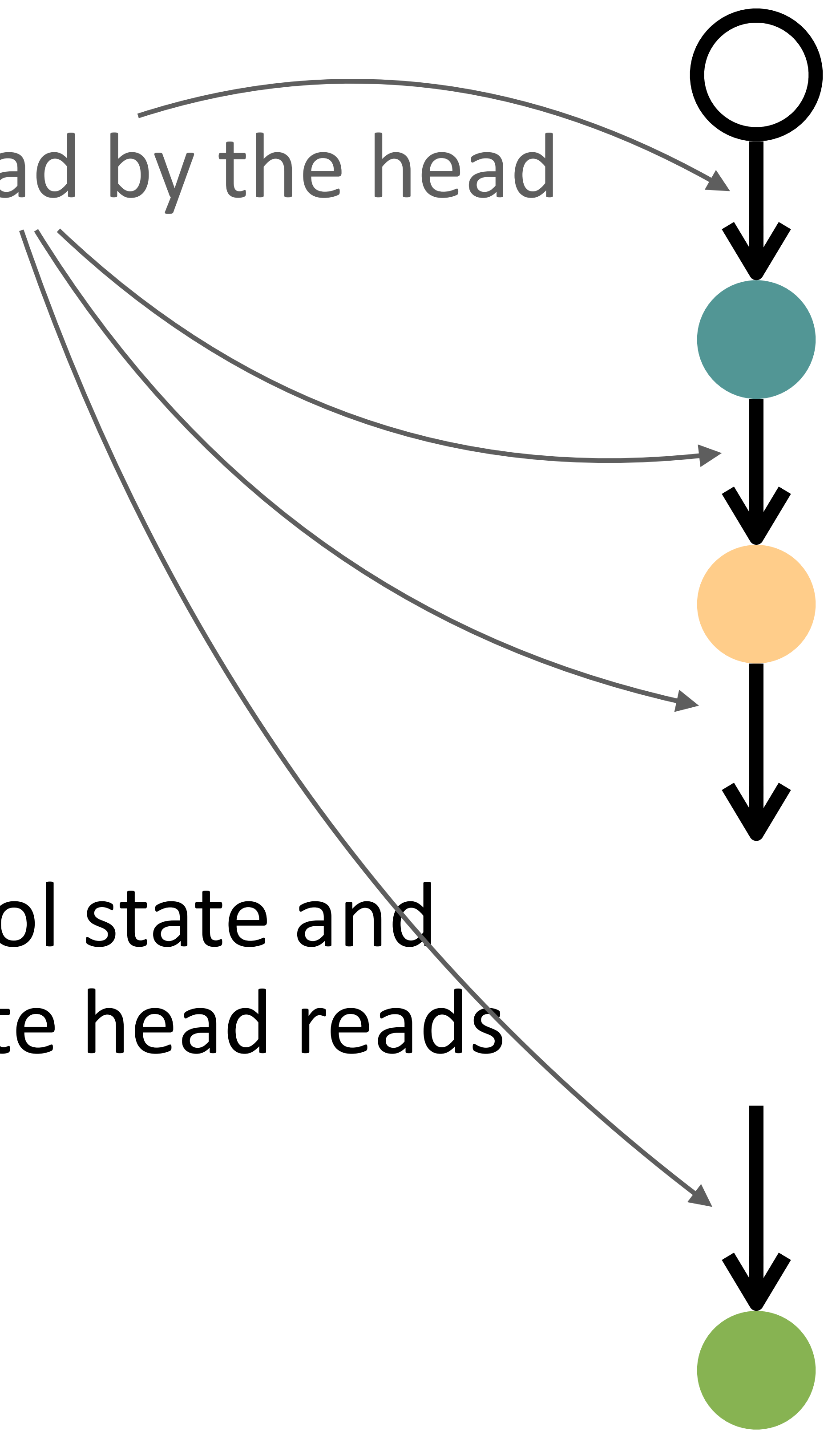


: Configuration

(The current control state and
what the read-write head reads

Nondeterministic Turing Machine

Deterministic

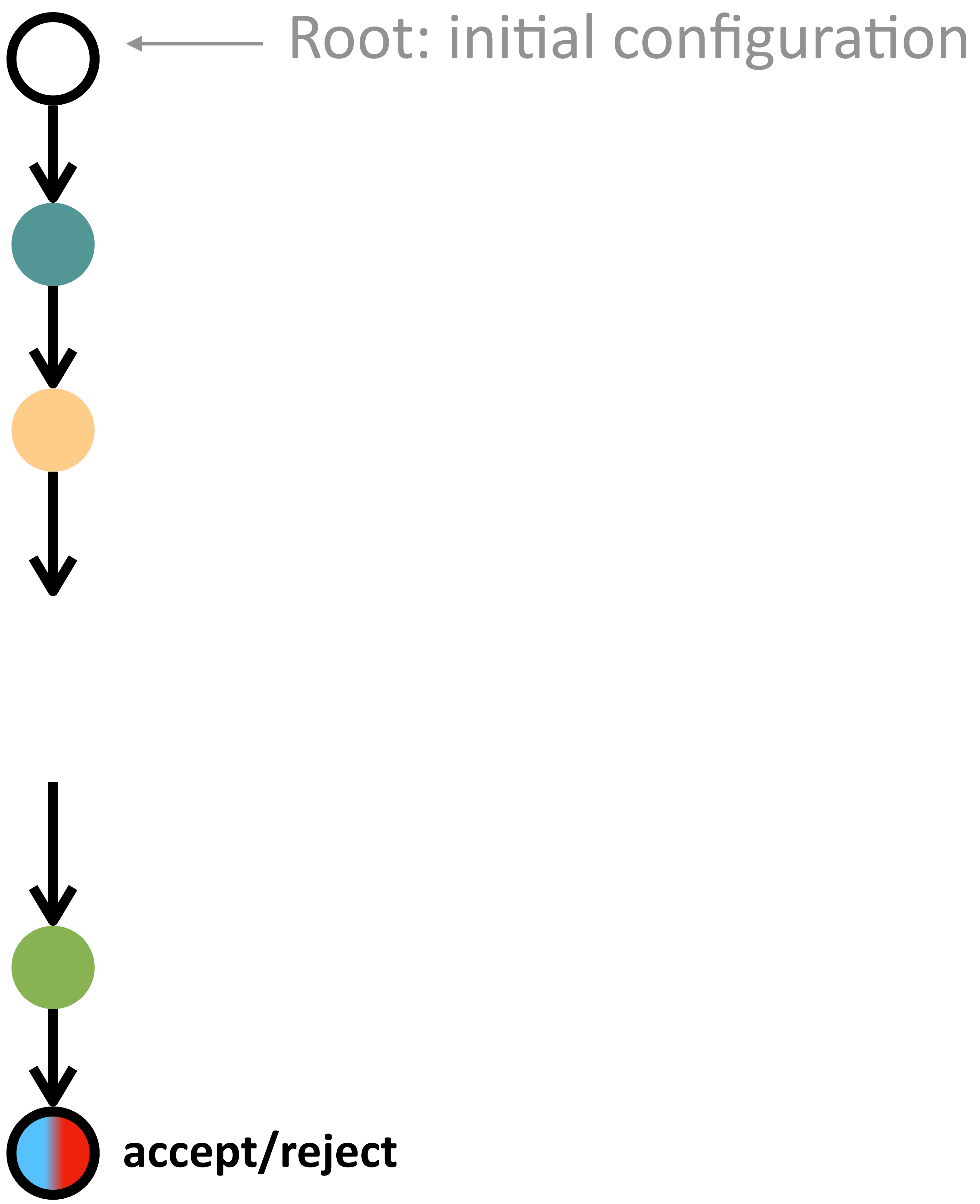


: Configuration

(The current control state and
what the read-write head reads

Nondeterministic Turing Machine

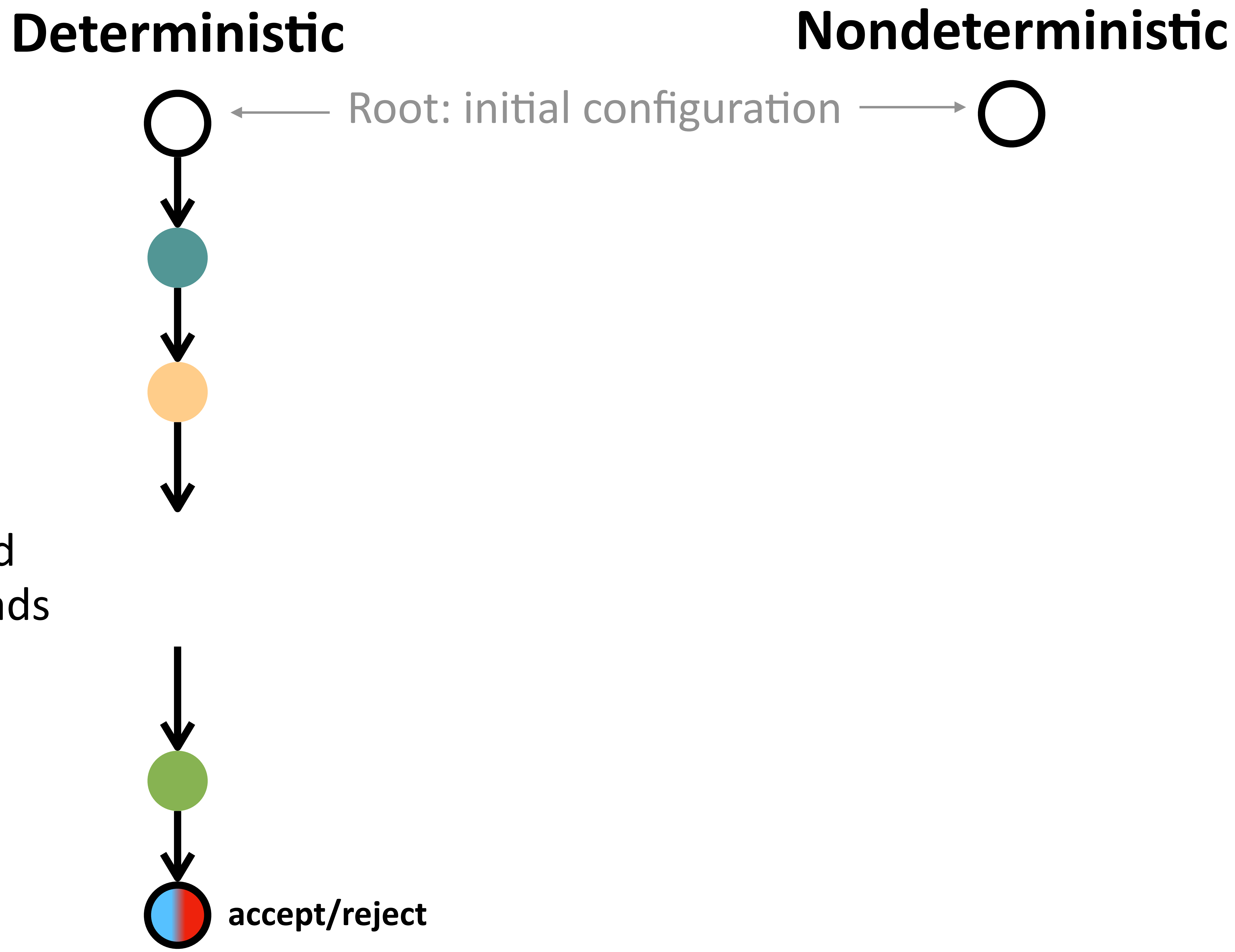
Deterministic



: Configuration

(The current control state and
what the read-write head reads

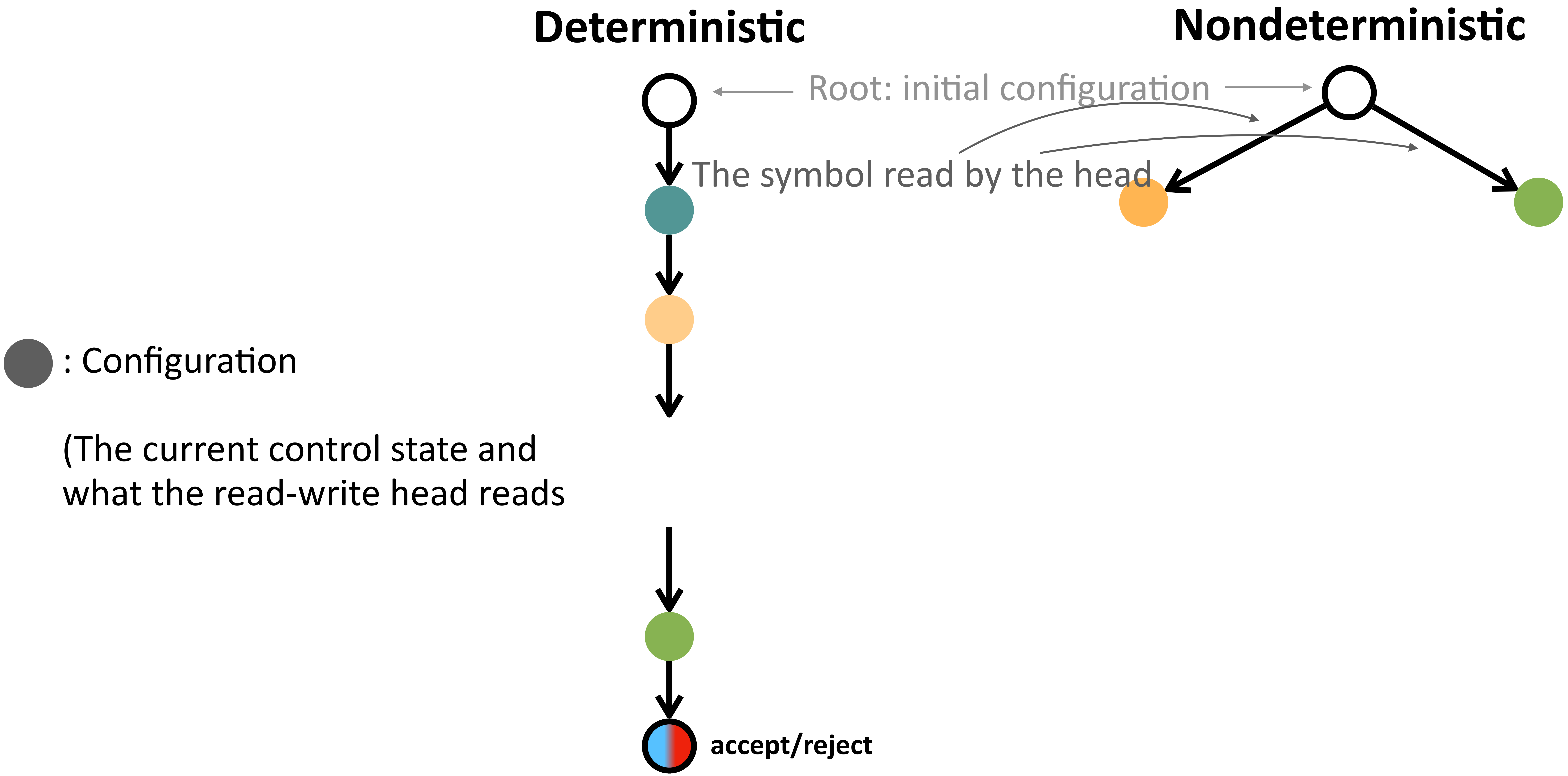
Nondeterministic Turing Machine



: Configuration

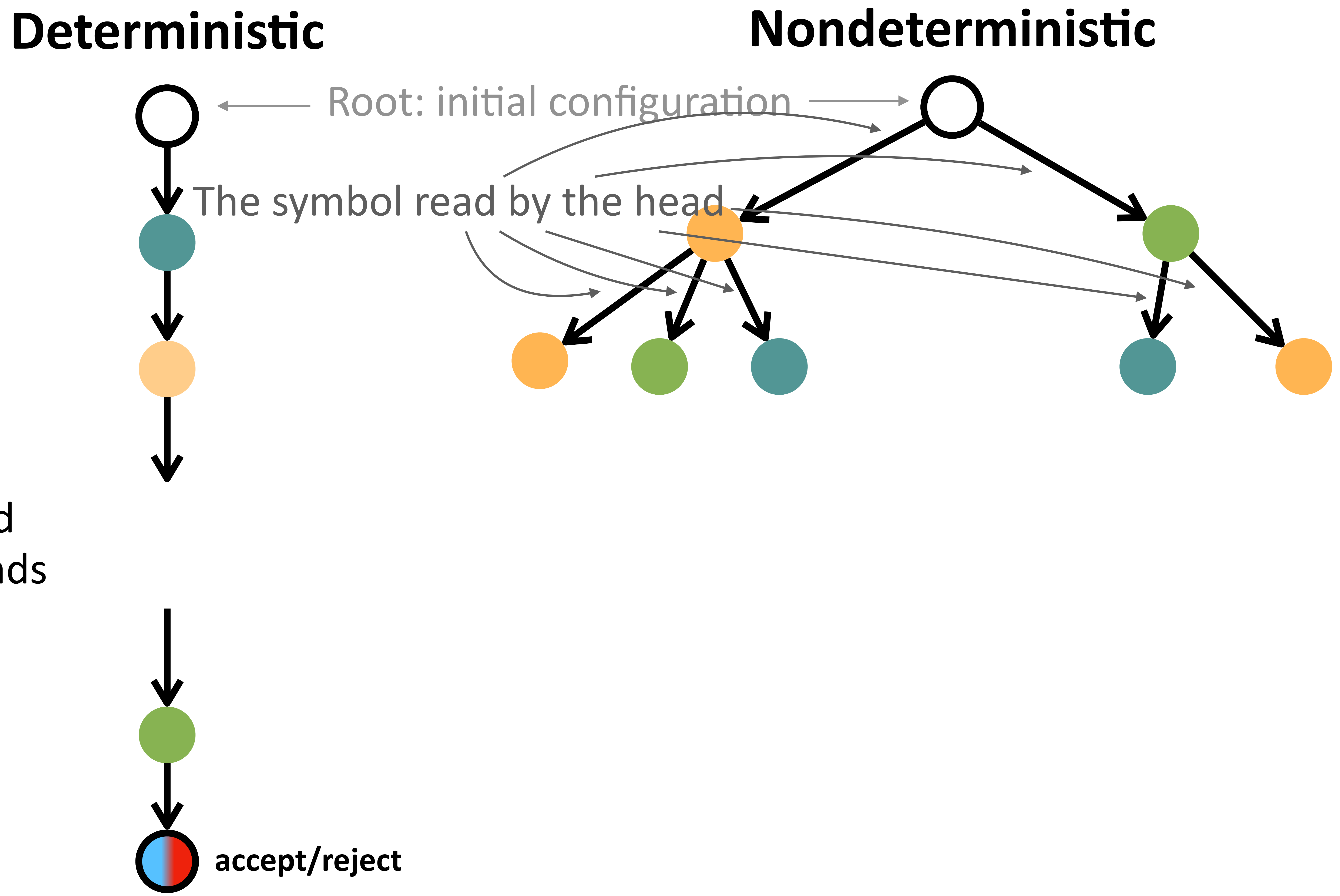
(The current control state and
what the read-write head reads

Nondeterministic Turing Machine



(The current control state and
what the read-write head reads

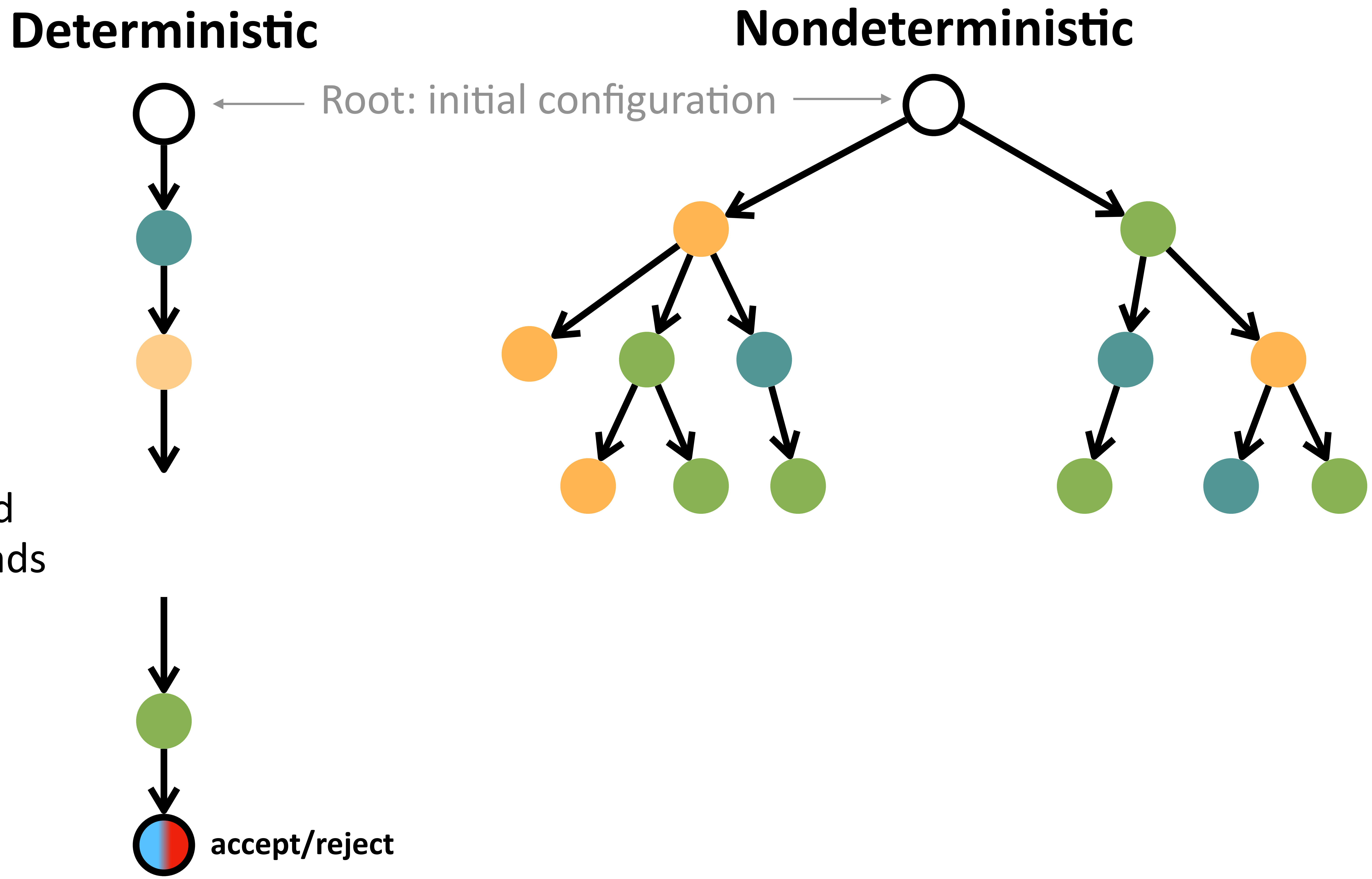
Nondeterministic Turing Machine



: Configuration

(The current control state and
what the read-write head reads

Nondeterministic Turing Machine

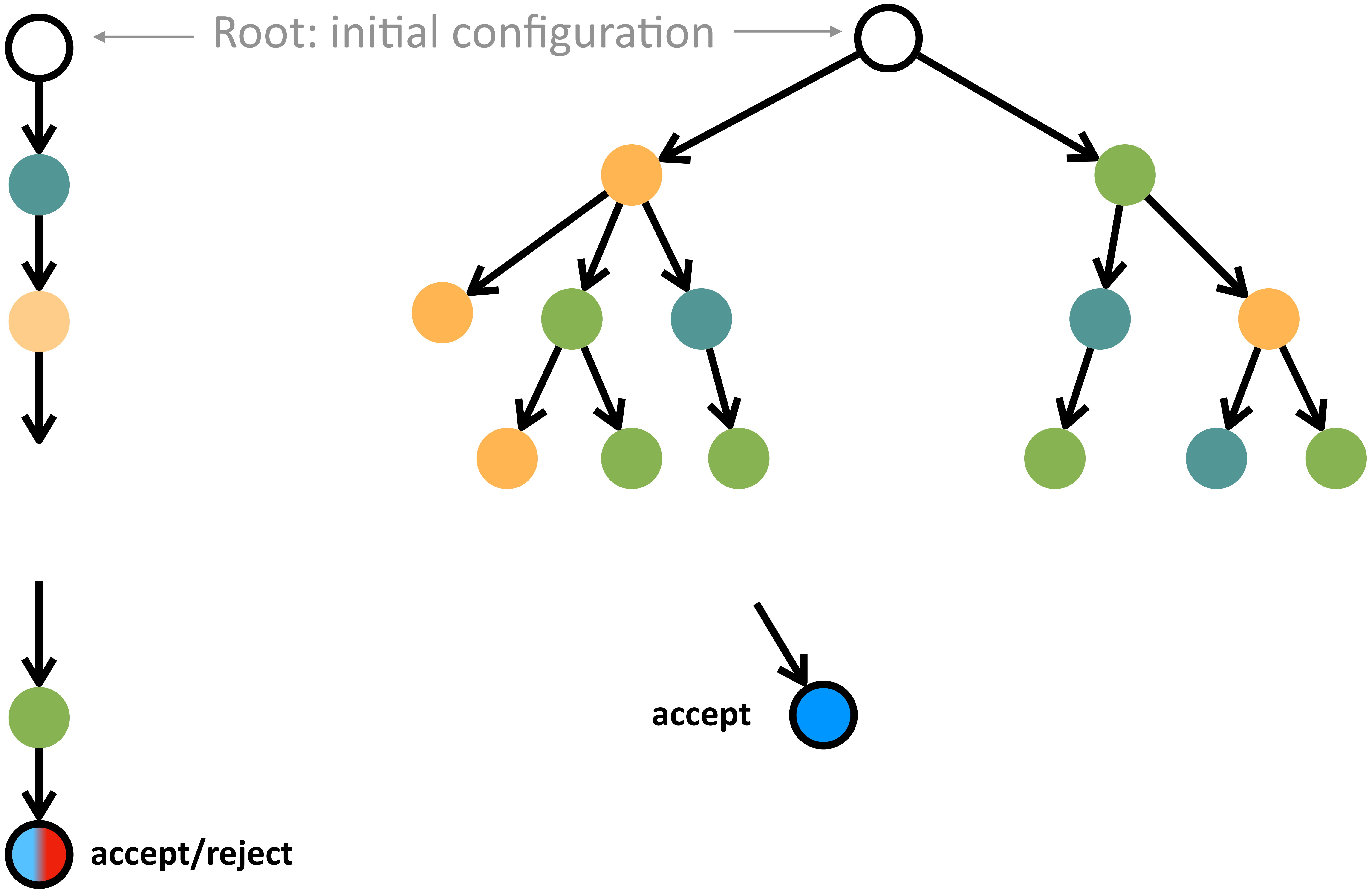


: Configuration

(The current control state and
what the read-write head reads

Nondeterministic Turing Machine

Deterministic **Nondeterministic**



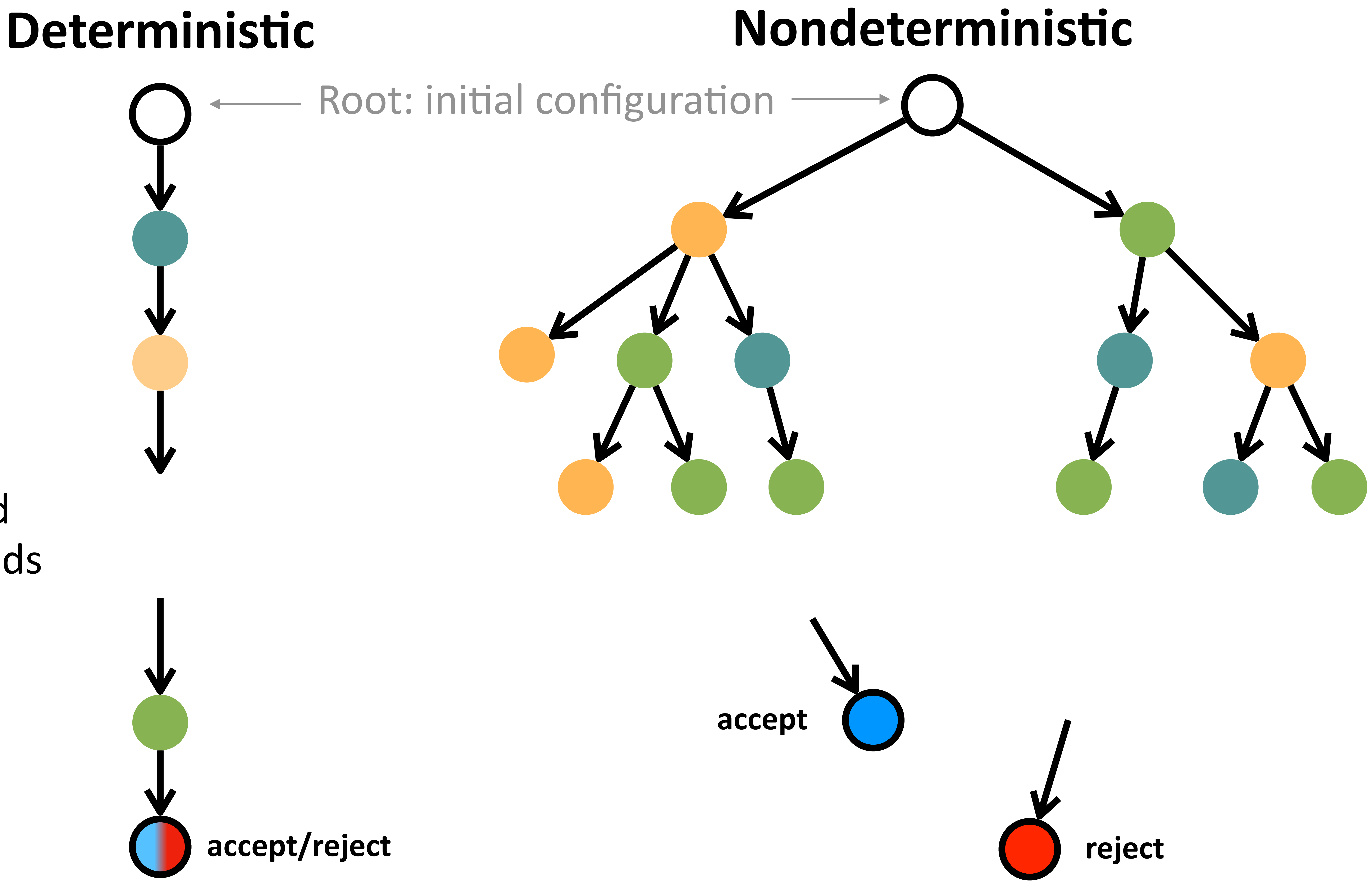
: Configuration

(The current control state and what the read-write head reads

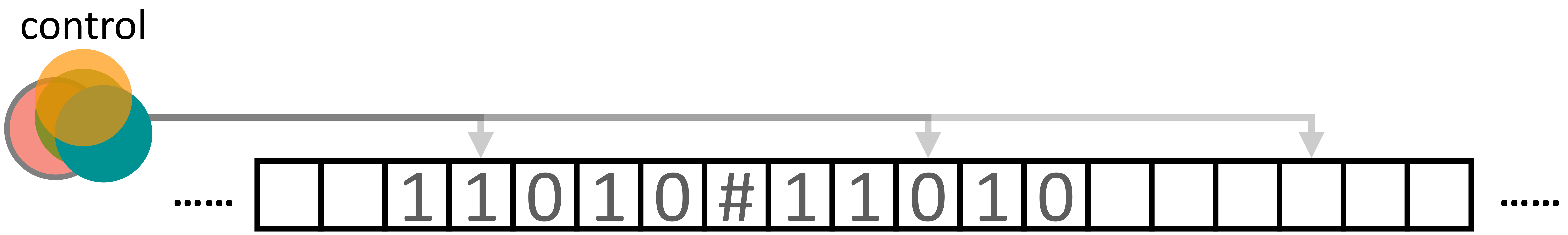
Nondeterministic Turing Machine

: Configuration

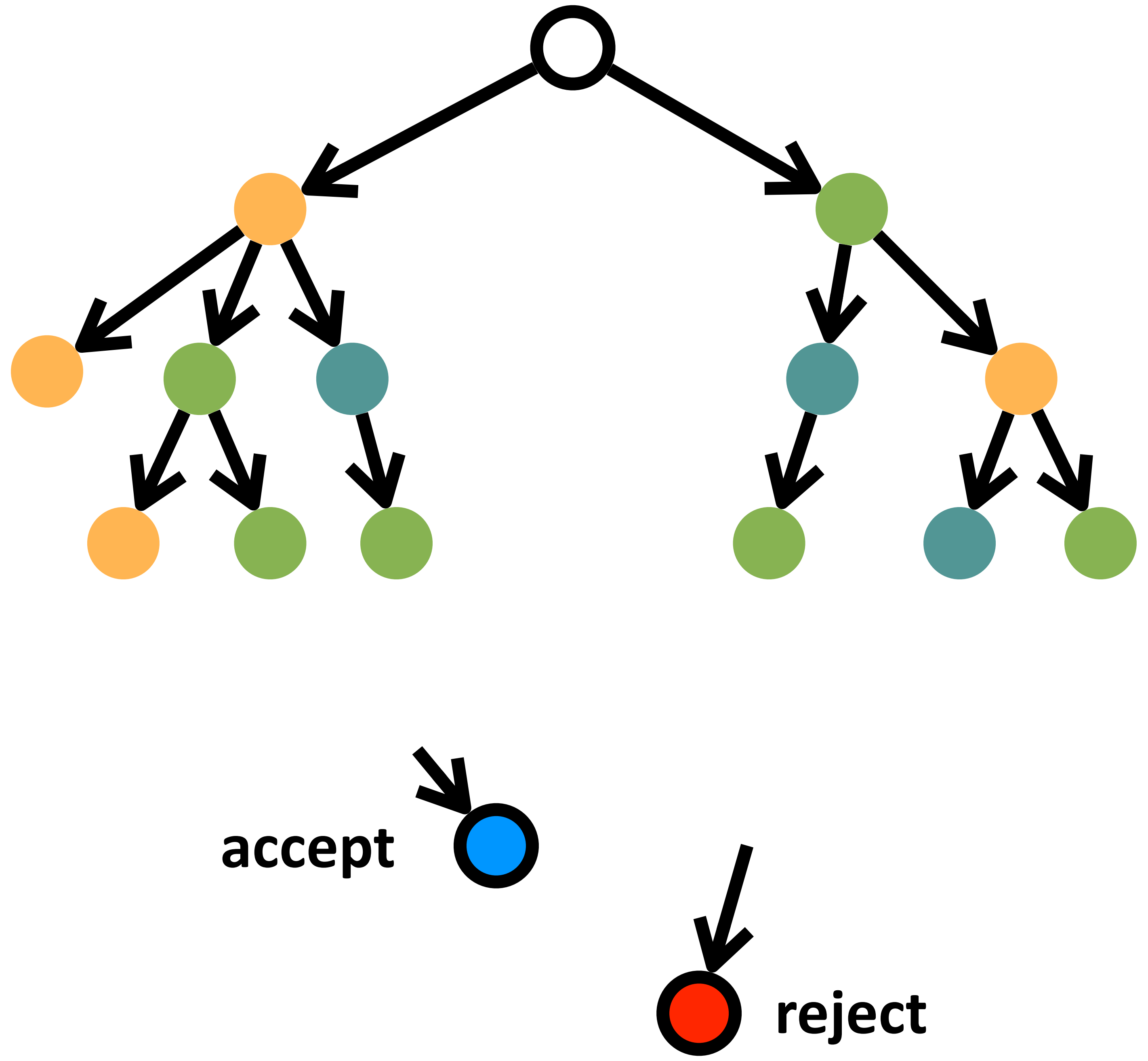
(The current control state and
what the read-write head reads



