The Universal Turing Machine

Where We Are

- The **Church-Turing thesis** tells us that all effective models of computation are no more powerful than a Turing machine.
- We have a family of programming languages (**WB*n***) that are equivalent to Turing machines.
- Let's start exploring what we can do with this new model of computation.

Important Ideas for Today

- The material from today will lay the groundwork for the next few weeks.
- Key concepts:
  - **Encodings**.
  - **Universal machines.**
  - **High-level specifications**
  - **Nondeterministic Turing machines.**
  - **Decidable and recognizable languages**.

Encodings

An Interesting Observation

- **WB** programs give us a way to check whether a string is contained within some language.
- **WB** programs themselves are strings.
- This means that we can run **WB** programs, taking other **WB** programs as input!
- Major questions:
  - Is this specific to **WB** programs?
  - What can we do with this knowledge?

There is a subtle flaw in this reasoning:

"Because **WB** programs are strings, **WB** programs can be run on the source code of other **WB** programs."

What is it?

The Alphabet Problem

- **WB** programs are described using multiple characters: letters, digits, colons, newlines, etc., plus potentially all tape symbols being used.
- Not all **WB** programs are written for languages over this alphabet; in fact, most do not.
- We cannot directly write the source of a **WB** program onto the input tape of another **WB** program.
- Can we fix this?

A Better Encoding

- We will restrict ourselves to talking about languages over alphabets containing at least two symbols in them.
- Let's assume, without loss of generality, that we're dealing with the alphabet $\mathbb{B}$ = { `0`, `1` }.
- It's always possible to encode a string with a large alphabet using a string with just the binary alphabet.
  - Think about how real computers work.

Encoding **WB** Programs

- Determine how many different characters are needed to represent the program.
  - This includes letters for keywords, etc. along with tape symbols.
- Build a fixed-length binary encoding for each symbol in some canonical order.
  - Perhaps write out keyword symbols first, then the blank symbol, then the rest of the input symbols, etc.
- Replace each character in the program with the binary encoding.

Notation for Encodings

- If *P* is a **WB** program, we will denote its binary encoding as ***⟨P⟩***.
- Don't worry too much about the details about how exactly you would compute ⟨*P*⟩; the important part is that there's at least one way to do it.

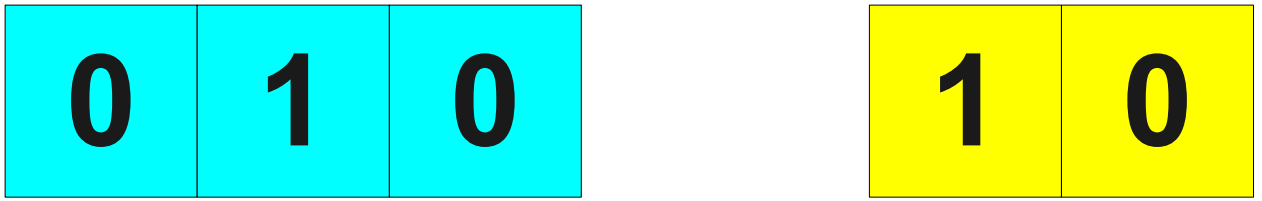
Encoding Other Automata

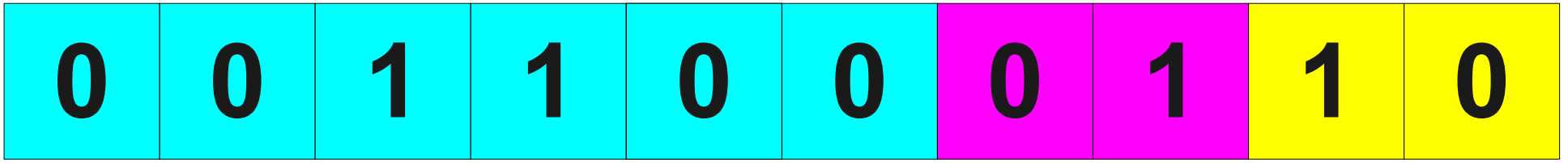
- Using similar techniques, we can encode all the other automata we've seen so far as binary strings.
- For example, if M is a Turing machine, $\langle M \rangle$ refers to a binary encoding of it.
- More generally, if we have any object O we want to encode as a binary string, we can write it out as $\langle O \rangle$.

Encoding Multiple Objects

- Suppose that we want to provide an encoding of multiple objects.
  - Several **WB** programs.
  - A **WB** program and a string.
  - A graph and a path in the graph.
  - A rock and a hard place.
- There are many ways that we can do this.

One Encoding Scheme





Encoding Multiple Objects

- Given several different objects O_1, O_2, …, O_n, we can represent the encoding of those *n* objects as $\langle O_1, O_2, \ldots, O_n \rangle$.
- Example:
  - If *P* is a **WB** program and *w* is a string, then ⟨*P*, *w*⟩ is an encoding of that program and that string.
  - If *G* is a context-free grammar and *P* is a PDA, then ⟨*G*, *P*⟩ is an encoding of that grammar and that PDA.

Universal Machines

Universal Machines and Programs

- **Theorem**: There is a Turing machine U_{TM} called the **universal Turing machine** that, when run on $\langle M, w \rangle$, where M is a Turing machine and w is a string, simulates M running on w.
- **Theorem**: There is a **WB** program U_{WB} called the **universal WB program** that, when run on $\langle P, w \rangle$, where P is a **WB** program and w is a string, simulates P running on w.

Why This Matters

- **As a historically significant achievement.**
  - The universal Turing machine might be the very first “complicated” algorithm ever designed for a computer.
  - Motivation for the “stored-program” model of computers.
- **As a justification for the Church-Turing thesis.**
  - All sufficiently powerful models of computation can simulate one another.
- **As a stepping stone to building elaborate models of computation.**
  - More on that later today.
- **As a stepping stone to finding unsolvable problems.**
  - More on that on Friday.

Sketch of the Universal **WB** Program

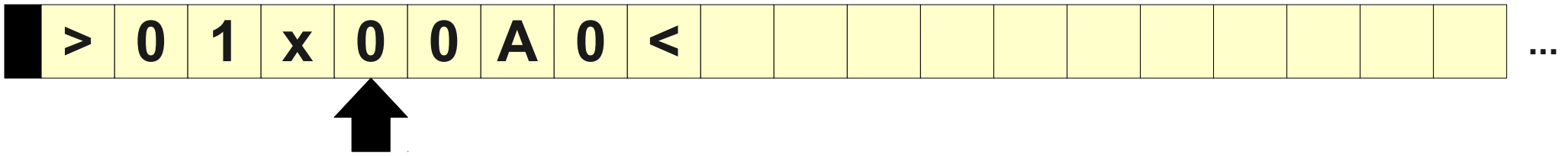
- It is a bit easier to understand the universal **WB** program than the universal TM.
- We will write the universal **WB** program in **WB6** (finite variables, multiple tracks, multiple tapes, multiple stacks) for simplicity, and can then "compile" it down into normal **WB**.

Sketch of the Universal **WB** Program

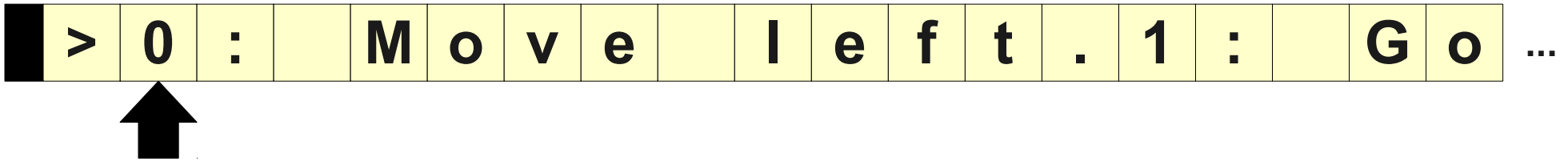
- To run a **WB** program, at each point in time we need to track three pieces of information:
  - The contents of the tape.
  - The location of the read head.
  - The index of the current instruction.
- Our program will enter a loop and repeatedly do the following:
  - Find the next instruction to execute.
  - Simulate that execution on the simulated tape.

Sketch of the Universal **WB** Program

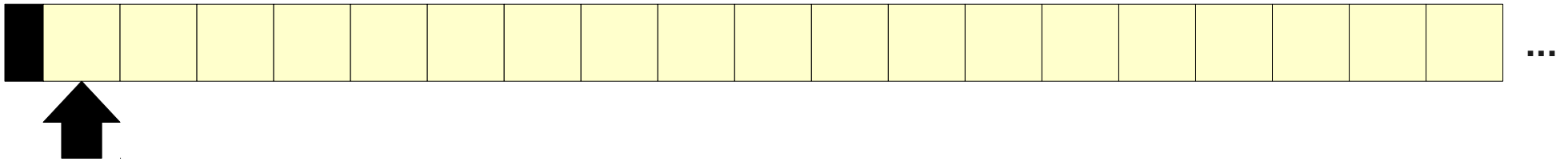
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

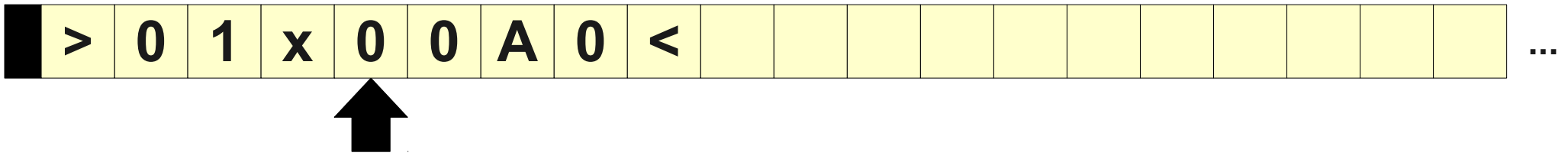


Variables for intermediate storage.

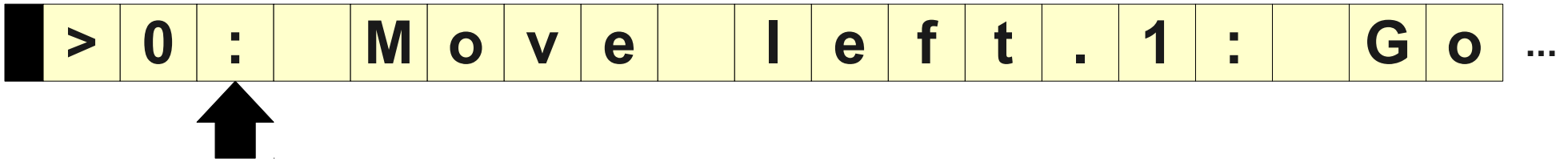
Instr **Letter**

Sketch of the Universal **WB** Program

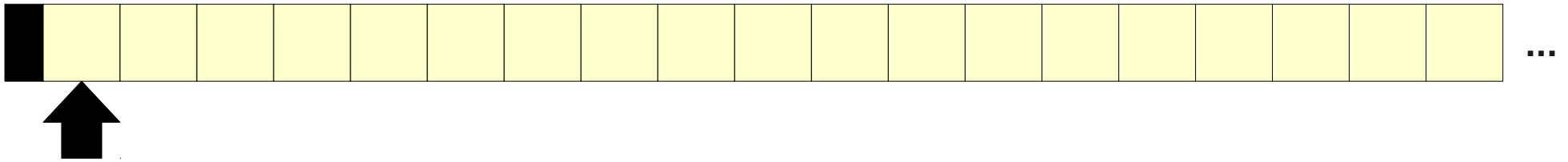
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

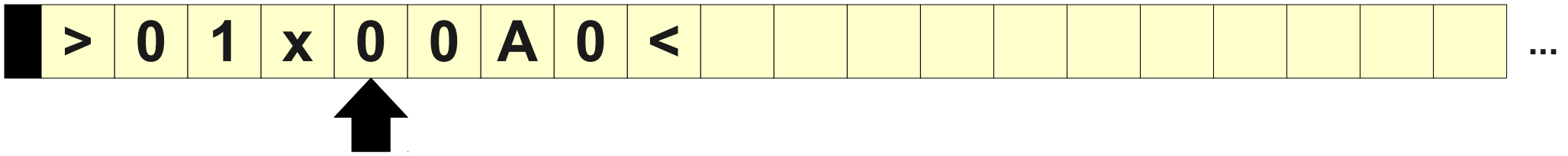


Variables for intermediate storage.

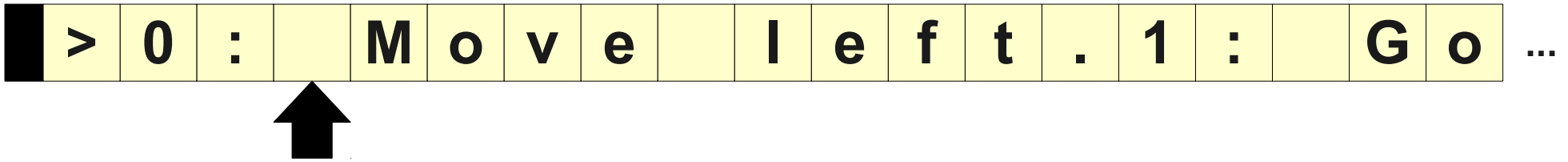
Instr **Letter**

Sketch of the Universal **WB** Program

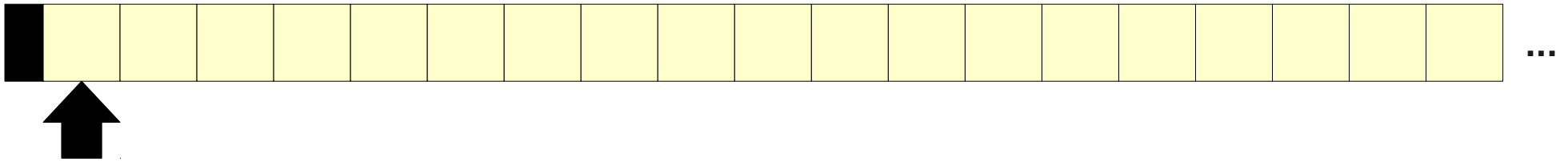
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

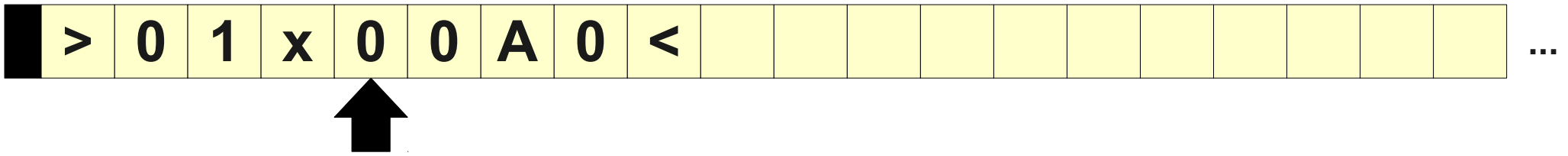


Variables for intermediate storage.

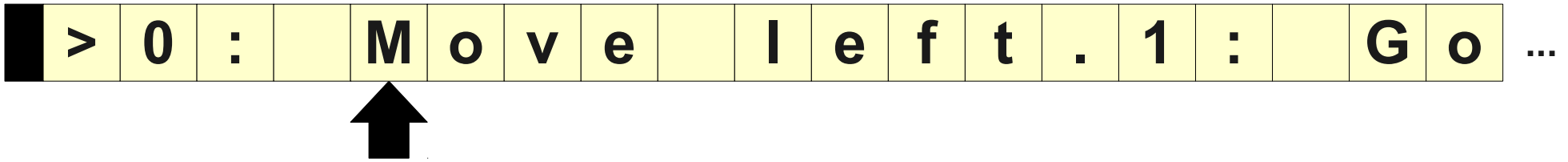
Instr **Letter**

Sketch of the Universal **WB** Program

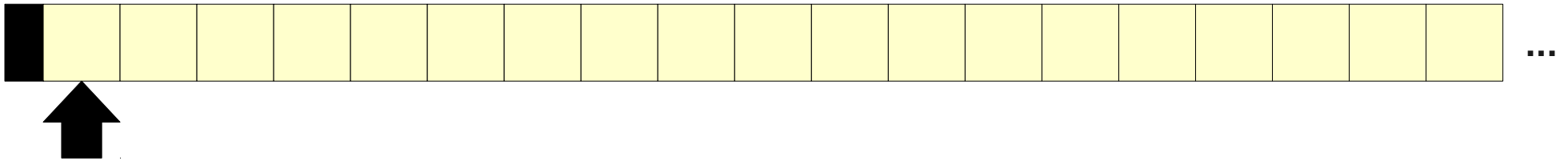
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

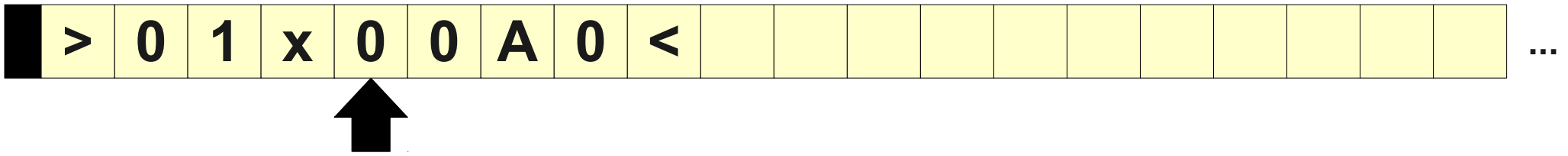


Variables for intermediate storage.

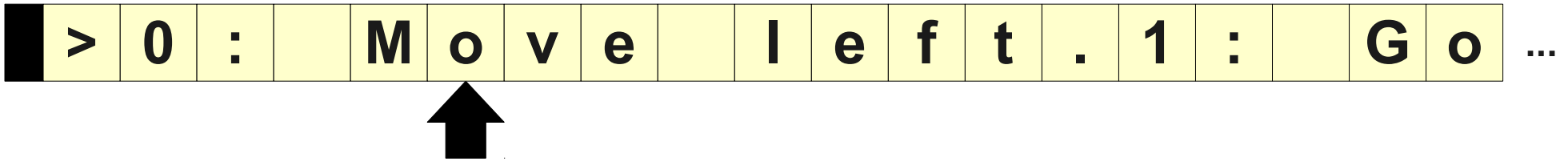
Instr **Letter**

Sketch of the Universal **WB** Program

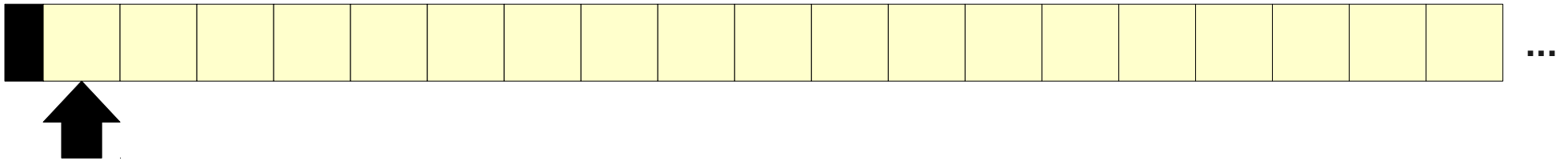
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

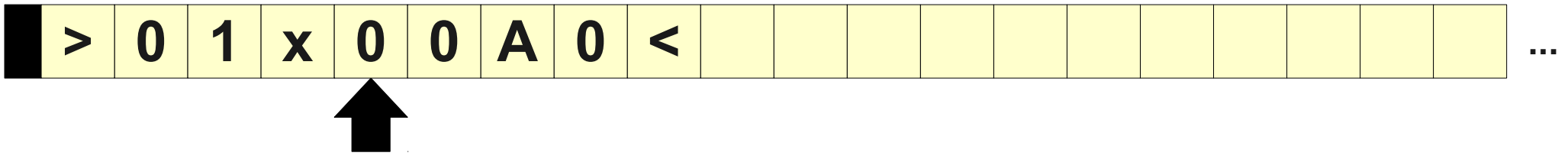


Variables for intermediate storage.

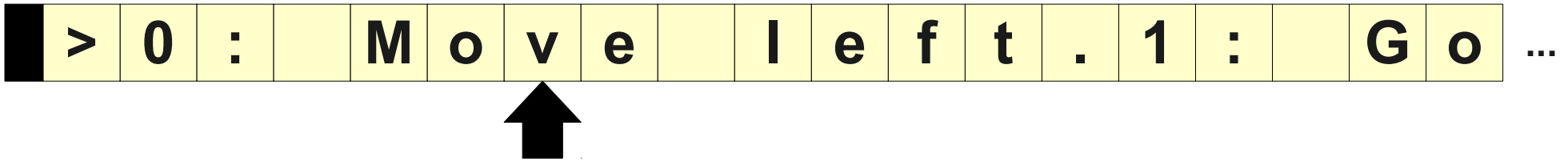
Instr **Letter**

Sketch of the Universal **WB** Program

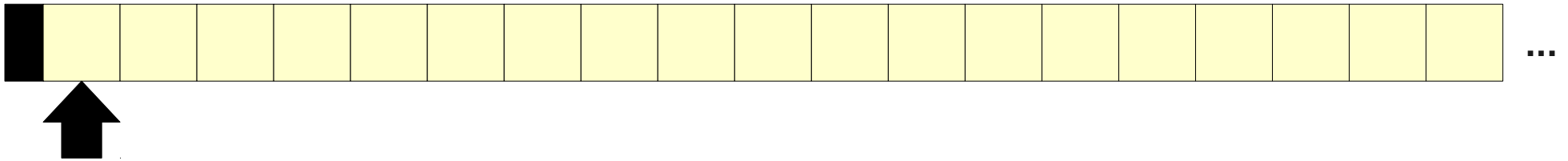
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

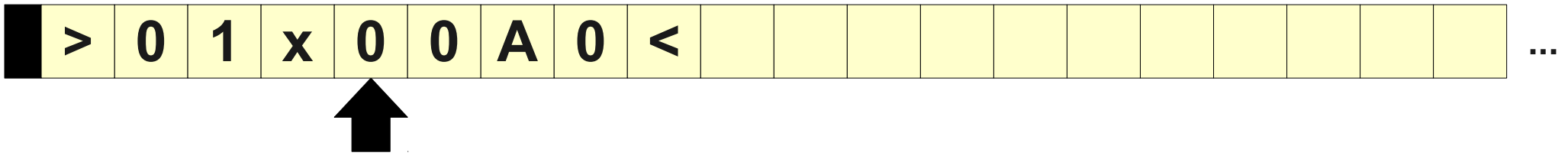


Variables for intermediate storage.

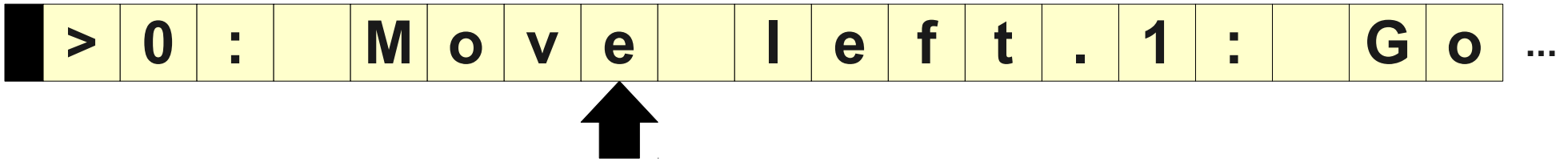
Instr **Letter**

Sketch of the Universal **WB** Program

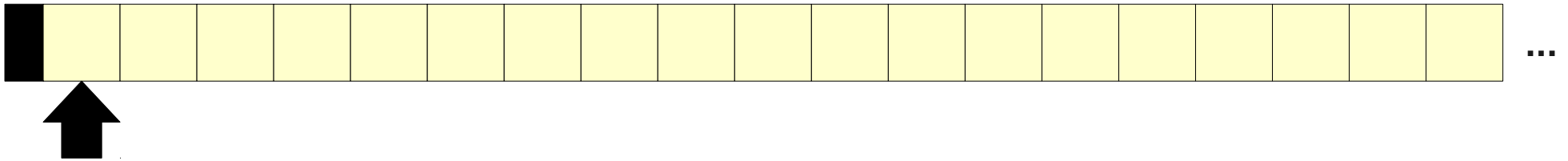
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

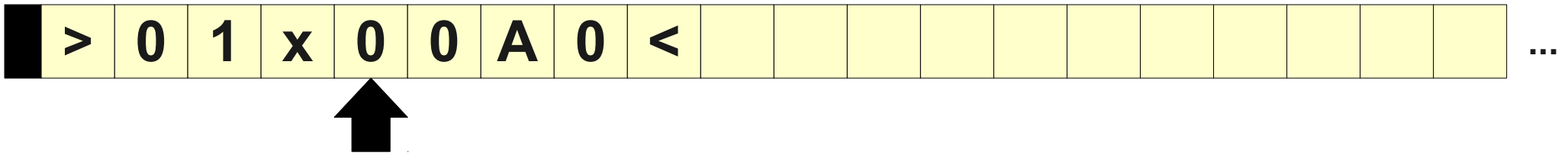


Variables for intermediate storage.

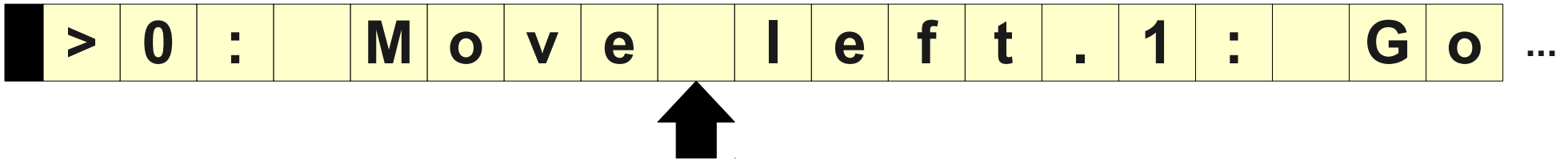
Instr **Letter**

Sketch of the Universal **WB** Program

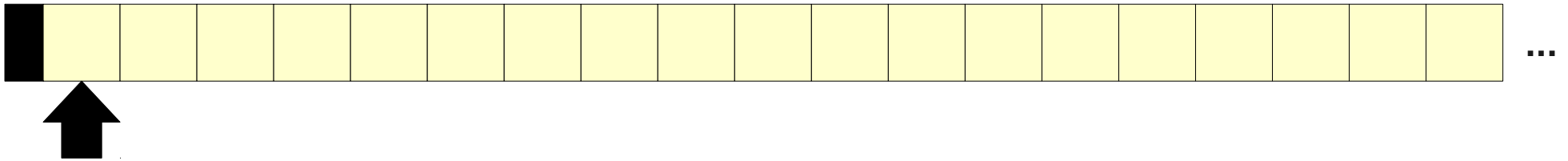
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

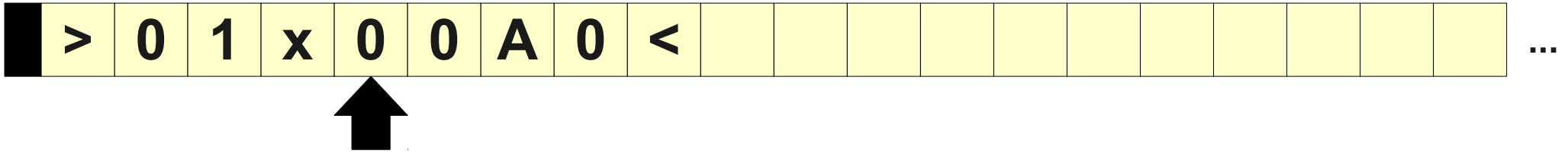


Variables for intermediate storage.

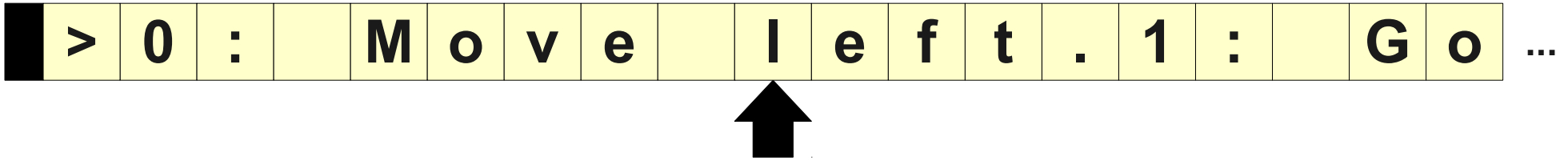
Instr **Letter**

Sketch of the Universal **WB** Program

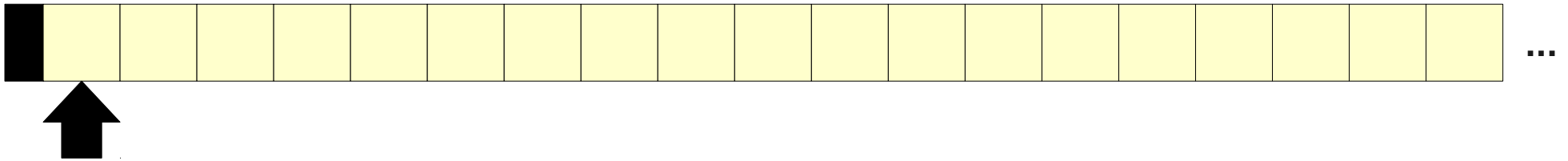
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

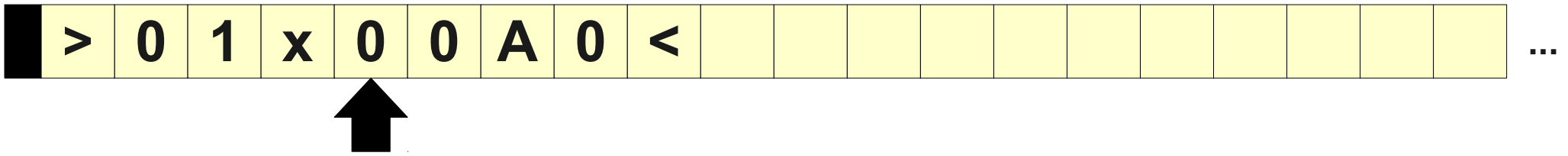


Variables for intermediate storage.

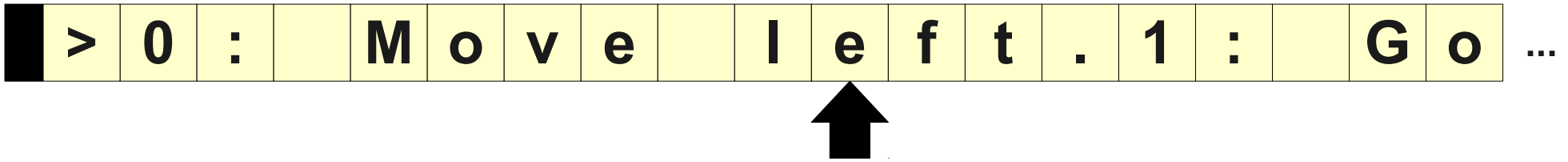
Instr **Letter**

Sketch of the Universal **WB** Program

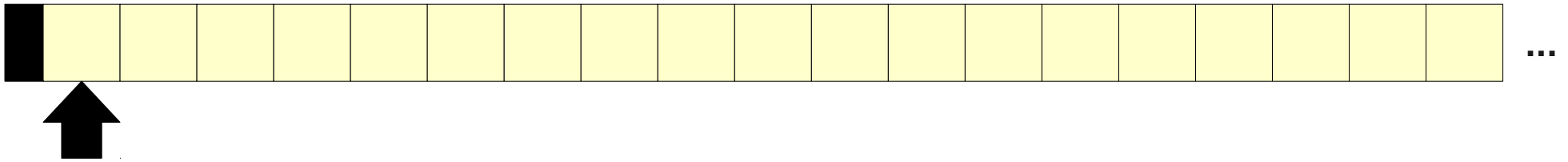
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

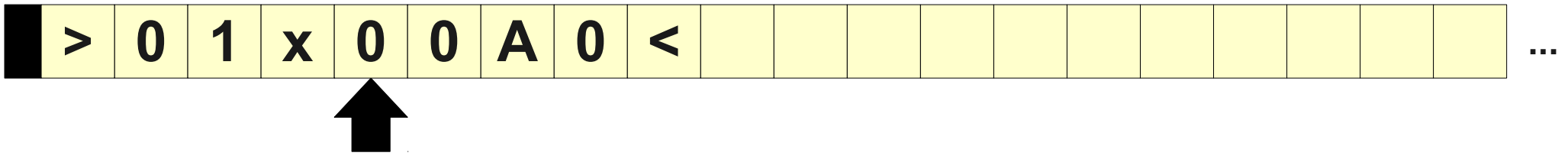


Variables for intermediate storage.

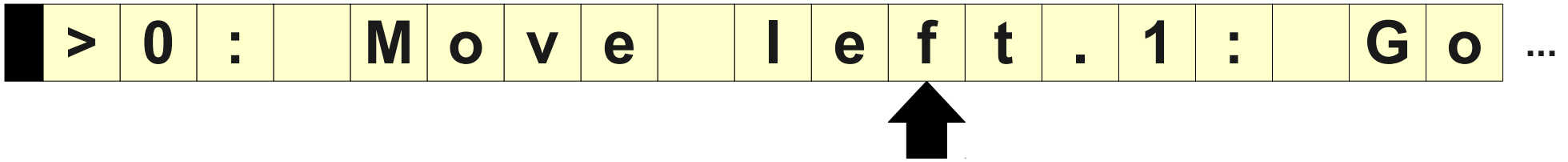
Instr **Letter**

Sketch of the Universal **WB** Program

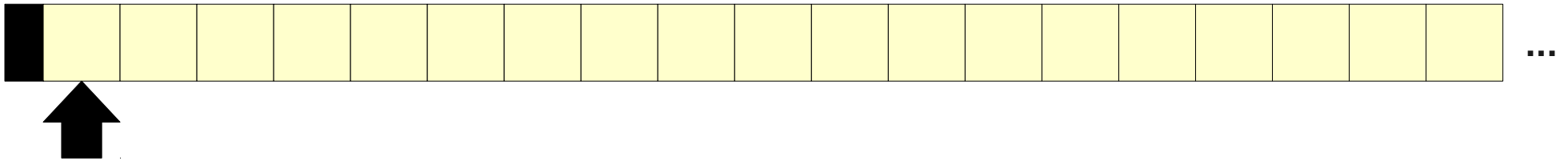
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

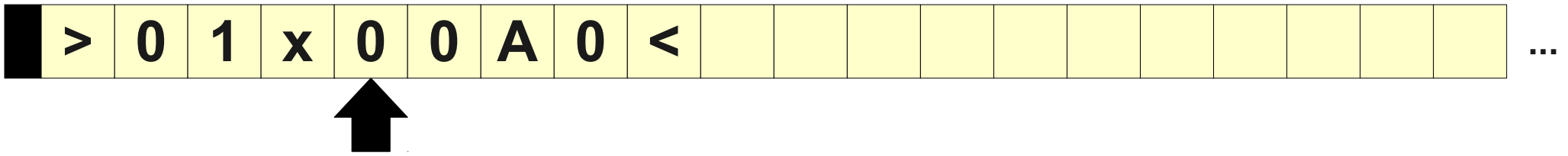


Variables for intermediate storage.

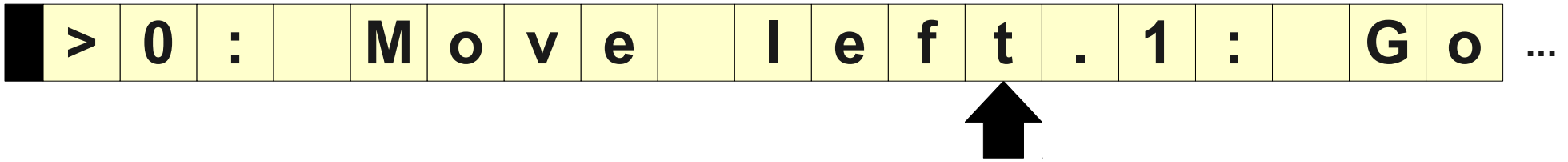
Instr **Letter**

Sketch of the Universal **WB** Program

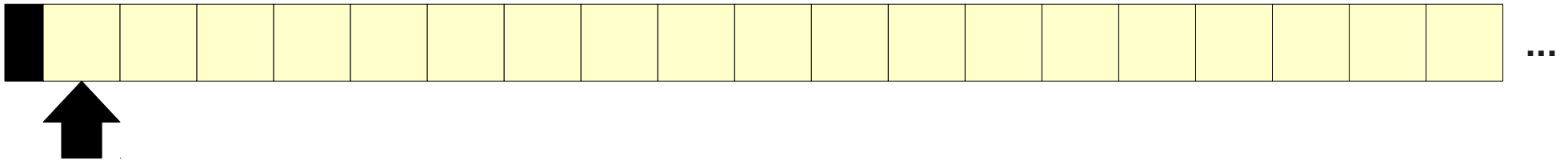
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

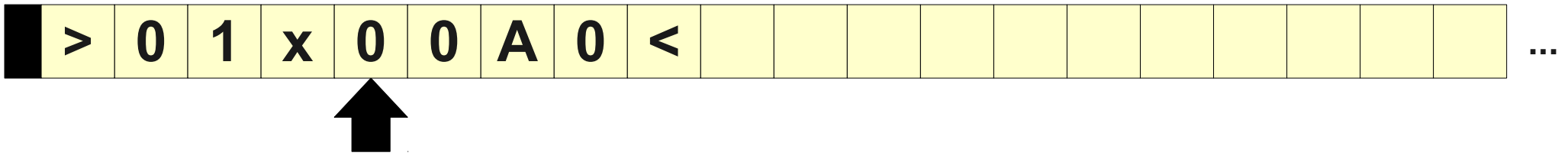


Variables for intermediate storage.

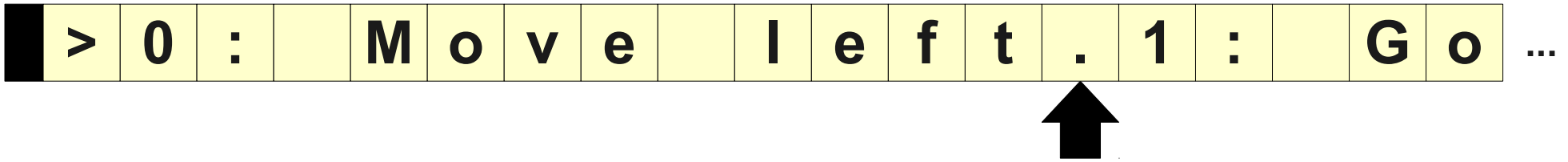
Instr **Letter**

Sketch of the Universal **WB** Program

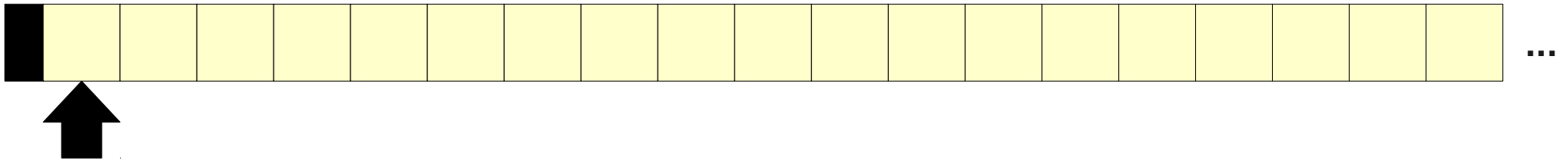
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

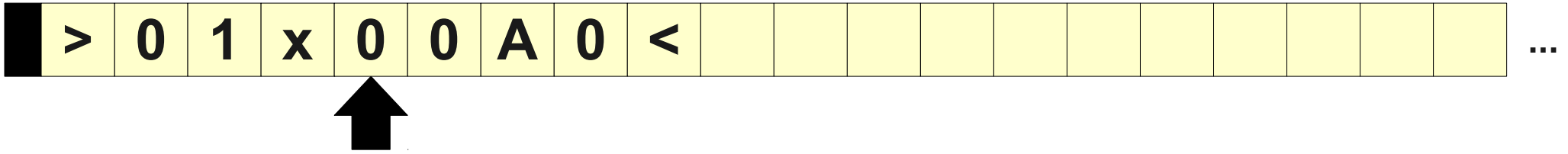


Variables for intermediate storage.

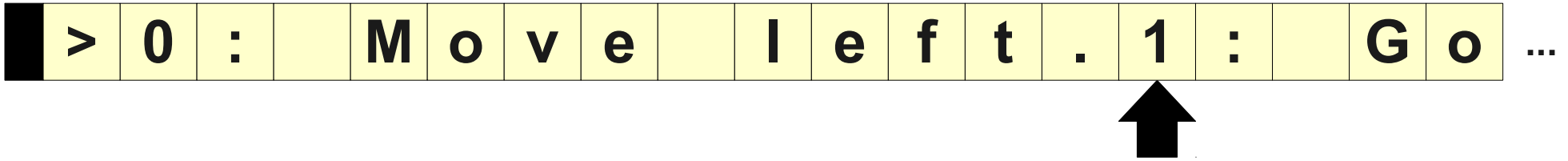
Instr **Letter**

Sketch of the Universal **WB** Program

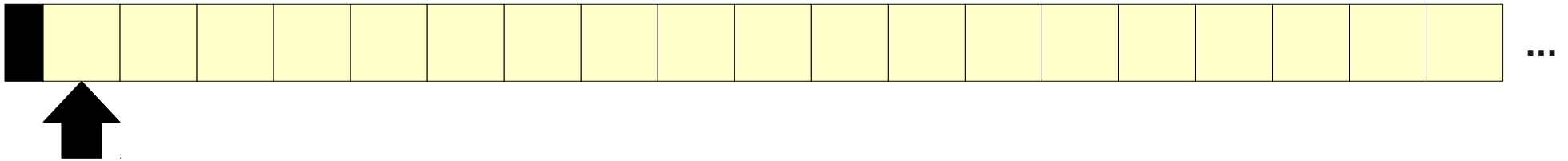
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

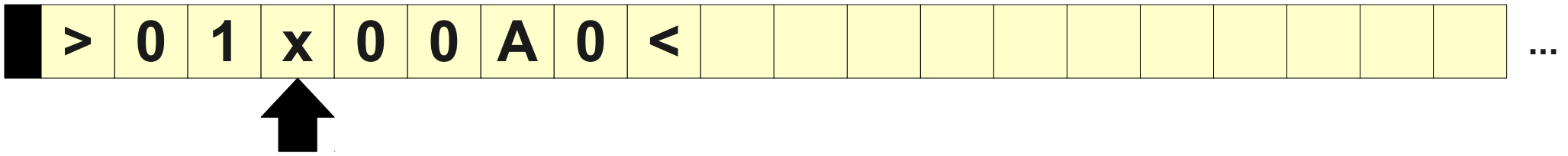


Variables for intermediate storage.

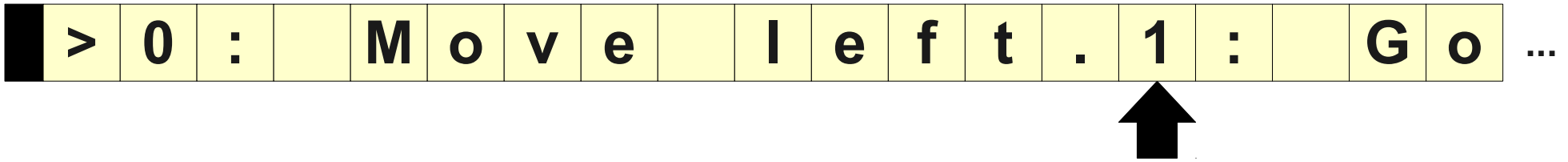
Instr **ML** **Letter**

Sketch of the Universal **WB** Program

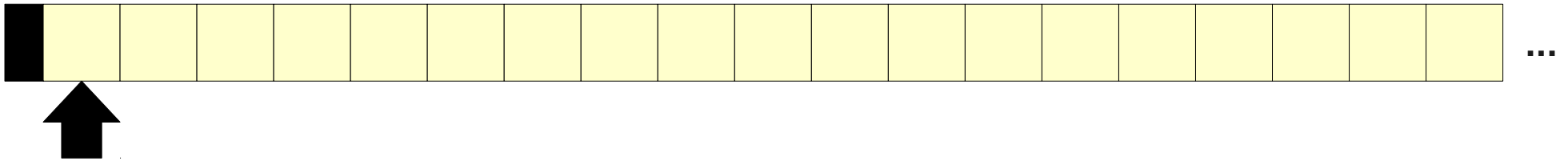
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

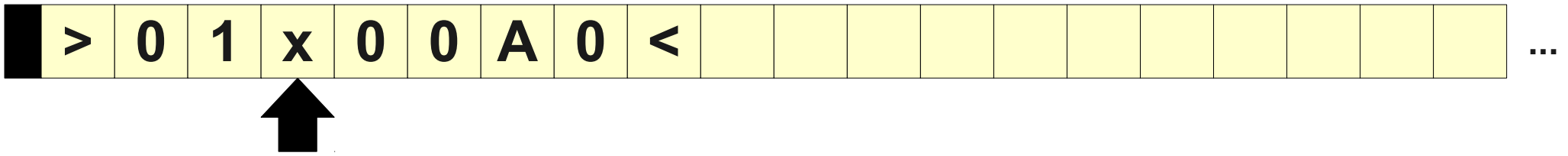


Variables for intermediate storage.

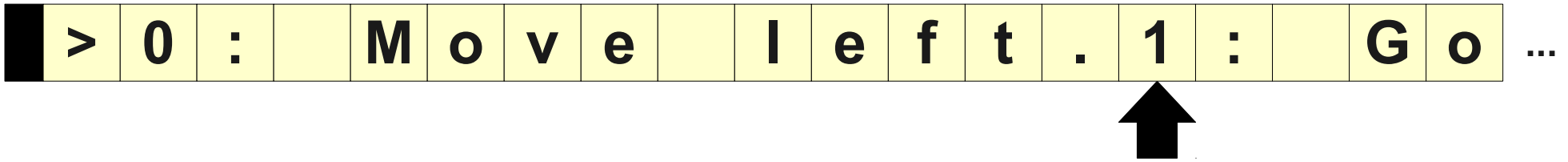
Instr **ML** **Letter**

Sketch of the Universal **WB** Program

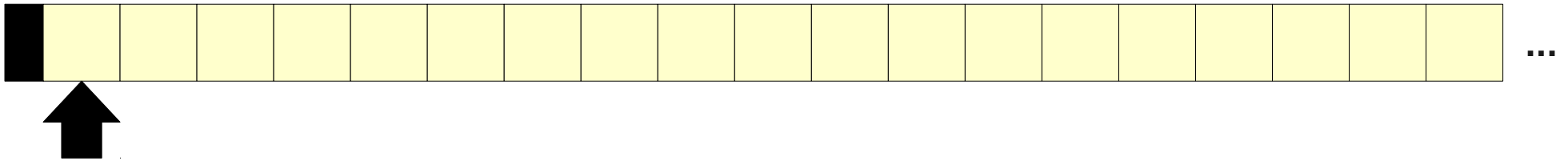
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

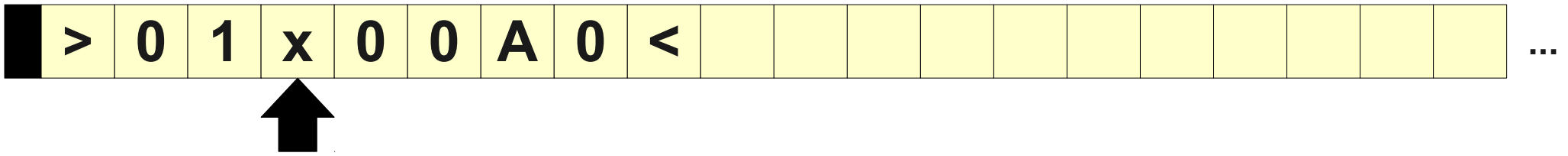


Variables for intermediate storage.