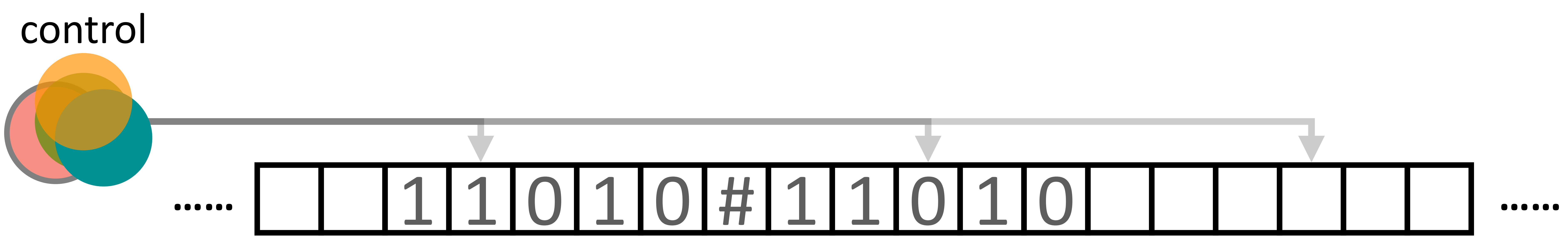
Nondeterministic Turing Machine



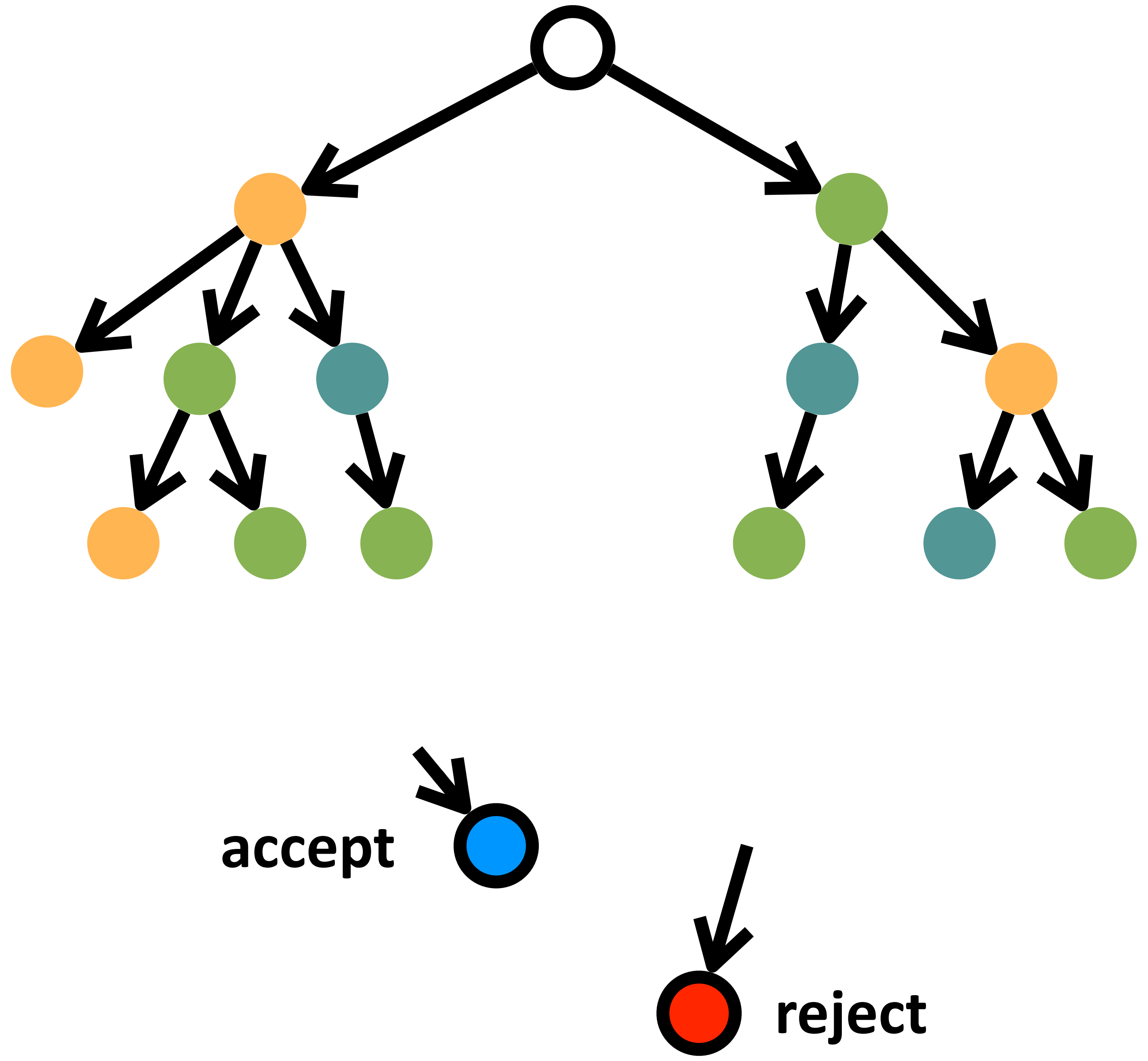
- It is like a (deterministic) Turing machine, but with **non-deterministic control**



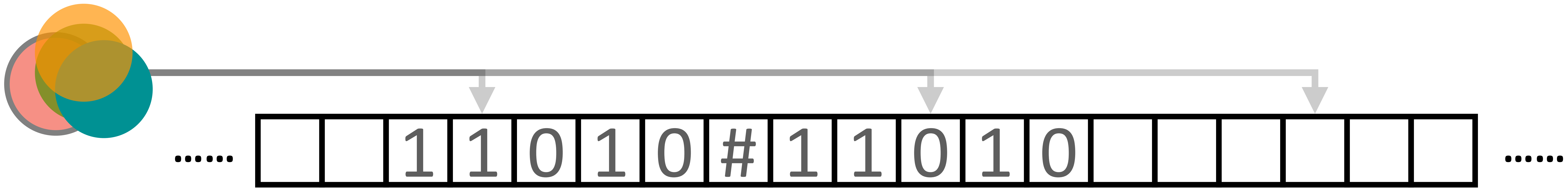
Nondeterministic Turing Machine



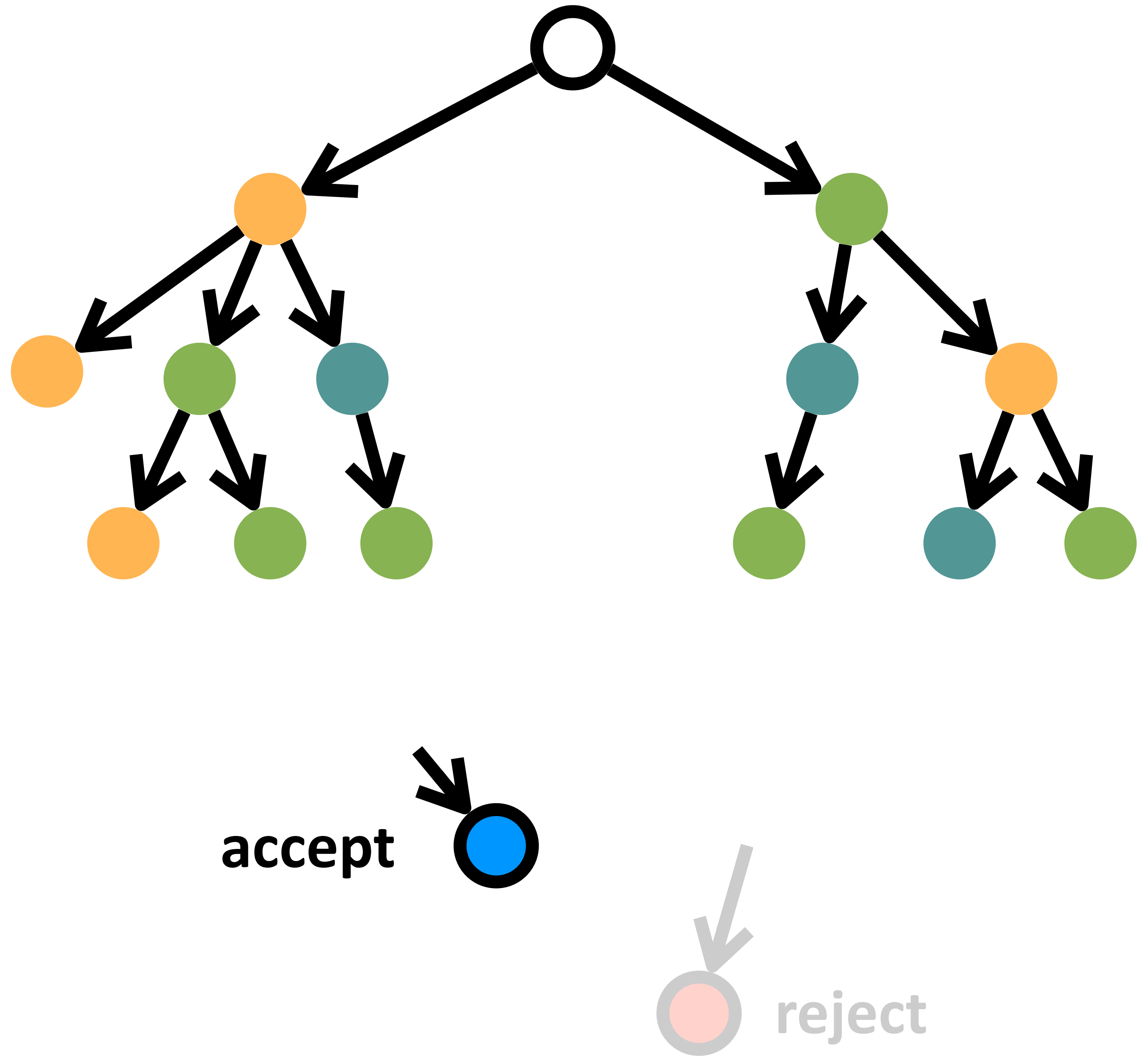
- It is like a (deterministic) Turing machine, but with **non-deterministic control**
  - Given an state and read a symbol, the non-deterministic Turing machine may enter different states



Nondeterministic Turing Machine

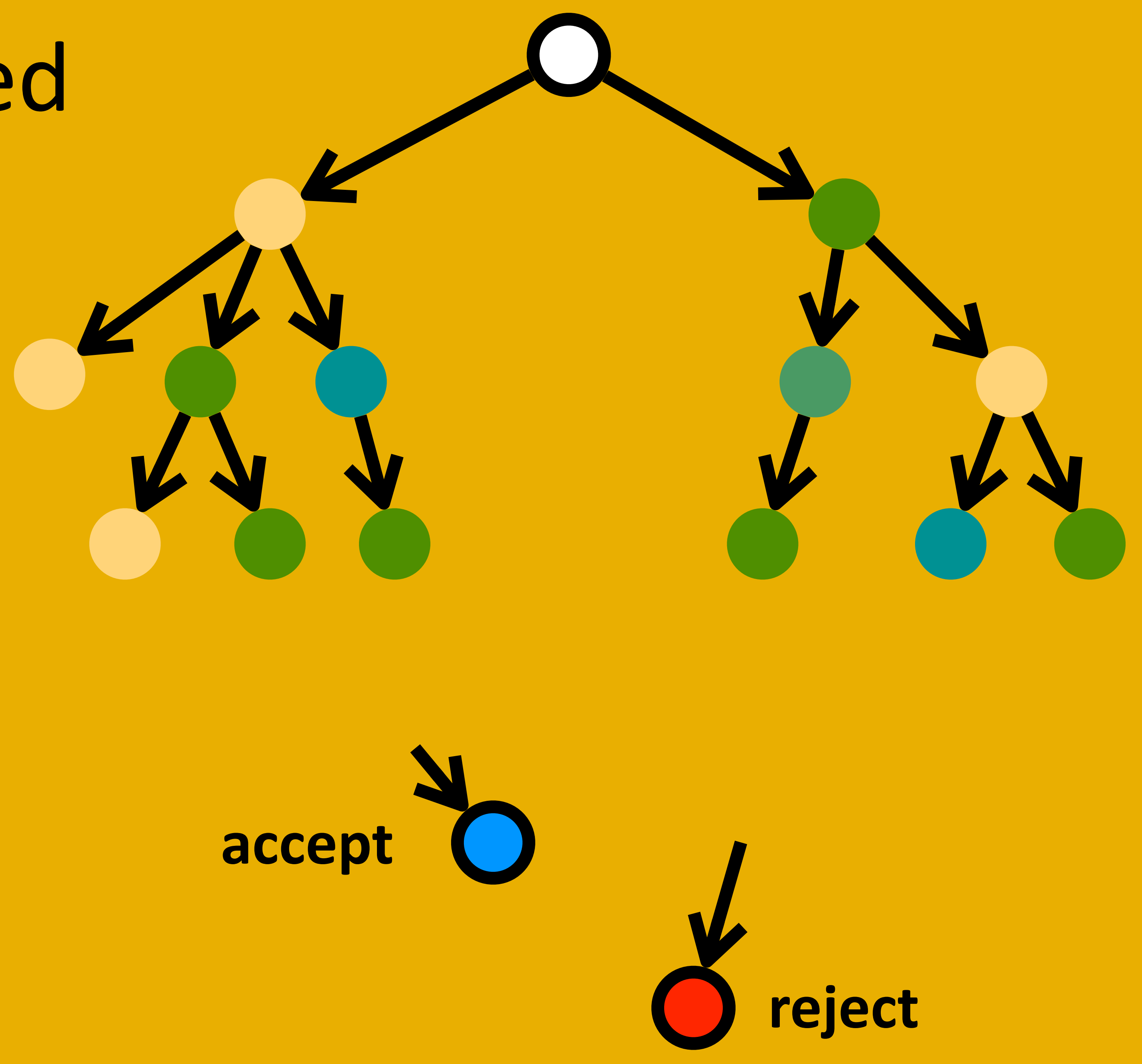


- It is like a (deterministic) Turing machine, but with non-deterministic control
  - Given an state and read a symbol, the non-deterministic Turing machine may enter different states
- The nondeterministic Turing machine ***accepts*** the input w if **some** branch of computation (i.e., a path from root to some node) leads to the **accept** state



Non-Deterministic Turing machine

- Like the (deterministic) Turing machine, but have non-deterministic behavior
- If there is **a** path ends at an accept state, the input is accepted



What is an algorithm/computer?

Church-Turing Thesis [1936]

Real world computation = Turing machine computation

Overview

- Algorithms:Turing machine, Deterministic and non-deterministic
- Formal language framework: string and language
- Time complexity
  - Input size
  - Classes **P** and **NP**
    - Polynomial time verification

A Formal Language Framework for *Problems*

- Initially, there is a string of symbols on the Turing machine tape

		1	1	0	1	0	#	1	1	0	1	0						

A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - $\{1, 01, 001, 0001, 00001, \ldots, 0^{*}1\}$
  - $\{0, 1\}^*$ Example: $010, 0, 1, 11111, \phi, \cdots$

A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - $\{1, 01, 001, 0001, 00001, \ldots, 0^*1\}$
  - $\{0, 1\}^*$
  - $\{a^k b^k \mid k \geq 0\}$ Example: aabb, ab, ϕ, aaaaabbbbb

A Formal Language Framework for *Problems*

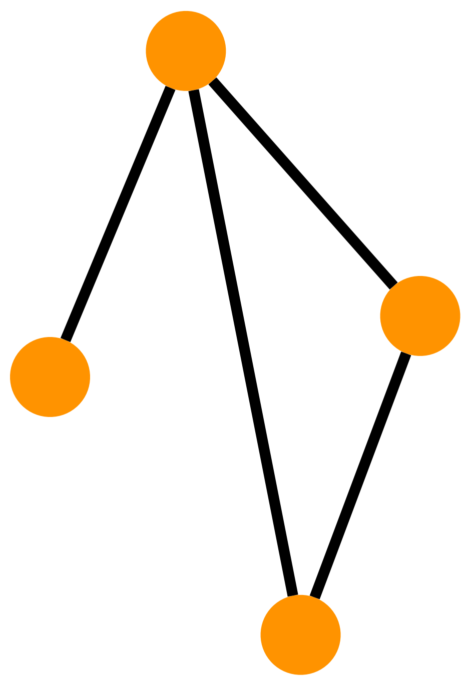
- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - {1, 01, 001, 0001, 00001, …, 0^*1}
  - $\{0, 1\}^*$
  - $\{a^k b^k \mid k \geq 0\}$
  - $\{w\#w \mid w \in \{0,1\}^*\}$ Example: 01#01, 1001#1001, ϕ, $\cdots$

A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - {1, 01, 001, 0001, 00001, …, 0^*1}
  - $\{0, 1\}^*$
  - $\{a^k b^k \mid k \geq 0\}$
  - $\{w\#w \mid w \in \{0,1\}^*\}$
  - {prime numbers} = {2, 3, 5, 7, 11, 13, 17, 19, …}

A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - $\{1, 01, 001, 0001, 00001, \ldots, 0^*1\}$
  - $\{0, 1\}^*$
  - $\{a^k b^k \mid k \geq 0\}$
  - $\{w\#w \mid w \in \{0,1\}^*\}$
  - $\{\text{prime numbers}\} = \{2, 3, 5, 7, 11, 13, 17, 19, \ldots\}$
  - $\{\langle G \rangle \mid G \text{ is connected graph}\}$



We use $\langle \cdot \rangle$ to represent an encoding of $\cdot$

(binary!)

A Formal Language Framework for *Problems*

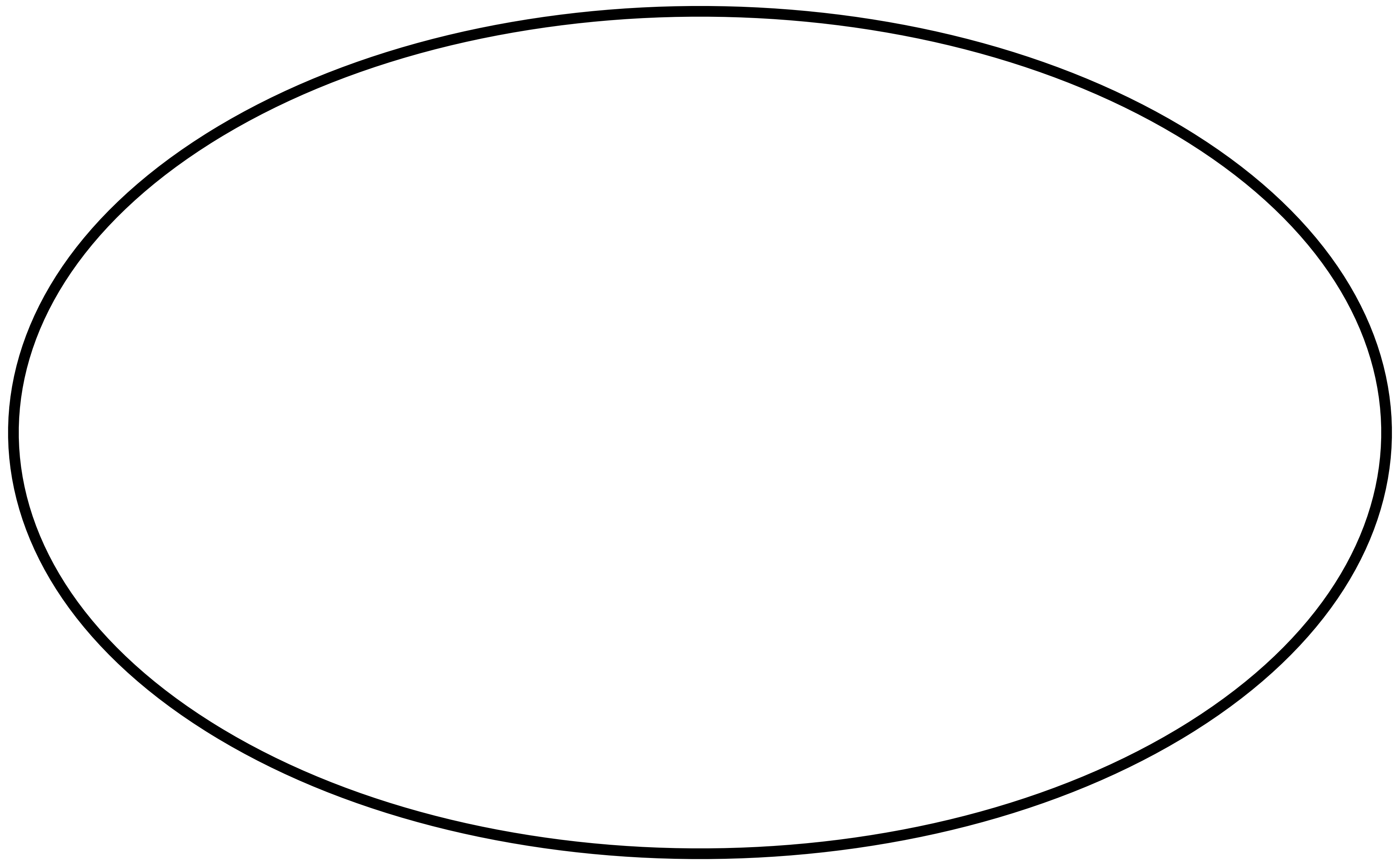
- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - {1, 01, 001, 0001, 00001, …, 0^*1}
  - $\{0, 1\}^*$
  - $\{a^k b^k \mid k \geq 0\}$
  - $\{w\#w \mid w \in \{0,1\}^*\}$
  - {prime numbers} = {2, 3, 5, 7, 11, 13, 17, 19, …}
  - $\{\langle G \rangle \mid G$ is connected graph$\}$
  - $\{\langle p \rangle \mid p$ is a polynomial with an integral root$\}$

We use $\langle \cdot \rangle$ to represent an encoding of $\cdot$
(binary!)

A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)

- Given a language L = $\{1, 01, 001, 0001, 00001, \ldots, 0^{*}1\}$ and a string w = 10110, is w in L?

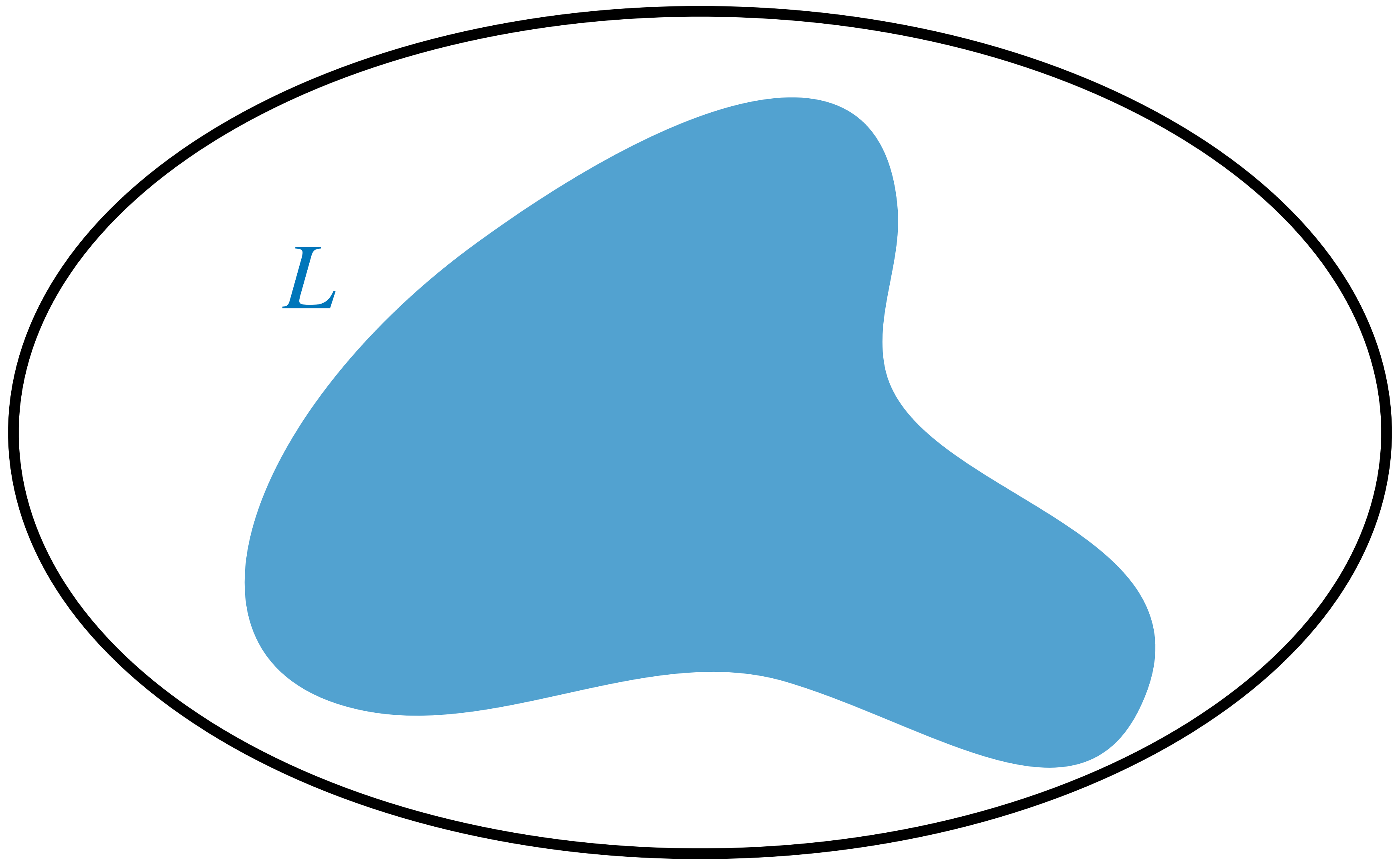


all possible strings

A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)

- Given a language $L = \{1, 01, 001, 0001, 00001, \ldots, 0^*1\}$ and a string $w = 10110$, is w in L?

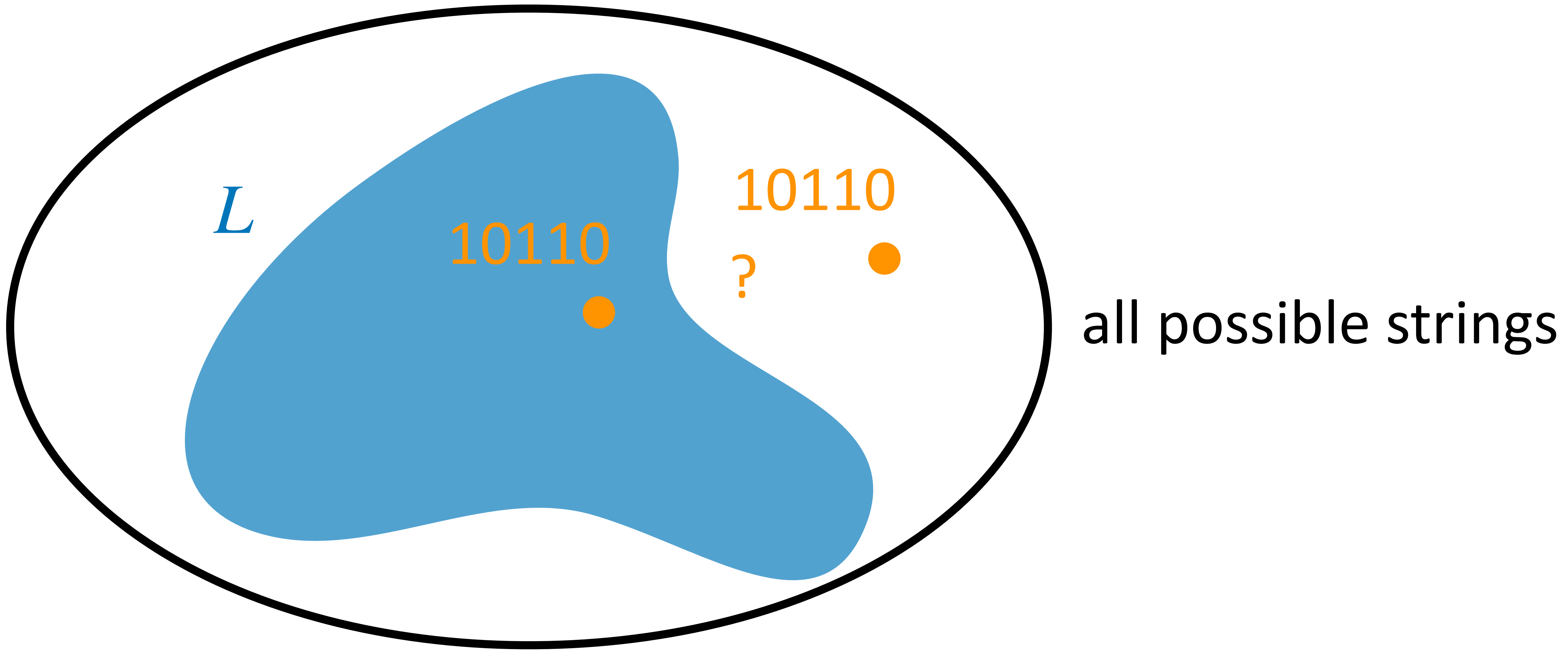


all possible strings

A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)

- Given a language L = $\{1, 01, 001, 0001, 00001, \ldots, 0^{*}1\}$ and a string w = 10110, is w in L?



A Formal Language Framework for *Problems*

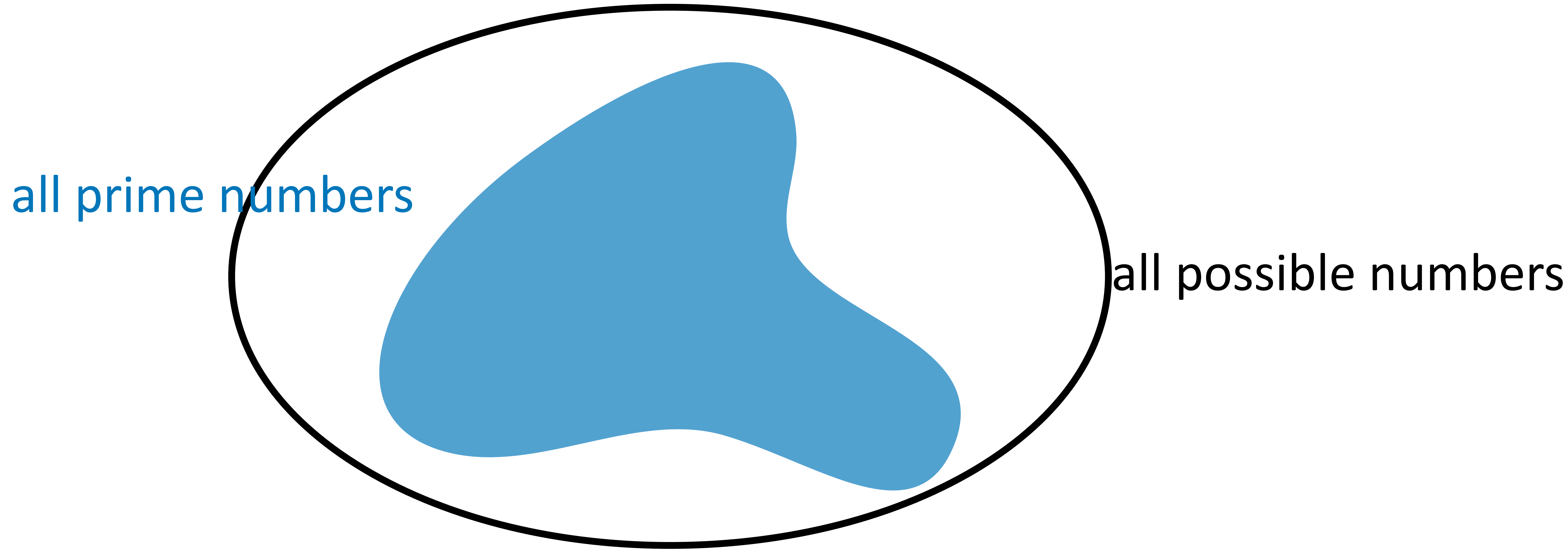
- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - Language ⇔ problem
  - String ⇔ instance

A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - Language ⇔ problem
  - String ⇔ instance
- Given a language L = $\{$prime number$\}$ and a string w = 15, is w in L?