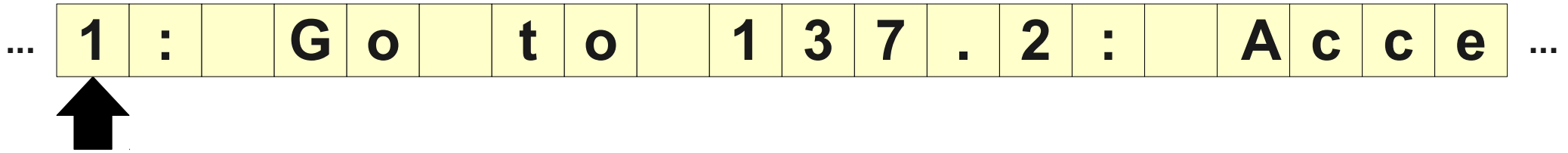
Instr **Letter**

Sketch of the Universal **WB** Program

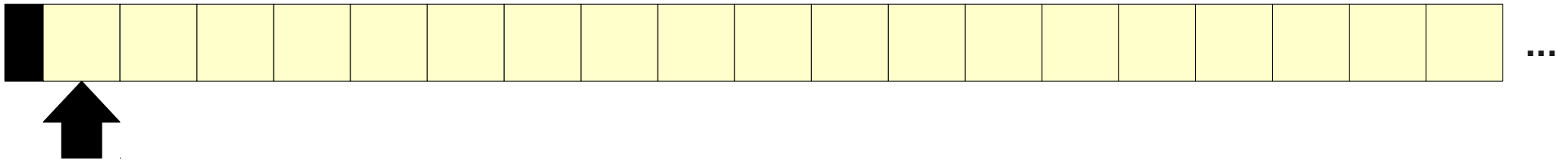
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

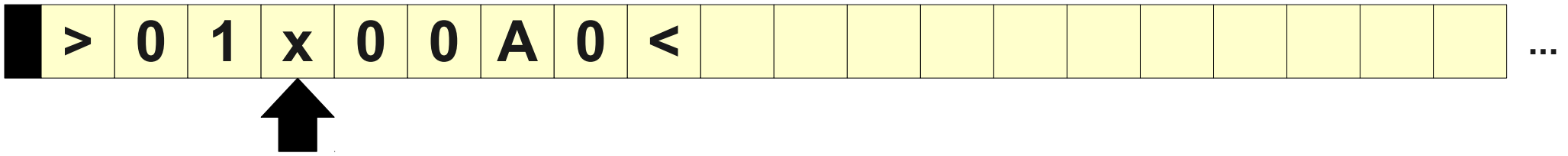


Variables for intermediate storage.

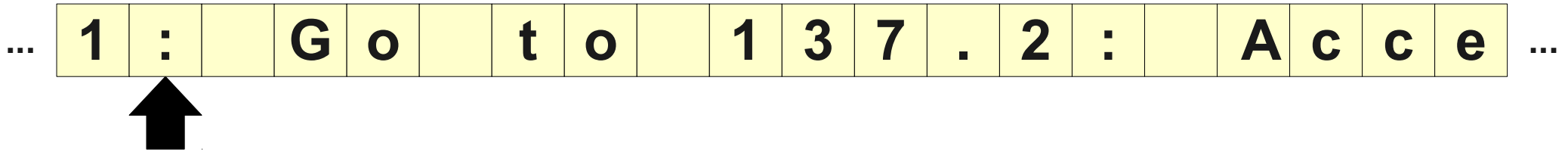
Instr **Letter**

Sketch of the Universal **WB** Program

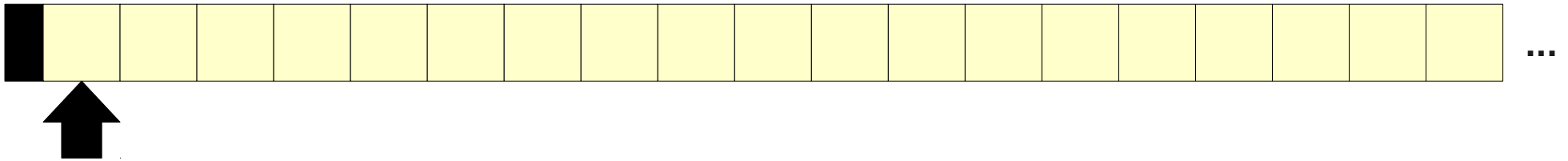
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

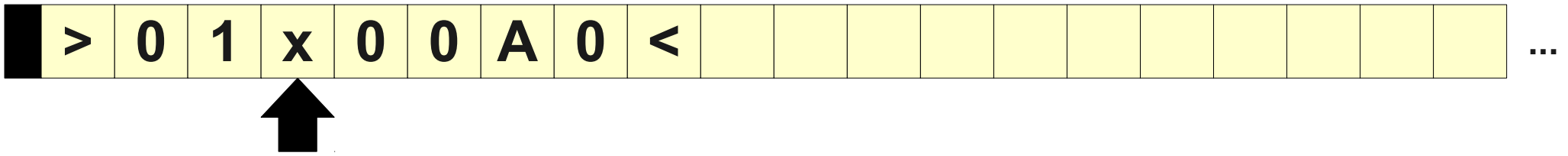


Variables for intermediate storage.

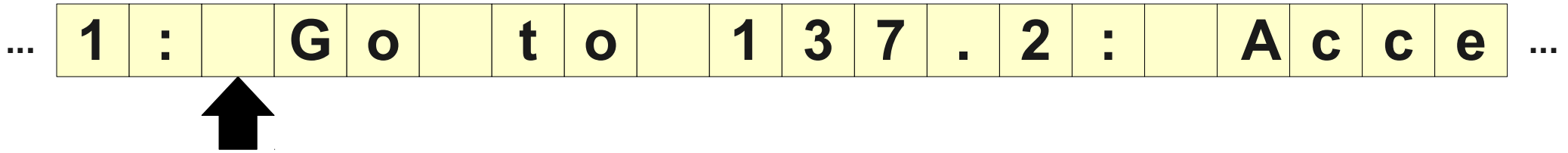
Instr **Letter**

Sketch of the Universal **WB** Program

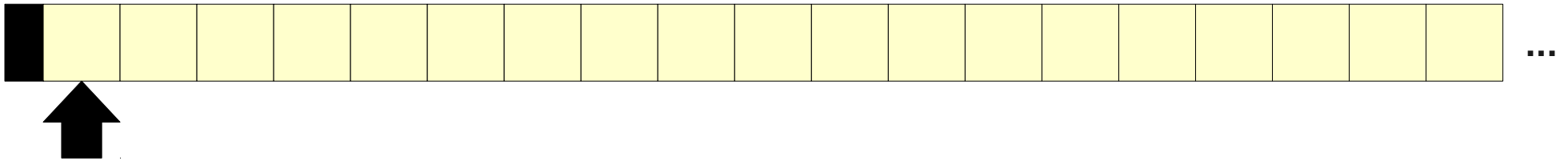
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

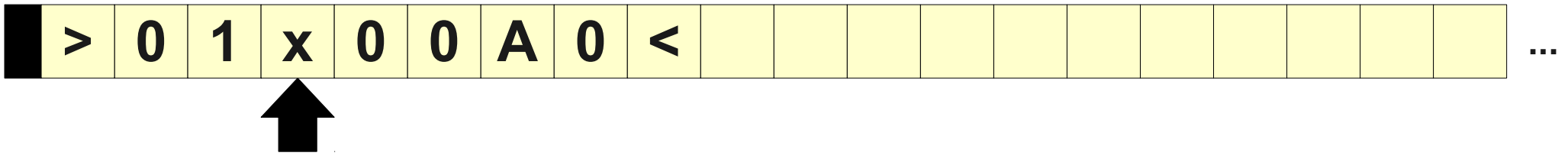


Variables for intermediate storage.

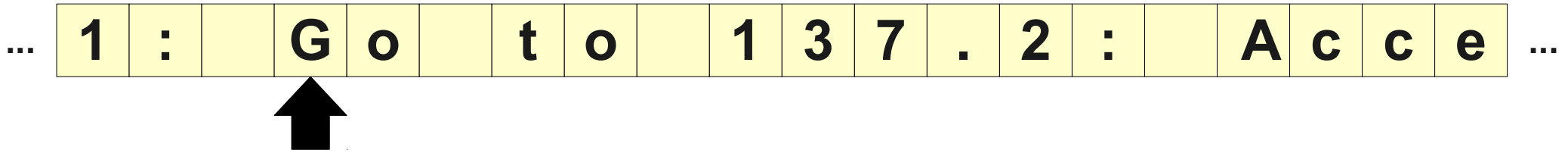
Instr **Letter**

Sketch of the Universal **WB** Program

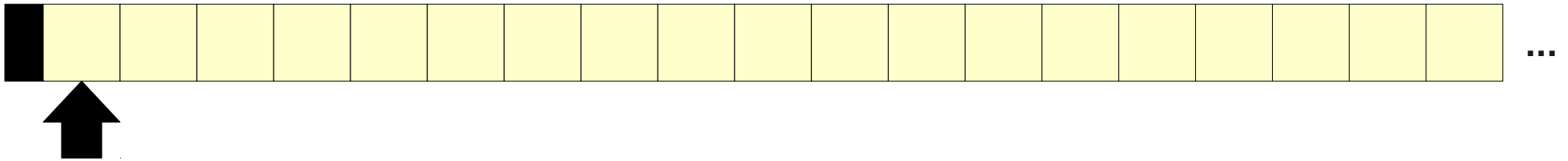
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

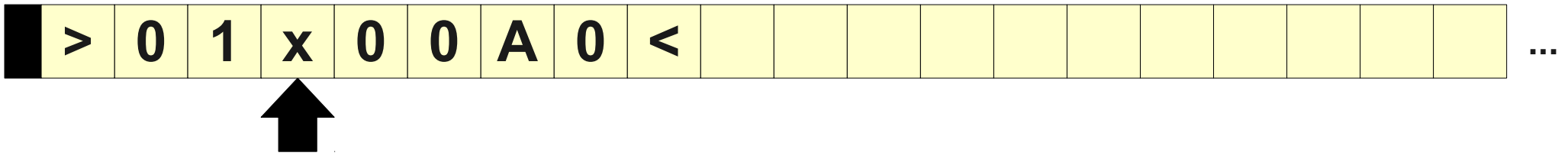


Variables for intermediate storage.

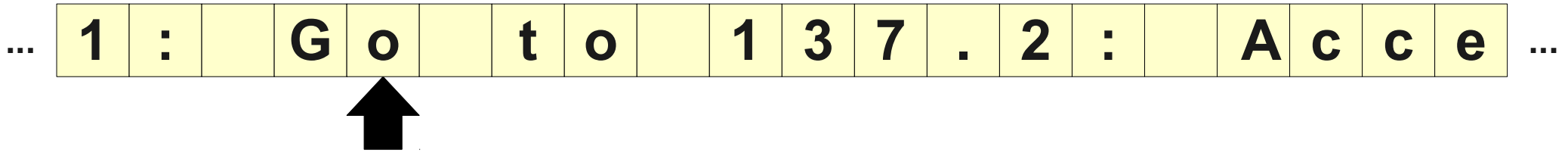
Instr **Letter**

Sketch of the Universal **WB** Program

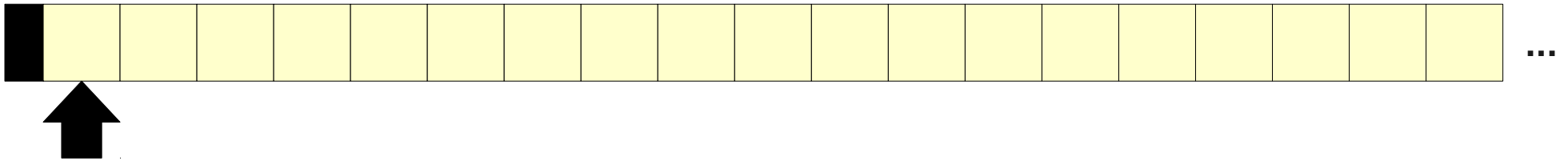
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

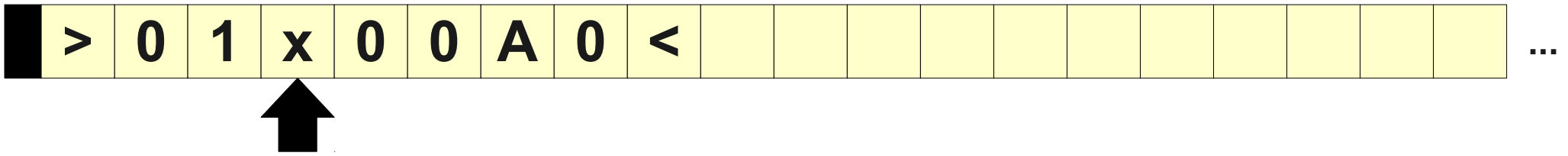


Variables for intermediate storage.

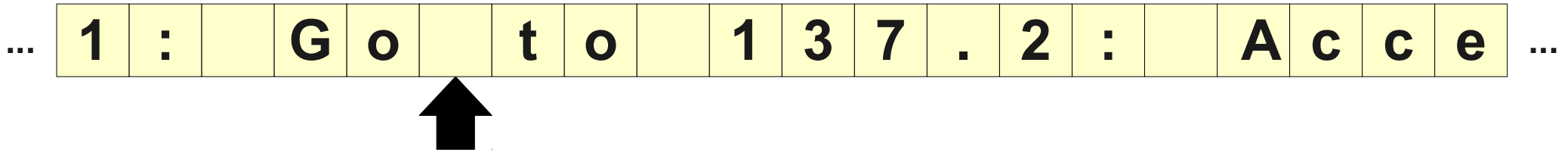
Instr **Letter**

Sketch of the Universal **WB** Program

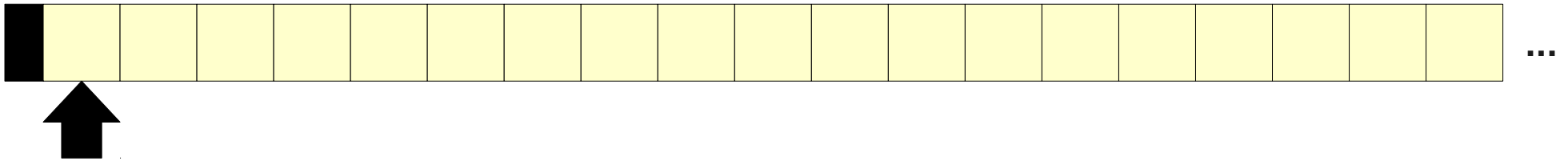
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

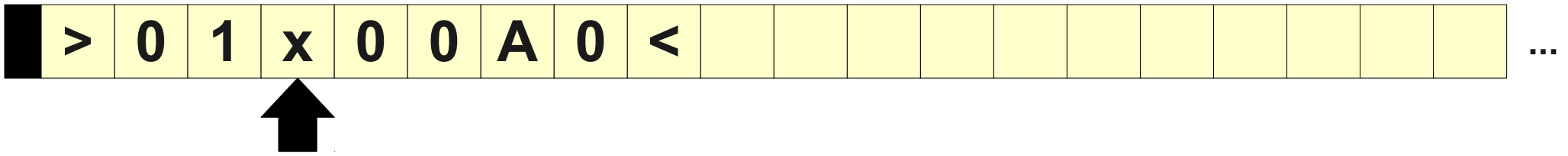


Variables for intermediate storage.

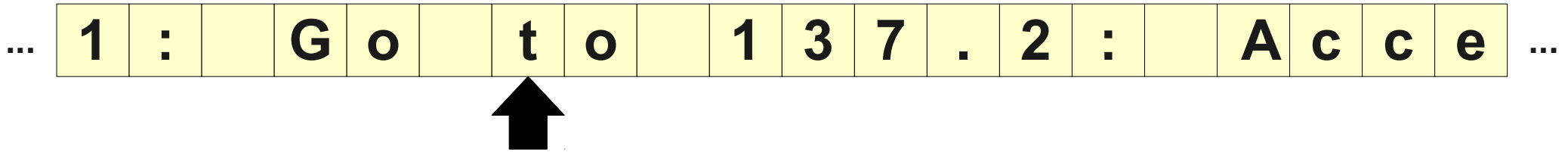
Instr **Letter**

Sketch of the Universal **WB** Program

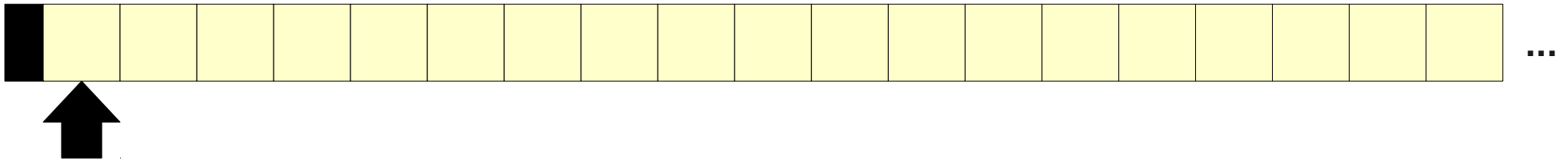
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

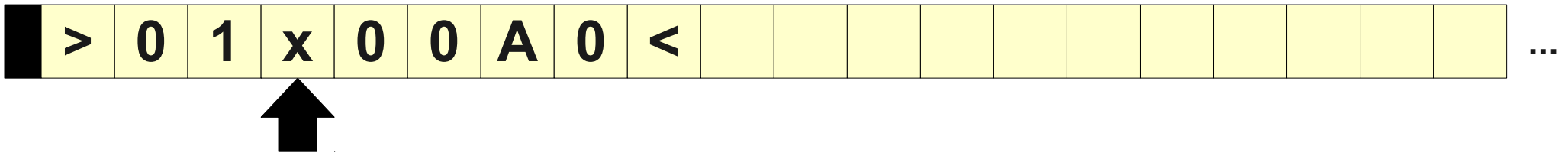


Variables for intermediate storage.

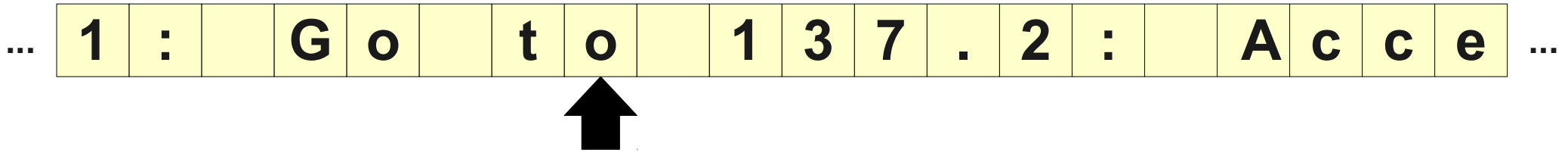
Instr **Letter**

Sketch of the Universal **WB** Program

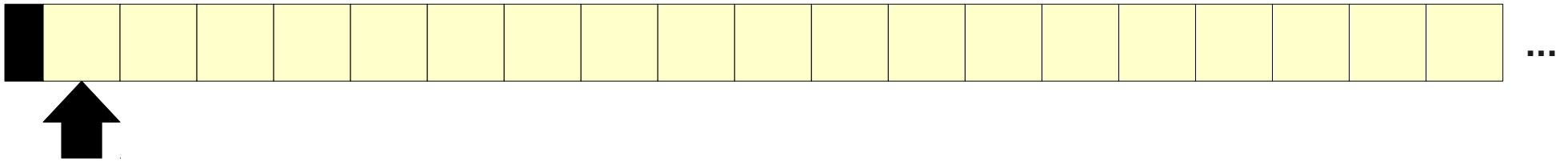
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

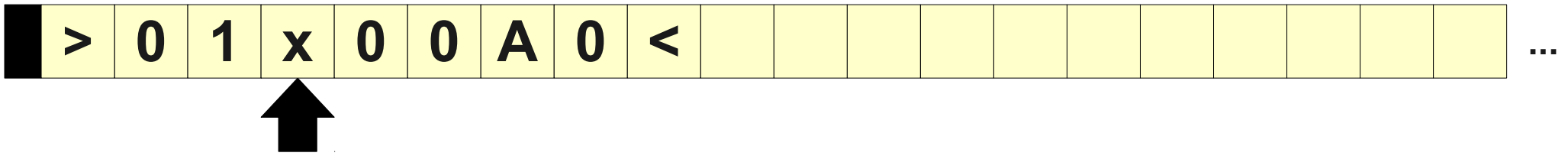


Variables for intermediate storage.

Instr **Letter**

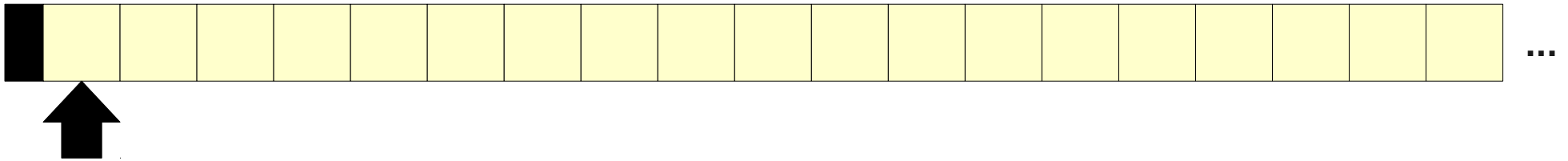
Sketch of the Universal **WB** Program

Simulated tape of the program being executed.



Program tape holding the program being executed.

Scratch tape for intermediate computation.

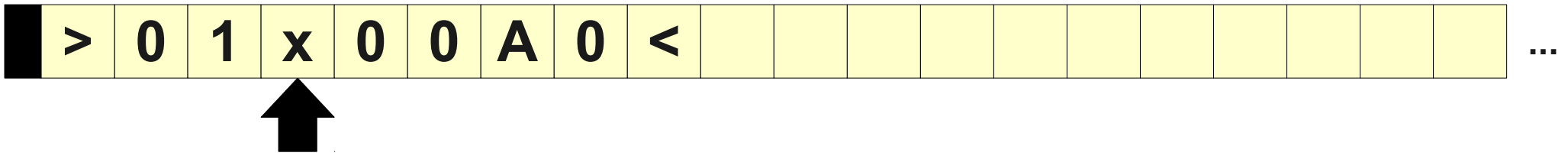


Variables for intermediate storage.

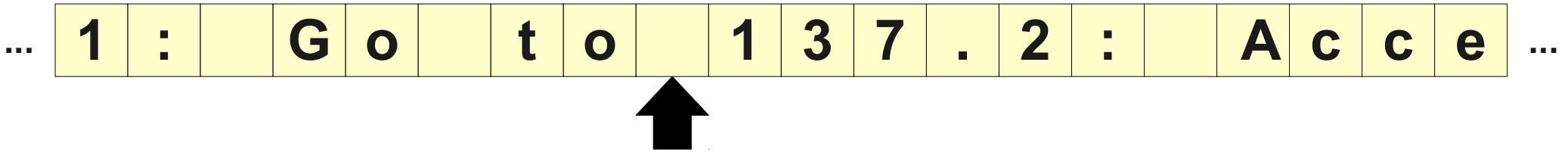
Instr **Letter**

Sketch of the Universal **WB** Program

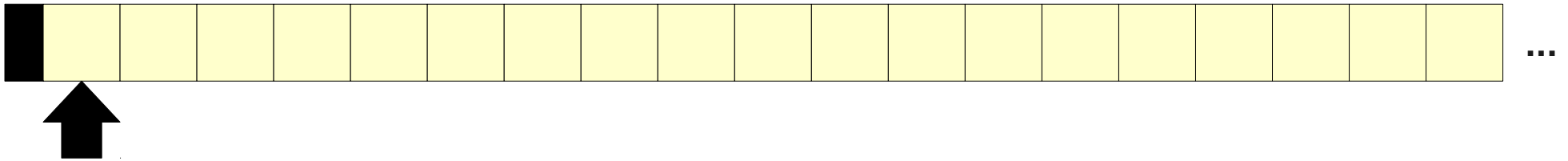
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

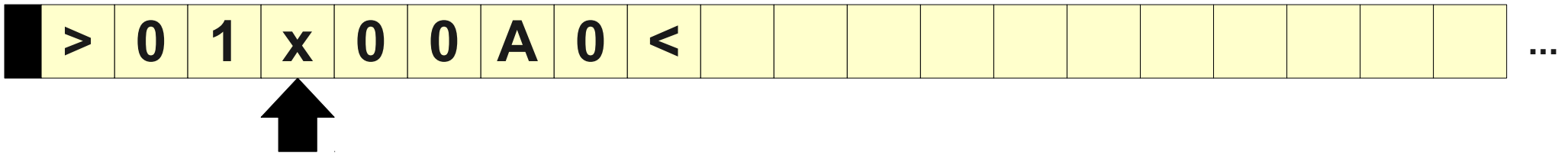


Variables for intermediate storage.

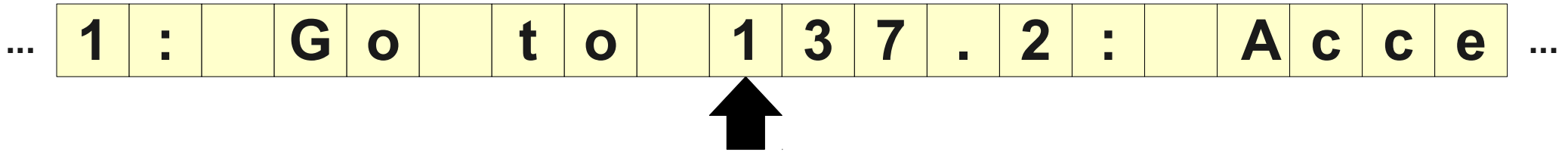
Instr **GoTo** **Letter**

Sketch of the Universal **WB** Program

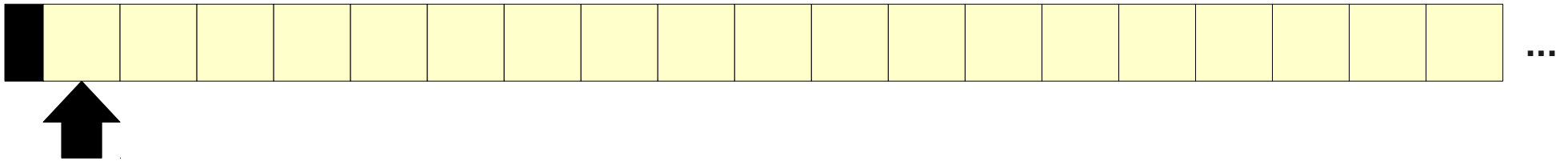
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

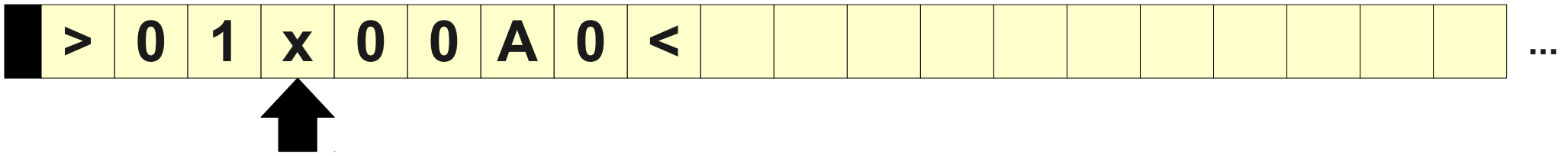


Variables for intermediate storage.

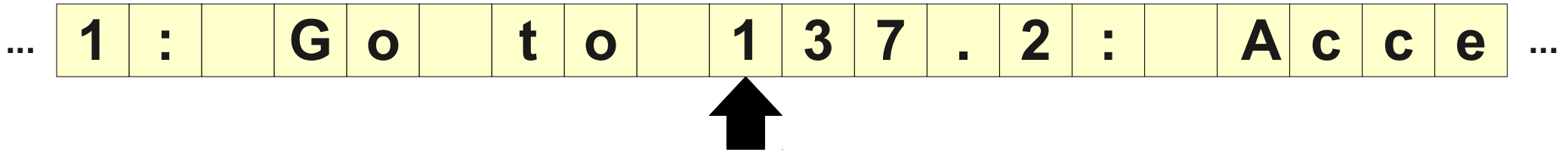
Instr **GoTo** **Letter**

Sketch of the Universal **WB** Program

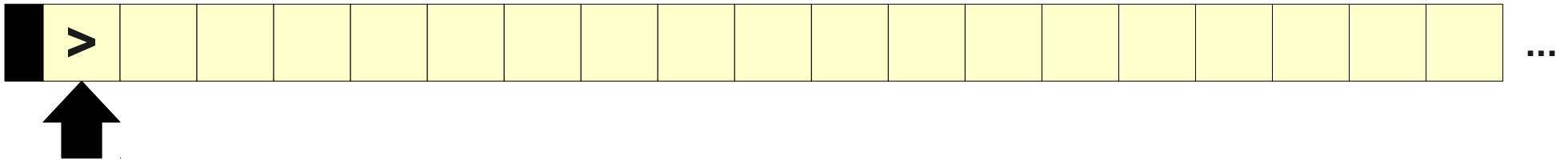
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

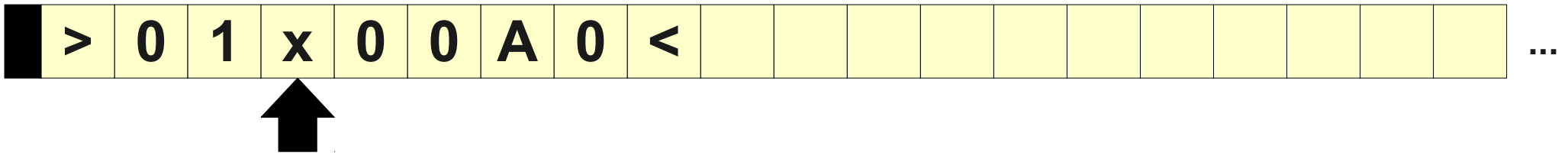


Variables for intermediate storage.

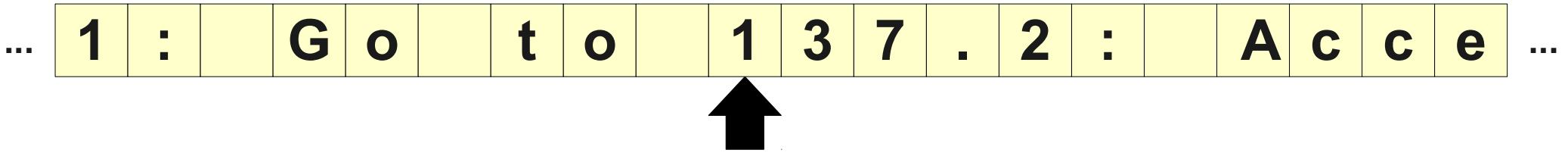
Instr **GoTo** **Letter**

Sketch of the Universal **WB** Program

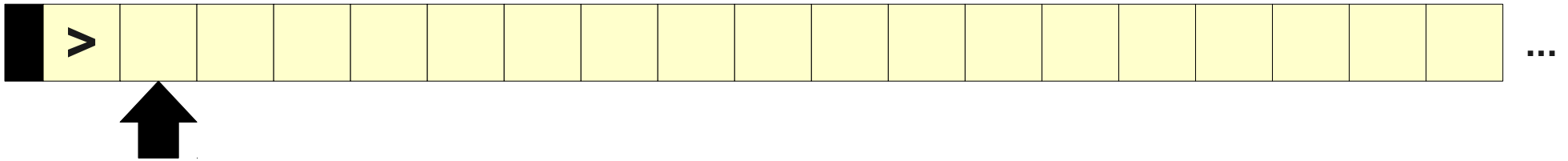
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

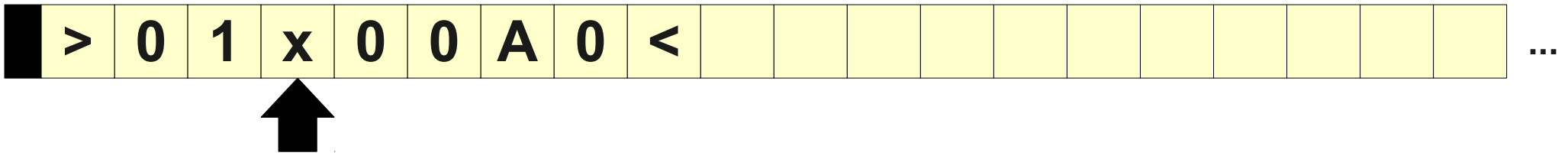


Variables for intermediate storage.

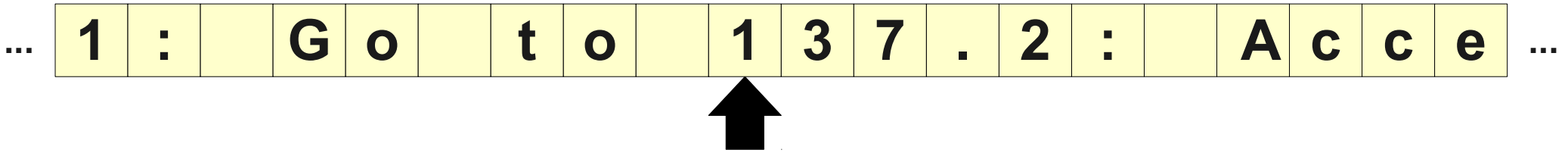
Instr **GoTo** **Letter**

Sketch of the Universal **WB** Program

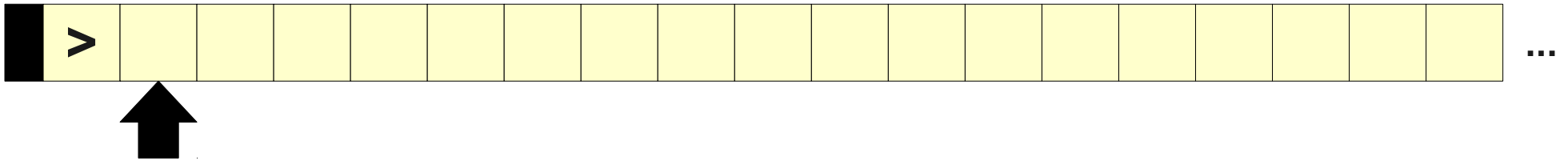
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

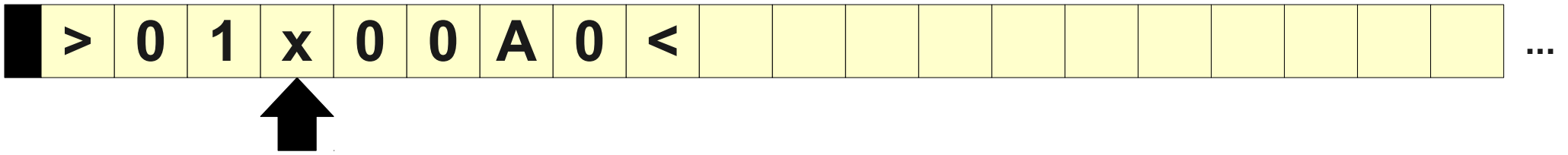


Variables for intermediate storage.

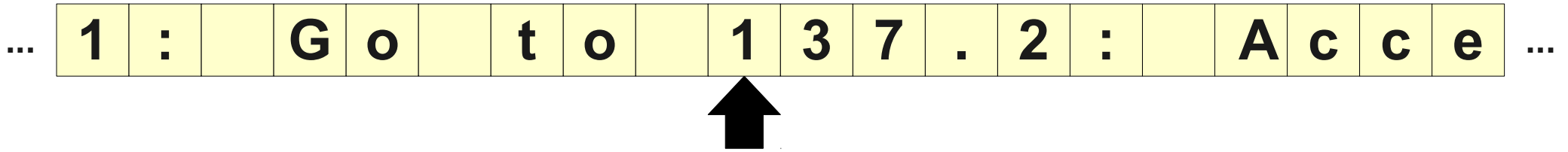
Instr GoTo **Letter** 1

Sketch of the Universal **WB** Program

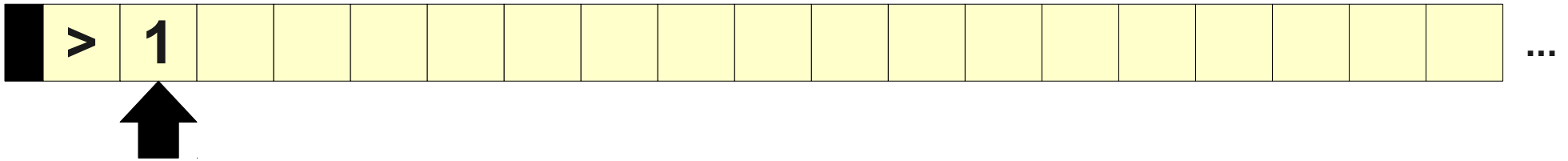
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

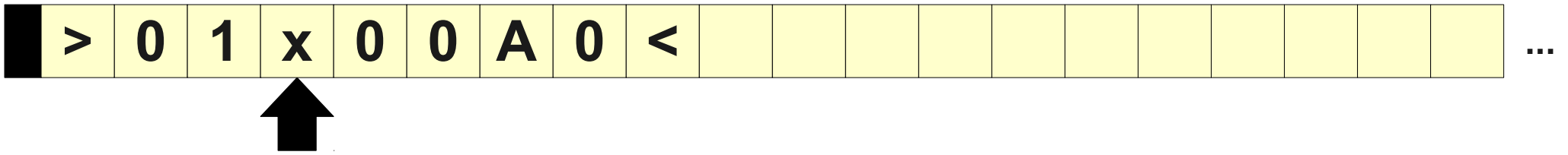


Variables for intermediate storage.

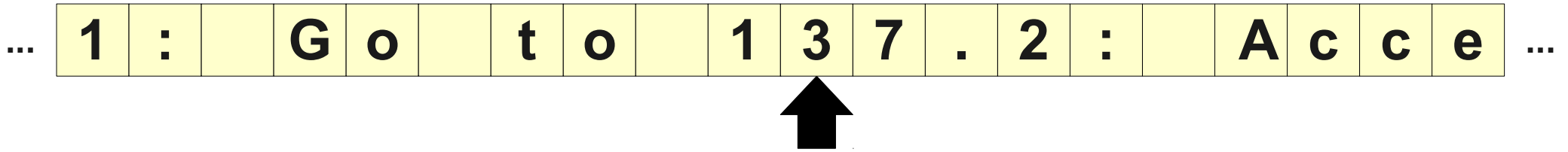
Instr GoTo **Letter** 1

Sketch of the Universal **WB** Program

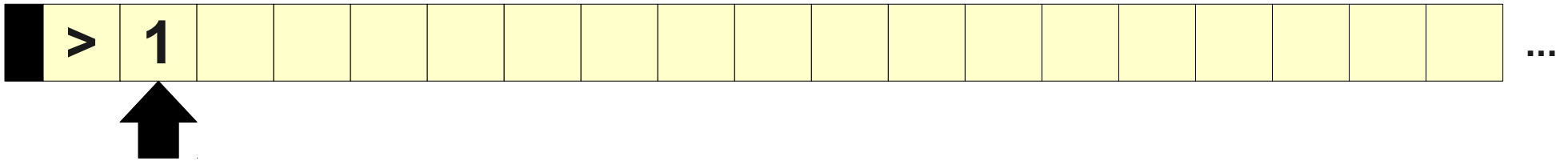
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

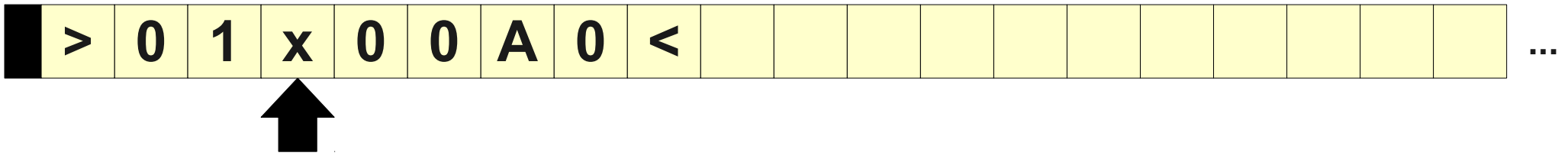


Variables for intermediate storage.

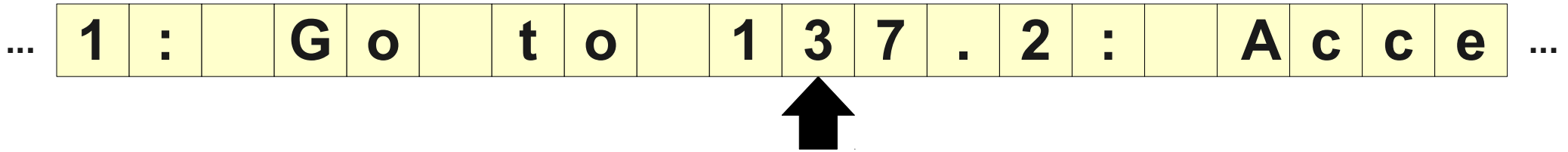
Instr GoTo **Letter** 1

Sketch of the Universal **WB** Program

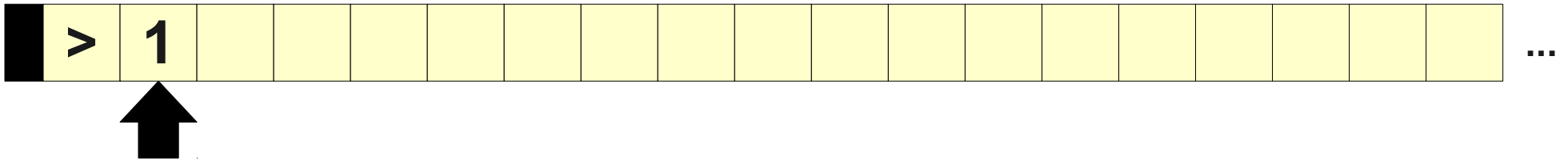
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

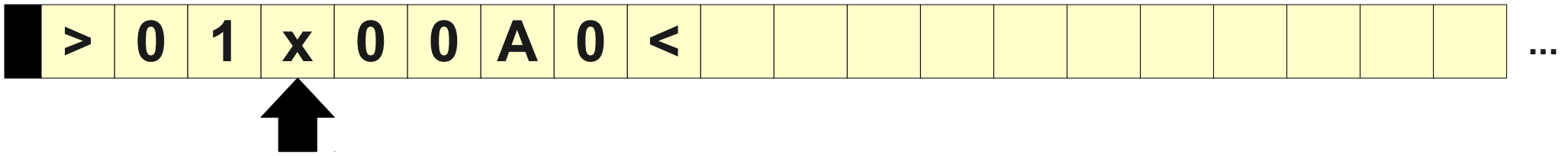


Variables for intermediate storage.

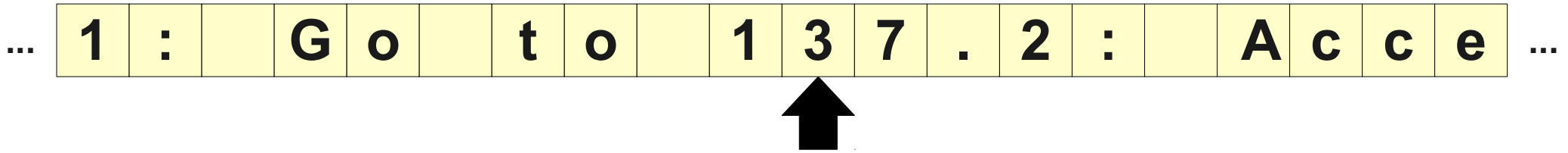
Instr GoTo **Letter** 3

Sketch of the Universal **WB** Program

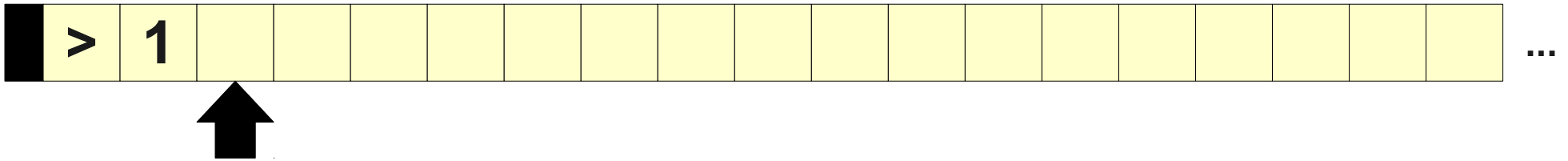
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

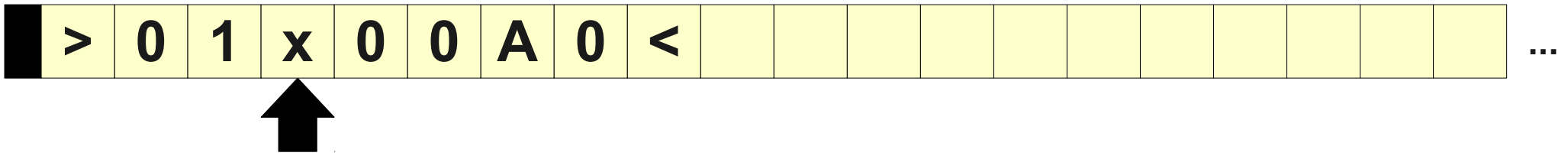


Variables for intermediate storage.