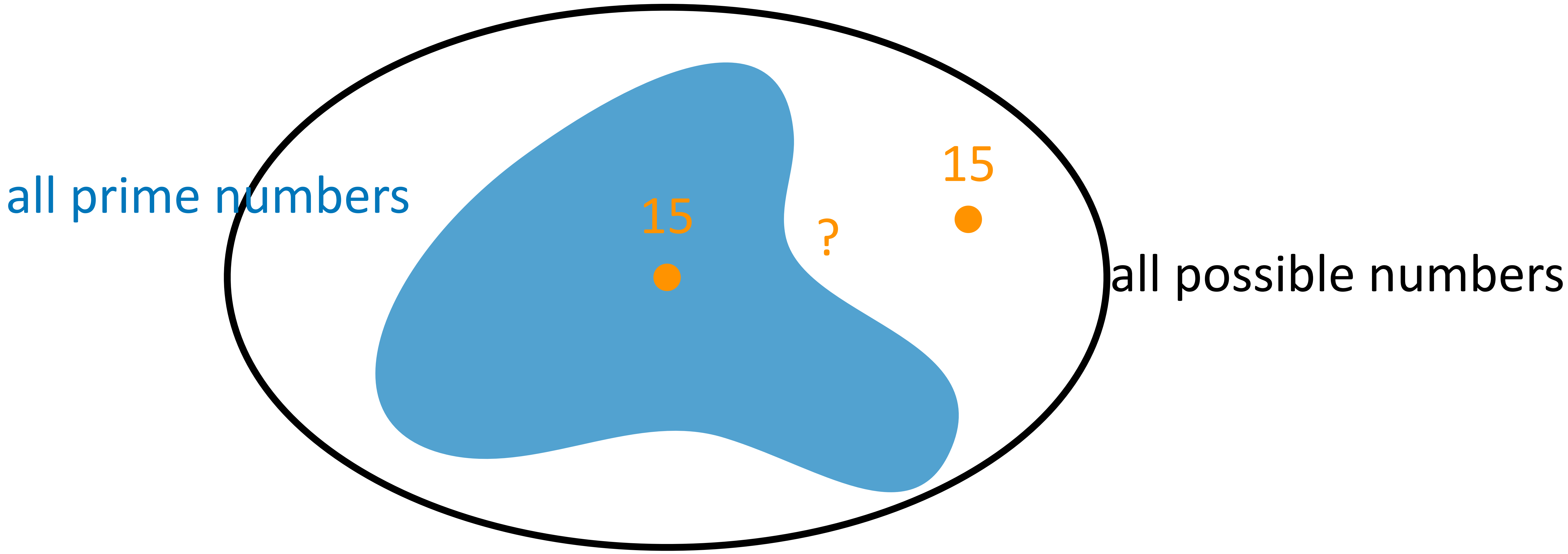
A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
    - Language ⇔ problem
    - String ⇔ instance
- Given a language $L = \{\text{prime number}\}$ and a string $w = 15$, is w in L?



A Formal Language Framework for *Problems*

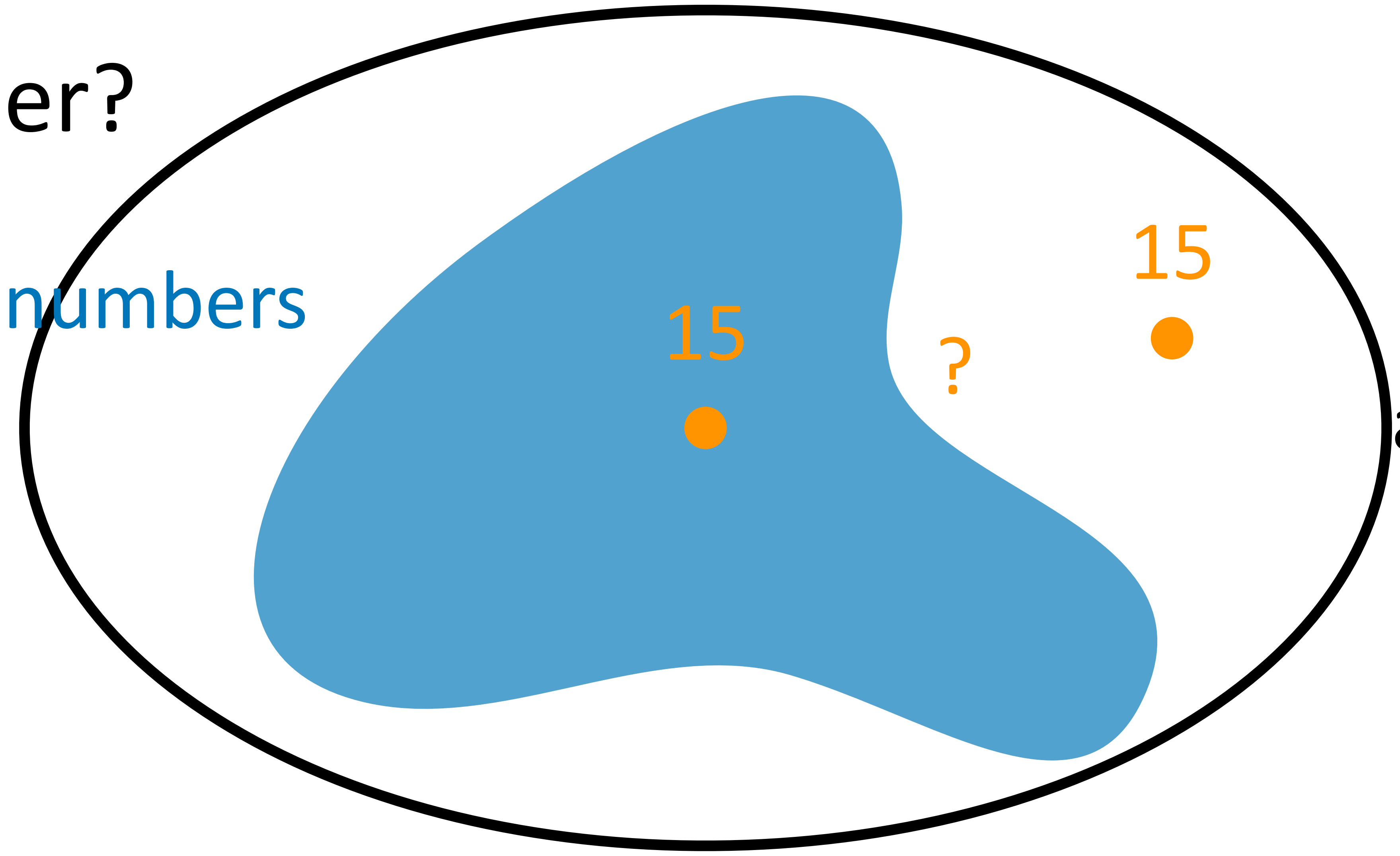
- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - Language ⇔ problem
  - String ⇔ instance
- Given a language L = $\{$prime number$\}$ and a string w = 15, is w in L?



A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - Language ⇔ problem
  - String ⇔ instance
- Given a language L = $\{$prime number$\}$ and a string w = 15, is w in L?

⇔ Is w a prime number?

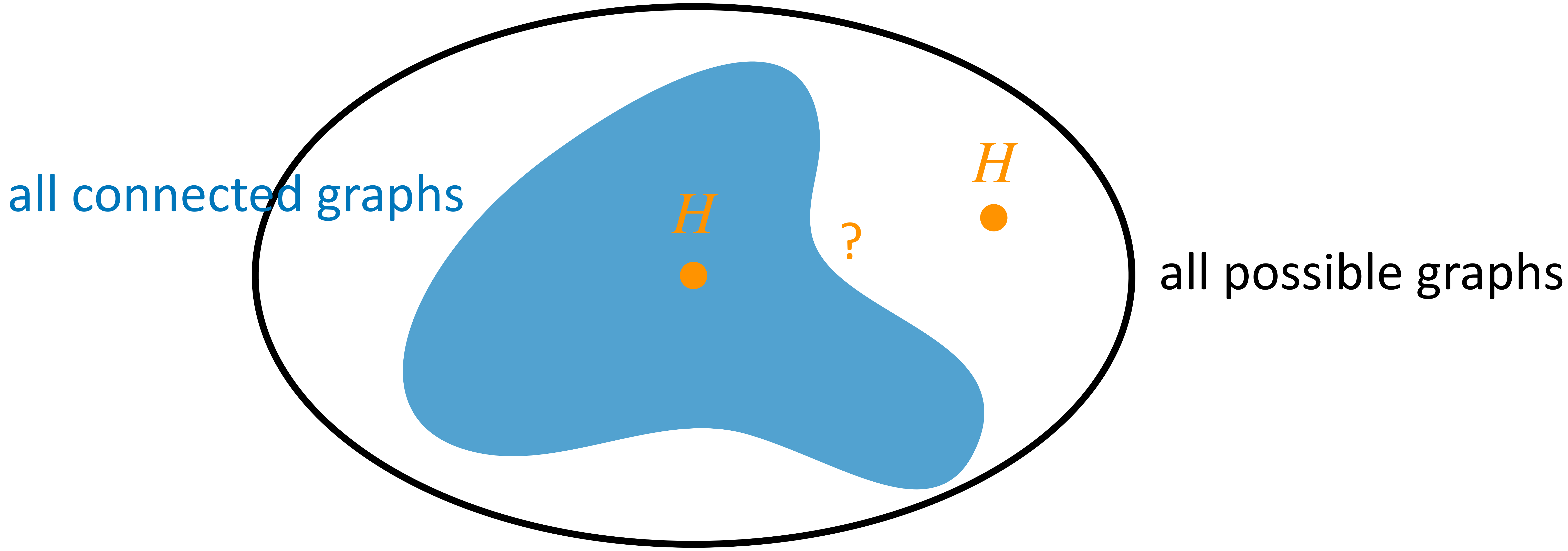


A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - Language ⇔ problem
  - String ⇔ instance
- Given a language $L = \{\langle G \rangle \mid G$ is connected graph$\}$ and a string $w = \langle H \rangle$, is w in L?

A Formal Language Framework for *Problems*

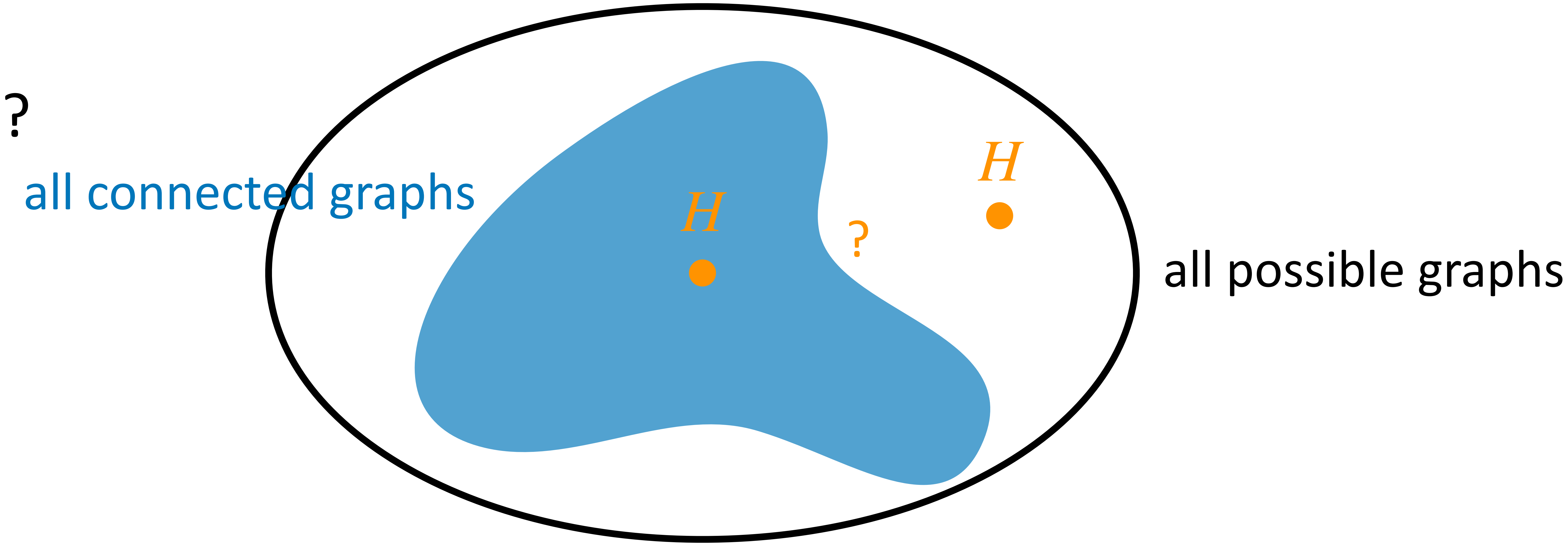
- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - Language ⇔ problem
  - String ⇔ instance
- Given a language $L = \{\langle G \rangle \mid G \text{ is connected graph}\}$ and a string $w = \langle H \rangle$, is w in L?



A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - Language ⇔ problem
  - String ⇔ instance
- Given a language $L = \{\langle G \rangle \mid G \text{ is connected graph}\}$ and a string $w = \langle H \rangle$, is w in L?

  ⇔ Is H a connected graph?

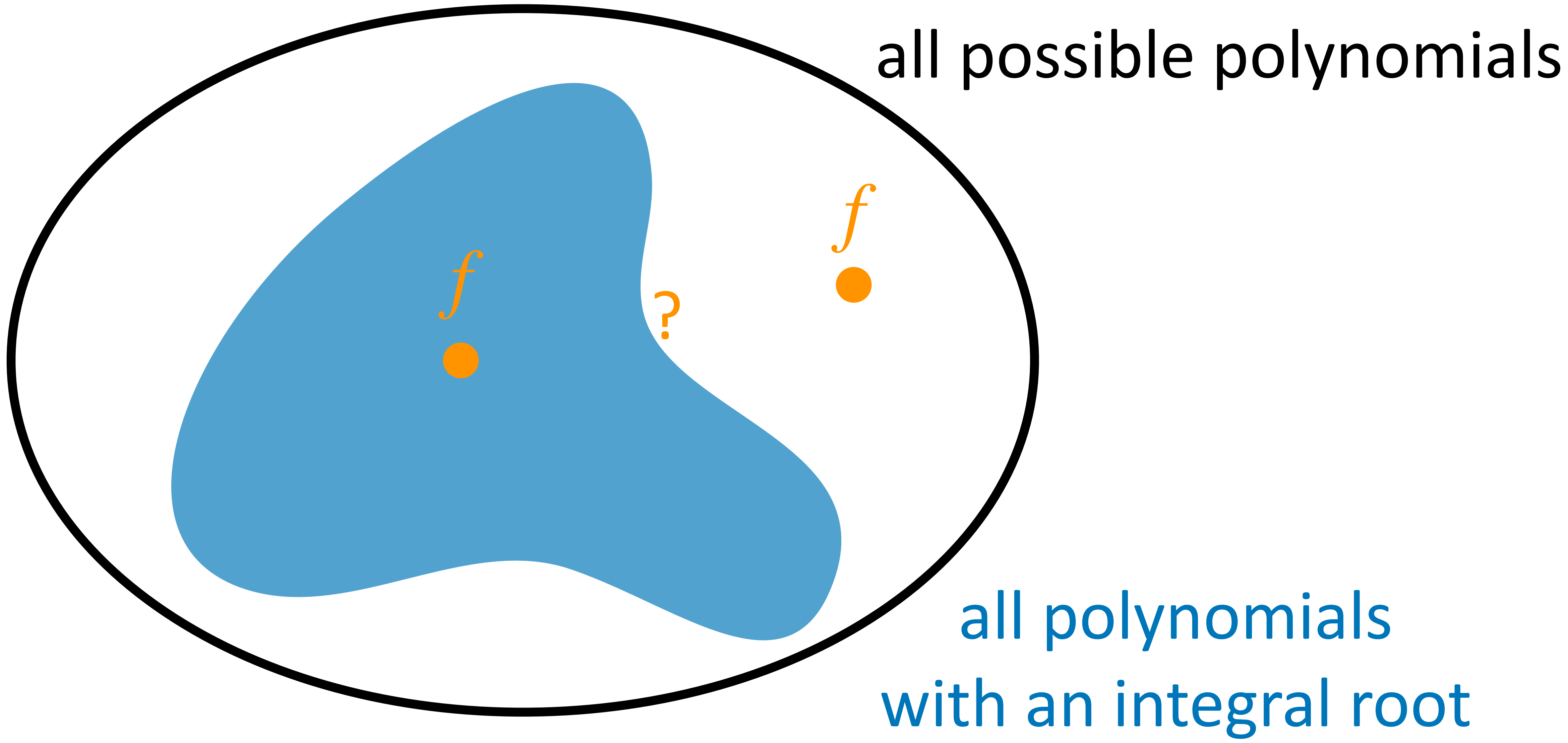


A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - Language ⇔ problem
  - String ⇔ instance
- Given a language $L = \{\langle p \rangle \mid p$ is a polynomial with an integral root$\}$ and a string $w = \langle f \rangle$, is w in L?

A Formal Language Framework for *Problems*

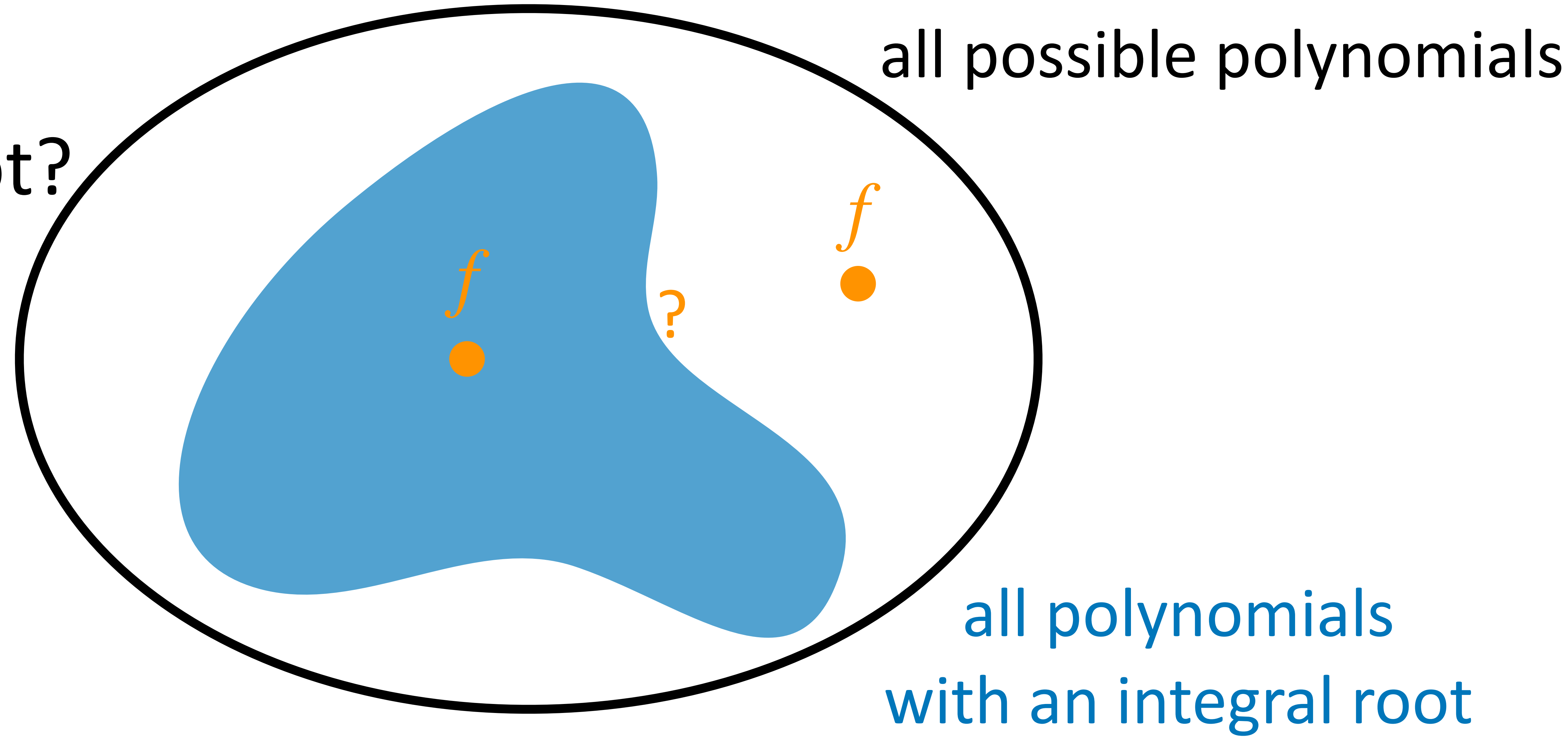
- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - Language ⇔ problem
  - String ⇔ instance
- Given a language $L = \{\langle p \rangle \mid p$ is a polynomial with an integral root$\}$ and a string $w = \langle f \rangle$, is w in L?



A Formal Language Framework for *Problems*

- Definition: A ***language*** is a set of strings (that satisfy some constraints)
  - Language ⇔ problem
  - String ⇔ instance
- Given a language $L = \{\langle p \rangle \mid p$ is a polynomial with an integral root$\}$ and a string $w = \langle f \rangle$, is w in L?

  ⇔ Is f a polynomial with an integral root?

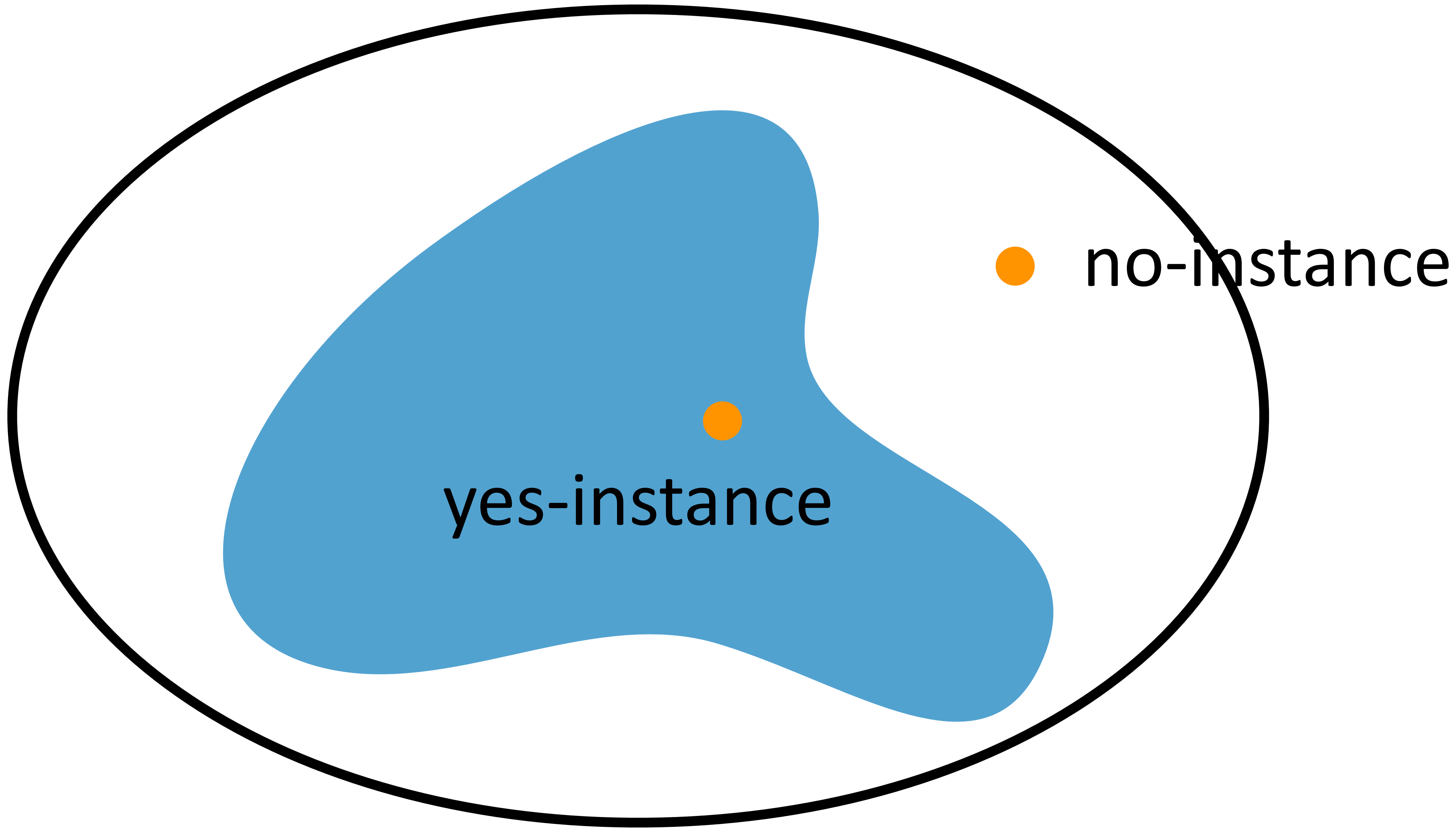


Yes-Instance and No-Instance

- Consider a language A. For any instance w, either $w \in A$ or $w \notin A$.
  - If $w \in A$, we call w a **yes-instance** (that is, a correct algorithm should return *accept* or *yes*)
  - If $w \notin A$, we call w a **no-instance** (that is, a correct algorithm should return *reject* or *no*)

Yes-Instance and No-Instance

- Consider a language A. For any instance w, either $w \in A$ or $w \notin A$.
  - If $w \in A$, we call w a **yes-instance** (that is, a correct algorithm should return *accept* or *yes*)
  - If $w \notin A$, we call w a **no-instance** (that is, a correct algorithm should return *reject* or *no*)



What Happened

- Following the vein of Turing machine concept, a **language** is a set of **strings**
  - Language ⇔ **problem**
  - String ⇔ **instance**
  - Asking if a string is in a language
    ⇔ if the instance satisfies the property that the problem asks

- Given a problem/language, a instance/string is a
  - **yes-instance**: an instance that satisfies the property that the problem asks
  - **no-instance**: an instance that does not satisfy the property that the problem asks

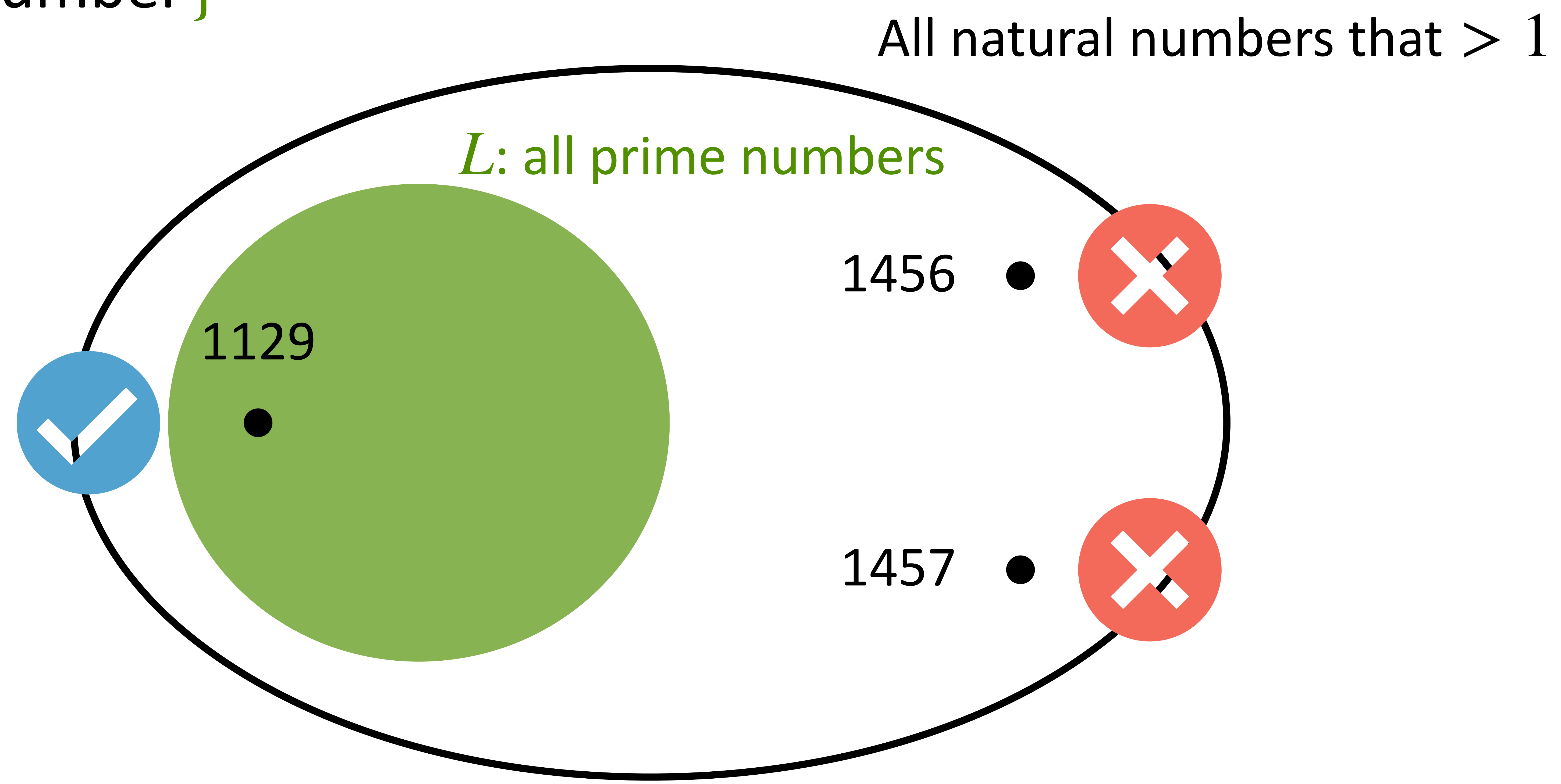
Turing-Decidable Language

Turing-Decidable Language

- A language L is (Turing-)**decidable** if **some Turing machine *decides* it**
    - The Turing machine accepts all strings in L and rejects all strings not in L

Turing-Decidable Language

- A language L is (Turing-)**decidable** if **some Turing machine *decides* it**
  - The Turing machine accepts all strings in L and rejects all strings not in L
- Ex: L = $\{$prime number$\}$



Decision Problems and Optimization Problems

- **Decision** problems:

- **Optimization** problems:

Decision Problems and Optimization Problems

- **Decision** problems: Given a problem and an input of the problem, asking if we feed this input to the problem, the answer is *yes* or *no*

- **Optimization** problems:

Decision Problems and Optimization Problems

- **Decision** problems: Given a problem and an input of the problem, asking if we feed this input to the problem, the answer is *yes* or *no*
  - Ex: Partition problem
- **Optimization** problems:

Decision Problems and Optimization Problems

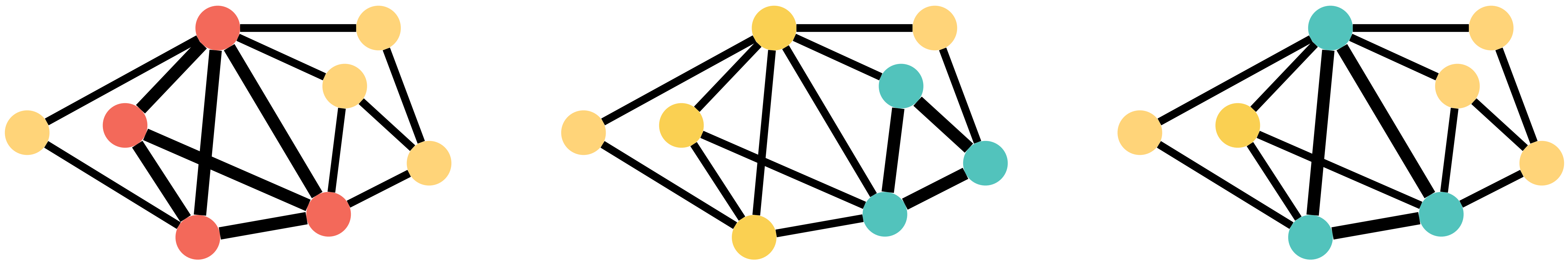
- **Decision** problems: Given a problem and an input of the problem, asking if we feed this input to the problem, the answer is *yes* or *no*
  - Ex: Partition problem
- **Optimization** problems: finding the *best* solution among all feasible solutions

Decision Problems and Optimization Problems

- **Decision** problems: Given a problem and an input of the problem, asking if we feed this input to the problem, the answer is *yes* or *no*
  - Ex: Partition problem
- **Optimization** problems: finding the *best* solution among all feasible solutions
  - Feasible solution: a solution that satisfies the requirement but probably not the best

Decision Problems and Optimization Problems

- **Decision** problems: Given a problem and an input of the problem, asking if we feed this input to the problem, the answer is *yes* or *no*
    - Ex: Partition problem
- **Optimization** problems: finding the *best* solution among all feasible solutions
    - Feasible solution: a solution that satisfies the requirement but probably not the best
        - A subgraph which is a clique is not necessary the one that contains minimum number of vertices



Decision Problems and Optimization Problems

- **Decision** problems: Given a problem and an input of the problem, asking if we feed this input to the problem, the answer is *yes* or *no*
  - Ex: Partition problem
- **Optimization** problems: finding the *best* solution among all feasible solutions
  - Feasible solution: a solution that satisfies the requirement but probably not the best
    - A subgraph which is a clique is not necessary the one that contains minimum number of vertices
    - Minimization or maximization

Decision Problems and Optimization Problems

- **Decision** problems: Given a problem and an input of the problem, asking if we feed this input to the problem, the answer is *yes* or *no*
  - Ex: Partition problem
- **Optimization** problems: finding the *best* solution among all feasible solutions
  - Feasible solution: a solution that satisfies the requirement but probably not the best
    - A subgraph which is a clique is not necessary the one that contains minimum number of vertices
    - Minimization or maximization
      - Ex: Minimum vertex cover or Maximum independent set

Optimization? An Equivalent Decision Problem

Optimization? An Equivalent Decision Problem

- The classes **P** and **NP** are both define on decision problems. How do we classify optimization problems?