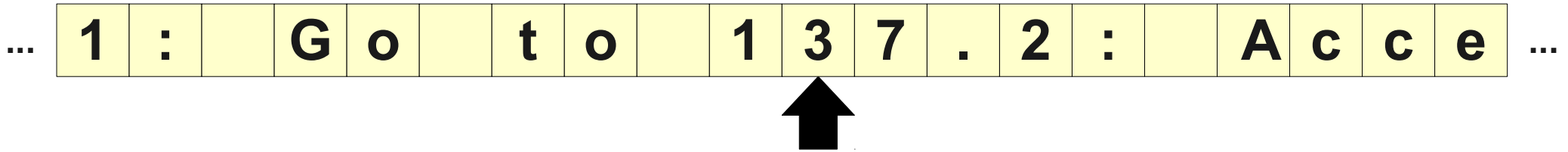
Instr GoTo **Letter** 3

Sketch of the Universal **WB** Program

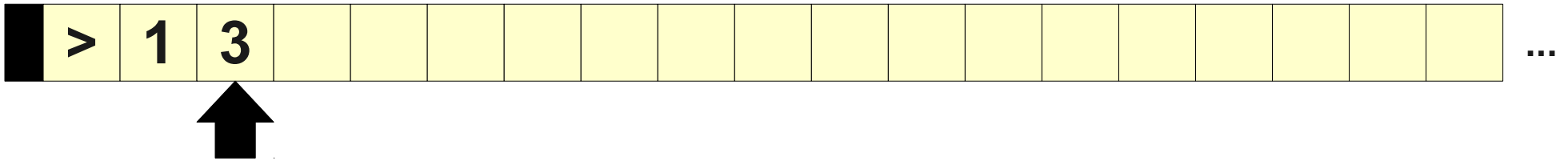
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

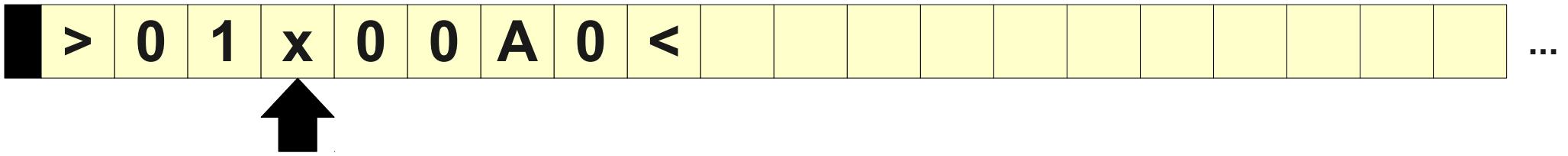


Variables for intermediate storage.

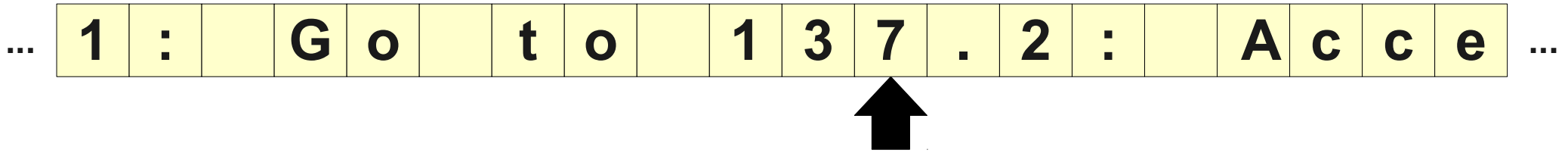
Instr **GoTo** **Letter** **3**

Sketch of the Universal **WB** Program

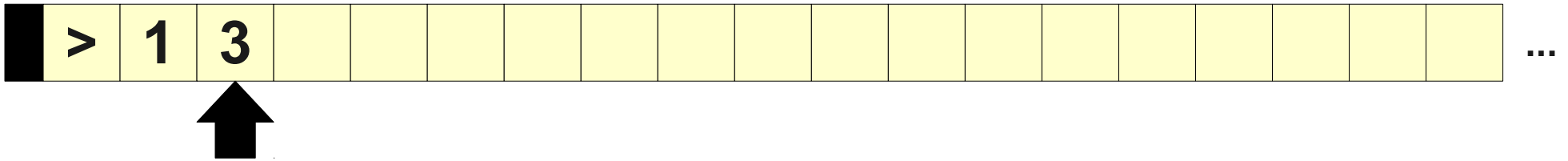
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

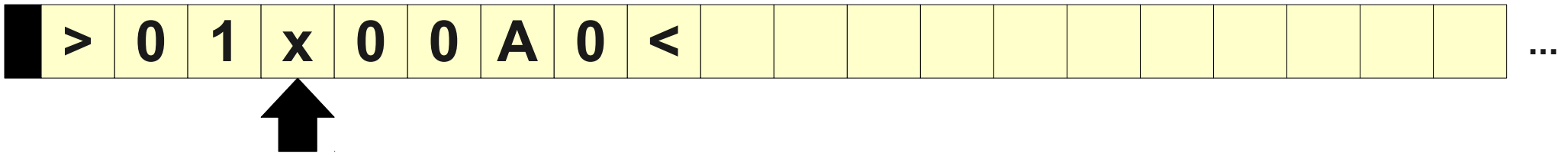


Variables for intermediate storage.

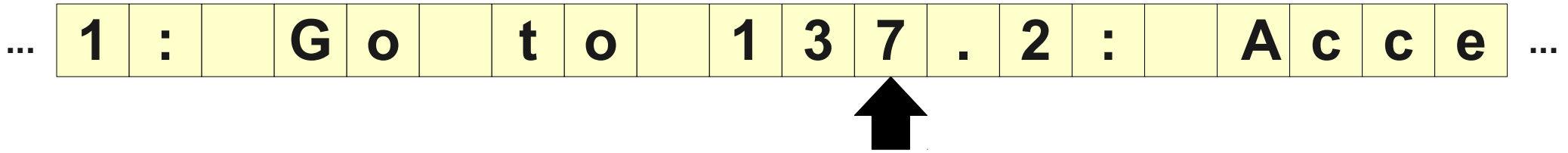
Instr GoTo **Letter** 3

Sketch of the Universal **WB** Program

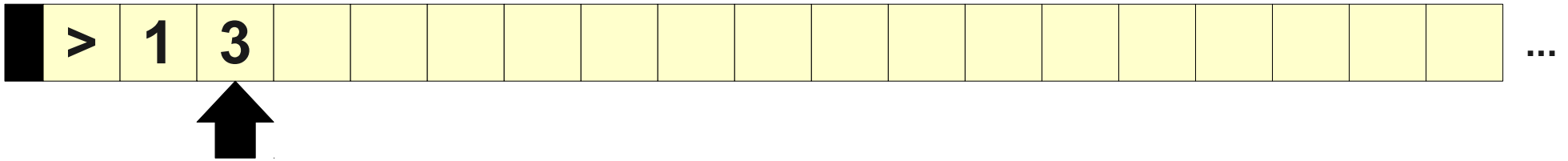
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

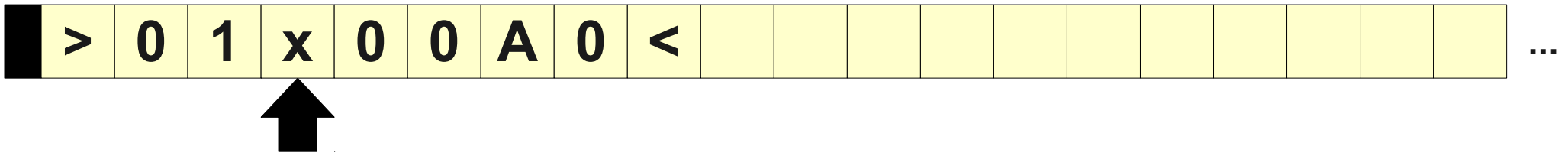


Variables for intermediate storage.

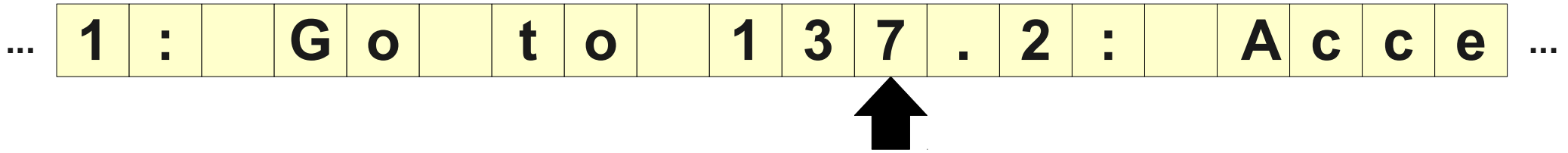
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

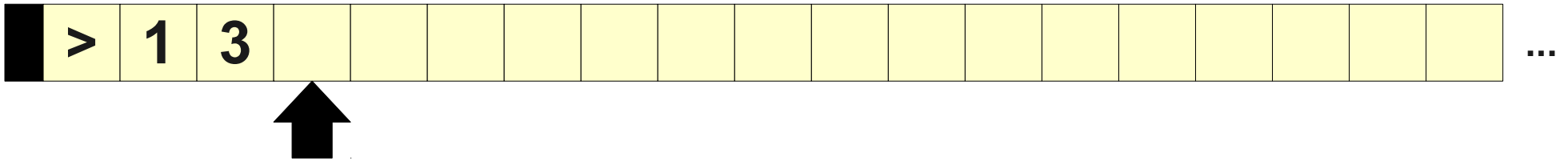
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

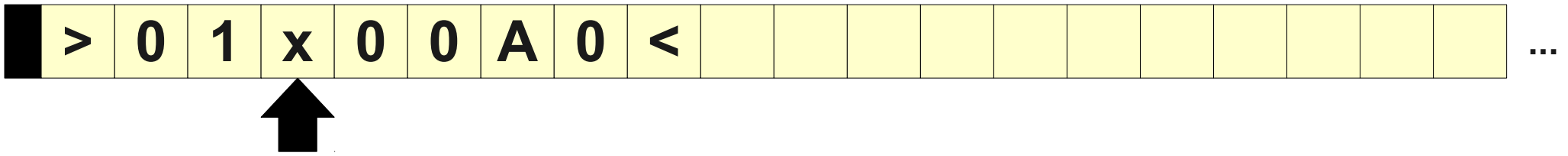


Variables for intermediate storage.

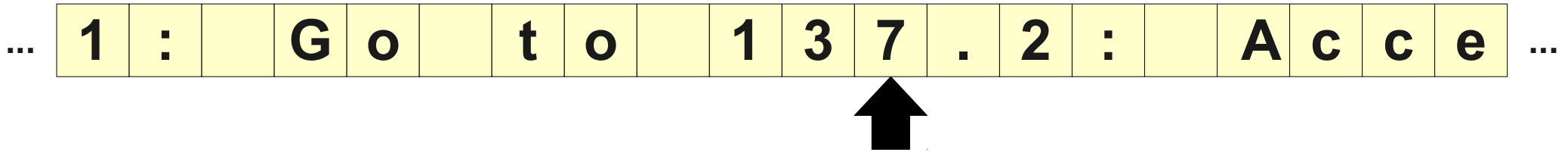
Instr GoTo **Letter** 7

Sketch of the Universal **WB** Program

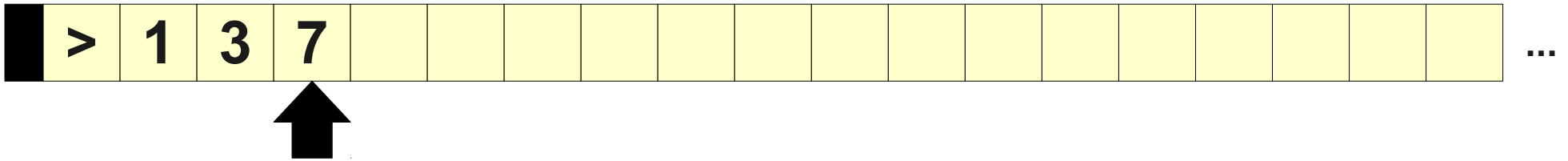
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

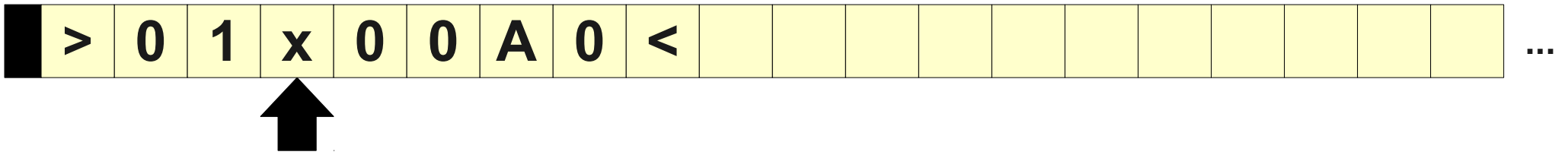


Variables for intermediate storage.

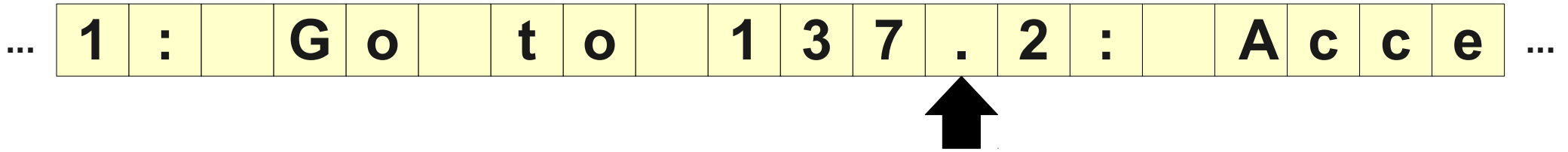
Instr GoTo **Letter** 7

Sketch of the Universal **WB** Program

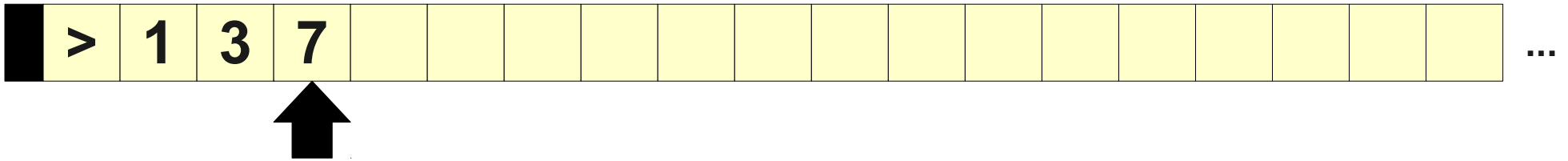
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

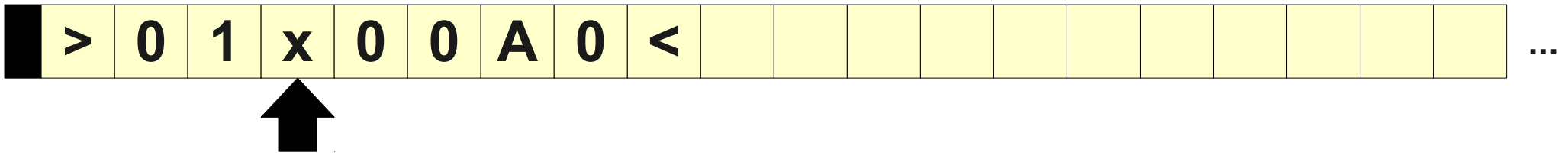


Variables for intermediate storage.

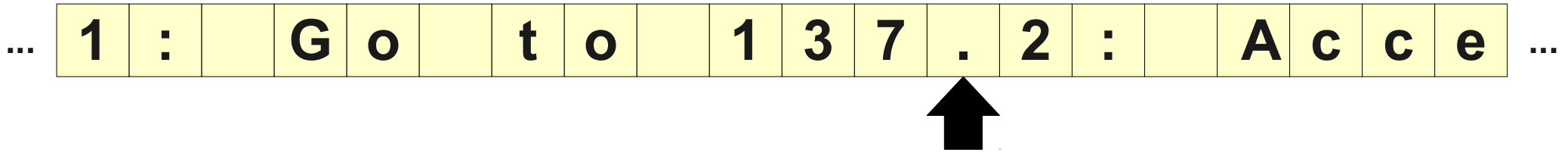
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

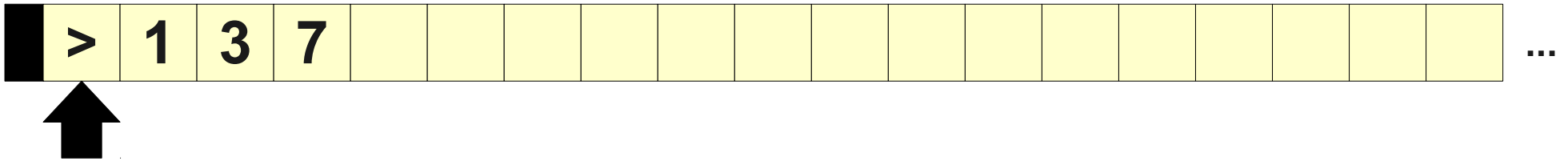
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

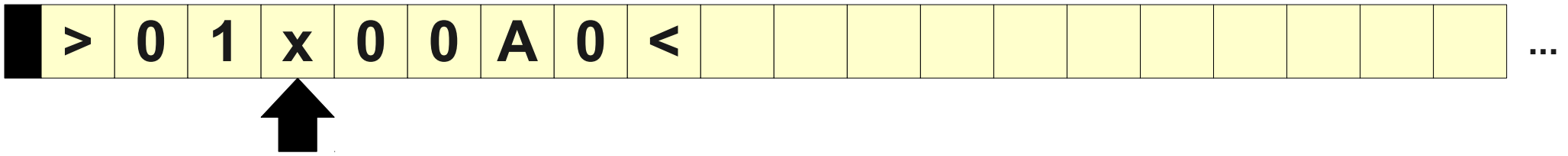


Variables for intermediate storage.

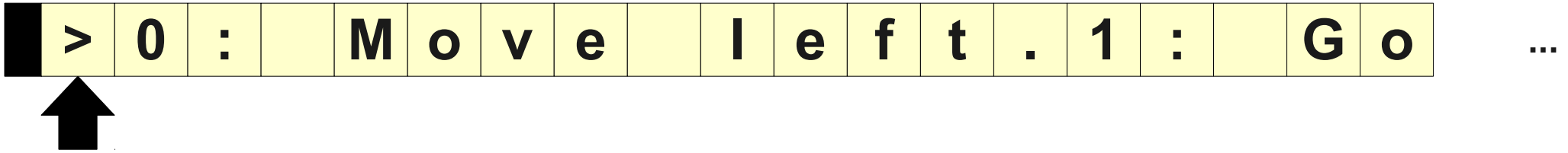
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

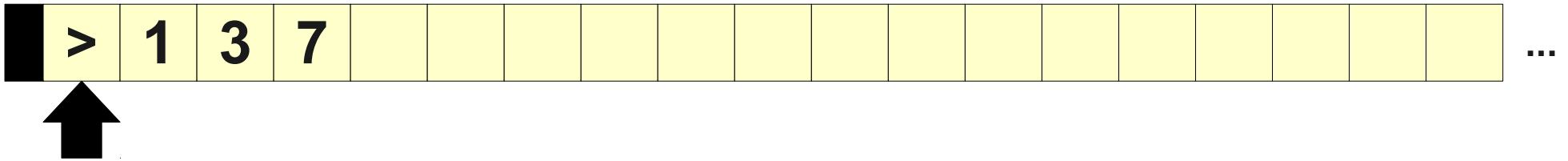
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

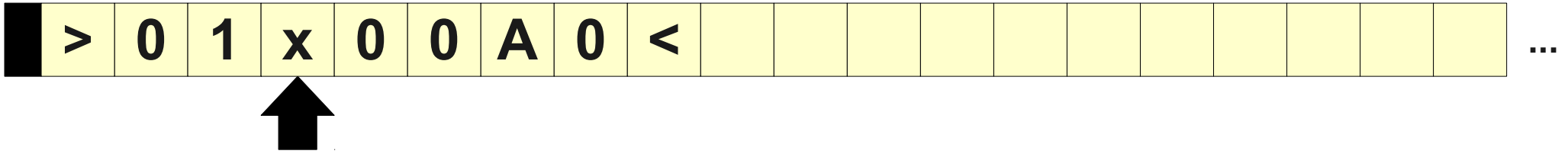


Variables for intermediate storage.

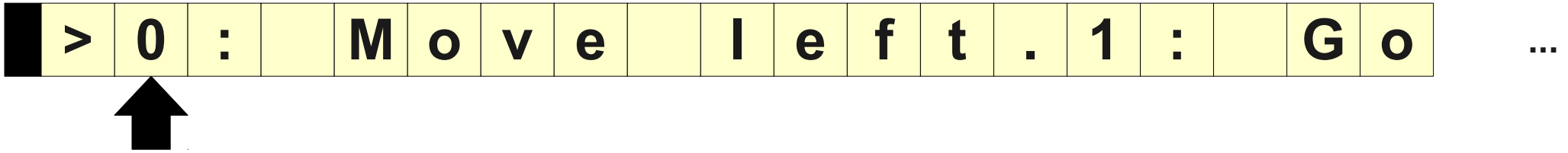
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

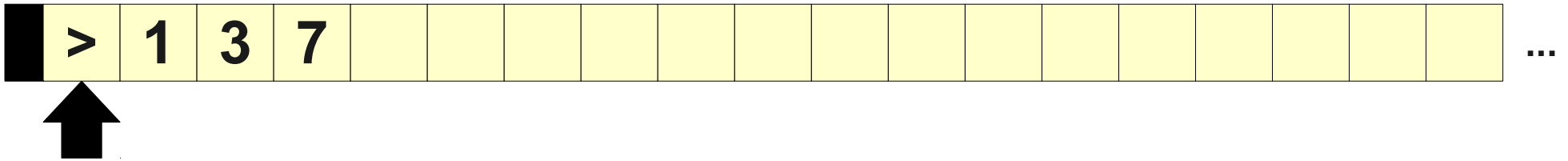
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

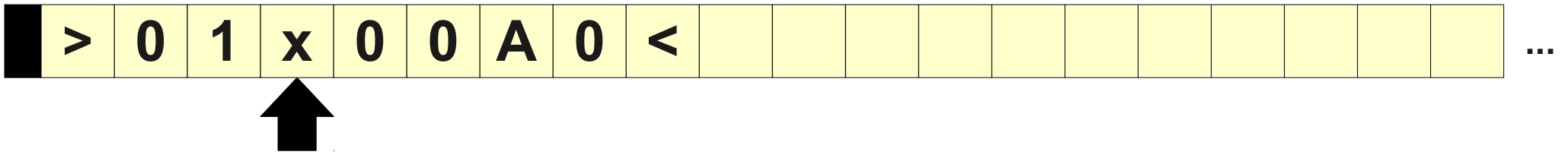


Variables for intermediate storage.

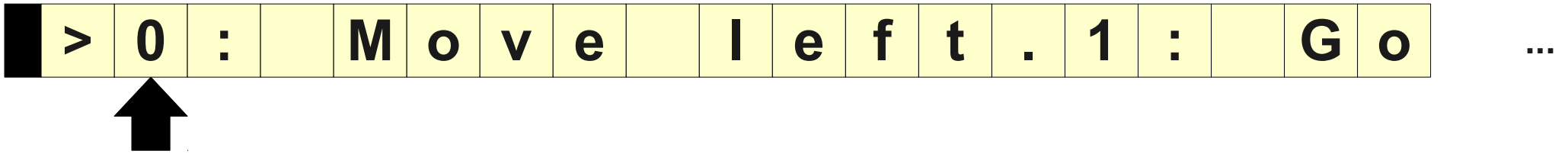
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

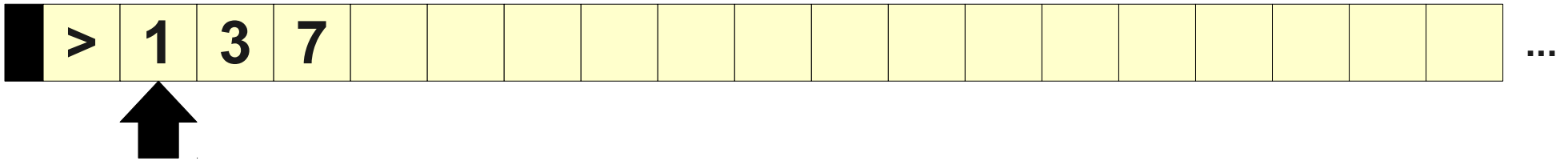
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

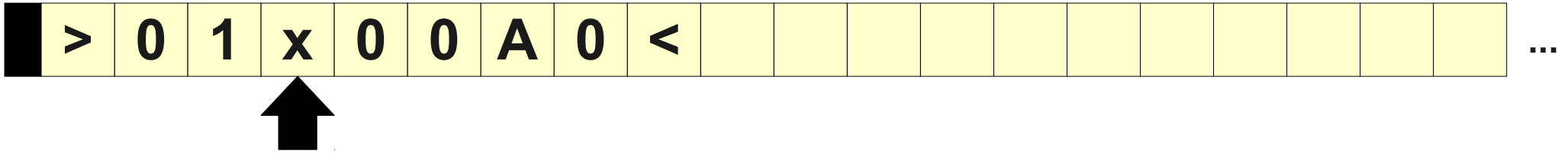


Variables for intermediate storage.