Optimization? An Equivalent Decision Problem

- The classes **P** and **NP** are both define on decision problems. How do we classify optimization problems?
- We can recast an optimization problem as a decision problem that is no harder!

Optimization? An Equivalent Decision Problem

- The classes **P** and **NP** are both define on decision problems. How do we classify optimization problems?
- We can recast an optimization problem as a decision problem that is no harder!
  - Optimization problem: we want to minimize/maximize…

Optimization? An Equivalent Decision Problem

- The classes **P** and **NP** are both define on decision problems. How do we classify optimization problems?
- We can recast an optimization problem as a decision problem that is no harder!
  - Optimization problem: we want to minimize/maximize…
  - Equivalent decision version problem: we want to find a solution with cost at most/least k

Optimization? An Equivalent Decision Problem

- The classes **P** and **NP** are both define on decision problems. How do we classify optimization problems?
- We can recast an optimization problem as a decision problem that is no harder!
  - Optimization problem: we want to minimize/maximize…
  - Equivalent decision version problem: we want to find a solution with cost at most/least k
    - k is an additional parameter

Overview

- Algorithms:Turing machine, Deterministic and non-deterministic

- Formal language framework: string and language

- Time complexity
  - Input size
  - Classes **P** and **NP**
    - Polynomial time verification

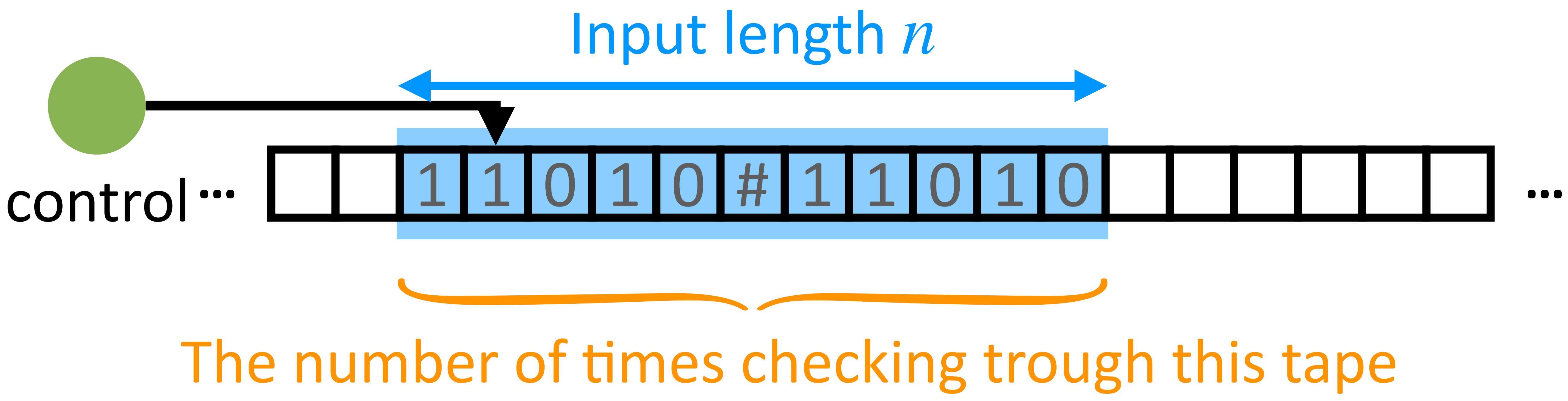
Time Complexity

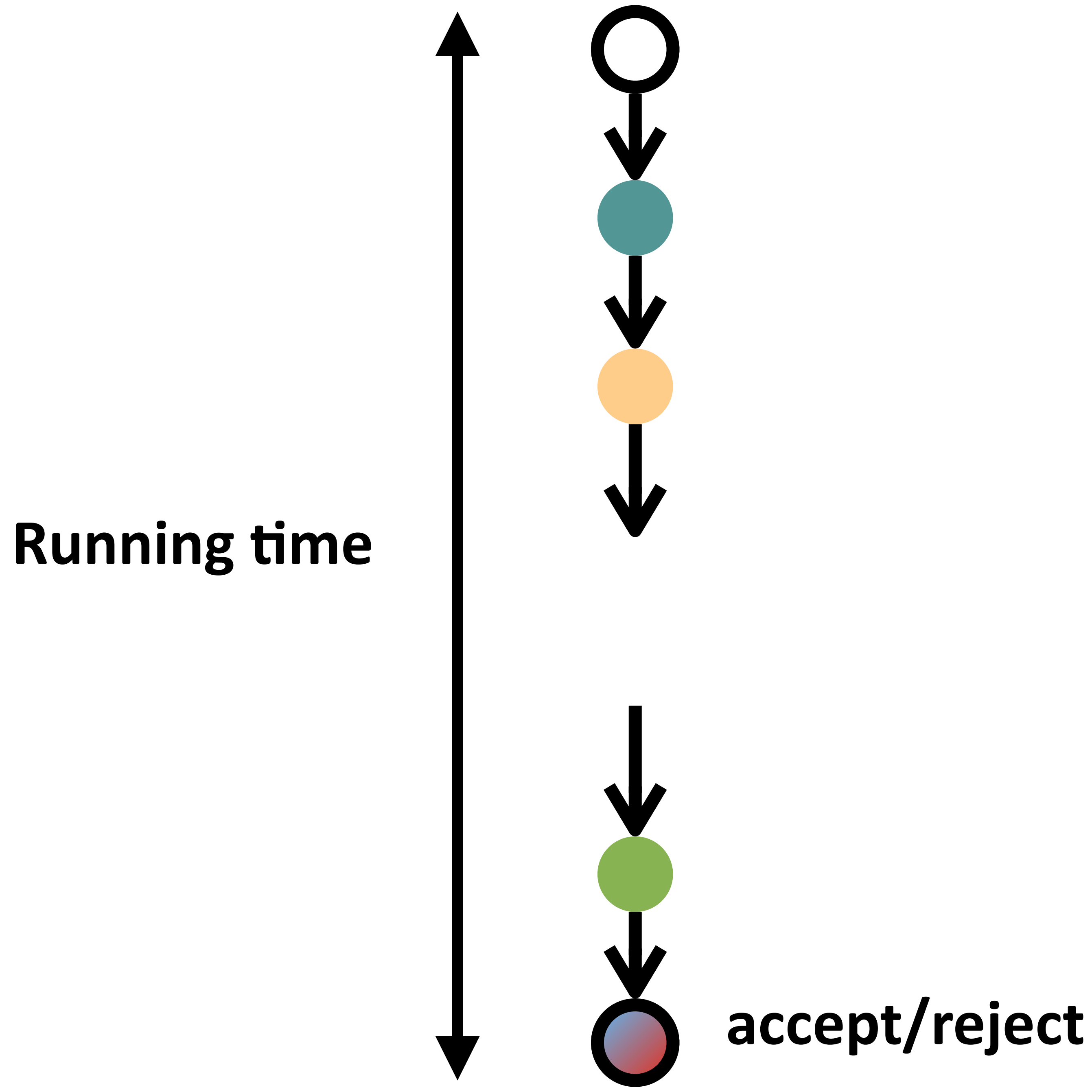
Time Complexity

- Definition: Let M be a deterministic Turing machine that accepts or rejects all inputs. The ***running time*** or ***time complexity*** of M is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.

Time Complexity

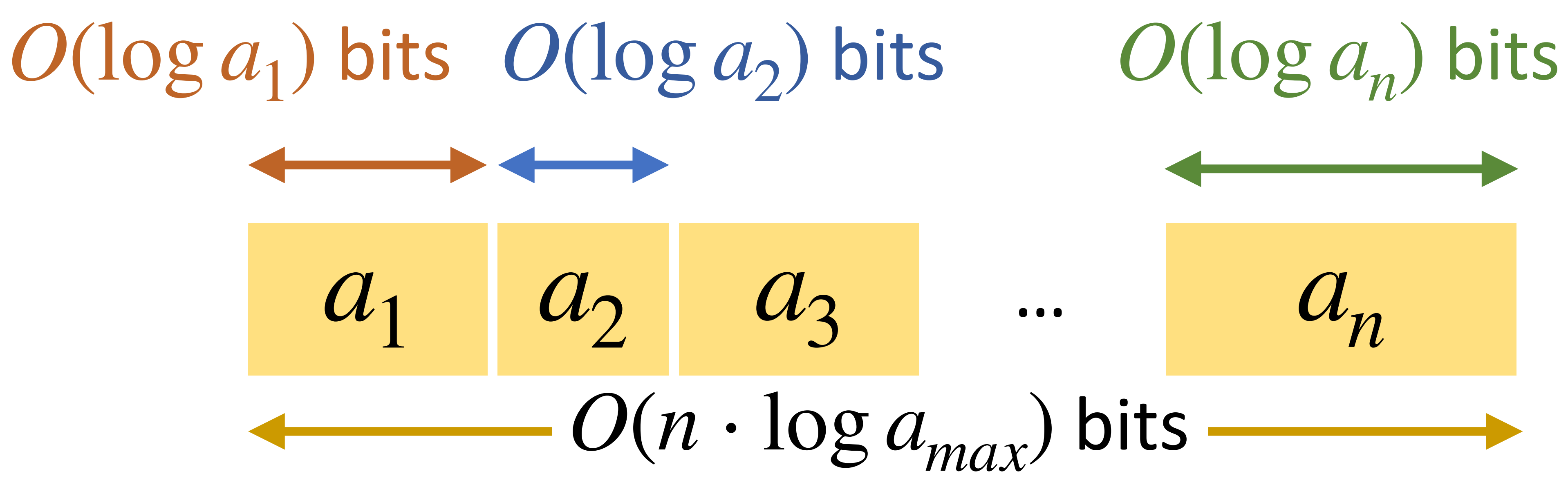
- Definition: Let M be a deterministic Turing machine that accepts or rejects all inputs. The ***running time*** or ***time complexity*** of M is the function $f: \mathcal{N} \rightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.
  - Ex: $f(n) = O(n^2)$
  - Ex: $f(n) = O(2^n)$





Input Size

- PARTITION $= \{\langle S = \{a_1, a_2, \cdots, a_n\}\rangle \mid S$ can be partitioned into two equal-sum subsets$\}$: a string w encoding the elements in S
- Input size: $O(n \log a_{max})$ bits
  - n: number of items in the set S
  - a_{max}: maximum value of the items in S. That is, for all $1 \leq i \leq n$, $a_i \leq a_{max}$.



Input Size

- CONNECT = $\{\langle G \rangle \mid G$ is connected graph$\}$: a string w encoding a graph G
- Input size: using binary encoding to encode vertices and edges in G
  - Use an adjacency array, the input size is $O(|V|\log|V| + |V|^2) = O(|V|^2)$ bits
  - Use an adjacency list, the input size is $O(|V|\log|V| + |E| \cdot 2\log|V|) = O(|V|^2\log|V|)$ bits
  - The $O(\log|V|)$ bits are for encoding the vertex ID

$v_1, v_2, v_3, \cdots, v_n$	$e_1 = (v_{i_1}, v_{j_1}), e_2 = (v_{i_2}, v_{j_2}), \cdots, e_m = (v_{i_m}, v_{j_m})$
$\longleftarrow O(|V|\log|V|) \longrightarrow$	$\longleftarrow O(|E| \cdot 2\log|V|) \longrightarrow$

Input Size

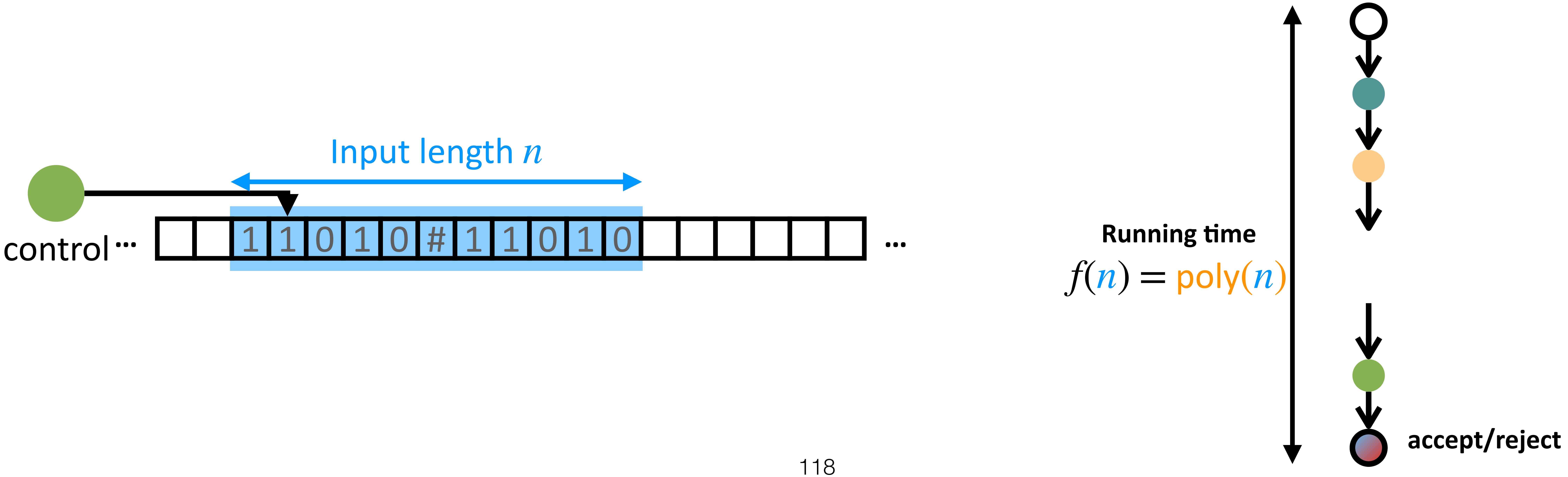
- PRIME = {prime number}: a string w representing number n
- Input size?

Overview

- Algorithms:Turing machine, Deterministic and non-deterministic
- Formal language framework: string and language
- Time complexity
  - Input size
  - Classes **P** and **NP**
    - Polynomial time verification

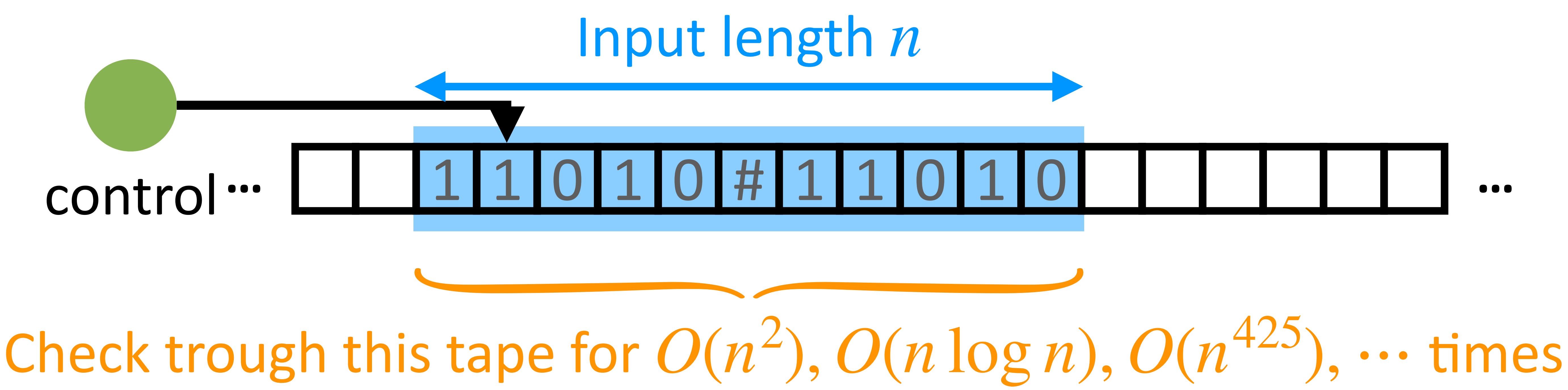
The Class **P**

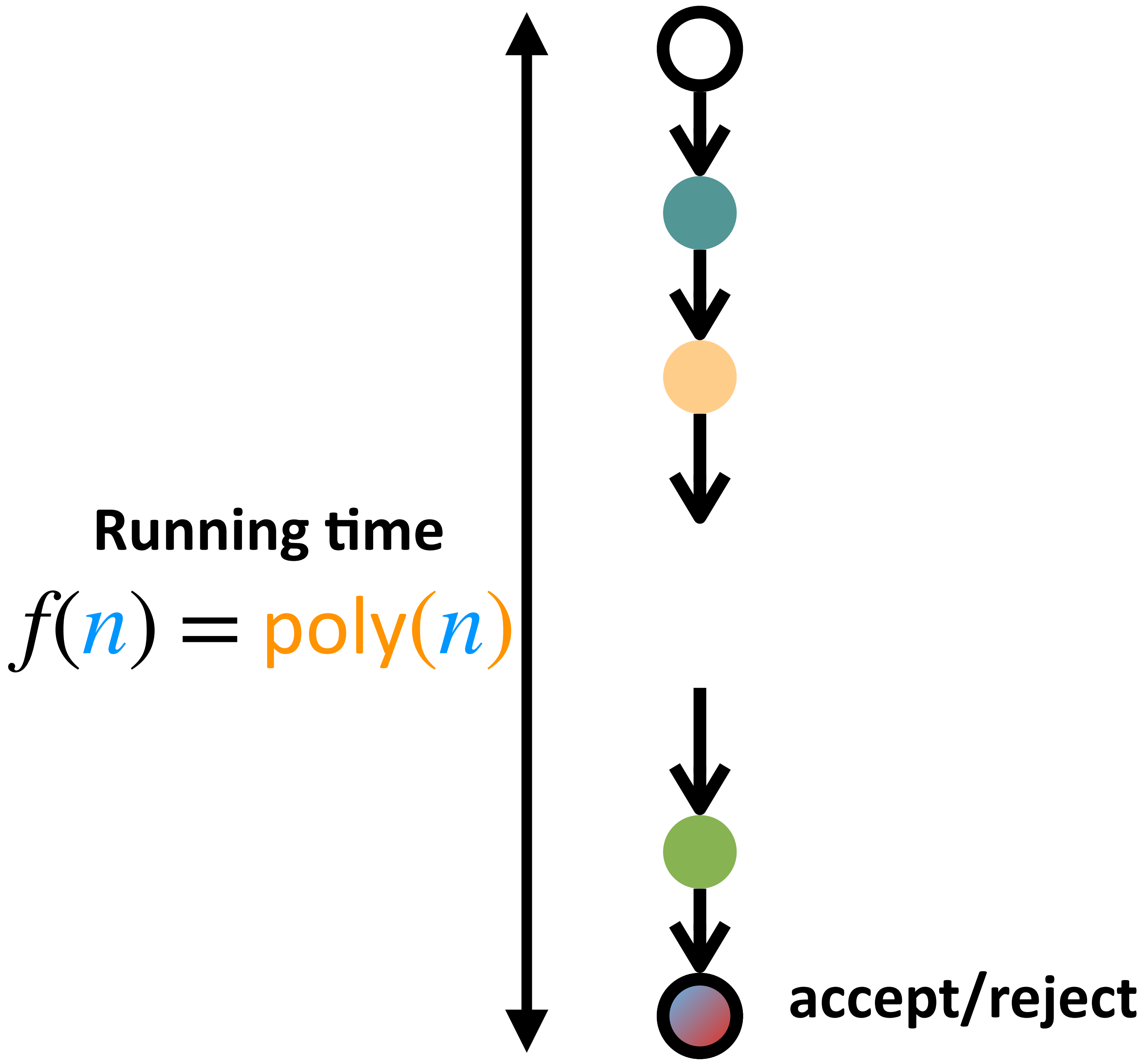
- Definition: **P** is the class of languages that are can be accepted or rejected in polynomial time by a deterministic single-tape Turing machine.



The Class **P**

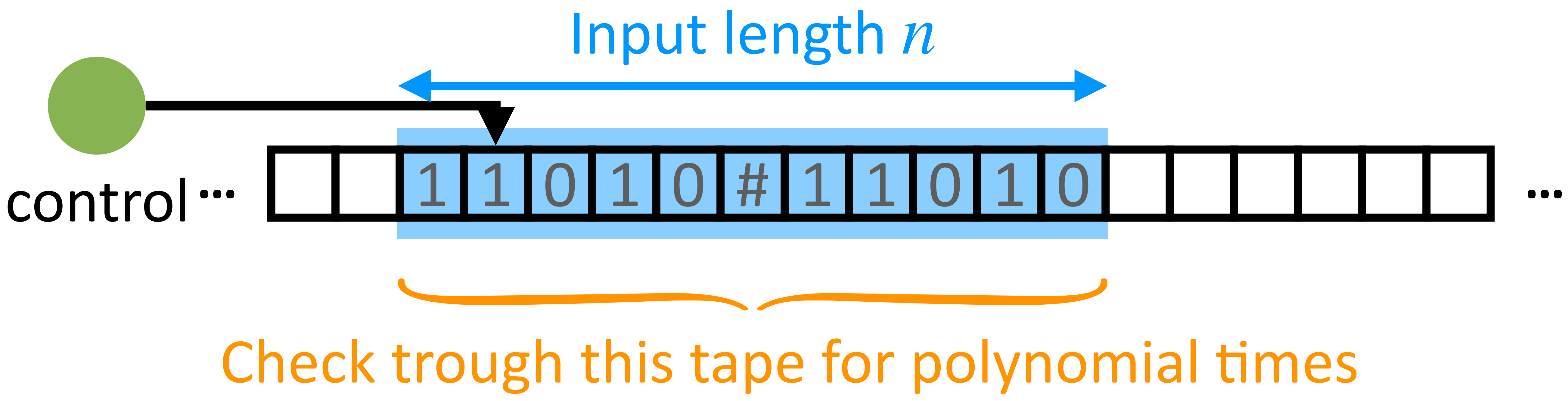
- Definition: **P** is the class of languages that are can be accepted or rejected in polynomial time by a deterministic single-tape Turing machine.
  - Ex: $O(n^2), O(n \log n), O(n^{425}), \cdots$

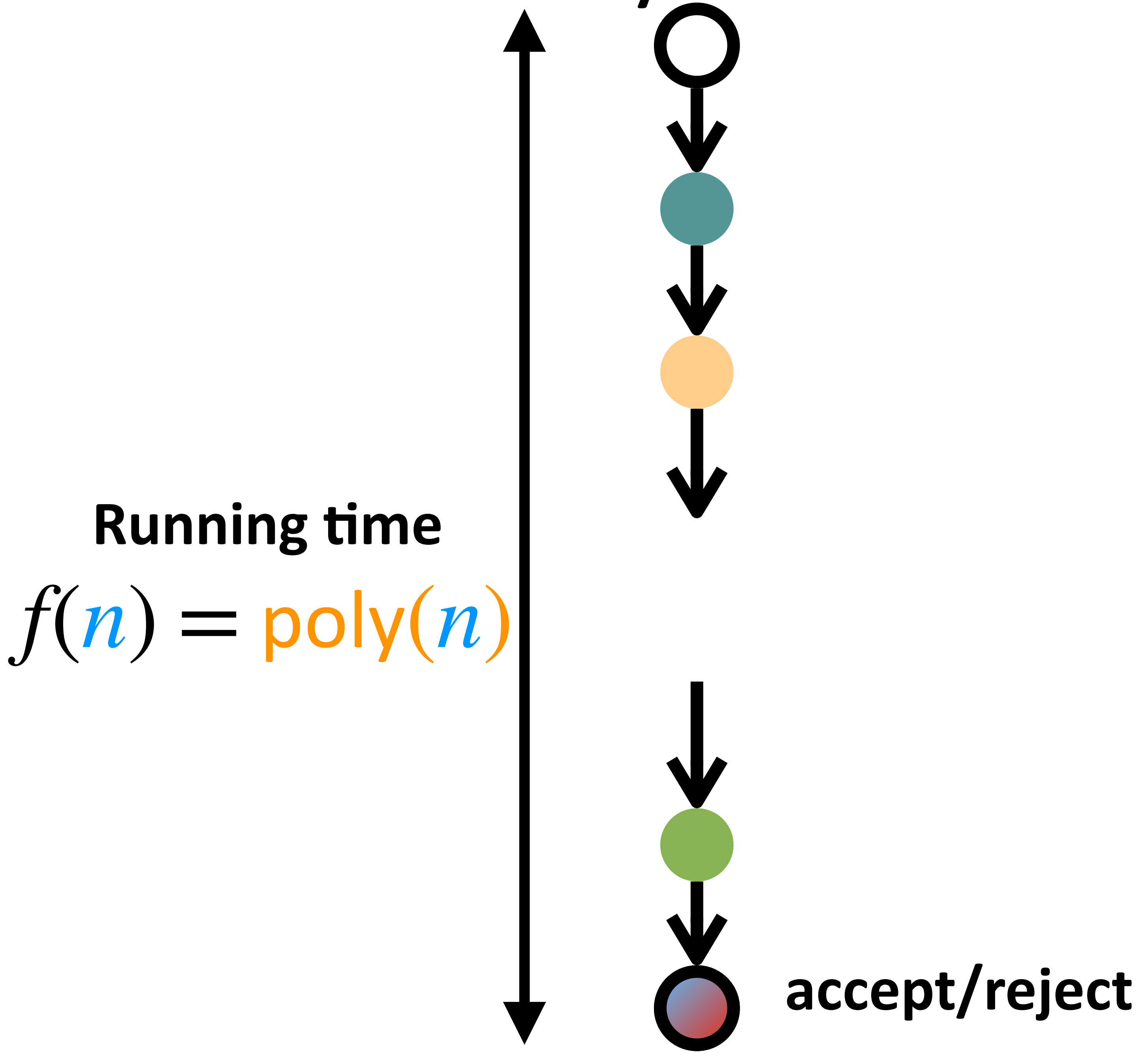




The Class **P**

- Definition: **P** is the class of languages that are can be accepted or rejected in polynomial time by a deterministic single-tape Turing machine.
  - Ex: $O(n^2), O(n \log n), O(n^{425}), \cdots$
  - **P** roughly corresponds to the class of problems that are realistically solvable on a computer



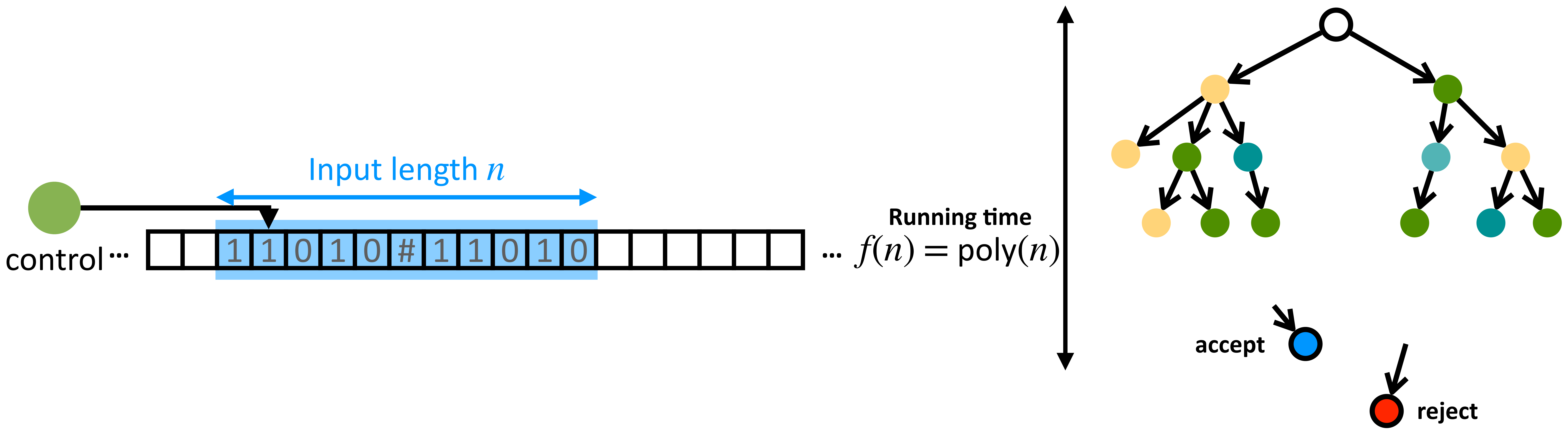


The Class **NP**

The Class **NP**

The Class **NP**

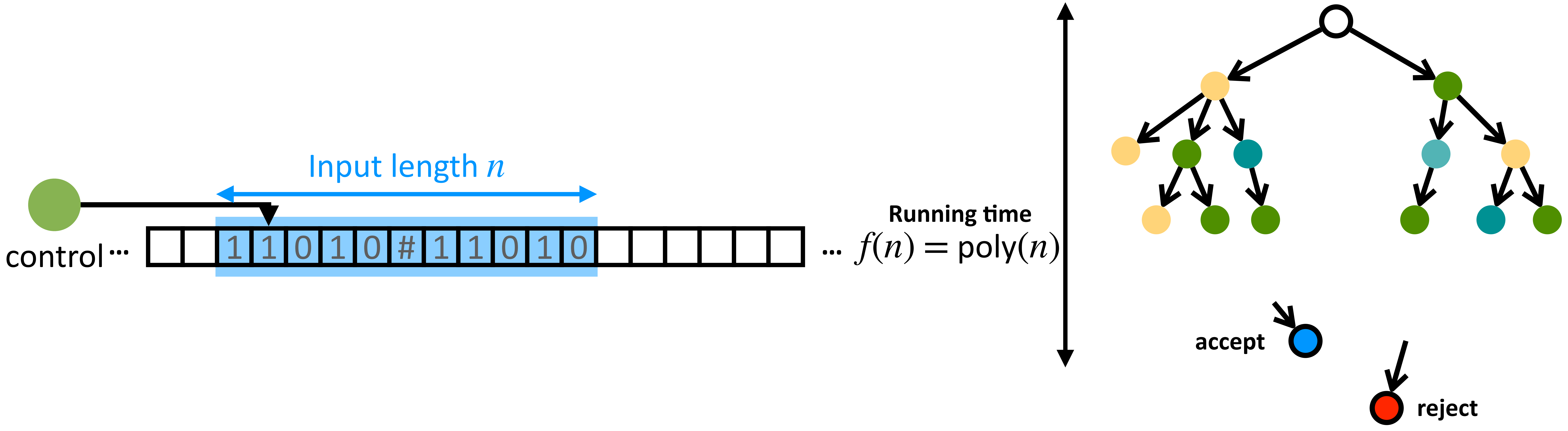
- Similarly, we can define the running time of a non-deterministic Turing machine N



The Class **NP**

- Similarly, we can define the running time of a non-deterministic Turing machine N

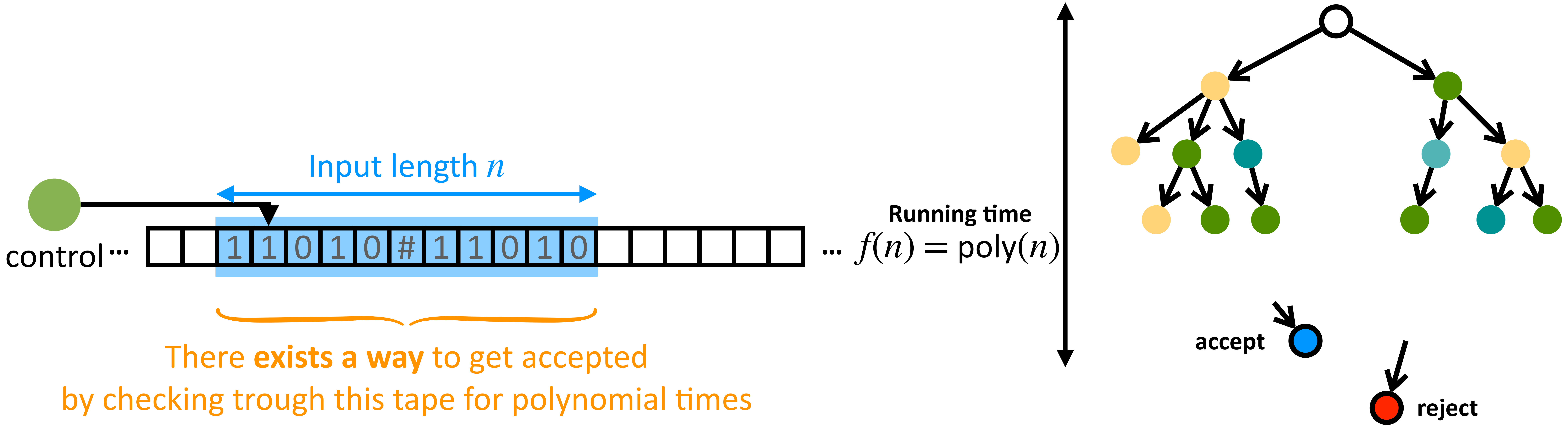
- Definition: **NP** is the class of languages that are accepted in polynomial time by a nondeterministic Turing machine.



The Class **NP**

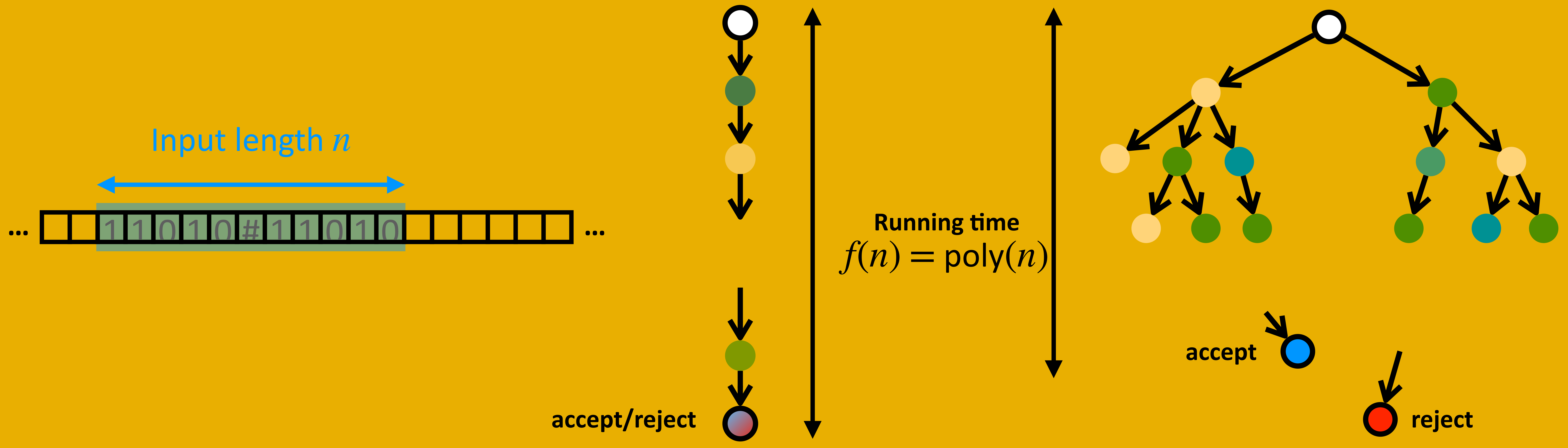
- Similarly, we can define the running time of a non-deterministic Turing machine N

- Definition: **NP** is the class of languages that are accepted in polynomial time by a nondeterministic Turing machine.



What Happened

- The class **P** is the class of languages that are accepted or rejected in polynomial time by a *deterministic* Turing machine
- The class **NP** is the class of languages that are accepted in polynomial time by a *non-deterministic* Turing machine.



Overview

- Algorithms:Turing machine, Deterministic and non-deterministic
- Formal language framework: string and language
- Time complexity
  - Input size
  - Classes **P** and **NP**
    - Polynomial time verification

Verify

Verify

- Intuition: Some problems are difficult. But with *a little hint*, it becomes much easier

Verify

- Intuition: Some problems are difficult. But with *a little hint*, it becomes much easier
  - For example, we want to know if 63187 is a composite number (that is, it is not a prime number).

Verify

- Intuition: Some problems are difficult. But with *a little hint*, it becomes much easier
  - For example, we want to know if 63187 is a composite number (that is, it is not a prime number).
    - It seems difficult to find the answer

Verify

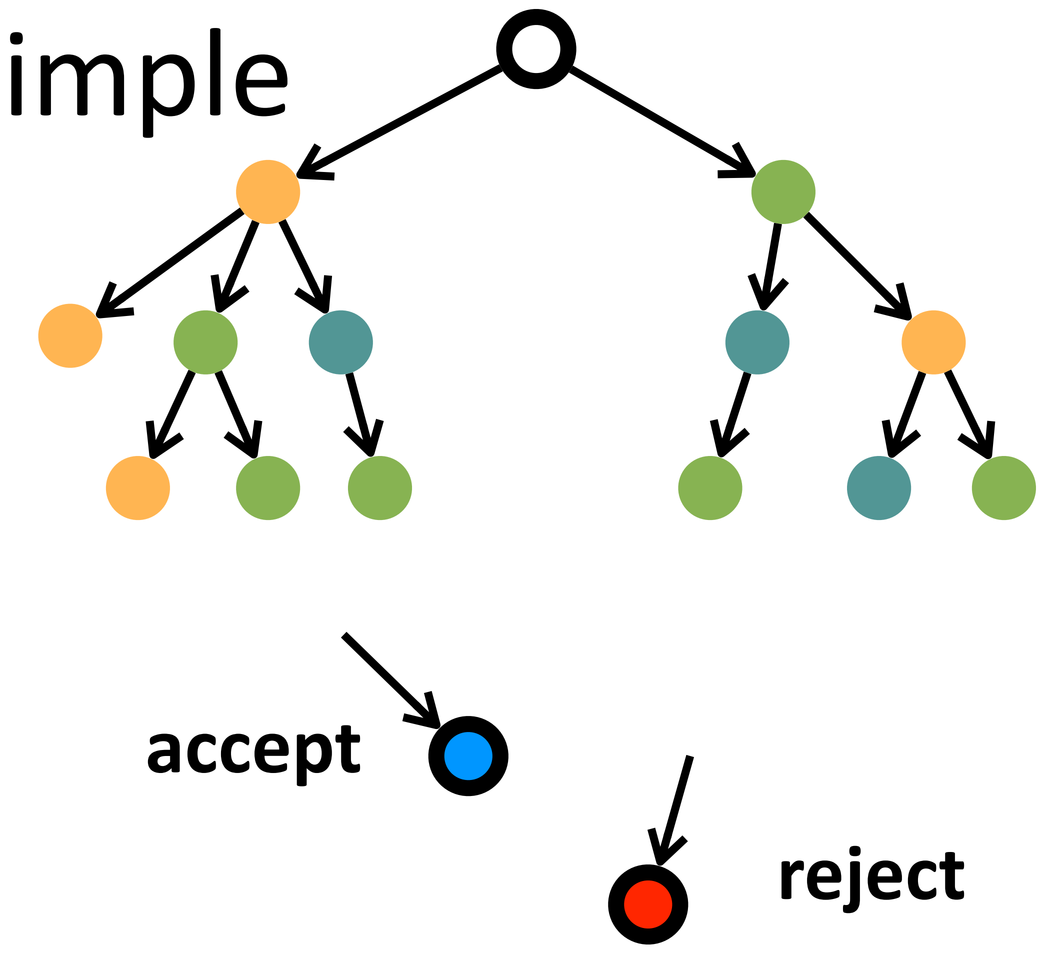
- Intuition: Some problems are difficult. But with *a little hint*, it becomes much easier
    - For example, we want to know if 63187 is a composite number (that is, it is not a prime number).
        - It seems difficult to find the answer
        - But if we are told that one of the divisor of 63187 is 353…

Verify

- Intuition: Some problems are difficult. But with *a little hint*, it becomes much easier
  - For example, we want to know if 63187 is a composite number (that is, it is not a prime number).
    - It seems difficult to find the answer
    - But if we are told that one of the divisor of 63187 is 353…
      - We can **verify** that 63187 is indeed a composite number by simple arithmetics.

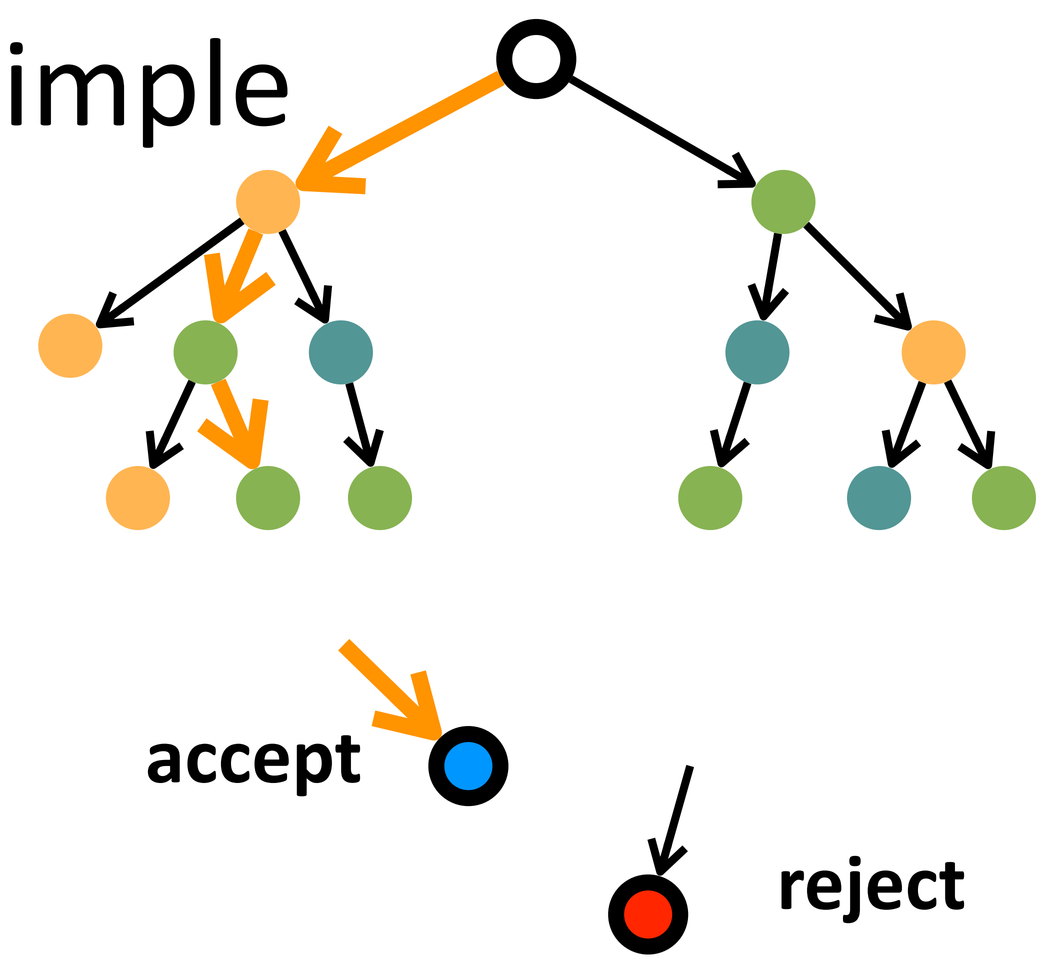
Verify

- Intuition: Some problems are difficult. But with *a little hint*, it becomes much easier
  - For example, we want to know if 63187 is a composite number (that is, it is not a prime number).
    - It seems difficult to find the answer
    - But if we are told that one of the divisor of 63187 is 353…
      - We can **verify** that 63187 is indeed a composite number by simple arithmetics.



Verify

- Intuition: Some problems are difficult. But with *a little hint*, it becomes much easier
  - For example, we want to know if 63187 is a composite number (that is, it is not a prime number).
    - It seems difficult to find the answer
    - But if we are told that one of the divisor of 63187 is 353…
      - We can **verify** that 63187 is indeed a composite number by simple arithmetics.



Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

hint

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

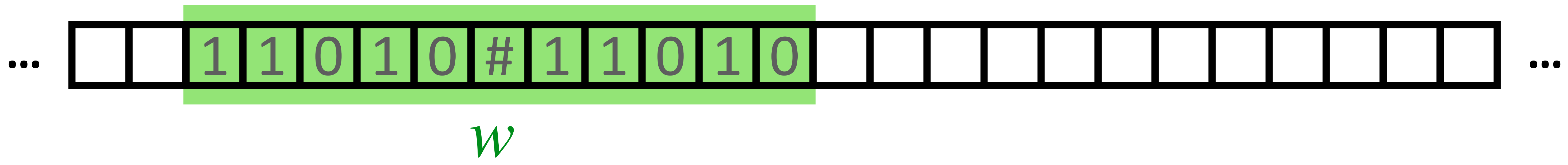
Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

hint

Is w in A?



Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

hint

Is w in A?

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

  hint

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A

Is w in A?

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

hint

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - You don't need to worry about the time complexity for coming up with c

Is w in A?

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

hint

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - You don't need to worry about the time complexity for coming up with c
    - Just assume there is an angel that can provide you any c you want for free

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- COMPOSITES $= \{x \mid x = pq, \text{ for integers } p, q > 1\}$

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - COMPOSITES $= \{x \mid x = pq, \text{ for integers } p, q > 1\}$
    - A devisor p of the number x can be a good certificate

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - COMPOSITES $= \{x \mid x = pq, \text{ for integers } p, q > 1\}$
    - A devisor p of the number x can be a good certificate
    - c is p

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A

  - CLIQUE $= \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- CLIQUE $= \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

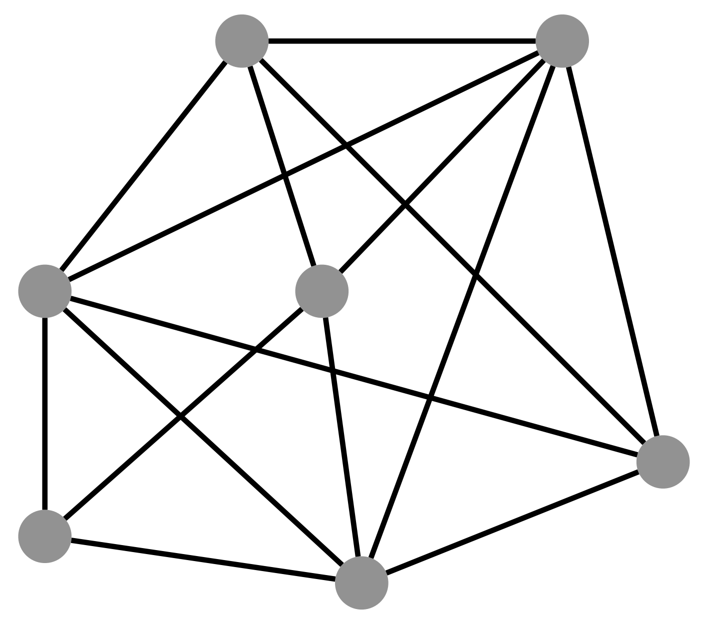
A set of k vertices with edge between every pair of vertices

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - CLIQUE $= \{\langle G,3 \rangle \mid G \text{ is an undirected graph with a } 3\text{-clique}\}$
    - Ex:

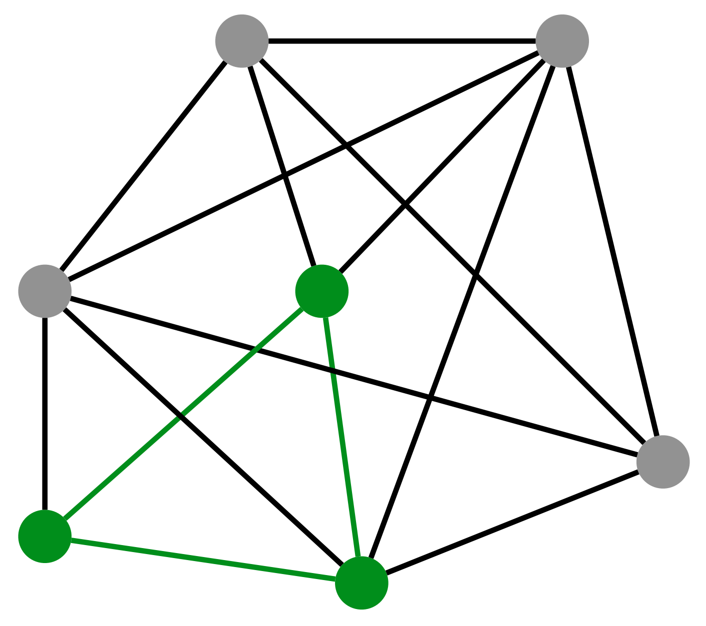


Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- CLIQUE $= \{\langle G,3 \rangle \mid G \text{ is an undirected graph with a } 3\text{-clique}\}$
  - Ex:



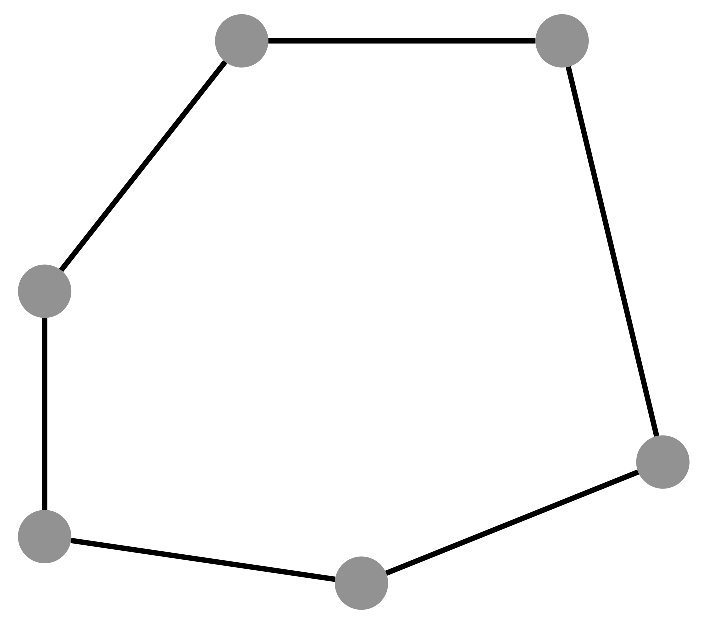
$\langle G,3 \rangle$ is a yes-instance

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- CLIQUE $= \{\langle G,3 \rangle \mid G \text{ is an undirected graph with a } 3\text{-clique}\}$
  - Ex:



❌ $\langle G,3 \rangle$ is a no-instance

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - CLIQUE $= \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

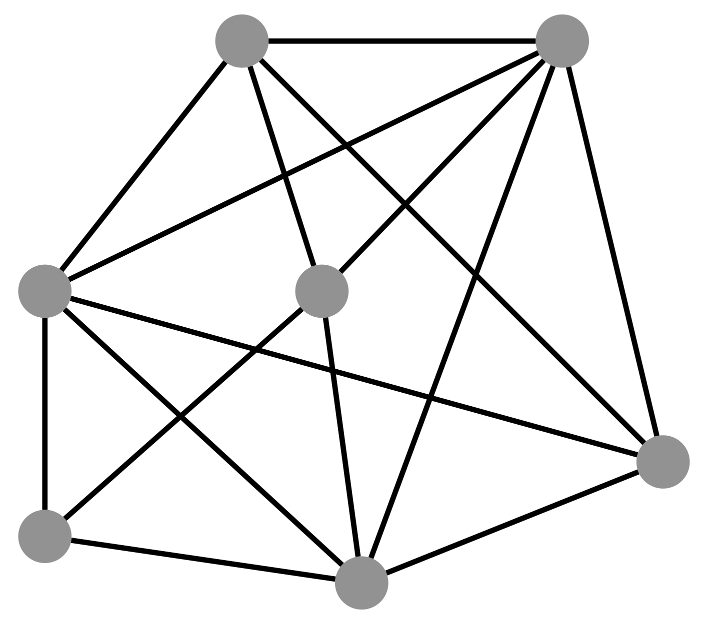
- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- CLIQUE $= \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$
  - A string c that encodes a clique with size k in G is a good certificate

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- CLIQUE $= \{\langle G, 4 \rangle \mid G \text{ is an undirected graph with a } 4\text{-clique}\}$
  - Ex:

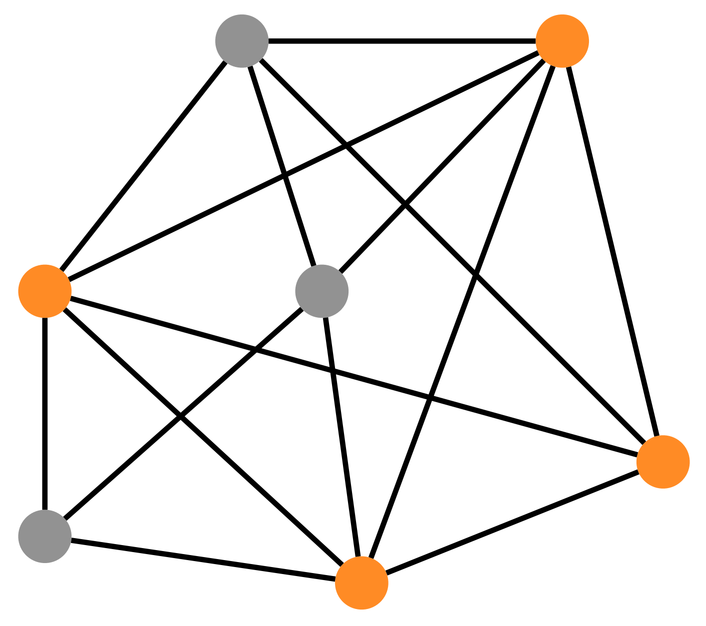


Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - CLIQUE $= \{\langle G, 4 \rangle \mid G \text{ is an undirected graph with a } 4\text{-clique}\}$

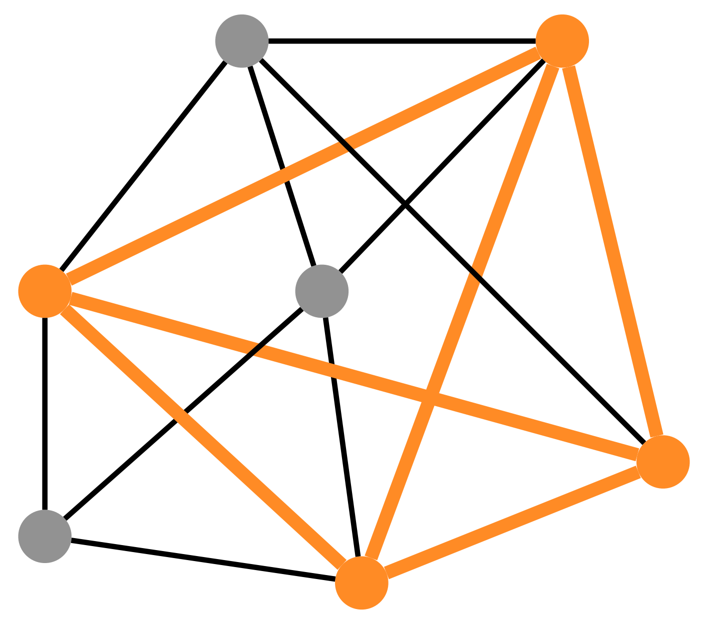


Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - CLIQUE $= \{\langle G, 4 \rangle \mid G \text{ is an undirected graph with a } 4\text{-clique}\}$



Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- SUBSET-SUM $= \{\langle S, t \rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
  - Ex: $S = \{4,2,8,5,7\}$ and $t = 17 \Rightarrow \{y_1, \cdots, y_t\} = \{2,8,7\}$

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- SUBSET-SUM $= \{\langle S, t \rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
  - Ex: $S = \{4,2,8,5,7\}$ and $t = 17 \Rightarrow \{y_1, \cdots, y_t\} = \{2,8,7\}$ ✔

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
    - Ex: $S = \{4,2,8,5,7\}$ and $t = 17 \Rightarrow \{y_1, \cdots, y_t\} = \{2,8,7\}$ ✓
    - Ex: $S = \{4,2,8,5,7\}$ and $t = 25 \Rightarrow$ No answer

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
  - Ex: $S = \{4,2,8,5,7\}$ and $t = 17 \Rightarrow \{y_1, \cdots, y_t\} = \{2,8,7\}$
  - Ex: $S = \{4,2,8,5,7\}$ and $t = 25 \Rightarrow$ No answer

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
  - A string c that encodes a subset of S with sum t is a good certificate
    - Ex: $S = \{4,2,8,5,7\}$ and $t = 17 \Rightarrow c = \{2,8,7\}$

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

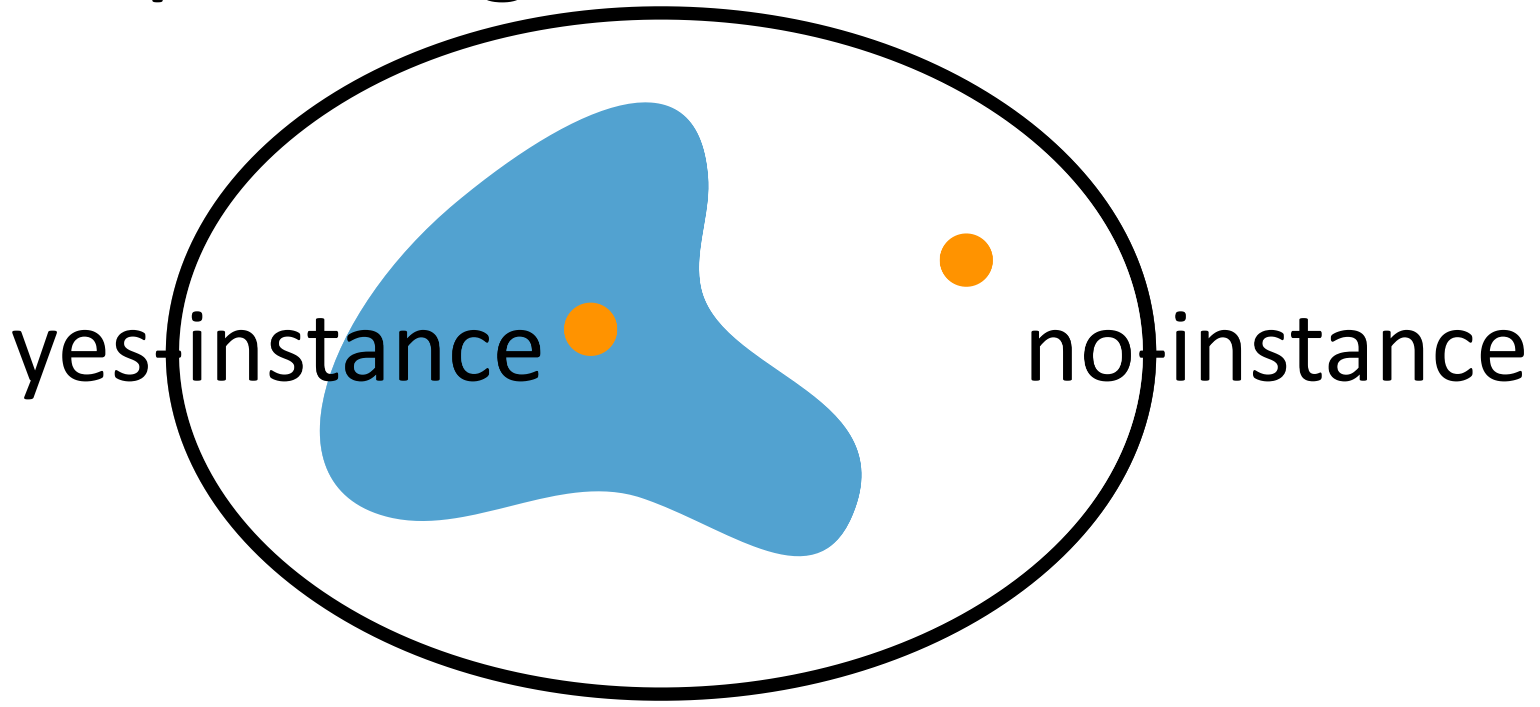
  - c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
  - An algorithm V verifies a language A if for any yes-instance $w \in A$, there is a certificate c that V can use to prove that $w \in A$
  - For any no-instance $w \notin A$, there must be no certificate proving that $w \in A$

Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- c is called a **certificate** or **proof**, which is an additional information to verify that the string w is a member of A
- An algorithm V verifies a language A if for any yes-instance $w \in A$, there is a certificate c that V can use to prove that $w \in A$
- For any no-instance $w \notin A$, there must be no certificate proving that $w \in A$



Polynomial Time Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

- A ***polynomial time verifier*** runs in polynomial time in the length of w

Polynomial Time Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- A ***polynomial time verifier*** runs in polynomial time in the length of w
  - We measure the time of a verifiers only in terms of the length of w

Polynomial Time Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

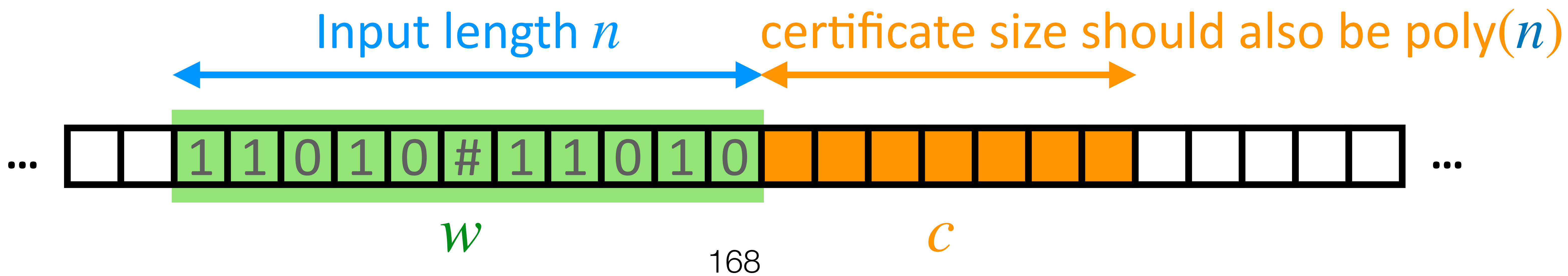
- A ***polynomial time verifier*** runs in polynomial time in the length of w
  - We measure the time of a verifiers only in terms of the length of w
  - A language A is ***polynomial-time verifiable*** if it has a polynomial time verifier

Polynomial Time Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

- A ***polynomial time verifier*** runs in polynomial time in the length of w
  - We measure the time of a verifiers only in terms of the length of w
  - A language A is ***polynomial-time verifiable*** if it has a polynomial time verifier
  - For polynomial verifiers, the certificate needs to have polynomial length (in the length of w)

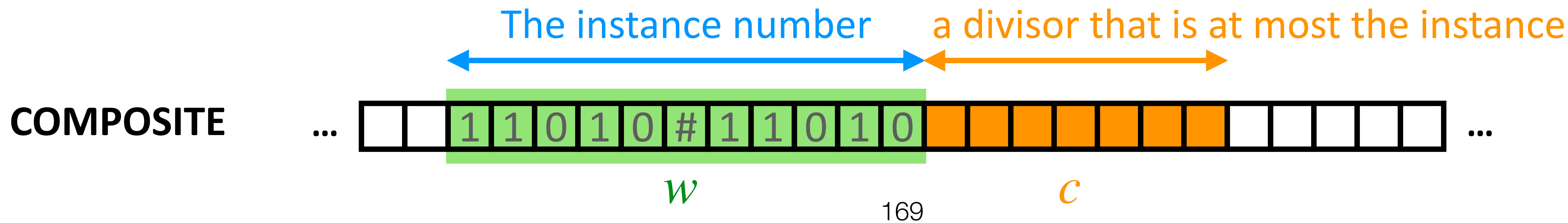


Polynomial Time Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

- A ***polynomial time verifier*** runs in polynomial time in the length of w
  - We measure the time of a verifiers only in terms of the length of w
  - A language A is ***polynomial-time verifiable*** if it has a polynomial time verifier
  - For polynomial verifiers, the certificate needs to have polynomial length (in the length of w)



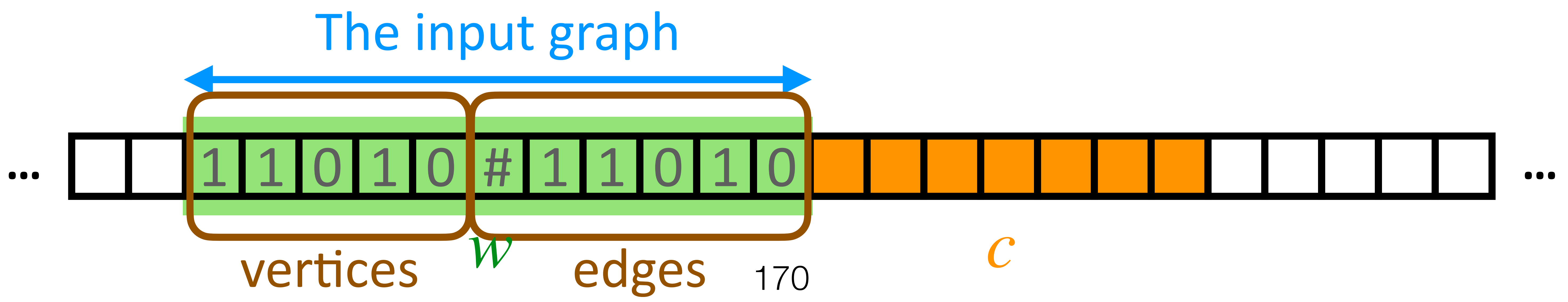
Polynomial Time Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

- A ***polynomial time verifier*** runs in polynomial time in the length of w
  - We measure the time of a verifiers only in terms of the length of w
  - A language A is ***polynomial-time verifiable*** if it has a polynomial time verifier
  - For polynomial verifiers, the certificate needs to have polynomial length (in the length of w)