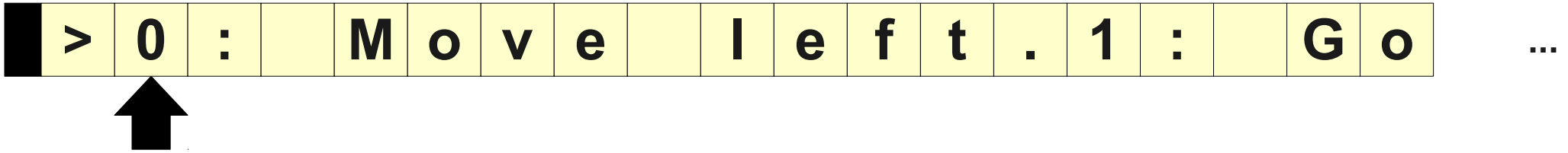
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

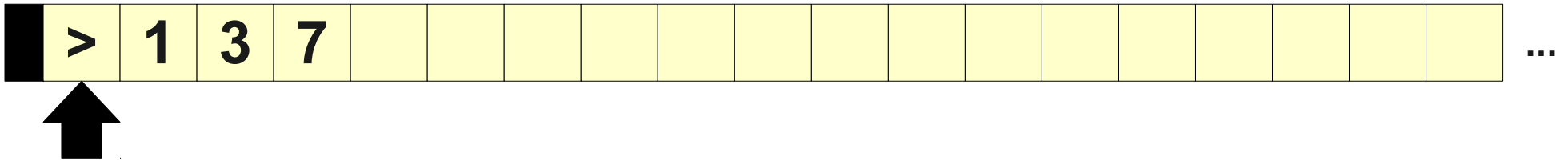
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

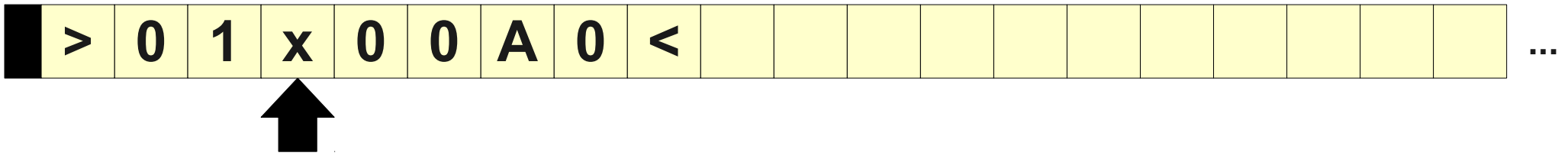


Variables for intermediate storage.

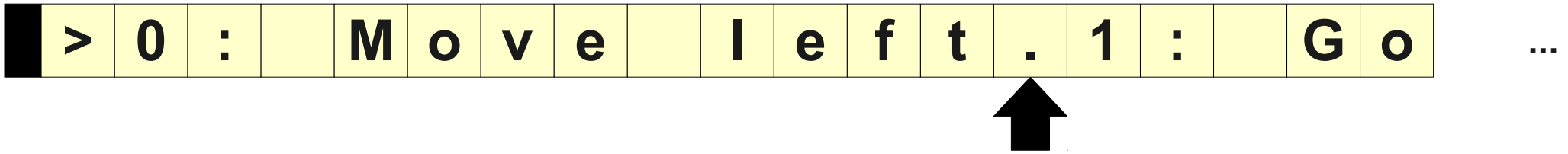
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

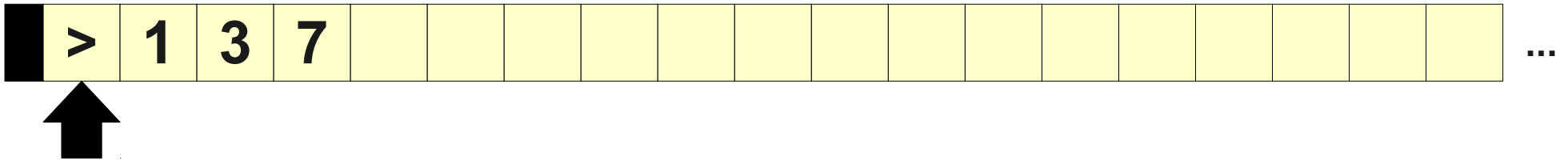
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

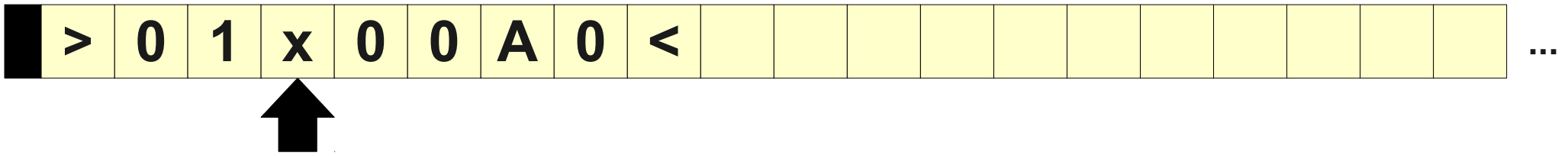


Variables for intermediate storage.

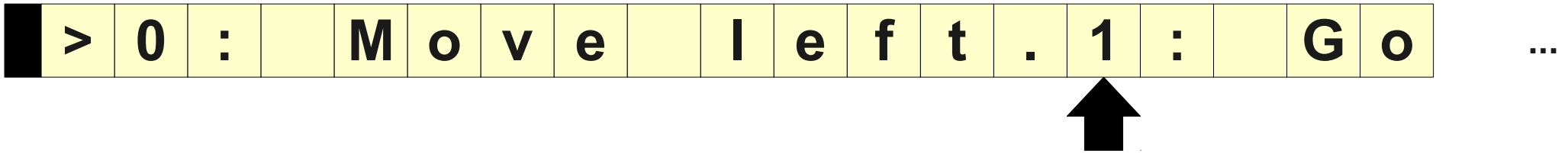
Instr GoTo **Letter** 7

Sketch of the Universal **WB** Program

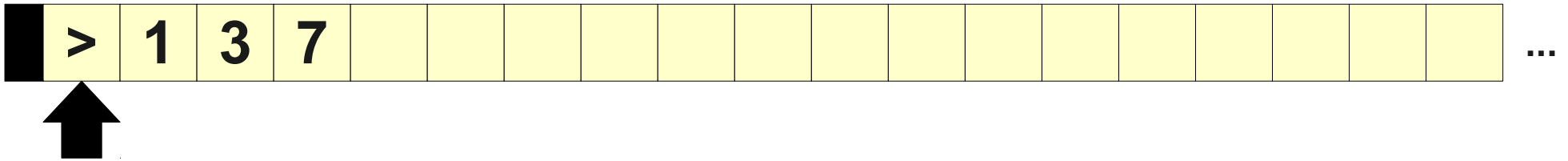
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

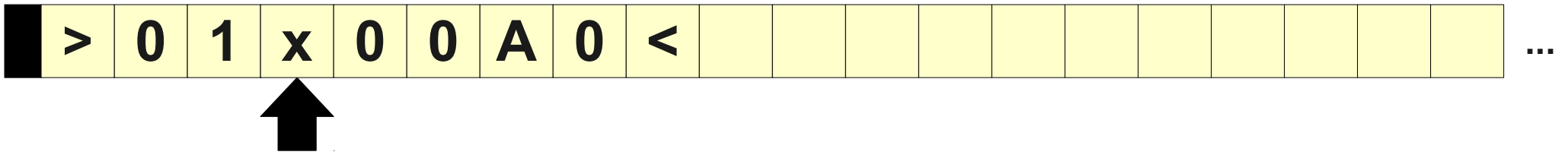


Variables for intermediate storage.

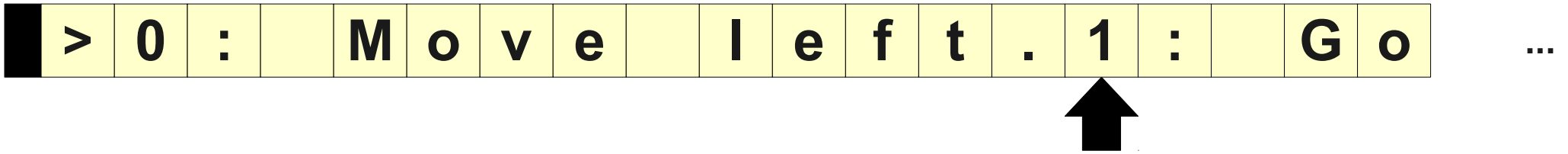
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

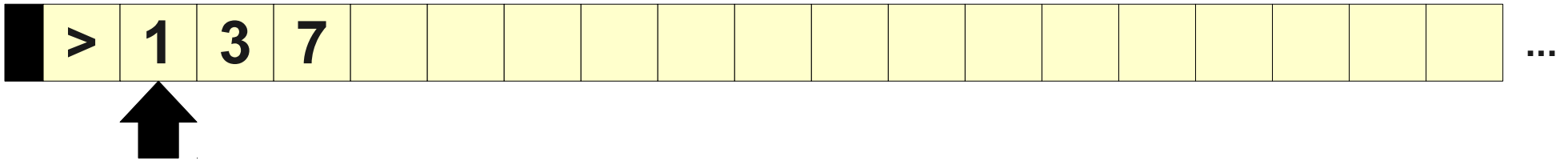
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

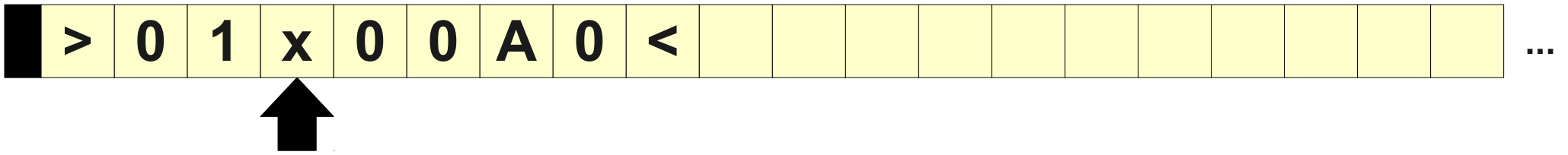


Variables for intermediate storage.

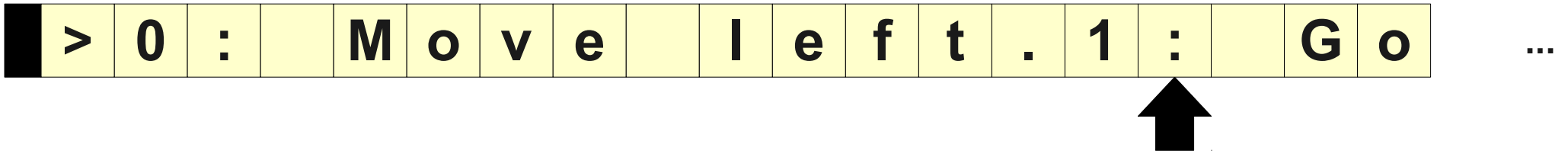
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

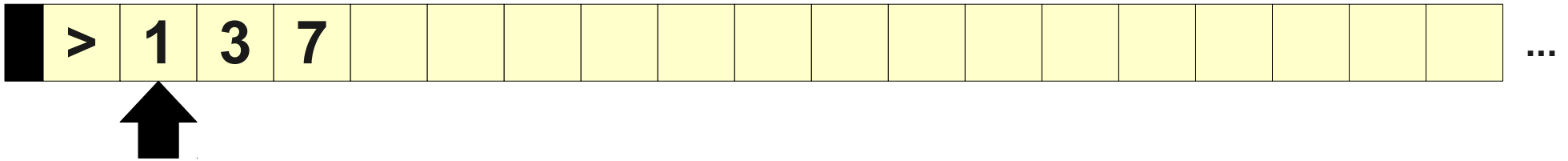
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

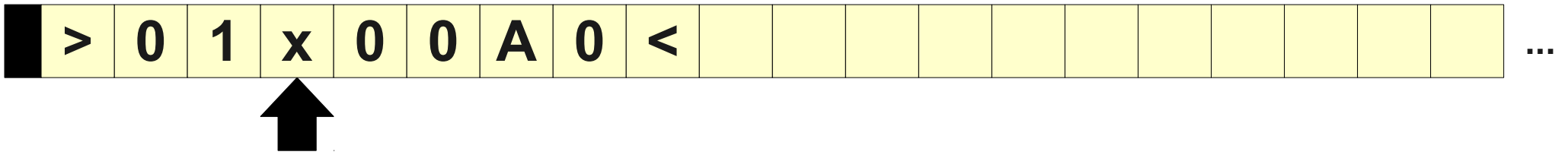


Variables for intermediate storage.

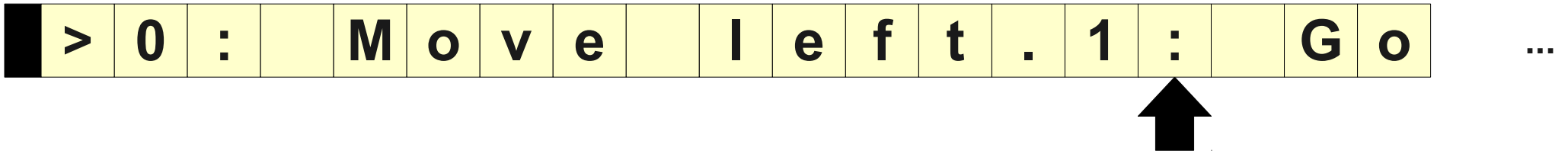
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

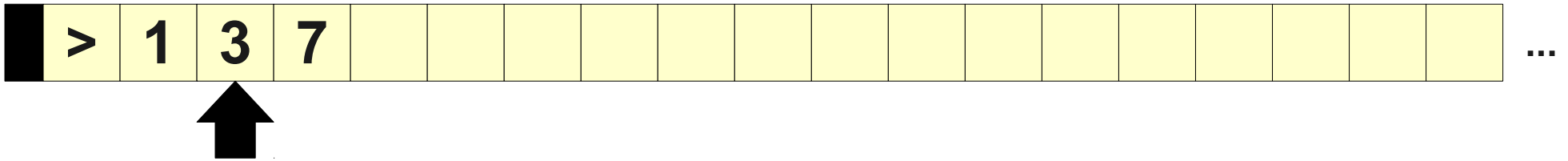
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

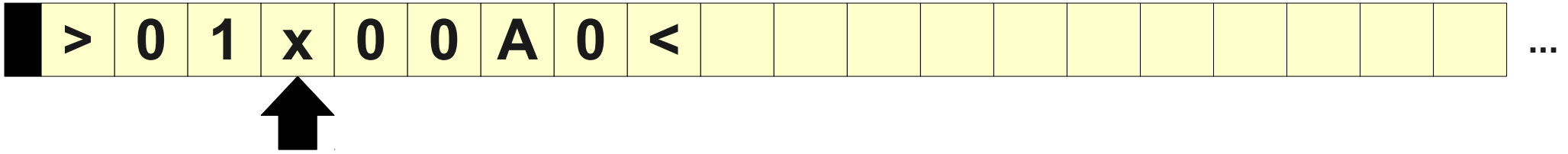


Variables for intermediate storage.

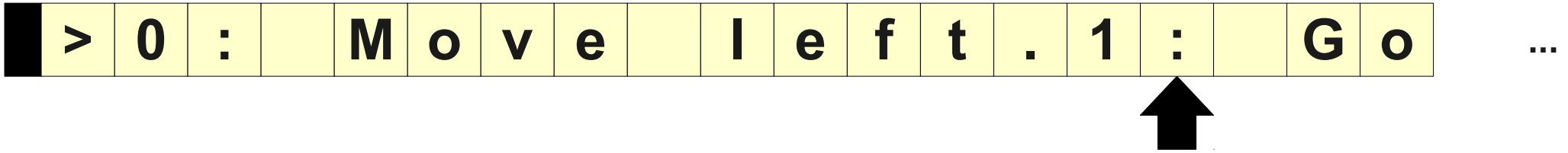
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

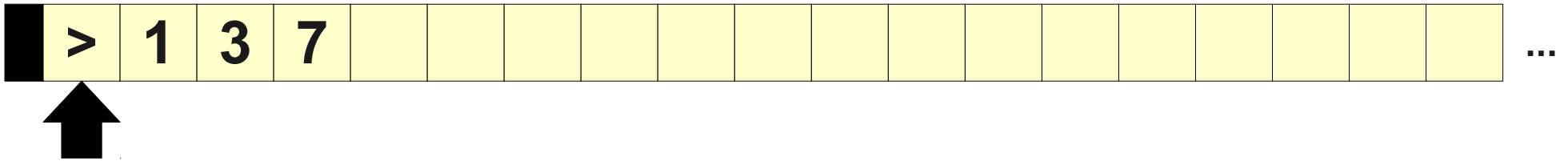
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.



Variables for intermediate storage.

Instr GoTo **Letter** 7

The Language of U_{TM}

- From a formal language perspective, what is the language of a universal machine?
- The universal Turing machine accepts all strings of the form $\langle M, w \rangle$, where M is a Turing machine that accepts string w.
- This language is called $\mathbf{A_{TM}}$:

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM that accepts } w. \}$$

- The universal **WB** program has language $\mathbf{A_{WB}}$:

$$A_{WB} = \{ \langle P, w \rangle \mid P \text{ is a } \mathbf{WB} \text{ program that accepts } w. \}$$

- Both A_{TM} and A_{WB} are recursively enumerable.

The Language of U_{TM}

- From a formal language perspective, what is the language of a universal machine?
- The universal Turing machine accepts all strings of the form $\langle M, w \rangle$, where M is a Turing machine that accepts string w.
- This language is called $\mathbf{A_{TM}}$:

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in \mathcal{L}(M) \}$$

- The universal **WB** program has language $\mathbf{A_{WB}}$:

$$A_{WB} = \{ \langle P, w \rangle \mid P \text{ is a } \mathbf{WB} \text{ program and } w \in \mathcal{L}(P) \}$$

- Both A_{TM} and A_{WB} are recursively enumerable.

Instrumenting U_{TM}

- Each universal machine has the following structure:
  - Find the next instruction to execute.
  - Determine how to execute that instruction.
  - Simulate the execution of that instruction.
  - Repeat.
- In-between each of those steps, we can add extra logic to modify the behavior of the simulated machine.

Tricks with Universal Machines

- Suppose that we have a TM (or **WB** program) and a string and want to see if that TM accepts the string within *n* steps.
- Can this be done with a Turing machine or **WB** program?
- As a language problem:

$$A_{FAST} = \{ \langle M, w, n \rangle \mid M \text{ is a TM that accepts } w \text{ in at most } n \text{ steps.} \}$$

- Is A_{FAST} recursively enumerable?

Sketch of a Program for A_{FAST}

- We could write a program for A_{FAST} using the following logic:
- "On input *x*:
  - Check that *x* has the form ⟨*M*, *w*, *n*⟩ for a TM *M*, string *w*, and natural number *n*. If not, reject the input string.
  - Copy *n* to a separate tape.
  - Using the universal Turing machine, run *M* on *w*. Before executing any step of *M*, decrement *n*.
  - If *n* reaches 0, reject.
  - Otherwise, if *M* accepts *w*, accept the input string *x*.
  - Otherwise, if *M* rejects *w*, reject the input string *x*."

Tricks with Universal Machines

- Suppose we have a **WB** program *P* and want to see if it accepts string *w* without ever executing the same instruction twice.
- Can we design a TM or **WB** program to determine this?
- As a language problem:

$$A_{ONCE} = \{ \langle P, w \rangle \mid P \text{ is a } \textbf{WB} \text{ program that accepts } w \text{ without executing the same instruction twice. } \}$$

- Is A_{ONCE} recursively enumerable?

Sketch of a Program for A_{ONCE}

- We could write a program for A_{ONCE} using the following logic:
- "On input *x*:
  - Check that *x* has the form ⟨*P*, *w*⟩ for a **WB** program *P* and string *w*. If not, reject the input string.
  - Use the universal **WB** program to run *P* on *w*.
  - After executing each instruction, replace that instruction with **`Reject`**.
  - If *P* accepts *w*, accept the input string *x*.
  - If *P* rejects *w*, reject the input string *x*."

High-Level Machine Specifications

- The machines we have described in the past few examples all the form
  - "On input *x*:
    - Description of what to do on *x*."
- These descriptions are called **high-level descriptions**.
- Because of the Church-Turing thesis, if we can know that every step could be done by a computer, then we know a TM/**WB** program exists for the description.
- Unless explicitly asked to write a **WB** program or Turing machine, feel free to use high-level descriptions in your proofs.

Sketch of a Program for A_{FAST}

- "On input *x*:
  - Check that *x* has the form ⟨*M*, *w*, *n*⟩ for a TM *M*, string *w*, and natural number *n*. If not, reject the input string.
  - Copy *n* to a separate tape.
  - Using the universal Turing machine, run *M* on *w*. Before executing any step of *M*, decrement *n*.
  - If *n* reaches 0, reject.
  - Otherwise, if *M* accepts *w*, accept the input string *x*.
  - Otherwise, if *M* rejects *w*, reject the input string *x*."

Sketch of a Program for A_{FAST}

- "On input *x*:
  - Check that *x* has the form ⟨*M*, *w*, *n*⟩ for a TM *M*, string *w*, and natural number *n*. If not, reject the input string.

    Copy *n* to a separate tape.

    Using the universal Turing machine, run *M* on *w*. Before executing any step of *M*, decrement *n*.

    If *n* reaches 0, reject.

    Otherwise, if *M* accepts *w*, accept the input string *x*.

    Otherwise, if *M* rejects *w*, reject the input string *x*."

Sketch of a Program for A_{ONCE}

- "On input *x*:
  - Check that *x* has the form ⟨*P*, *w*⟩ for a **WB** program *P* and string *w*. If not, reject the input string.
  - Use the universal **WB** program to run *P* on *w*.
  - After executing each instruction, replace that instruction with **`Reject`**.
  - If *P* accepts *w*, accept the input string *x*.
  - If *P* rejects *w*, reject the input string *x*."

Sketch of a Program for A_{ONCE}

- "On input *x*:
  - Check that *x* has the form ⟨*P*, *w*⟩ for a **WB** program *P* and string *w*. If not, reject the input string.

    Use the universal **WB** program to run *P* on *w*.

    After executing each instruction, replace that instruction with `Reject`.

    If *P* accepts *w*, accept the input string *x*.

    If *P* rejects *w*, reject the input string *x*."

A Nice Shorthand

- Most interesting programs require their input to be structured in some fashion.
- As a new notation, we will allow ourselves to write high-level descriptions like this:

  "On input $\langle O_1, O_2, \ldots, O_n \rangle$:

  Do something with $O_1, \ldots, O_n$."

  to mean

  "On input *x*:

  Check that *x* has the form $\langle O_1, O_2, \ldots, O_n \rangle$. If not, reject.

  Do something with $O_1, \ldots, O_n$."