CLIQUE

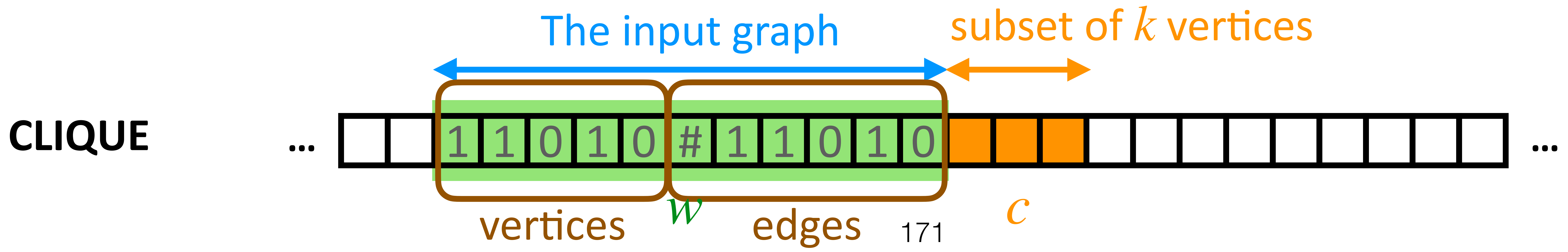


Polynomial Time Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

- A ***polynomial time verifier*** runs in polynomial time in the length of w
  - We measure the time of a verifiers only in terms of the length of w
  - A language A is ***polynomial-time verifiable*** if it has a polynomial time verifier
  - For polynomial verifiers, the certificate needs to have polynomial length (in the length of w)

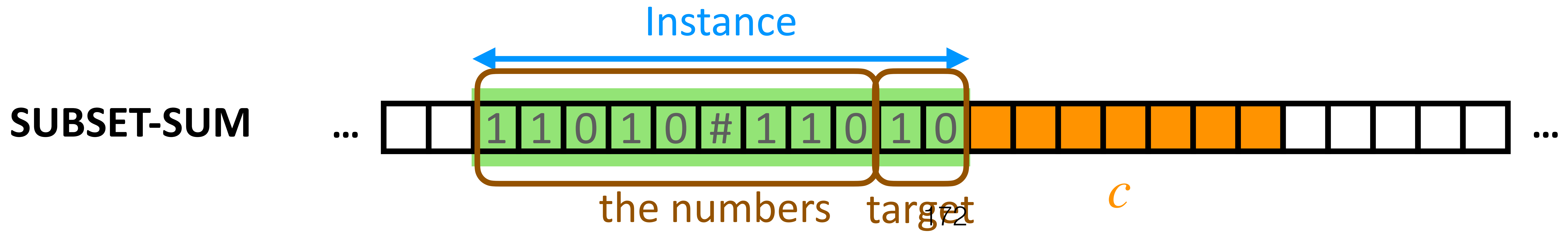


Polynomial Time Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

- A ***polynomial time verifier*** runs in polynomial time in the length of w
  - We measure the time of a verifiers only in terms of the length of w
  - A language A is ***polynomial-time verifiable*** if it has a polynomial time verifier
  - For polynomial verifiers, the certificate needs to have polynomial length (in the length of w)

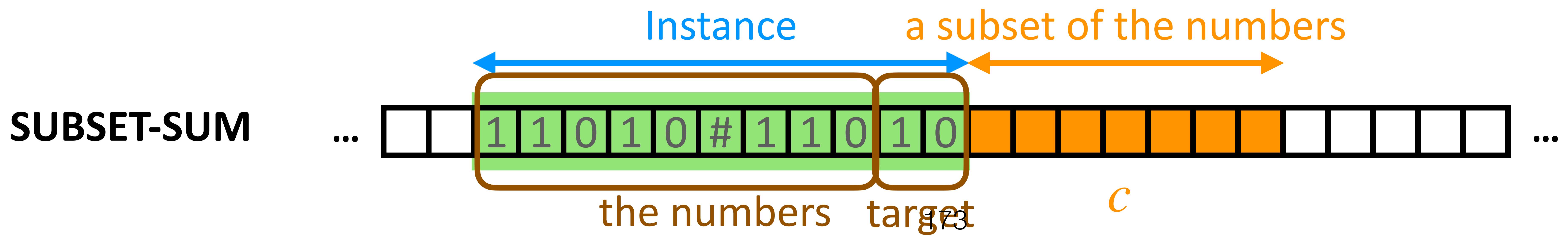


Polynomial Time Verifier

- Definition: A ***verifier*** for a language A is an algorithm V, where

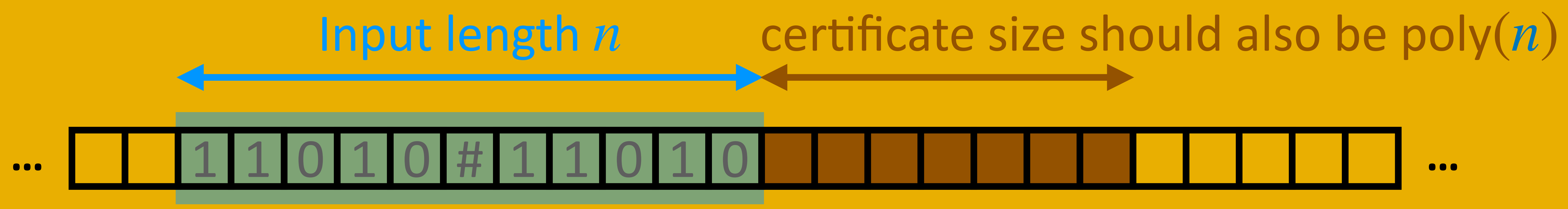
$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

- A ***polynomial time verifier*** runs in polynomial time in the length of w
  - We measure the time of a verifiers only in terms of the length of w
  - A language A is ***polynomial-time verifiable*** if it has a polynomial time verifier
  - For polynomial verifiers, the certificate needs to have polynomial length (in the length of w)



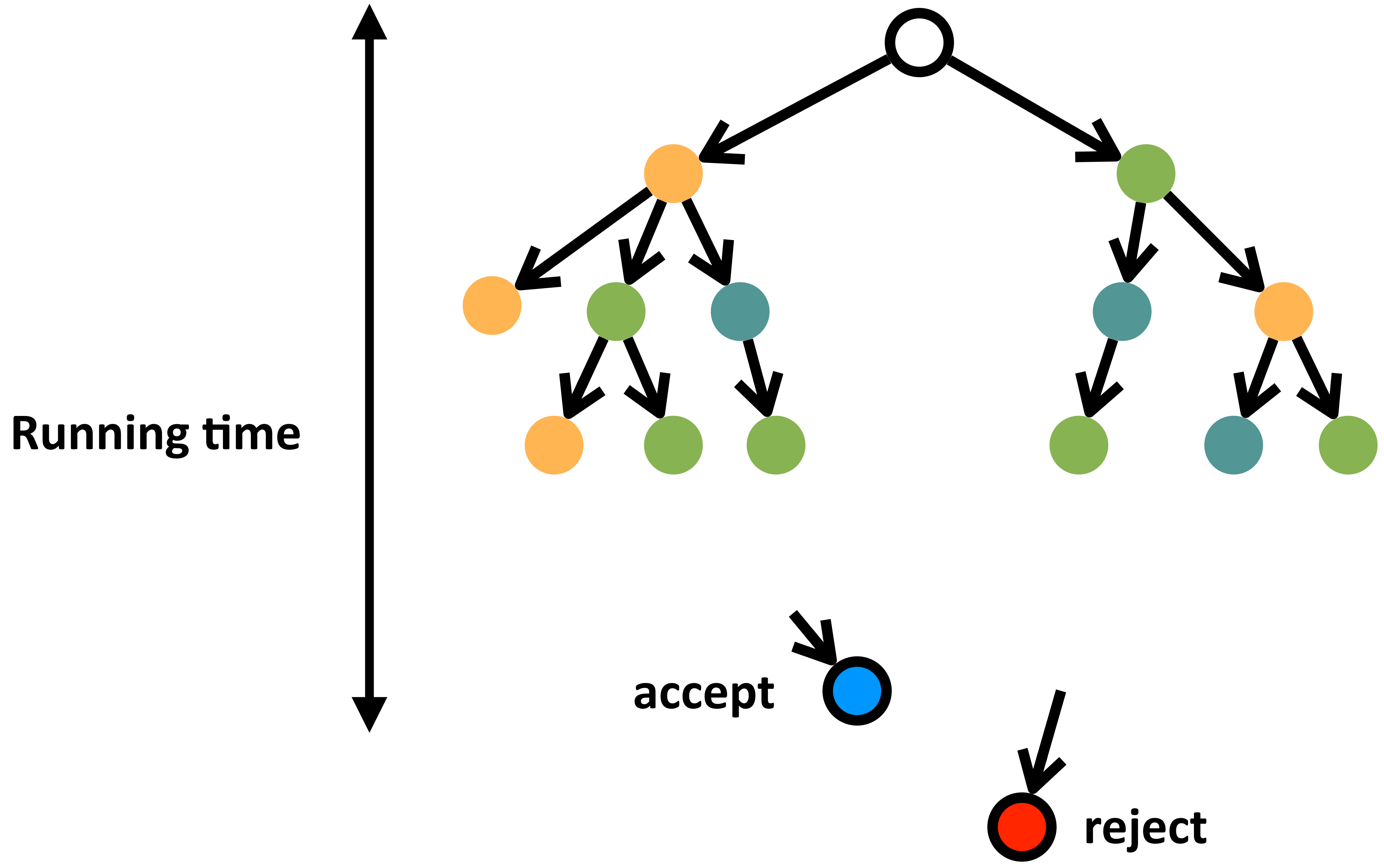
What Happened

- A language A is **verifiable** if for any of its yes-instances w, there exists a piece of hint (certificate) c such that using this hint c, one can be convinced that w is indeed a yes-instance of A
  - Only yes-instances have certificates
- **Polynomial-time verifiable**: the verification can be done in time of polynomial in input length
  - The hint size should also be polynomial
  - It does NOT mean that the hint c should be constructed within polynomial time!



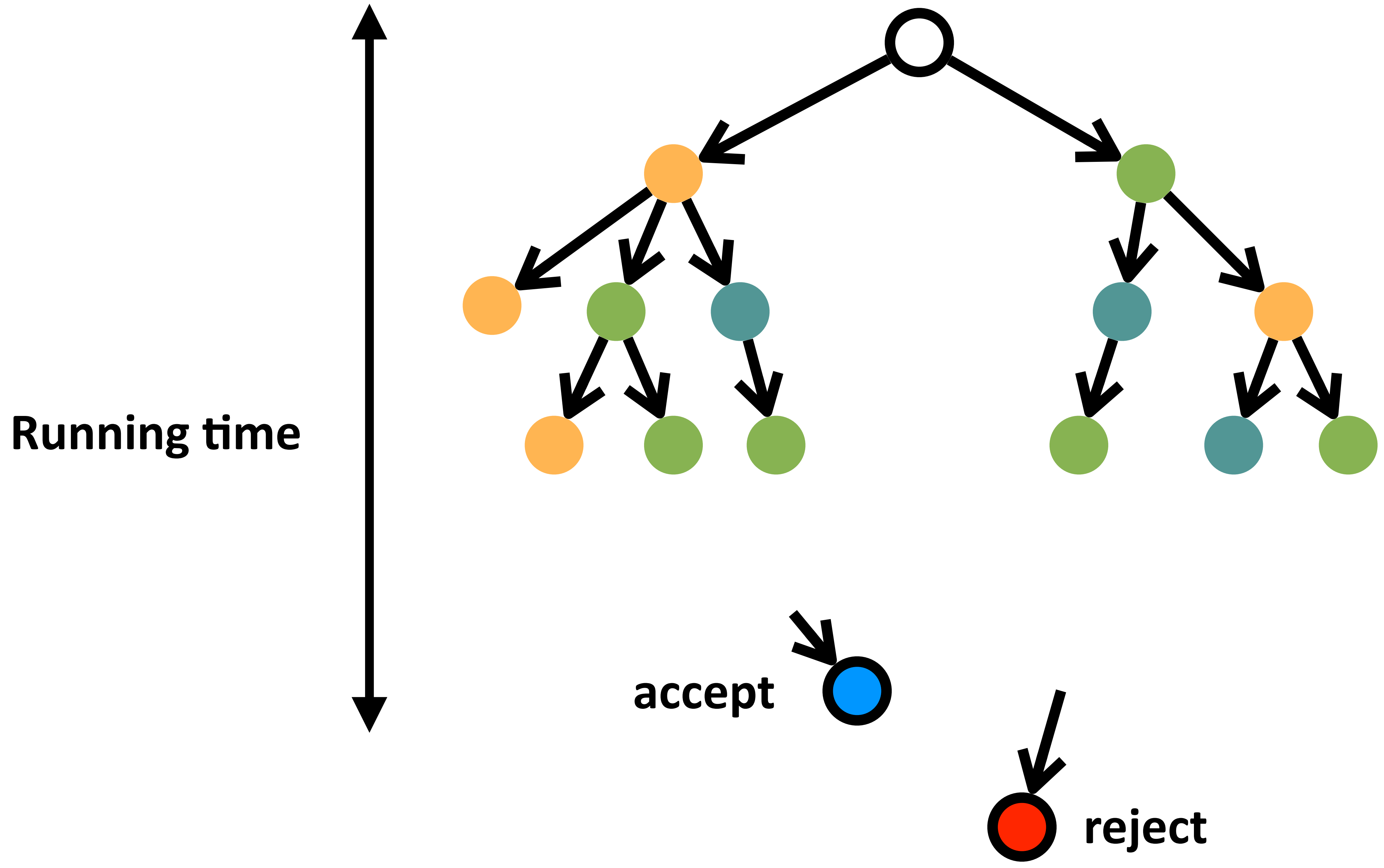
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are accepted or rejected in polynomial time by a nondeterministic Turing machine.



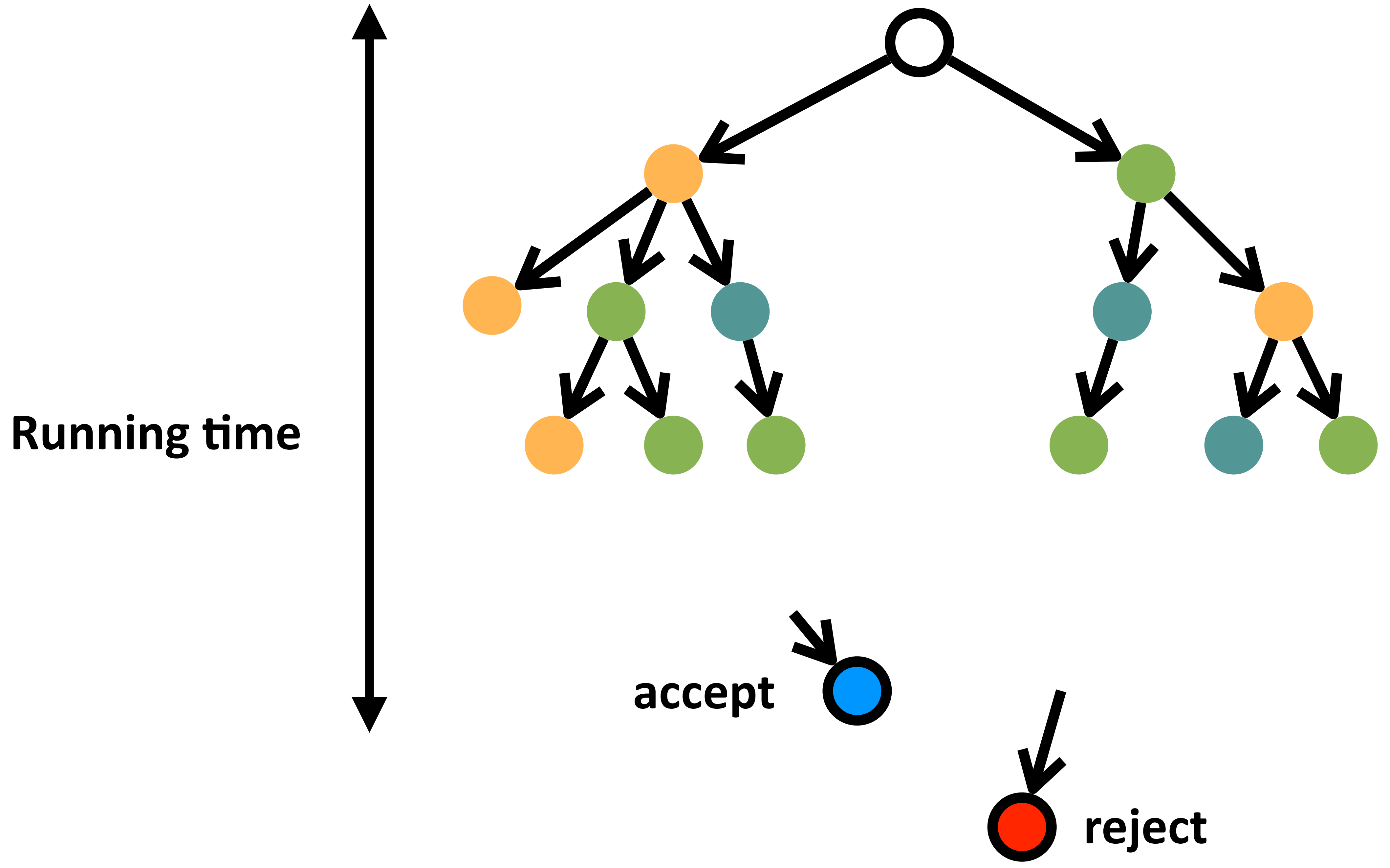
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.



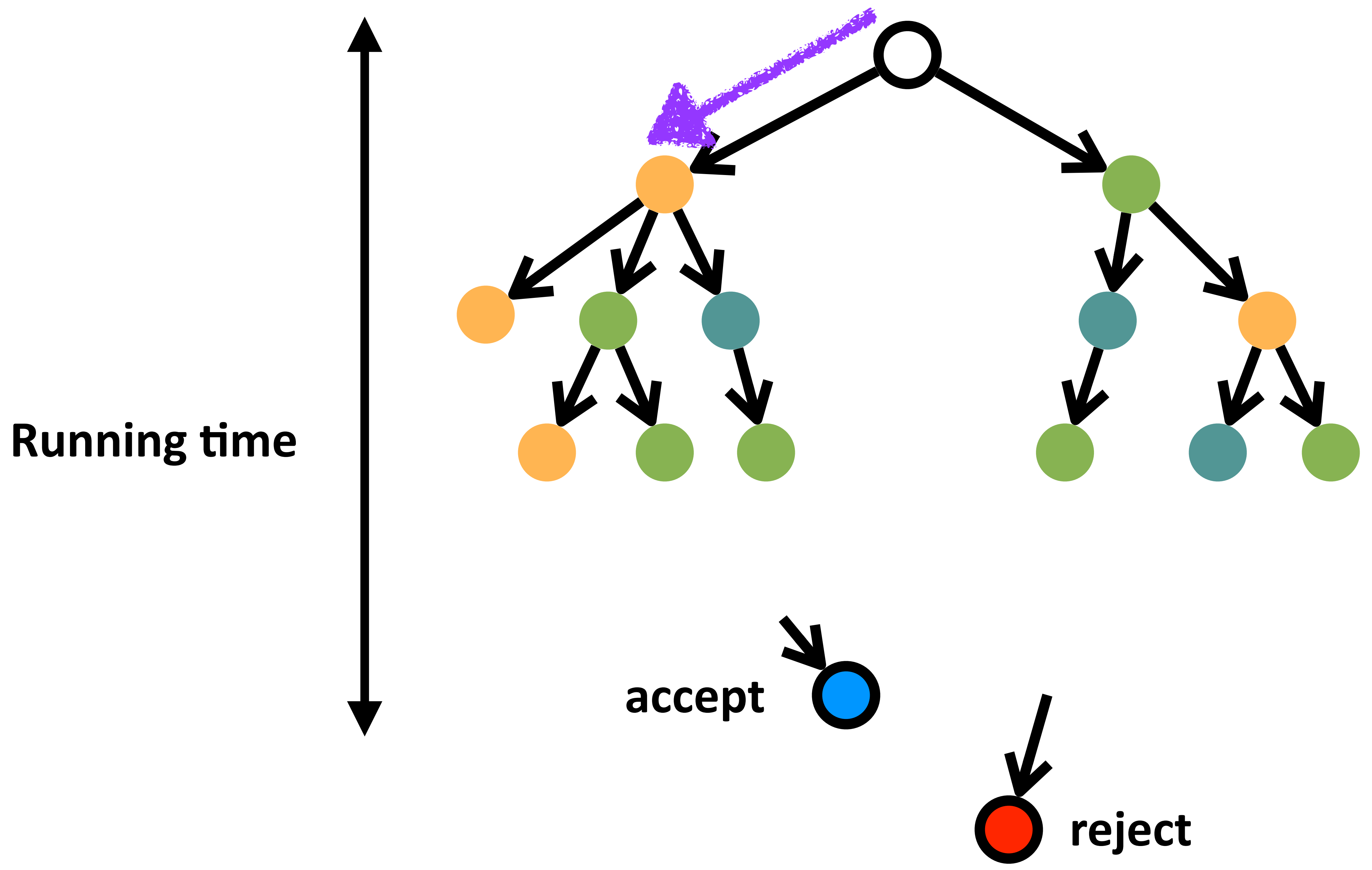
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.
- Any non-deterministic Turing machine can be simulated by a deterministic Turing machine



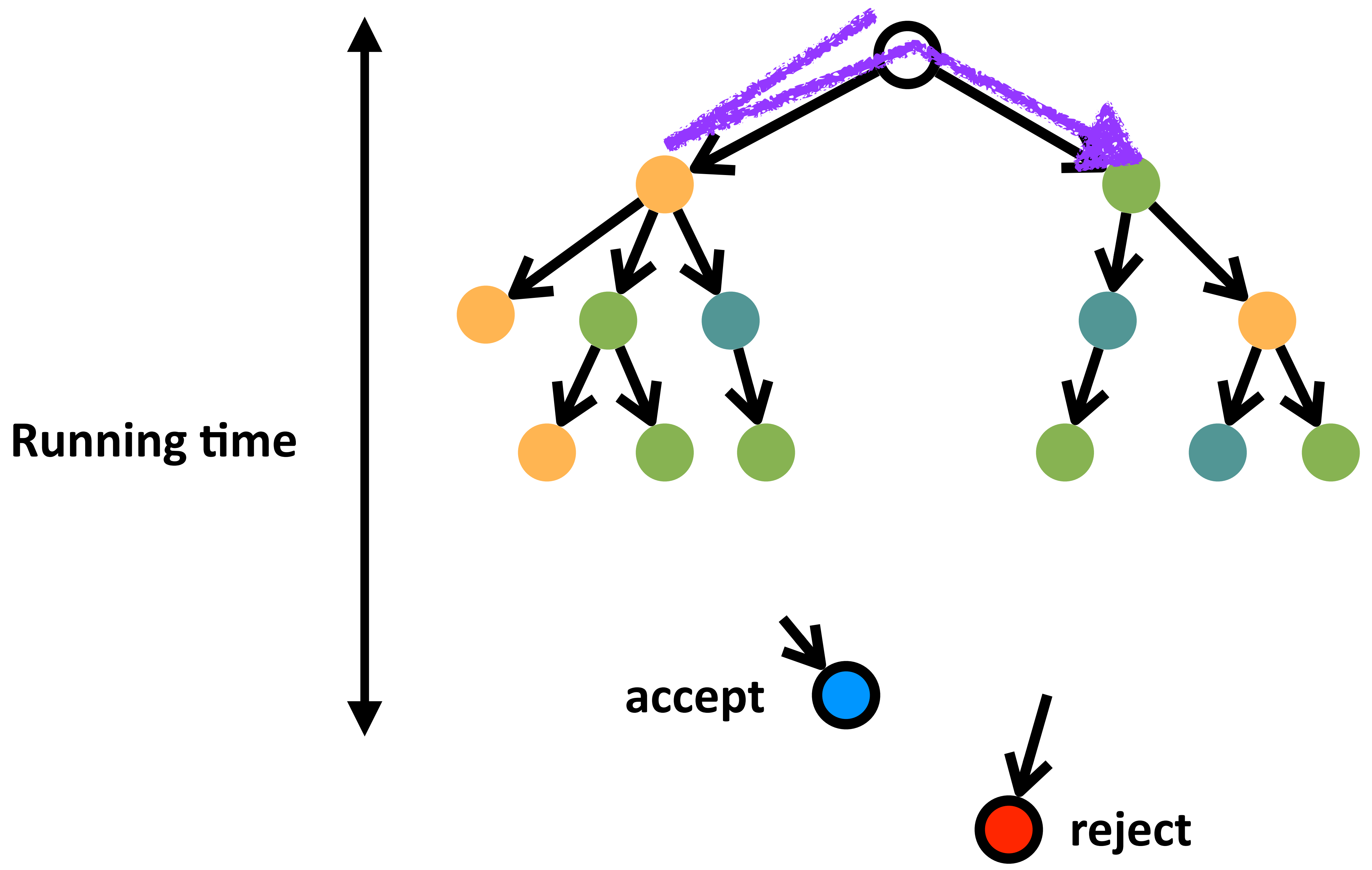
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.
- Any non-deterministic Turing machine can be simulated by a deterministic Turing machine
    - It needs more than polynomial time



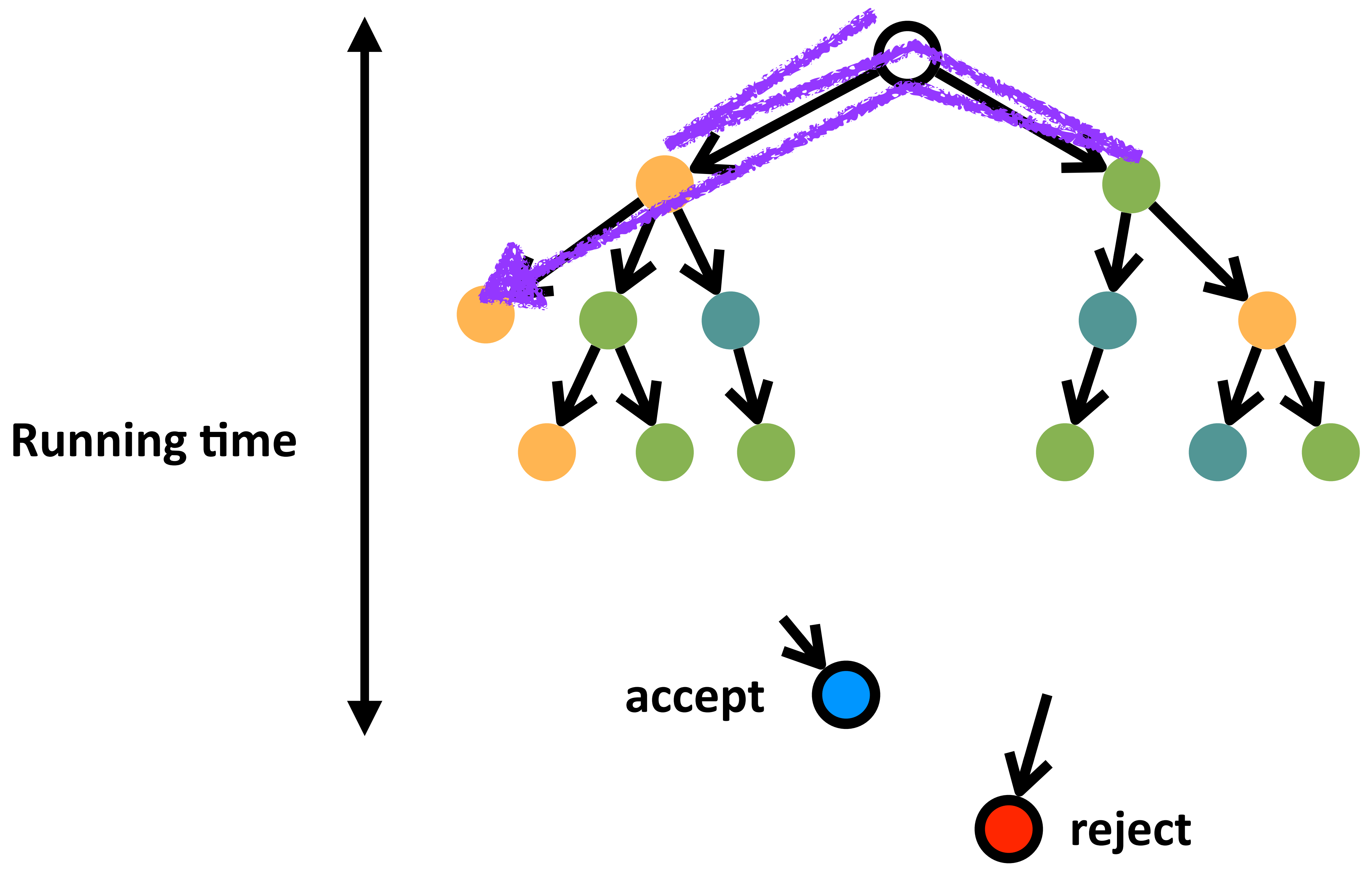
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.
- Any non-deterministic Turing machine can be simulated by a deterministic Turing machine
  - It needs more than polynomial time



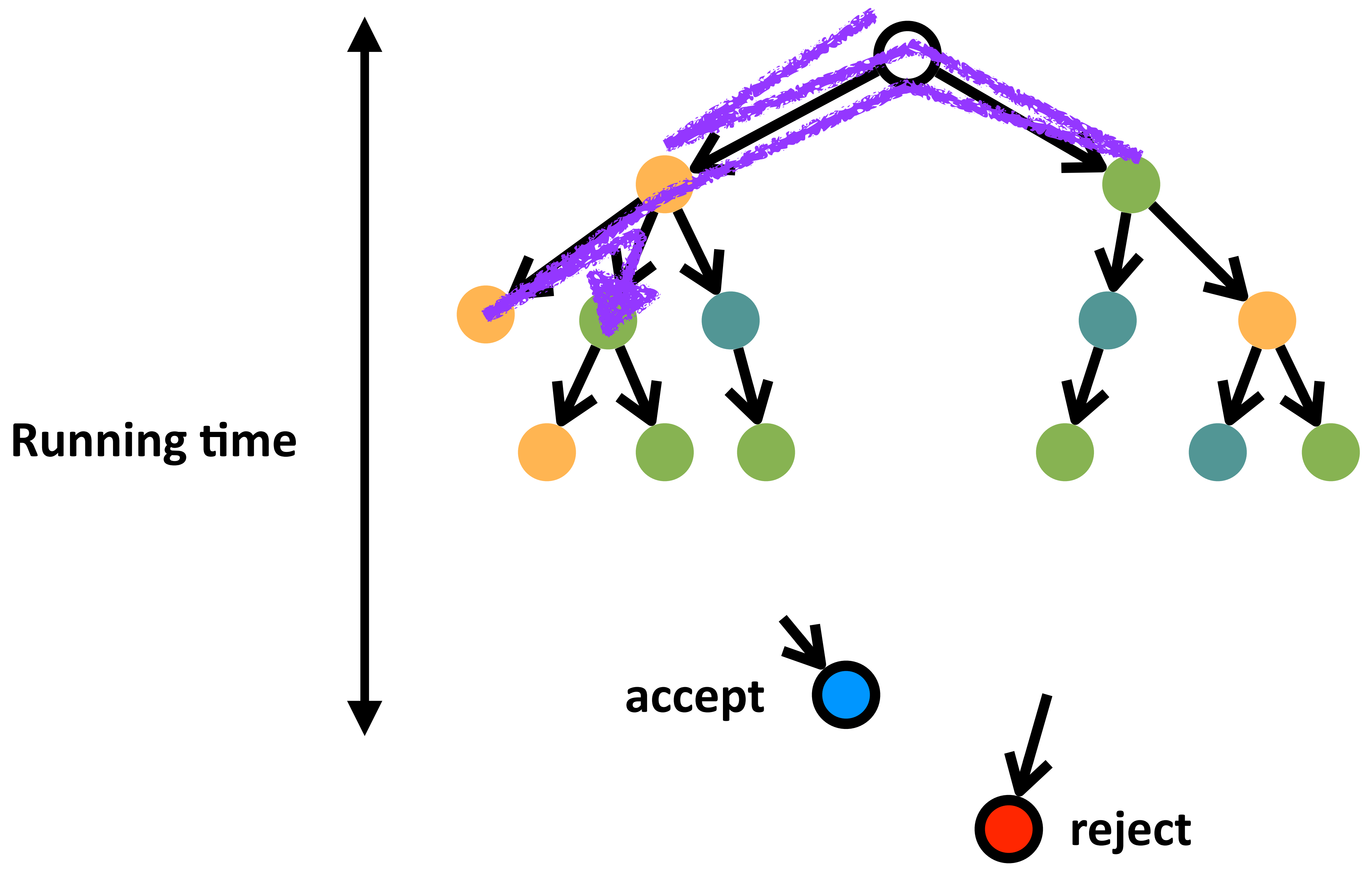
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.
- Any non-deterministic Turing machine can be simulated by a deterministic Turing machine
  - It needs more than polynomial time



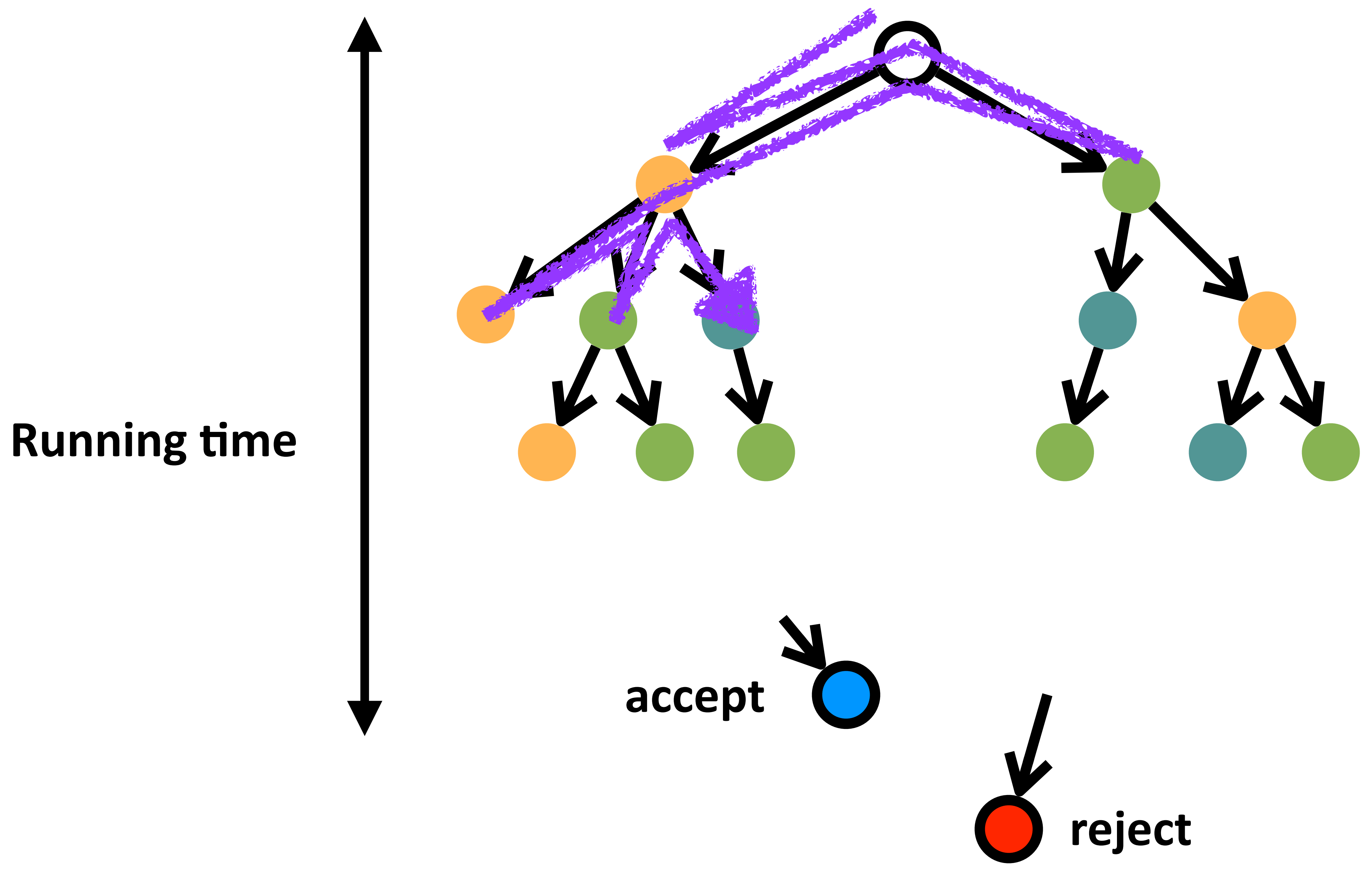
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.
- Any non-deterministic Turing machine can be simulated by a deterministic Turing machine
  - It needs more than polynomial time



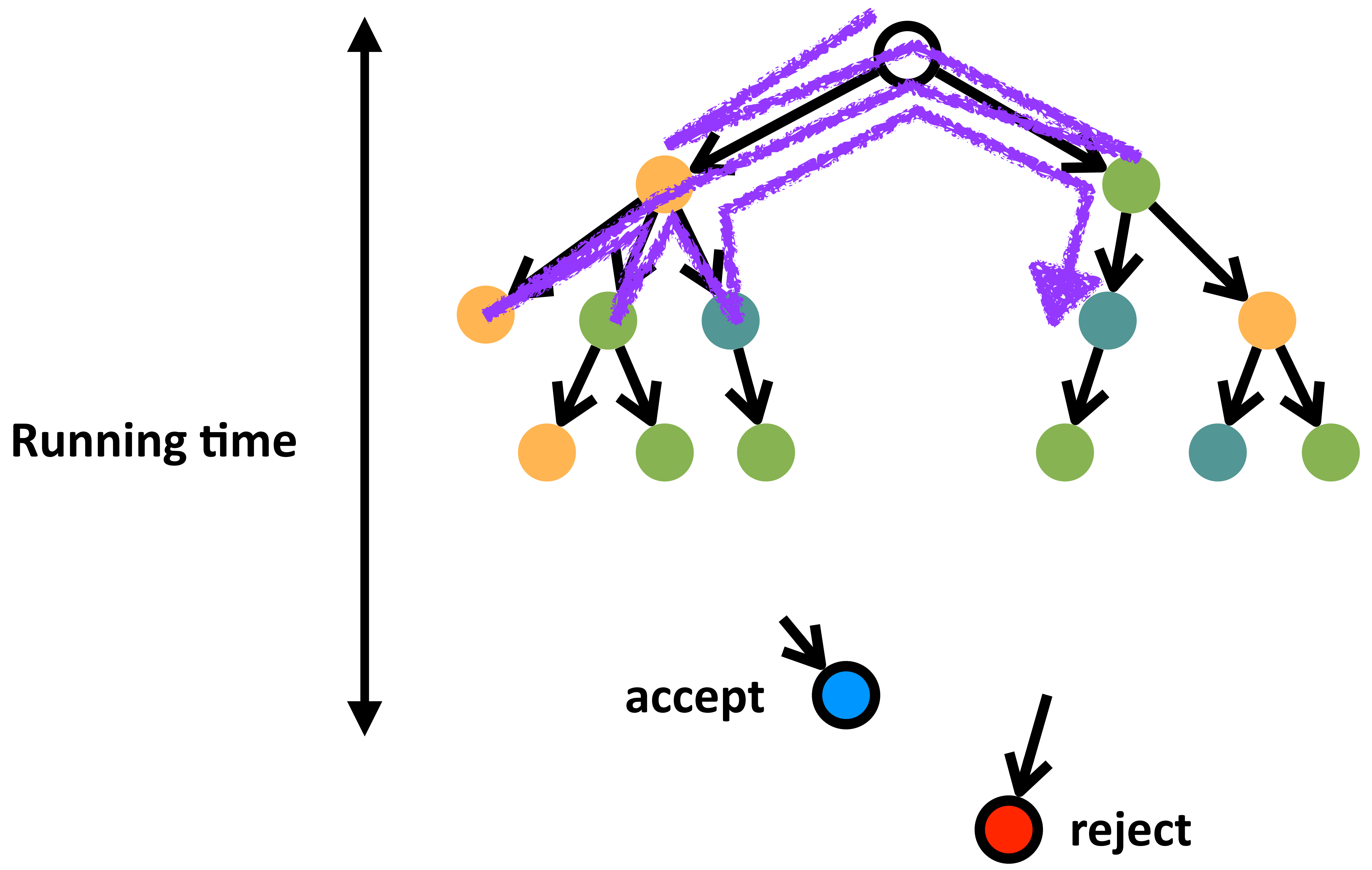
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.
- Any non-deterministic Turing machine can be simulated by a deterministic Turing machine
  - It needs more than polynomial time



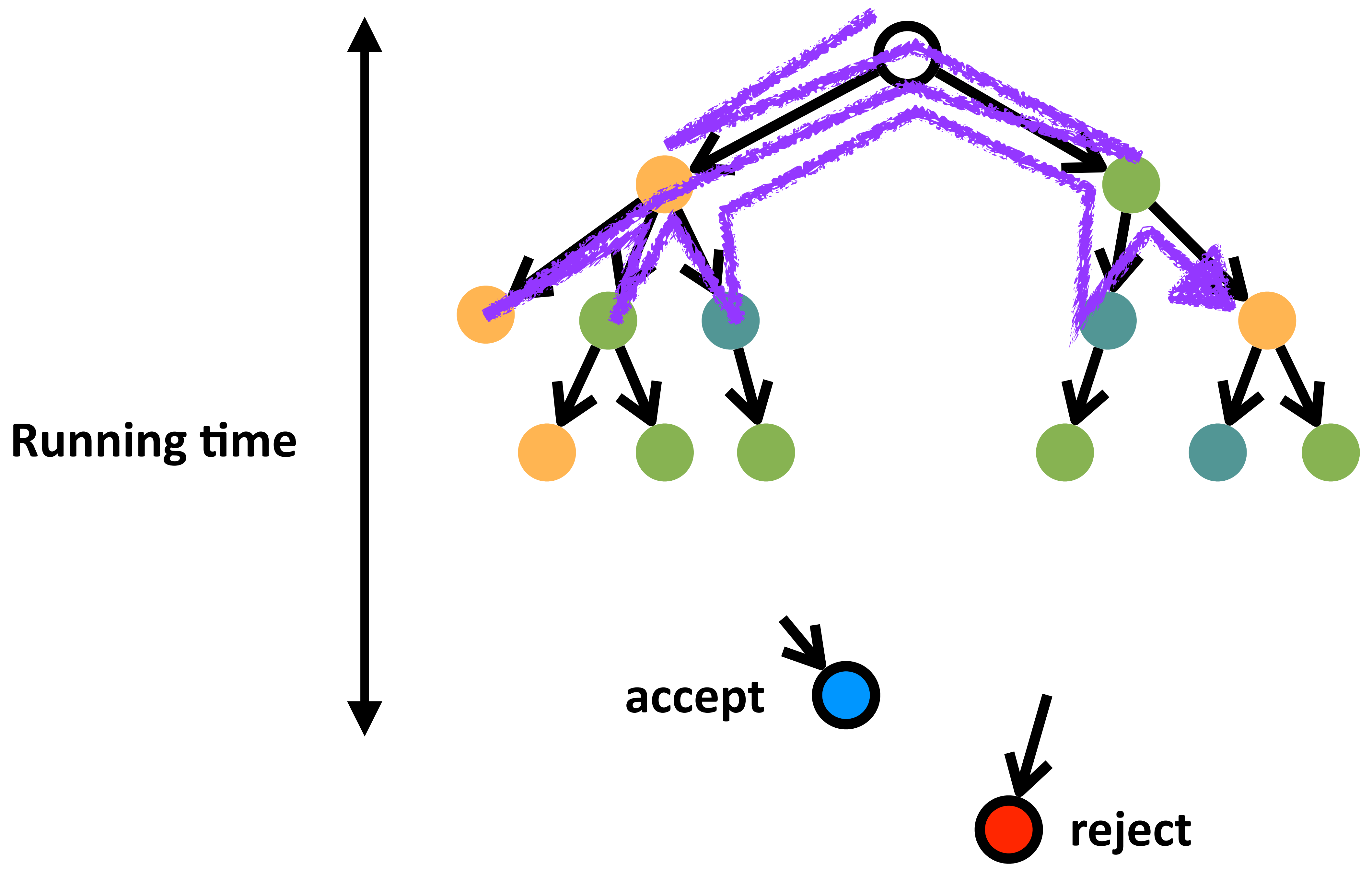
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.
- Any non-deterministic Turing machine can be simulated by a deterministic Turing machine
  - It needs more than polynomial time



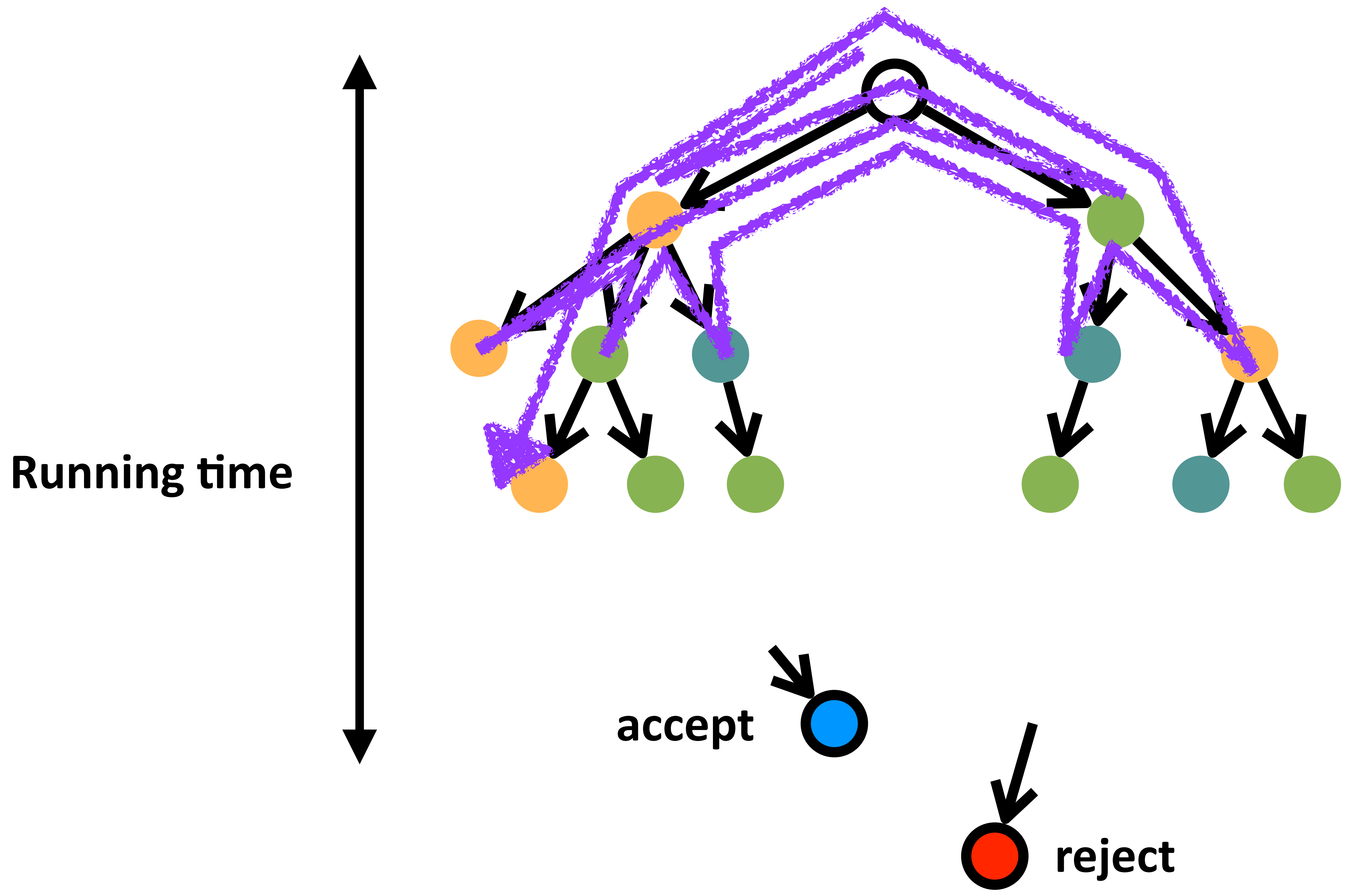
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.
- Any non-deterministic Turing machine can be simulated by a deterministic Turing machine
  - It needs more than polynomial time



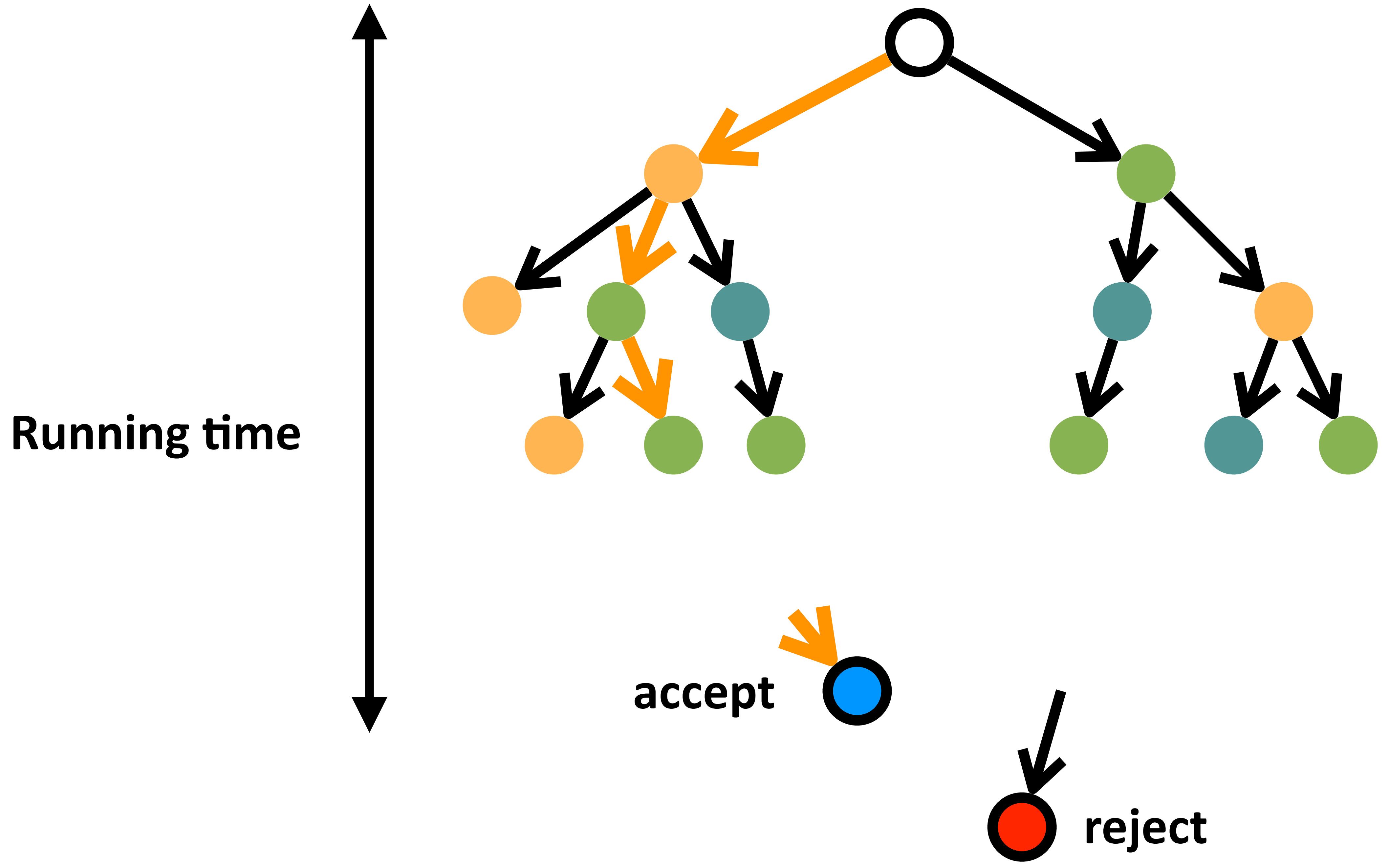
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.
- Any non-deterministic Turing machine can be simulated by a deterministic Turing machine
  - It needs more than polynomial time



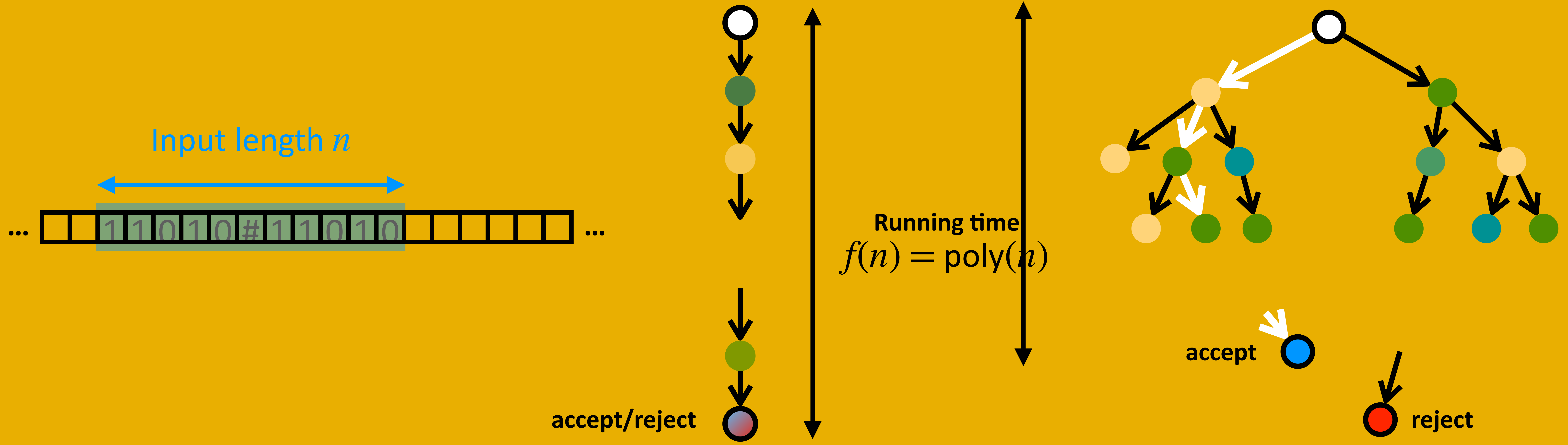
The Class **NP** — Alternative Definition

- Definition: **NP** is the class of languages that are verifiable in polynomial time on a deterministic Turing machine.
- Any non-deterministic Turing machine can be simulated by a deterministic Turing machine
  - It needs more than polynomial time
  - But if we know some **hint**, we know the path to an accept state with polynomial length



What Happened

- The class **P** is the class of languages that are *accepted* or *rejected* in polynomial time by a deterministic Turing machine
- The class **NP** is the class of languages that can be verified in polynomial time by a deterministic Turing machine.



Prove Language L is in **P**

- To prove that a language L is in **P**, we need to:
  - Design a Turing machine M
  - Show that M correctly accepts or rejects all input
  - Show that M runs in polynomial time

Prove Language L is in **P**

- To prove that a language L is in **P**, we need to:
  - Design a Turing machine M
  - Show that M correctly accepts or rejects all input
  - Show that M runs in polynomial time

Design an algorithm

Correctness proof

Time complexity analysis

Prove Language L is in **P**

- To prove that a language L is in **P**, we need to:
  - Design a Turing machine M — Design an algorithm
  - Show that M correctly accepts or rejects all input — Correctness proof
  - Show that M runs in polynomial time — Time complexity analysis

design a Turing machine $\equiv$ design an algorithm

Use Polynomial Time Verifier to Prove that A is in **NP**

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

Prove A is in **NP** ⇔ Design a **polynomial time verifier** to decide A (with help from some c)

Use Polynomial Time Verifier to Prove that A is in **NP**

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

Prove A is in **NP** $\Leftrightarrow$ Design a **polynomial time verifier** to decide A (with help from some c)

<Proof Idea>

1. Assume that there is a certificate c with size polynomial in the length of w
2. Design a verifier V on input $\langle w, c \rangle$ that *accepts* all $w \in A$ and *rejects* all $w \notin A$
3. Show that V runs in polynomial time (in the length of w)

CLIQUE is in **NP**

- CLIQUE $= \{\langle G, k\rangle \mid G$ is an undirected graph with a k-clique$\}$
  - A string c that encodes a clique with size k in G is a good certificate

Prove CLIQUE is in **NP** ⇔ Design a **polynomial time verifier** to decide CLIQUE (with help from some c)

<Proof Idea>

1. Assume that there is a certificate c that encodes a clique with size k in G.
2. Design a verifier V on input $\langle w, c\rangle$ that *accepts* all $w \in A$ and *rejects* all $w \notin A$
3. Show that V runs in polynomial time (in the length of w)

CLIQUE is in **NP**

- CLIQUE $= \{\langle G, k\rangle \mid G$ is an undirected graph with a k-clique$\}$

Prove CLIQUE is in **NP** ⇔ Design a polynomial time verifier to decide CLIQUE (with help from some c)

<Proof>

Let string c that encodes a clique with size k in G as a certificate

CLIQUE is in **NP**

- CLIQUE $= \{\langle G, k\rangle \mid G$ is an undirected graph with a k-clique$\}$

Prove CLIQUE is in **NP** ⇔ Design a polynomial time verifier to decide CLIQUE (with help from some c)

<Proof>

Let string c that encodes a clique with size k in G as a certificate

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both 1 and 2 pass, *accept*; otherwise, *reject*."

CLIQUE is in **NP**

- CLIQUE $= \{\langle G, k\rangle \mid G$ is an undirected graph with a k-clique$\}$

Prove CLIQUE is in **NP** ⇔ Design a polynomial time verifier to decide CLIQUE (with help from some c)

<Proof>

Let string c that encodes a clique with size k in G as a certificate

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both 1 and 2 pass, *accept*; otherwise, *reject*."

Step 1 takes at most $|c| = k$ times of scanning through the input.

CLIQUE is in **NP**

- CLIQUE $= \{\langle G, k\rangle \mid G$ is an undirected graph with a k-clique$\}$

Prove CLIQUE is in **NP** ⇔ Design a polynomial time verifier to decide CLIQUE (with help from some c)

<Proof>

Let string c that encodes a clique with size k in G as a certificate

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both 1 and 2 pass, *accept*; otherwise, *reject*."

Step 1 takes at most $|c| = k$ times of scanning through the input. Step 2 takes at most $|c|^2 = k^2$ times of scanning through the input.

CLIQUE is in **NP**

- CLIQUE $= \{\langle G, k\rangle \mid G$ is an undirected graph with a k-clique$\}$

Prove CLIQUE is in **NP** ⇔ Design a polynomial time verifier to decide CLIQUE (with help from some c)

<Proof>

Let string c that encodes a clique with size k in G as a certificate

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both 1 and 2 pass, *accept*; otherwise, *reject*."

Step 1 takes at most $|c| = k$ times of scanning through the input. Step 2 takes at most $|c|^2 = k^2$ times of scanning through the input. Hence, V runs in polynomial time in the input length.

CLIQUE is in **NP**

- CLIQUE $= \{\langle G, k\rangle \mid G$ is an undirected graph with a k-clique$\}$

Prove CLIQUE is in **NP** ⇔ Design a polynomial time verifier to decide CLIQUE (with help from some c)

<Proof>

Let string c that encodes a clique with size k in G as a certificate

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both 1 and 2 pass, *accept*; otherwise, *reject*."

Step 1 takes at most $|c| = k$ times of scanning through the input. Step 2 takes at most $|c|^2 = k^2$ times of scanning through the input. Hence, V runs in polynomial time in the input length.

Use Polynomial Time Verifier to Prove that A is in **NP**

- Definition: A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

Prove A is in **NP** $\Leftrightarrow$ Design a **polynomial time verifier** to decide A (with help from some c)

<Proof Idea>

1. Show that for any yes instance w, there is a polynomial-size certificate c.
2. Design a verifier V on input $\langle w, c \rangle$ that *accepts* all $w \in A$ and *rejects* all $w \notin A$
3. Show that V runs in polynomial time (in the length of w)

CLIQUE is in **NP**

- CLIQUE $= \{\langle G, k\rangle \mid G$ is an undirected graph with a k-clique$\}$

Prove CLIQUE is in **NP** ⇔ Design a polynomial time verifier to decide CLIQUE (with help from some c)

<Proof>

Let string c that encodes a clique with size k in G as a certificate

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both 1 and 2 pass, *accept*; otherwise, *reject*."

Step 1 takes at most $|c| = k$ times of scanning through the input. Step 2 takes at most $|c|^2 = k^2$ times of scanning through the input. Hence, V runs in polynomial time in the input length.

SUBSET-SUM is in **NP**

- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
- Prove SUBSET-SUM is in **NP** $\Leftrightarrow$ Design a **polynomial time** **verifier** to decide CLIQUE (with help from some c)

SUBSET-SUM is in **NP**

- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
- Prove SUBSET-SUM is in **NP** ⇔ Design a **polynomial time verifier** to decide CLIQUE (with help from some c)

<Proof Idea>

1. Assume that there is a certificate c with size polynomial in the length of w
2. Design a verifier V on input $\langle w, c\rangle$ that *accepts* all $w \in A$ and *rejects* all $w \notin A$
3. Show that V runs in polynomial time (in the length of w)

SUBSET-SUM is in **NP**

- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
- Prove SUBSET-SUM is in **NP** ⇔ Design a polynomial time verifier to decide SUBSET-SUM (with help from some c)

<Proof>

SUBSET-SUM is in **NP**

- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
- Prove SUBSET-SUM is in **NP** ⇔ Design a polynomial time verifier to decide SUBSET-SUM (with help from some c)

<Proof>

Let string c that encodes a subset of S with sum t as a certificate

SUBSET-SUM is in **NP**

- SUBSET-SUM $= \{\langle S, t \rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
- Prove SUBSET-SUM is in **NP** ⇔ Design a polynomial time verifier to decide SUBSET-SUM (with help from some c)

<Proof>

Let string c that encodes a subset of S with sum t as a certificate

V = "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether $|c| < |S|$
2. Test whether c is a collection of numbers that sum to t
3. Test whether S contains all the numbers in c
4. If all 1, 2, and 3 pass, *accept*; otherwise, *reject*."

SUBSET-SUM is in **NP**

- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
- Prove SUBSET-SUM is in **NP** ⇔ Design a polynomial time verifier to decide SUBSET-SUM (with help from some c)

<Proof>

Let string c that encodes a subset of S with sum t as a certificate

V = "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether $|c| < |S|$
2. Test whether c is a collection of numbers that sum to t
3. Test whether S contains all the numbers in c
4. If all 1, 2, and 3 pass, *accept*; otherwise, *reject*."

Step 1 takes at most 1 time of scanning through the input.

SUBSET-SUM is in **NP**

- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
- Prove SUBSET-SUM is in **NP** ⇔ Design a polynomial time verifier to decide SUBSET-SUM (with help from some c)

<Proof>

Let string c that encodes a subset of S with sum t as a certificate

V = "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether $|c| < |S|$
2. Test whether c is a collection of numbers that sum to t
3. Test whether S contains all the numbers in c
4. If all 1, 2, and 3 pass, *accept*; otherwise, *reject*."

Step 1 takes at most 1 time of scanning through the input. Step 2 takes $|c| < |S|$ summations.

SUBSET-SUM is in **NP**

- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
- Prove SUBSET-SUM is in **NP** ⇔ Design a polynomial time verifier to decide SUBSET-SUM (with help from some c)

<Proof>

Let string c that encodes a subset of S with sum t as a certificate

V = "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether $|c| < |S|$
2. Test whether c is a collection of numbers that sum to t
3. Test whether S contains all the numbers in c
4. If all 1, 2, and 3 pass, *accept*; otherwise, *reject*."

Step 1 takes at most 1 time of scanning through the input. Step 2 takes $|c| < |S|$ summations. Step 3 takes at most $|c|$ times of scanning through the input.

SUBSET-SUM is in **NP**

- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
- Prove SUBSET-SUM is in **NP** ⇔ Design a polynomial time verifier to decide SUBSET-SUM (with help from some c)

<Proof>

Let string c that encodes a subset of S with sum t as a certificate

V = "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether $|c| < |S|$
2. Test whether c is a collection of numbers that sum to t
3. Test whether S contains all the numbers in c
4. If all 1, 2, and 3 pass, *accept*; otherwise, *reject*."

Step 1 takes at most 1 time of scanning through the input. Step 2 takes $|c| < |S|$ summations. Step 3 takes at most $|c|$ times of scanning through the input. Hence, V runs in polynomial time in the input length.

SUBSET-SUM is in **NP**

- SUBSET-SUM $= \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $\{y_1, \cdots, y_m\} \subseteq \{x_1, \cdots, x_k\}$, we have $\Sigma y_i = t\}$
- Prove SUBSET-SUM is in **NP** ⇔ Design a polynomial time verifier to decide SUBSET-SUM (with help from some c)

<Proof>

Let string c that encodes a subset of S with sum t as a certificate

V = "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether $|c| < |S|$
2. Test whether c is a collection of numbers that sum to t
3. Test whether S contains all the numbers in c
4. If all 1, 2, and 3 pass, *accept*; otherwise, *reject*."

Step 1 takes at most linear time to scan through the input. Step 2 takes $|c| < |S|$ summations. Step 3 takes at most $|c|$ times of scanning through the input. Hence, V runs in polynomial time in the input length.