Nondeterministic Turing Machines

Nondeterministic Turing Machines

- A **nondeterministic Turing machine** (abbreviated **NTM**) is a Turing machine in which there may be multiple transitions defined for a particular state/input combination.
- If **any** possible set of choices causes the machine to accept, it accepts.
- Otherwise, if there is at least one choice that loops forever, the NTM loops forever.
- Otherwise, all choices cause the machine to reject, and the NTM rejects.

Nondeterministic Algorithms

- A natural number greater than 0 is **composite** if it is not prime.
- Let Σ = { `1` } and consider the language

$$\textit{COMPOSITE} = \{ \texttt{1}^n \mid n \text{ is composite} \}$$

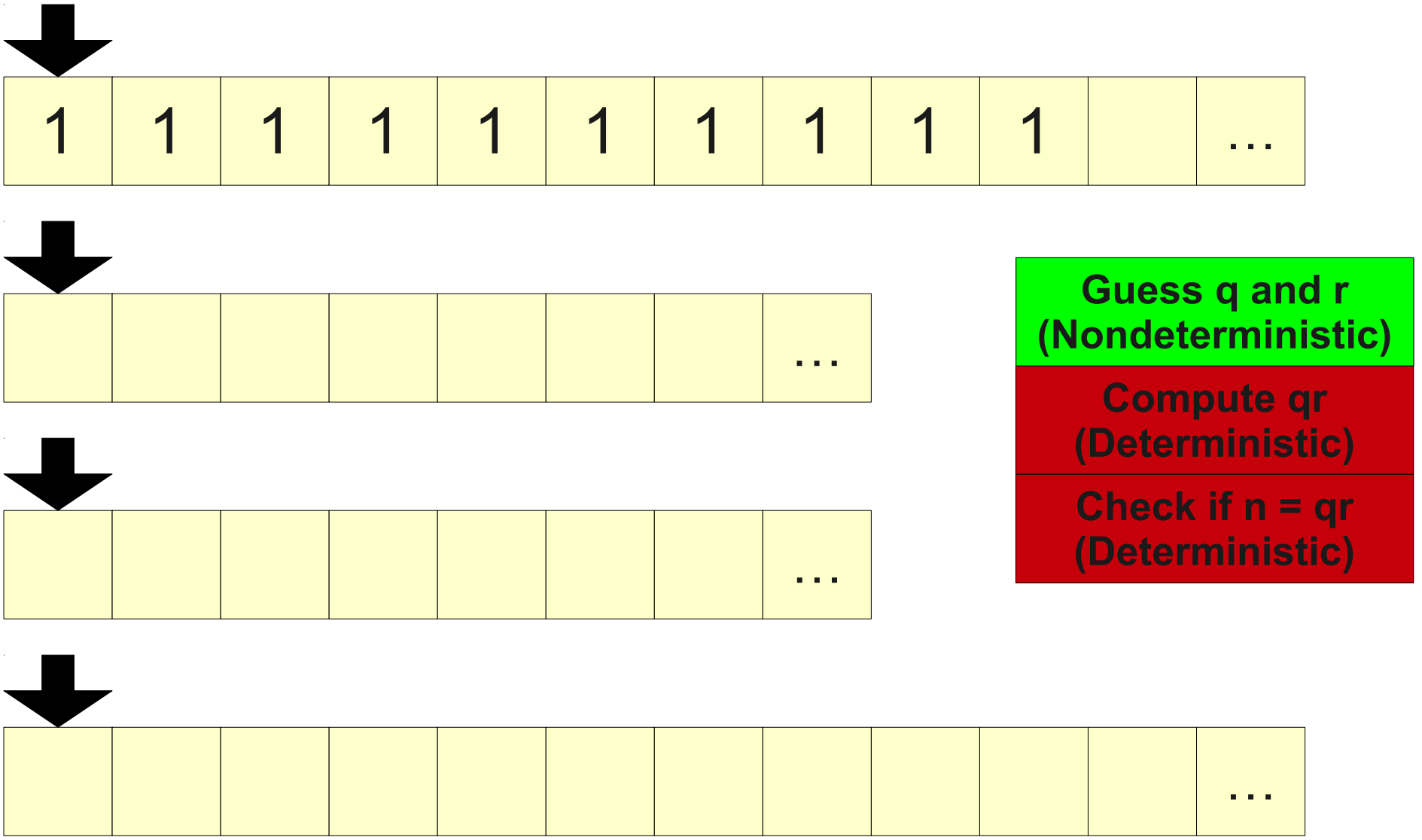
Nondeterministic Algorithms

- A natural number greater than 0 is **composite** if it is not prime.
- Let Σ = { 1 } and consider the language

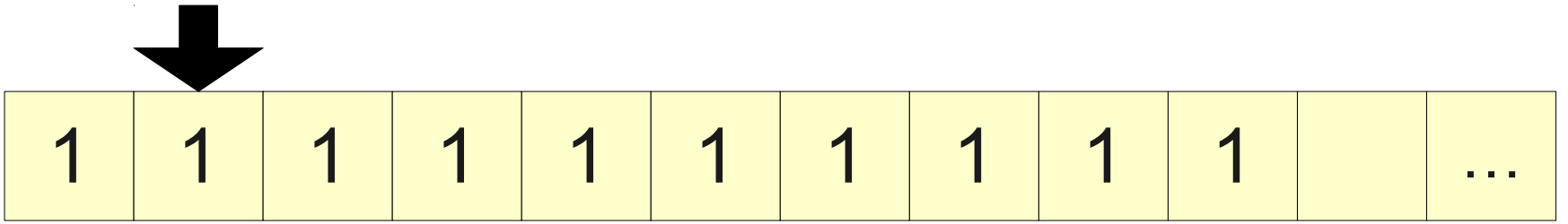
$$COMPOSITE = \{ 1^n \mid n \text{ is composite} \}$$

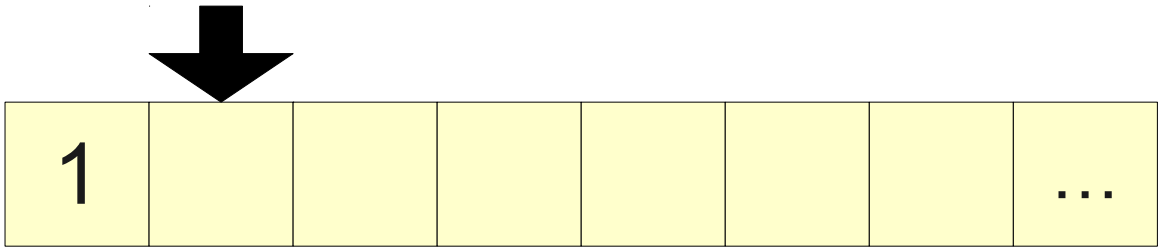
- We can build a **multitape, nondeterministic TM** for *COMPOSITE* as follows:
- M = “On input 1^n:
  - **Nondeterministically** write out q 1s on a second tape ($2 \leq q < n$)
  - **Nondeterministically** write out r 1s on a third tape ($2 \leq r < n$)
  - **Deterministically** check if qr = n.
  - If so, accept.
  - Otherwise, reject”

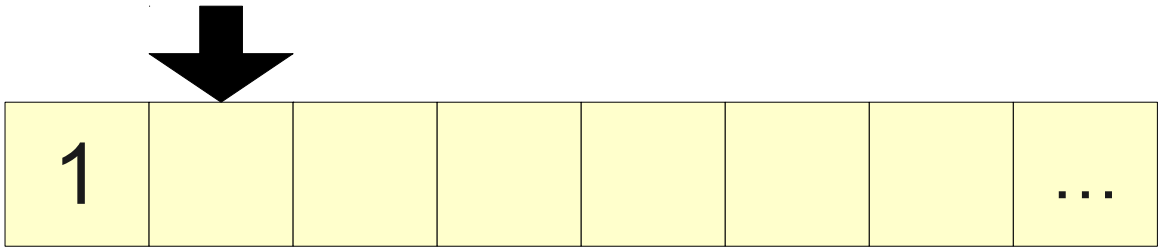
Nondeterministic Algorithms

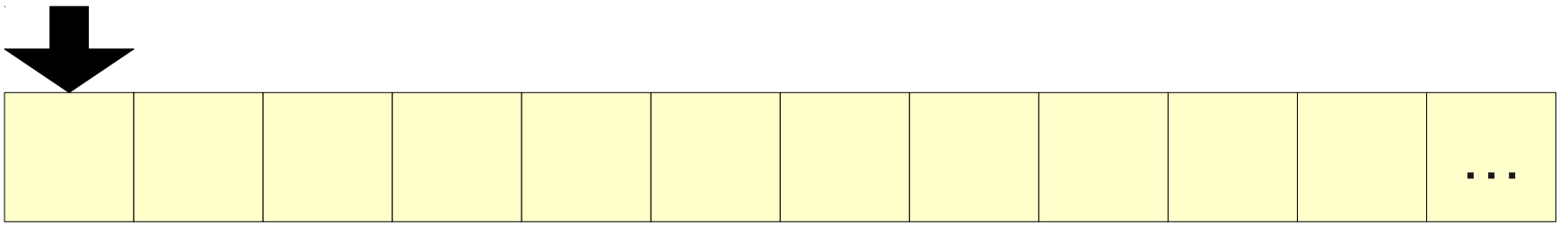


Nondeterministic Algorithms







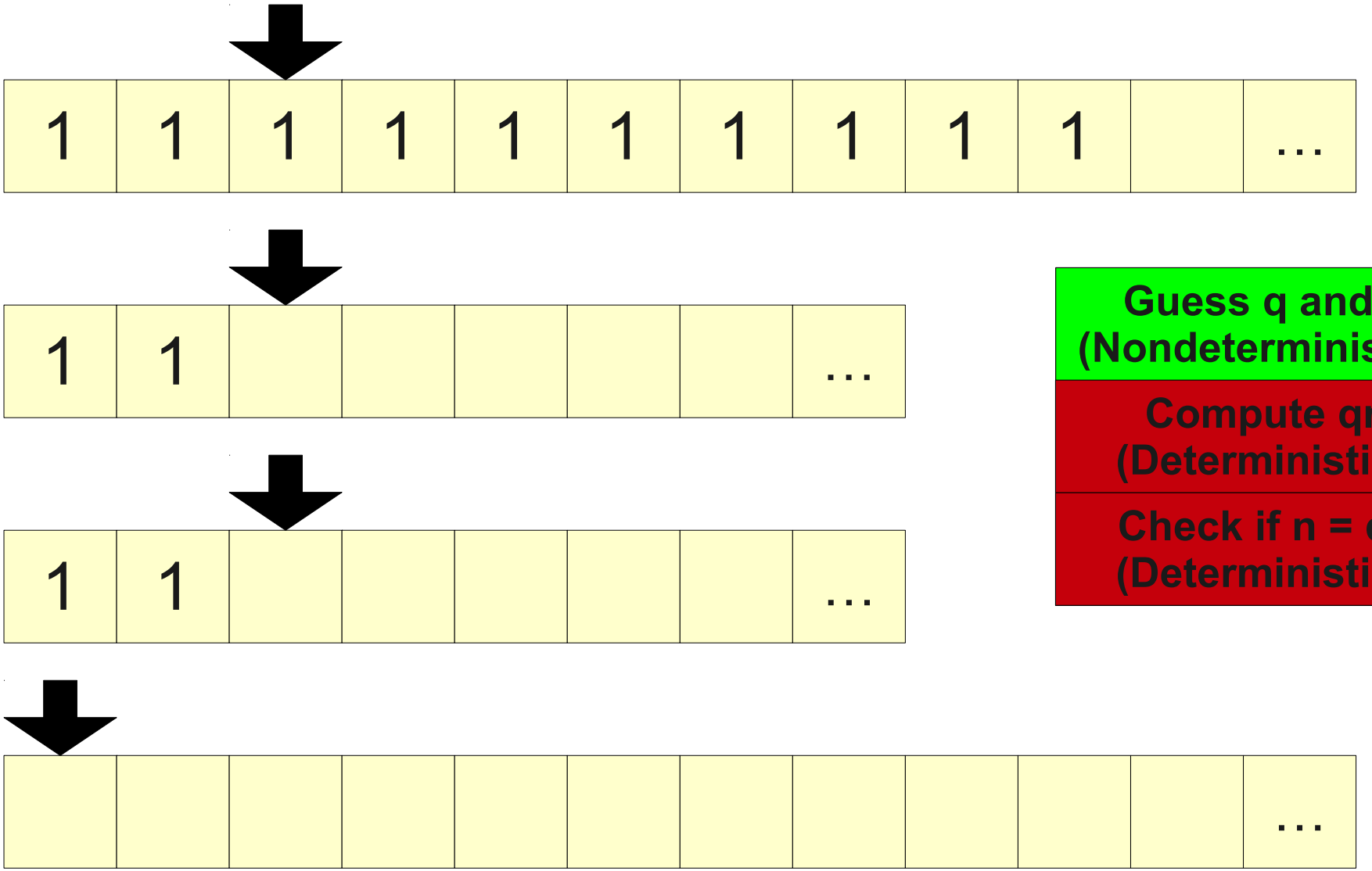


Guess q and r (Nondeterministic)

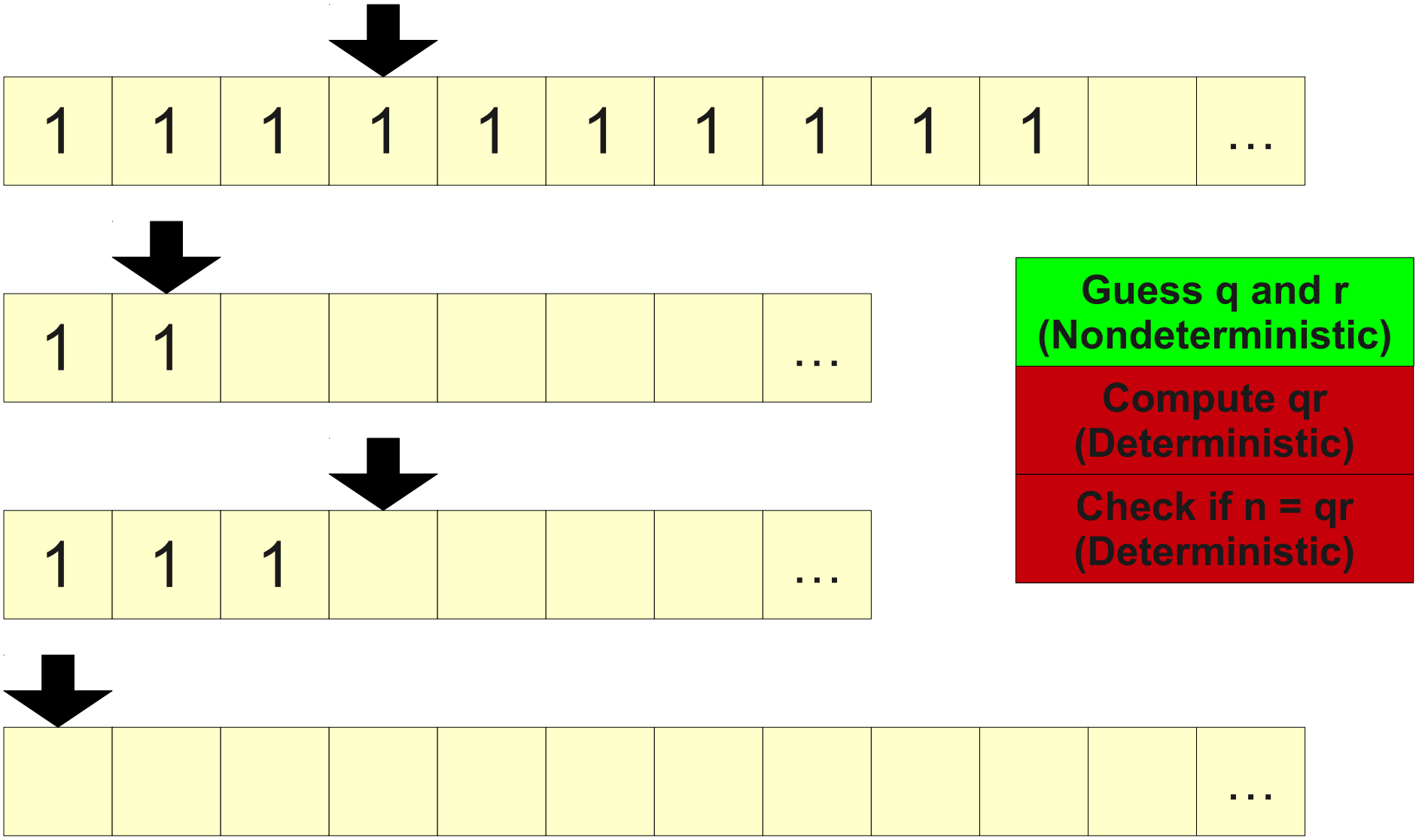
Compute qr (Deterministic)

Check if n = qr (Deterministic)