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - *literals*
    - $\phi = \bar{x} \wedge y \wedge z$
    - $\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \underset{0}{\bar{x}} \wedge \underset{1}{y} \wedge \underset{1}{z}$
    - $\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \bar{x} \wedge y \wedge z$ $\quad\quad\quad\quad\quad\quad$ $x = \text{TRUE}, y = \text{TRUE}, z = \text{TRUE}$
    - $\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \underset{0}{\bar{x}} \wedge \underset{1}{y} \wedge \underset{1}{z}$ $x = \text{TRUE}, y = \text{TRUE}, z = \text{TRUE}$
    - $\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \underset{0}{\bar{x}} \wedge \underset{1}{y} \wedge \underset{1}{z}$ = FALSE $\quad x =$ TRUE, $y =$ TRUE, $z =$ TRUE
    - $\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \bar{x} \wedge y \wedge z$
    - $\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$ $\quad x =$ TRUE, $y =$ TRUE, $z =$ FALSE
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \bar{x} \wedge y \wedge z$
    - $\phi = (\underset{0}{\bar{x}} \wedge \underset{1}{y}) \vee (\underset{1}{x} \wedge \underset{1}{\bar{z}})$ $\qquad x = \text{TRUE}, y = \text{TRUE}, z = \text{FALSE}$
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \bar{x} \wedge y \wedge z$
    - $\phi = (\underset{0}{\bar{x}} \wedge \underset{1}{y}) \vee (\underset{1}{x} \wedge \underset{1}{\bar{z}})$ = FALSE $\quad x$ = TRUE, y = TRUE, z = FALSE
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \bar{x} \wedge y \wedge z$
    - $\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)
- A Boolean formula is **satisfiable** if some assignment of *TRUE*s and *FALSE*s to the variables make the formula true

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \bar{x} \wedge y \wedge z$ $\quad x = \text{FALSE}, y = \text{TRUE}, z = \text{TRUE}$
    - $\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)
- A Boolean formula is **satisfiable** if some assignment of *TRUE*s and *FALSE*s to the variables make the formula true

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \overline{x} \wedge y \wedge z$ $\qquad$ $x =$ FALSE, $y =$ TRUE, $z =$ TRUE
    - $\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)
- A Boolean formula is **satisfiable** if some assignment of *TRUE*s and *FALSE*s to the variables make the formula true
- SAT = $\{\langle \phi \rangle \mid \phi$ is a satisfiable Boolean formula$\}$

Boolean Formula

- Boolean formula: an expression involving Boolean variables and operations
  - Example:
    - $\phi = \overline{x} \wedge y \wedge z$ $\quad x = \text{FALSE}, y = \text{TRUE}, z = \text{TRUE}$ yes-instance
    - $\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$ no-instance
  - (Boolean) variables: x, y, z
- The Boolean variables can take on the values *TRUE* (1) and *FALSE* (0)
- A Boolean formula is **satisfiable** if some assignment of *TRUE*s and *FALSE*s to the variables make the formula true
- SAT = $\{\langle\phi\rangle \mid \phi$ is a satisfiable Boolean formula$\}$

SAT is in **NP**

- SAT $= \{\langle \phi \rangle \mid \phi$ is a satisfiable Boolean formula$\}$
- Prove SAT is in **NP** ⇔ Design a polynomial time verifier to decide SAT (with help from some c)

<Proof>

Let string c that encodes a truth assignment of variables in ϕ as a certificate

V = "On input $\langle\langle \phi \rangle, c\rangle$:

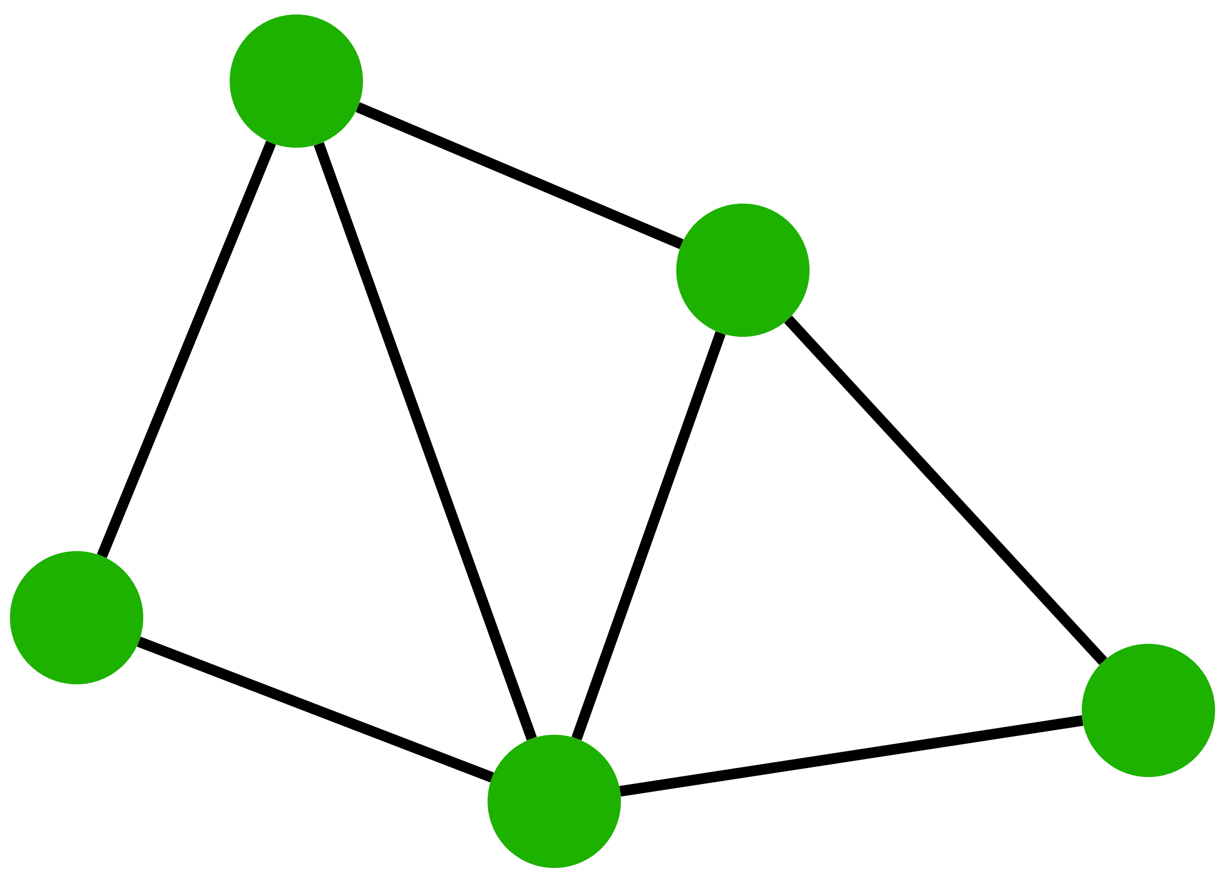
1. Replace the literals in ϕ by the truth assignments in c
2. Test whether the resulting ϕ is true
3. If 2 pass, *accept*; otherwise, *reject*."

For each replacement in Step 1, it takes at most linear time of scanning through the input. In total, it scan through the input ℓ times, where ℓ is the number of literals in ϕ. Step 2 can be done in one scan through the input. Hence, V runs in polynomial time in the input length.

D-HAM-PATH

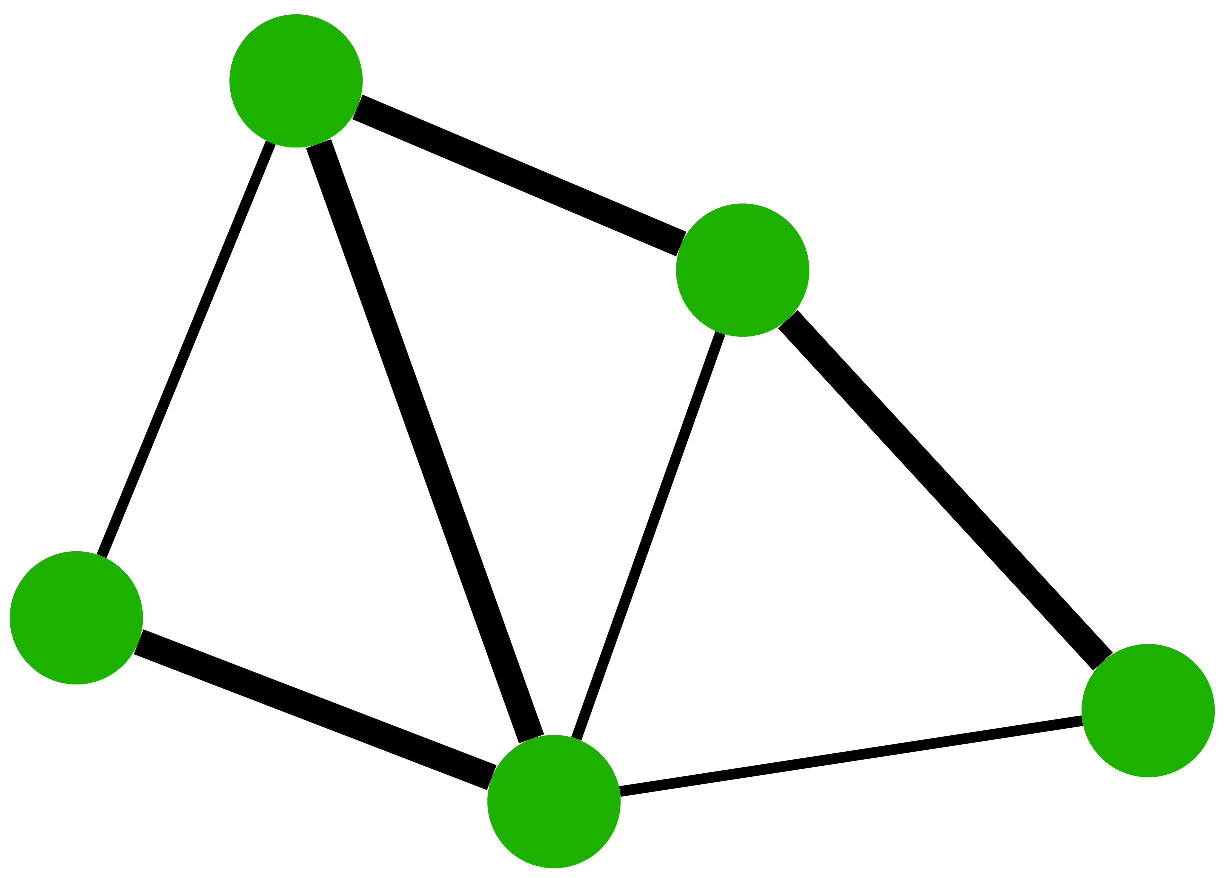
D-HAM-PATH

- A **Hamiltonian path** of a graph $G = (V, E)$ is a simple path that contains each vertex in V.



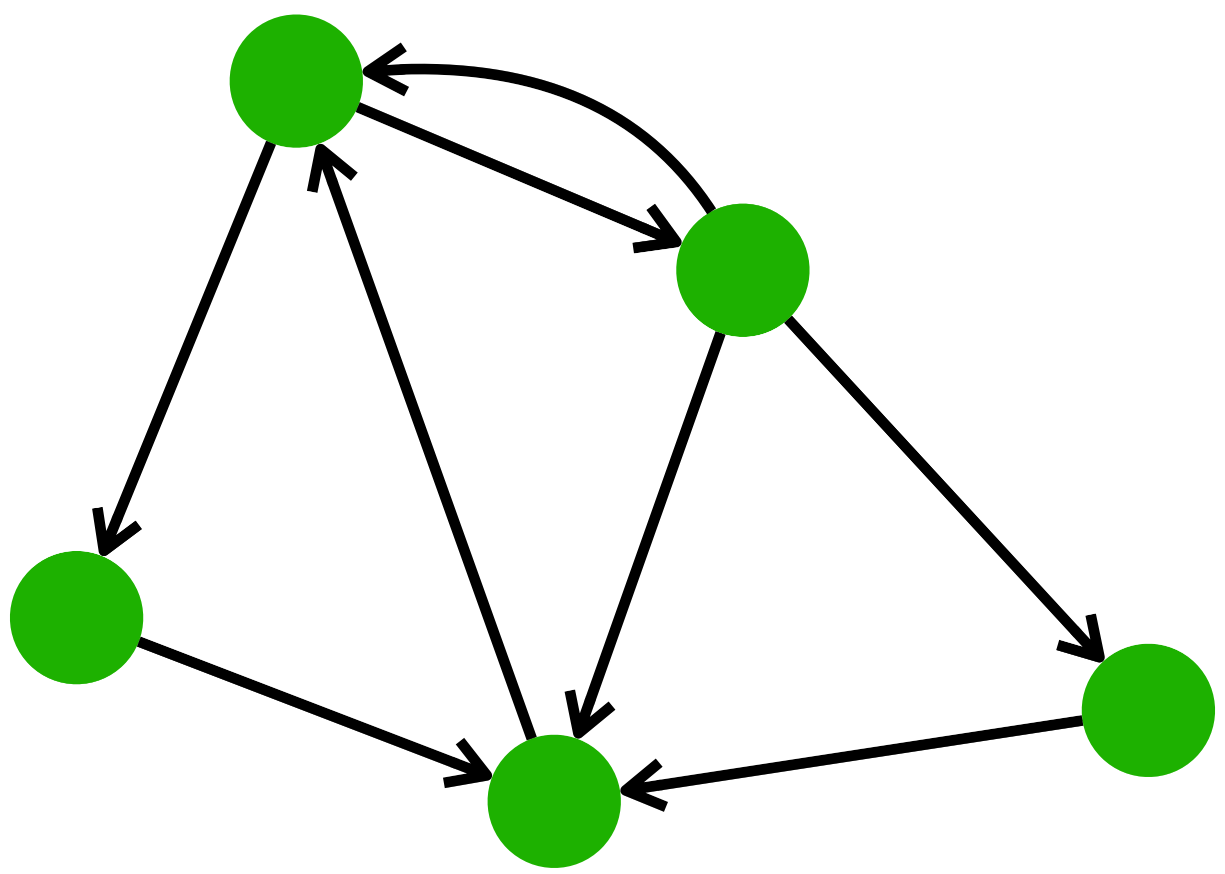
D-HAM-PATH

- A **Hamiltonian path** of a graph $G = (V, E)$ is a simple path that contains each vertex in V.



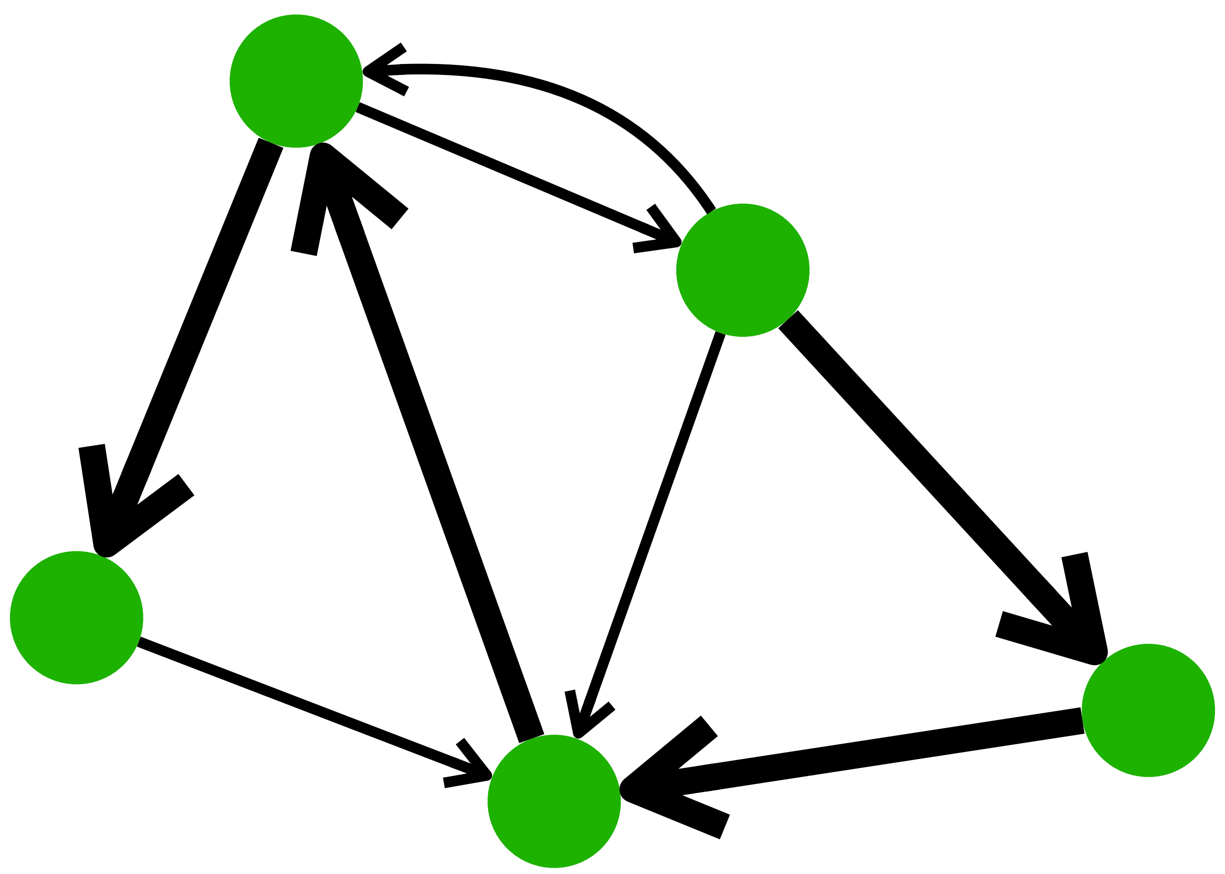
D-HAM-PATH

- A **Hamiltonian path** of a graph $G = (V, E)$ is a simple path that contains each vertex in V.



D-HAM-PATH

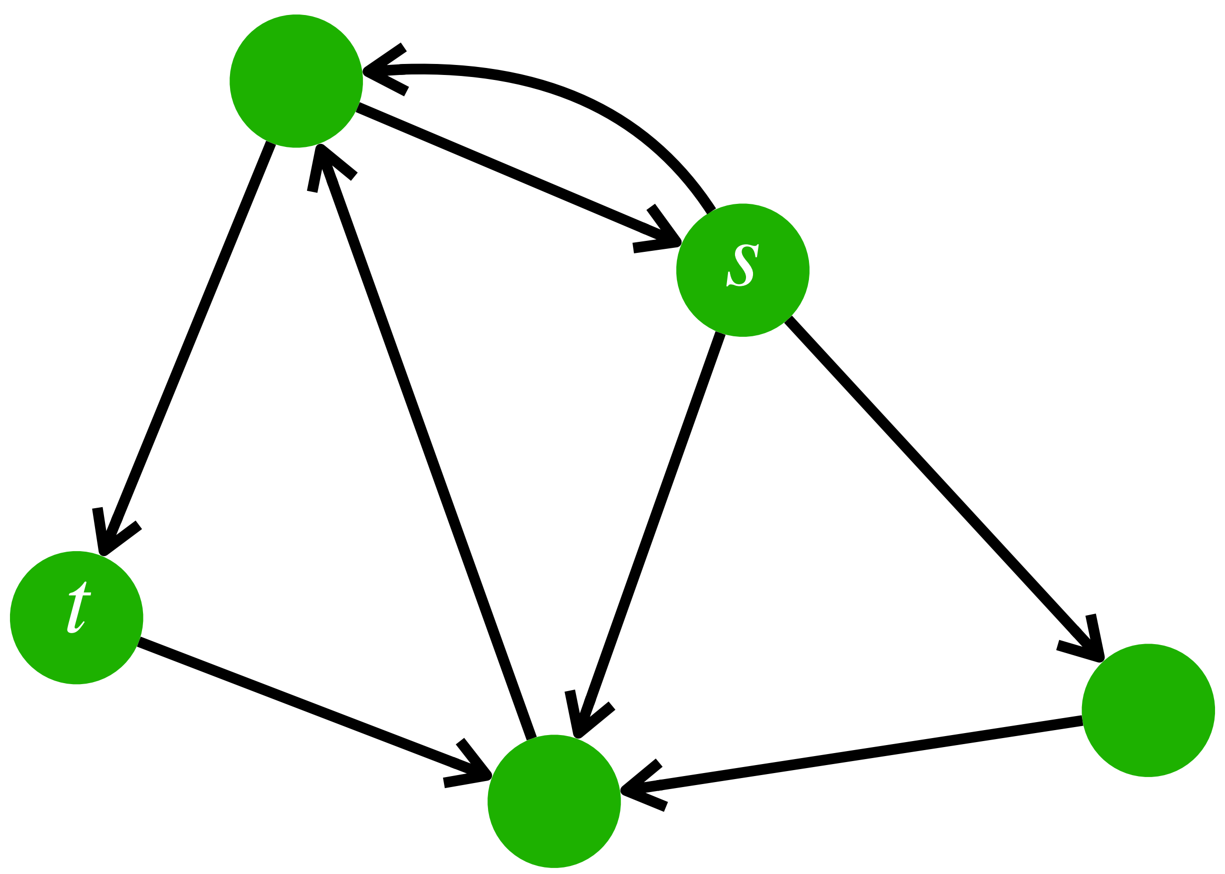
- A **Hamiltonian path** of a graph $G = (V, E)$ is a simple path that contains each vertex in V.



D-HAM-PATH

- A **Hamiltonian path** of a graph $G = (V, E)$ is a simple path that contains each vertex in V.

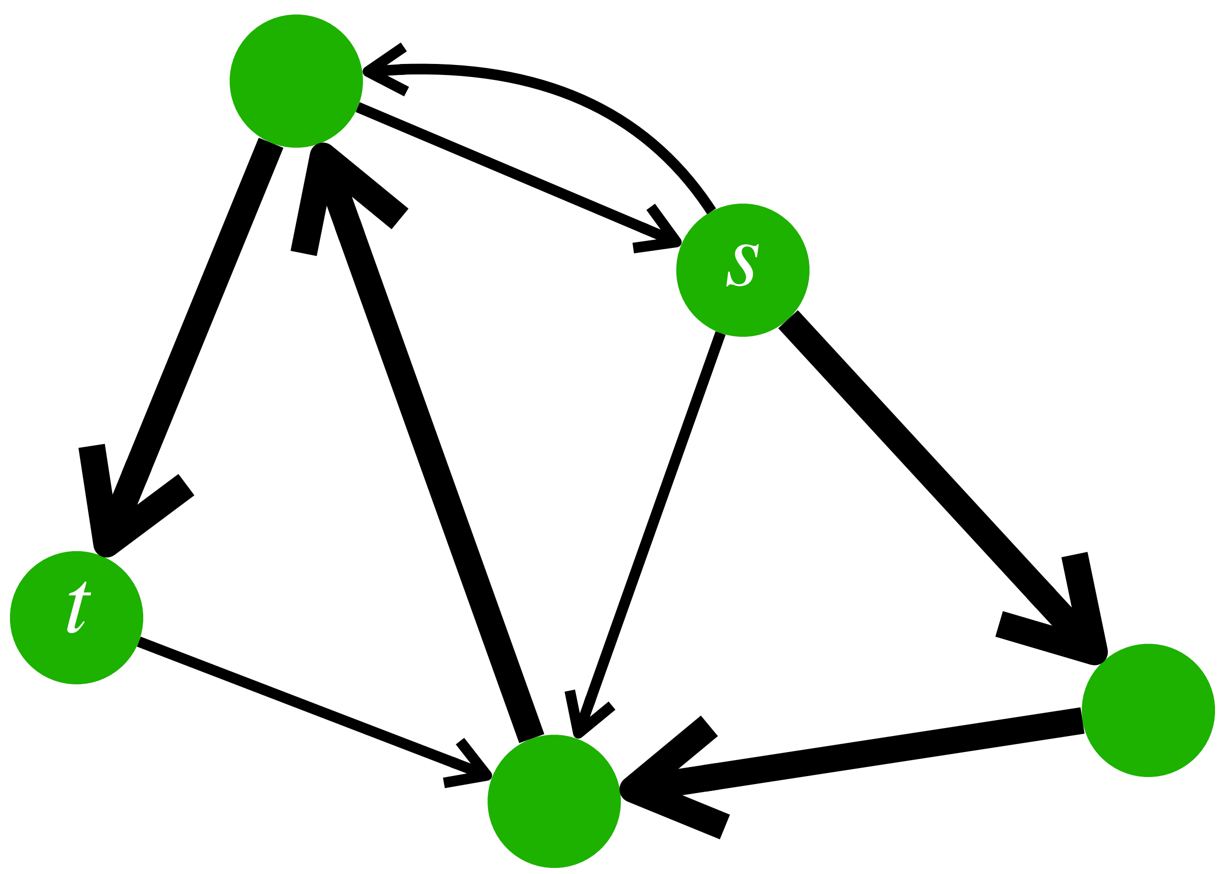
- D-HAM-PATH $= \{\langle G, s, t\rangle |\ G$ is a directed graph with a Hamiltonian path from s to $t\}$



D-HAM-PATH

- A **Hamiltonian path** of a graph $G = (V, E)$ is a simple path that contains each vertex in V.

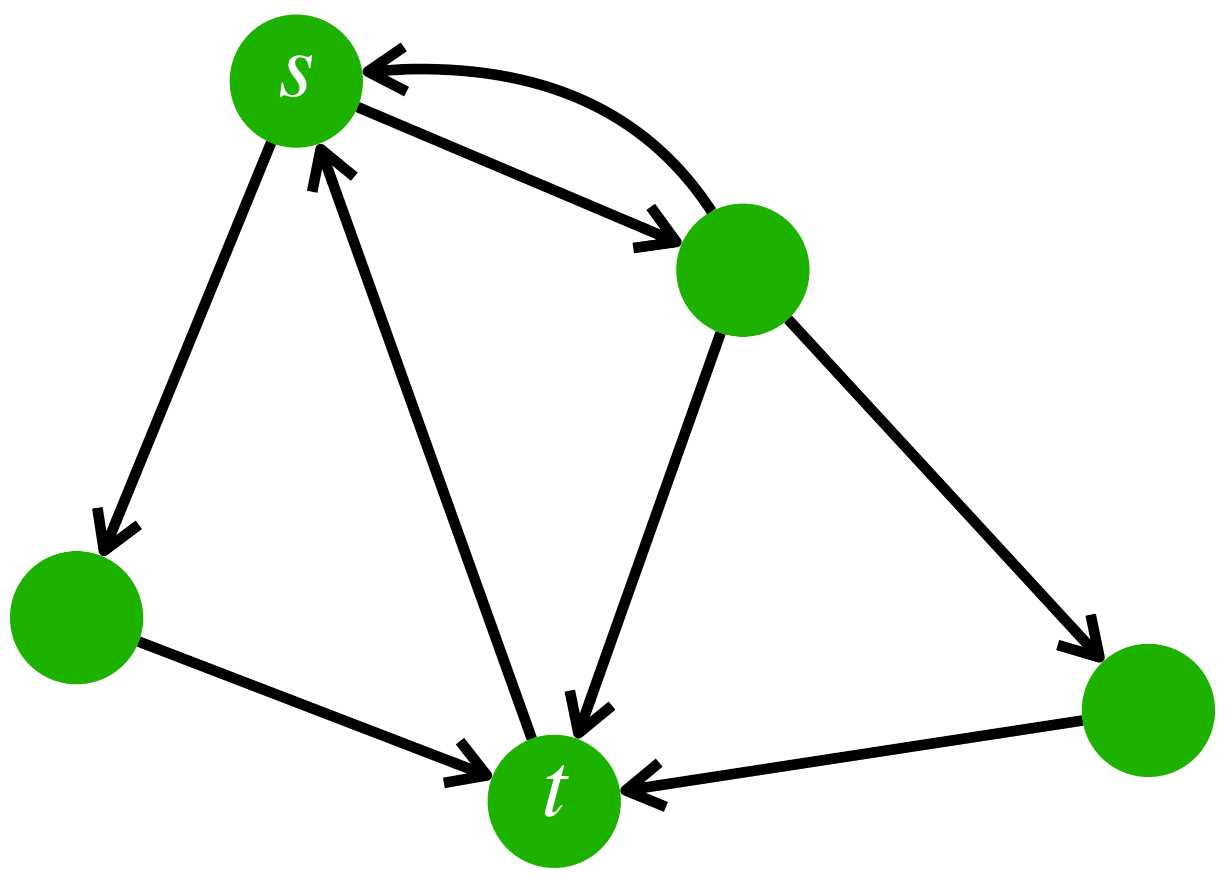
- D-HAM-PATH $= \{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$



D-HAM-PATH

- A **Hamiltonian path** of a graph $G = (V, E)$ is a simple path that contains each vertex in V.

- D-HAM-PATH $= \{\langle G, s, t\rangle |\ G$ is a directed graph with a Hamiltonian path from s to $t\}$



D-HAM-PATH is in **NP**

- D-HAM-PATH $= \{\langle G, s, t\rangle | \, G$ is a directed graph with a Hamiltonian path from s to $t\}$
- Prove D-HAM-PATH is in **NP** ⇔ Design a polynomial time verifier to decide D-HAM-PATH (with help from some c)

<Proof>

Let string c that encodes a permutation of vertices in G that forms a Hamiltonian walk starting from s and end at t as a certificate

V = "On input $\langle\langle G, s, t\rangle, c\rangle$:

1. Check if c is indeed a permutation of vertices in G starting from s and end at t
2. For every consecutive pair of vertices in c, v_i and v_{i+1}, check if there is an edge from v_i to v_{i+1} in G
3. If 1 and 2 both pass, *accept*; otherwise, *reject*."

For each element in c in Step 1, it takes at most linear time of scanning through the input. In total, it scan through the input n times, where n is the number of vertices in G. Each consecutive pair in Step 2 can be checked in one scan through the input, and there are at most $O(n)$ pairs. Hence, V runs in polynomial time in the input length.

What Happened

- To show that a problem is in **NP**, we can show that it is polynomial-time verifiable

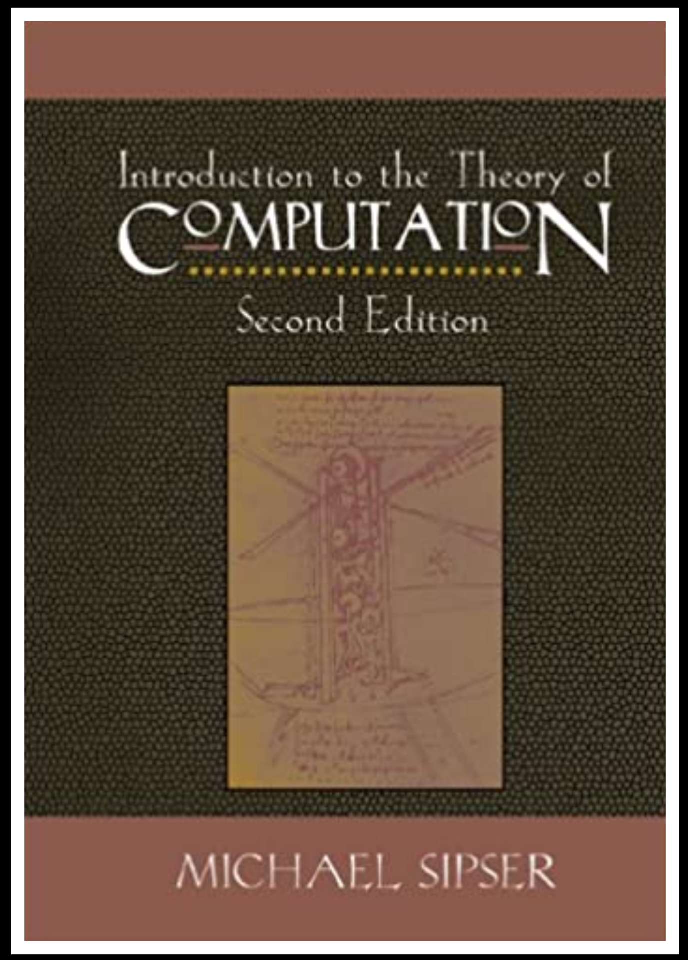
<Proof Idea>

1. Show that for any yes instance w, there is a polynomial-size certificate c.
2. Design a verifier V on input $\langle w, c \rangle$ that *accepts* all $w \in A$ and *rejects* all $w \notin A$
3. Show that V runs in polynomial time (in the length of w)

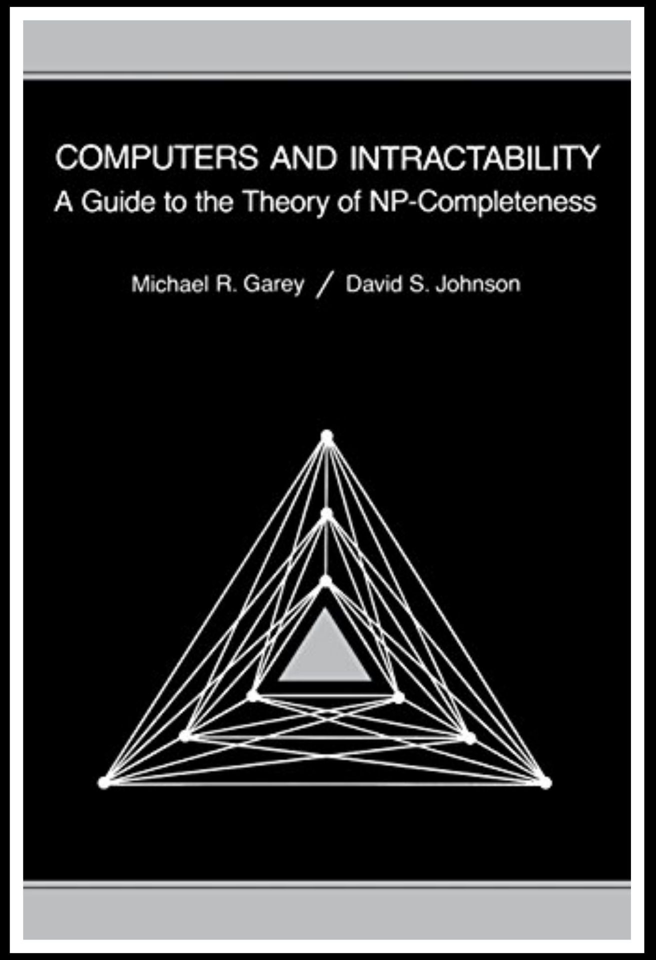
Why **P** and **NP**?

- There are many $f(n)$ time Turing machine variations that have an equivalent poly($f(n)$) time single-tape Turing machine
- Every $f(n)$ time non-deterministic Turing machine has an equivalent $2^{O(f(n))}$ time deterministic Turing machine
- There is at most a square or polynomial difference between the time complexity of problems measured on deterministic single-tape and many Turing machine variations
- There is at most an exponential difference between the time complexity of problems on deterministic and nondeterministic Turing machines

Reference



- *Introduction to the Theory of Computation* by Michael Sipser



- *Computers and Intractability - A Guide to the Theory of NP-Completeness* by Michael R. Garey and David S. Johnson

- *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

- *Computational Complexity* by Christos h. Papadimitriou