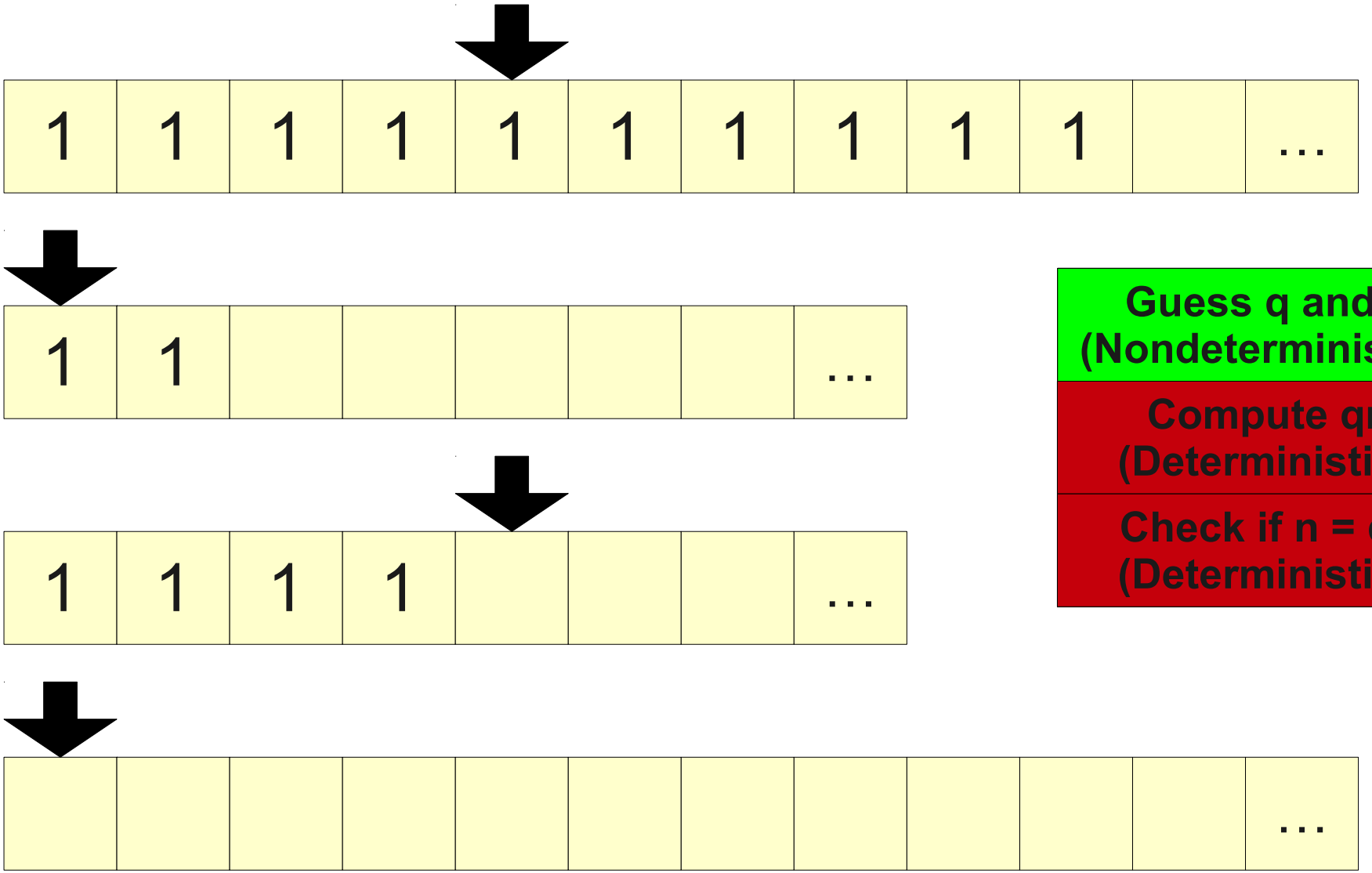
Nondeterministic Algorithms



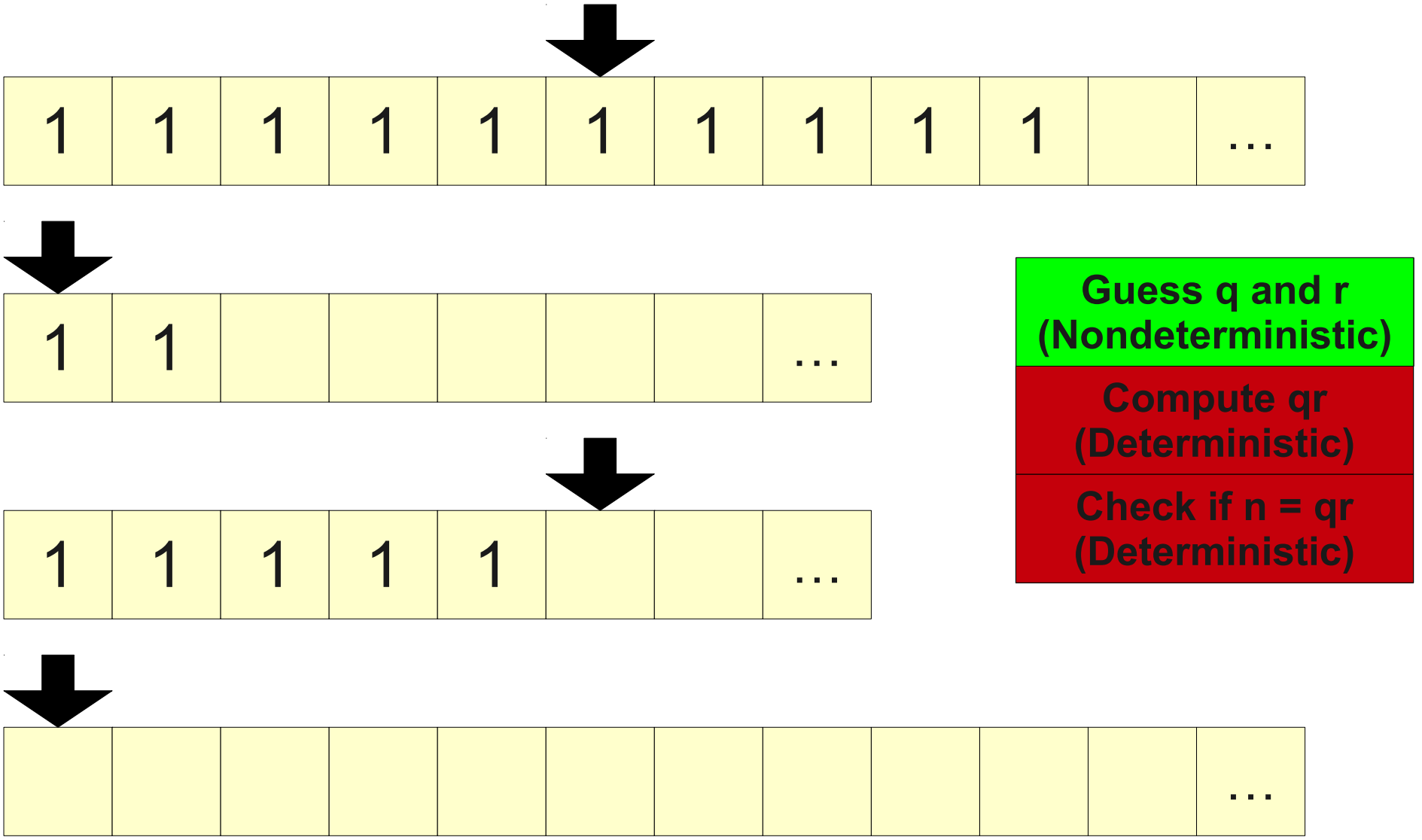
Nondeterministic Algorithms



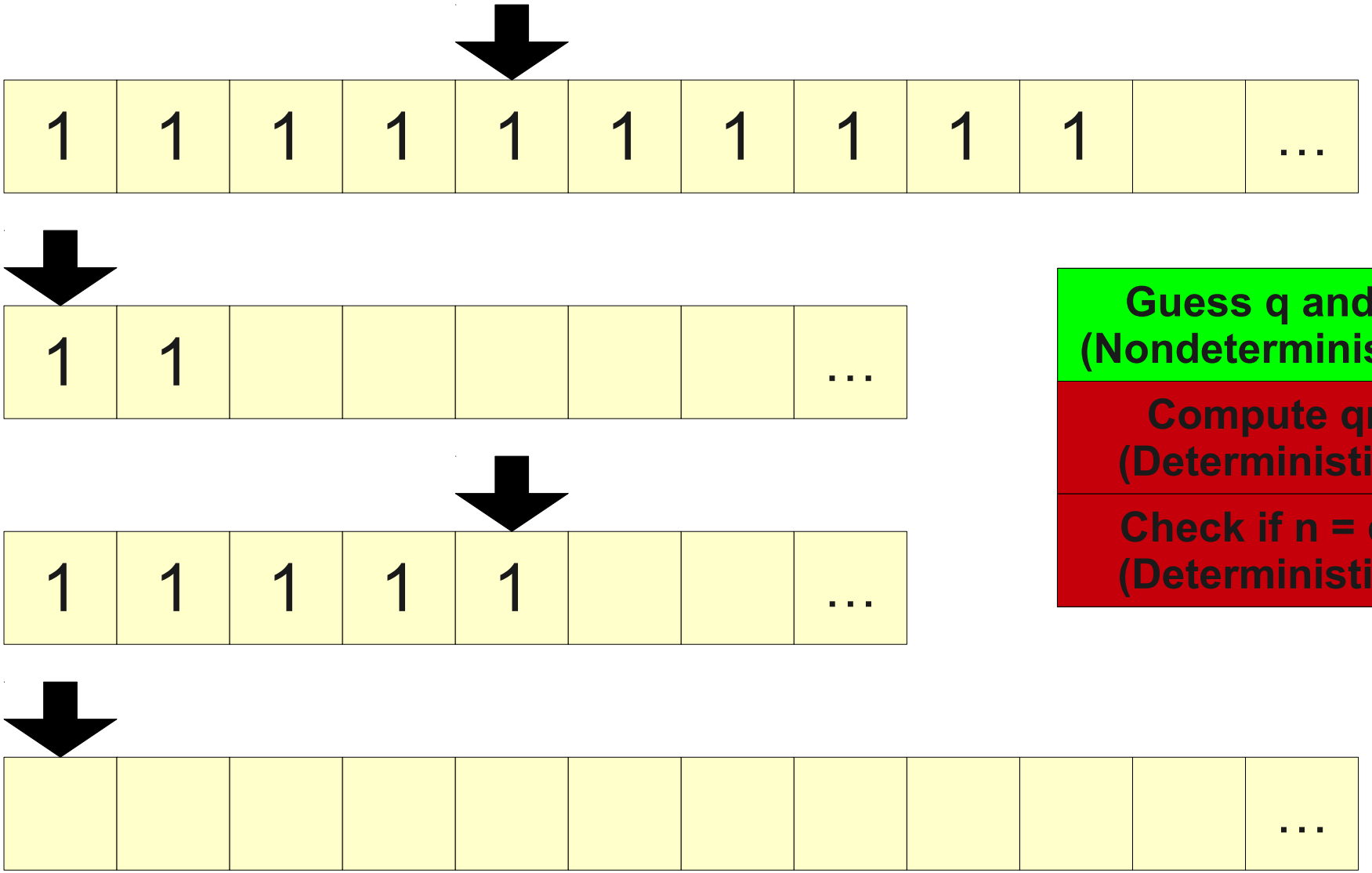
Nondeterministic Algorithms



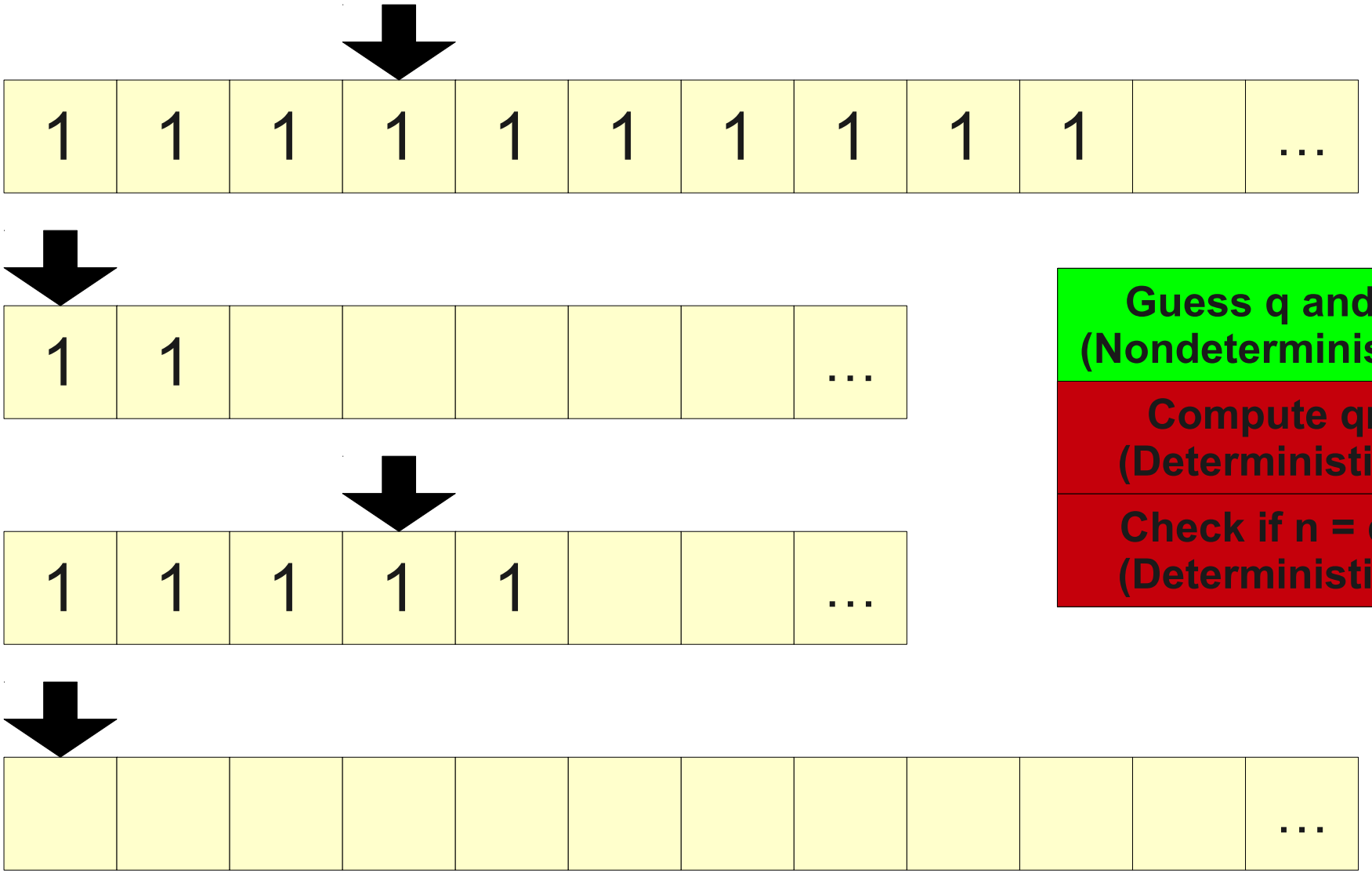
Nondeterministic Algorithms



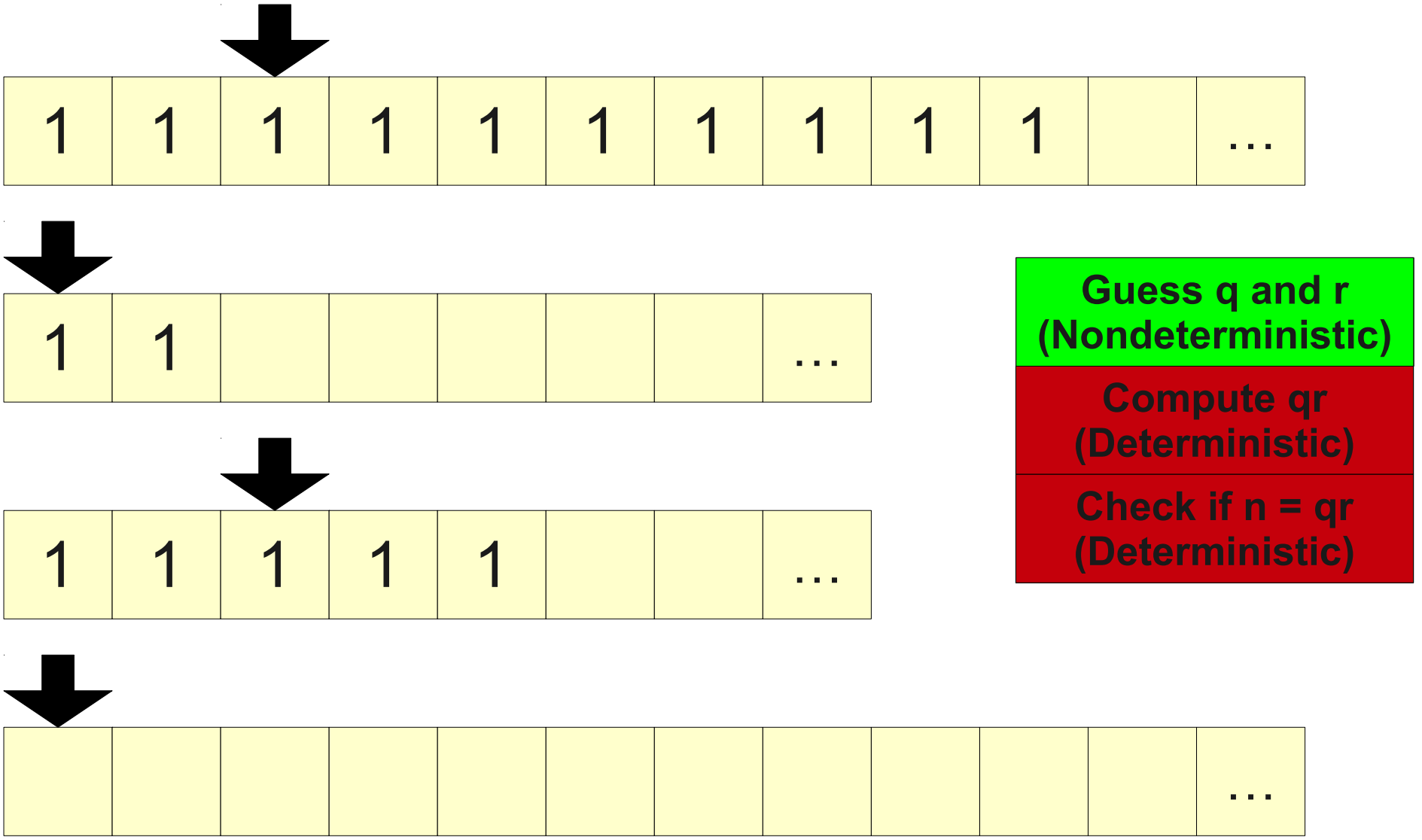
Nondeterministic Algorithms



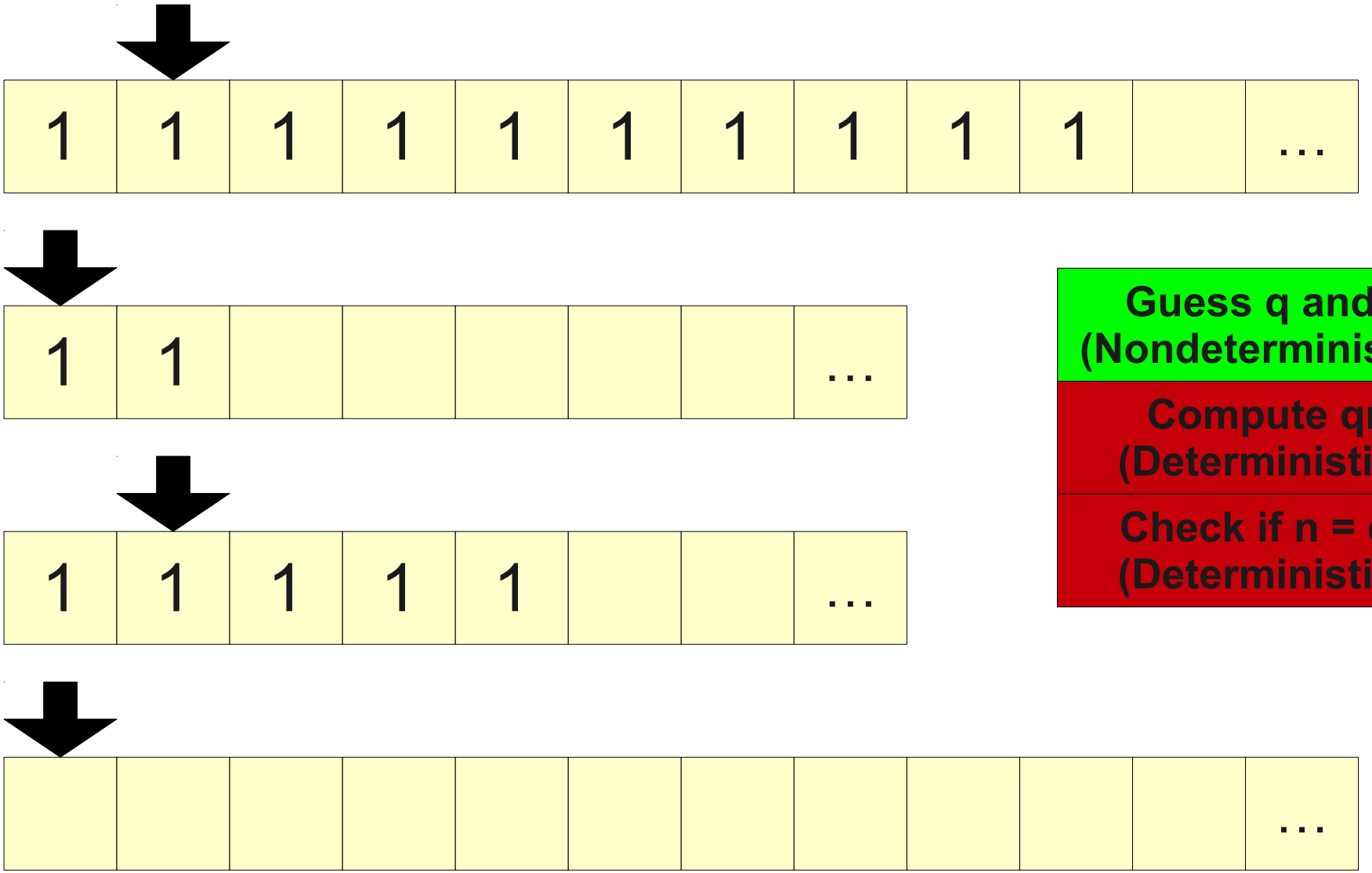
Nondeterministic Algorithms



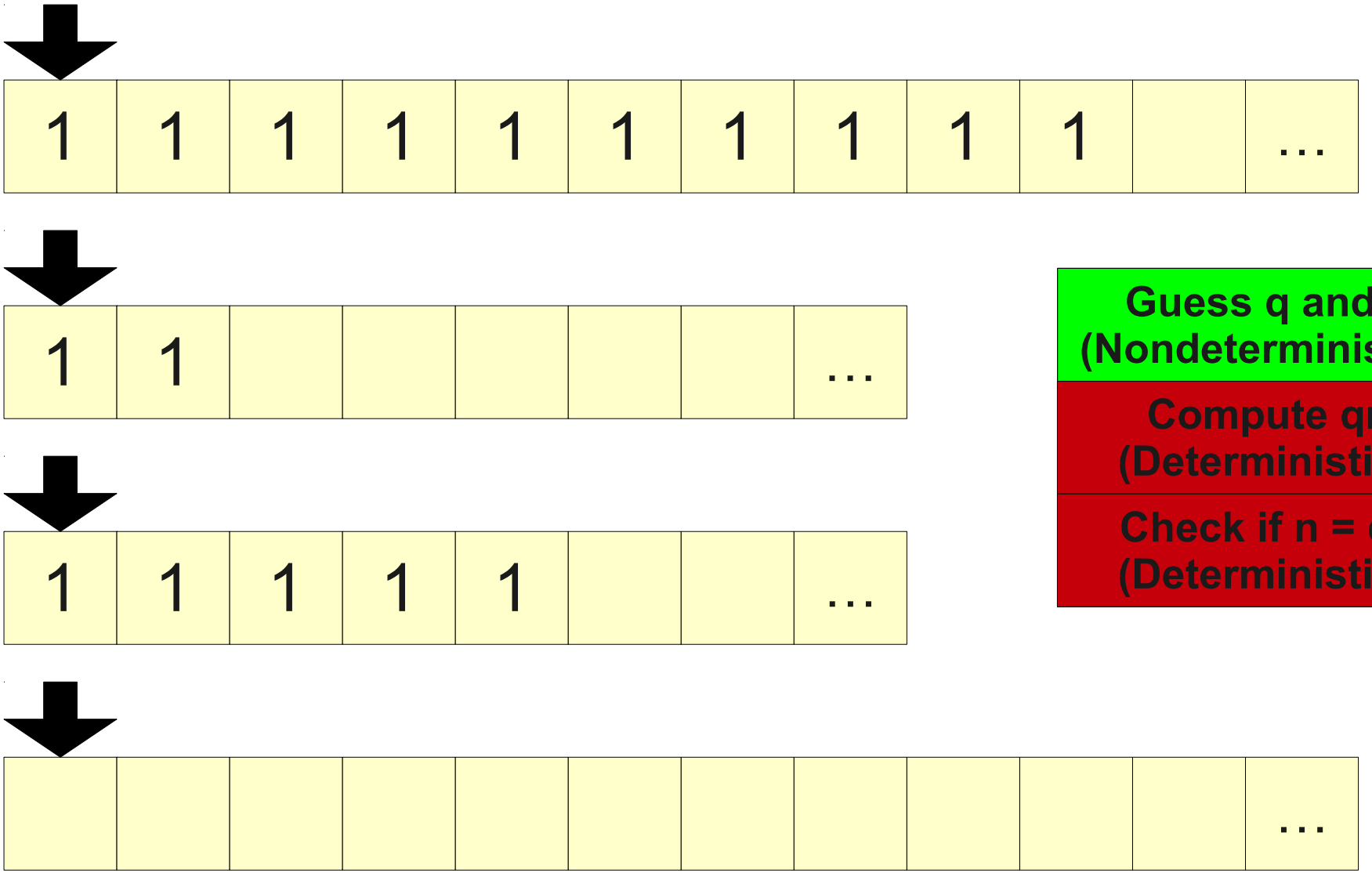
Nondeterministic Algorithms



Nondeterministic Algorithms



Nondeterministic Algorithms

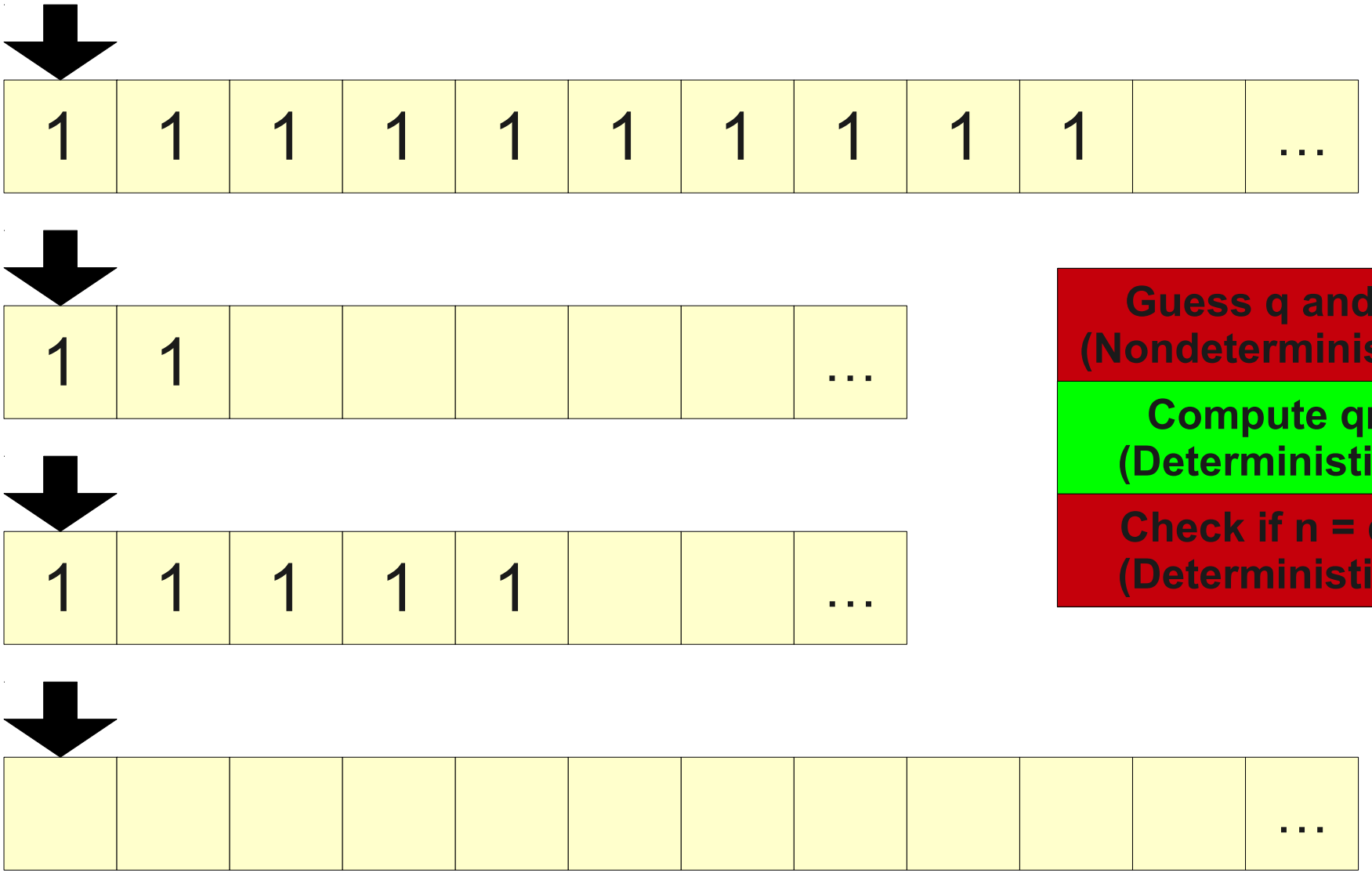


Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

Nondeterministic Algorithms

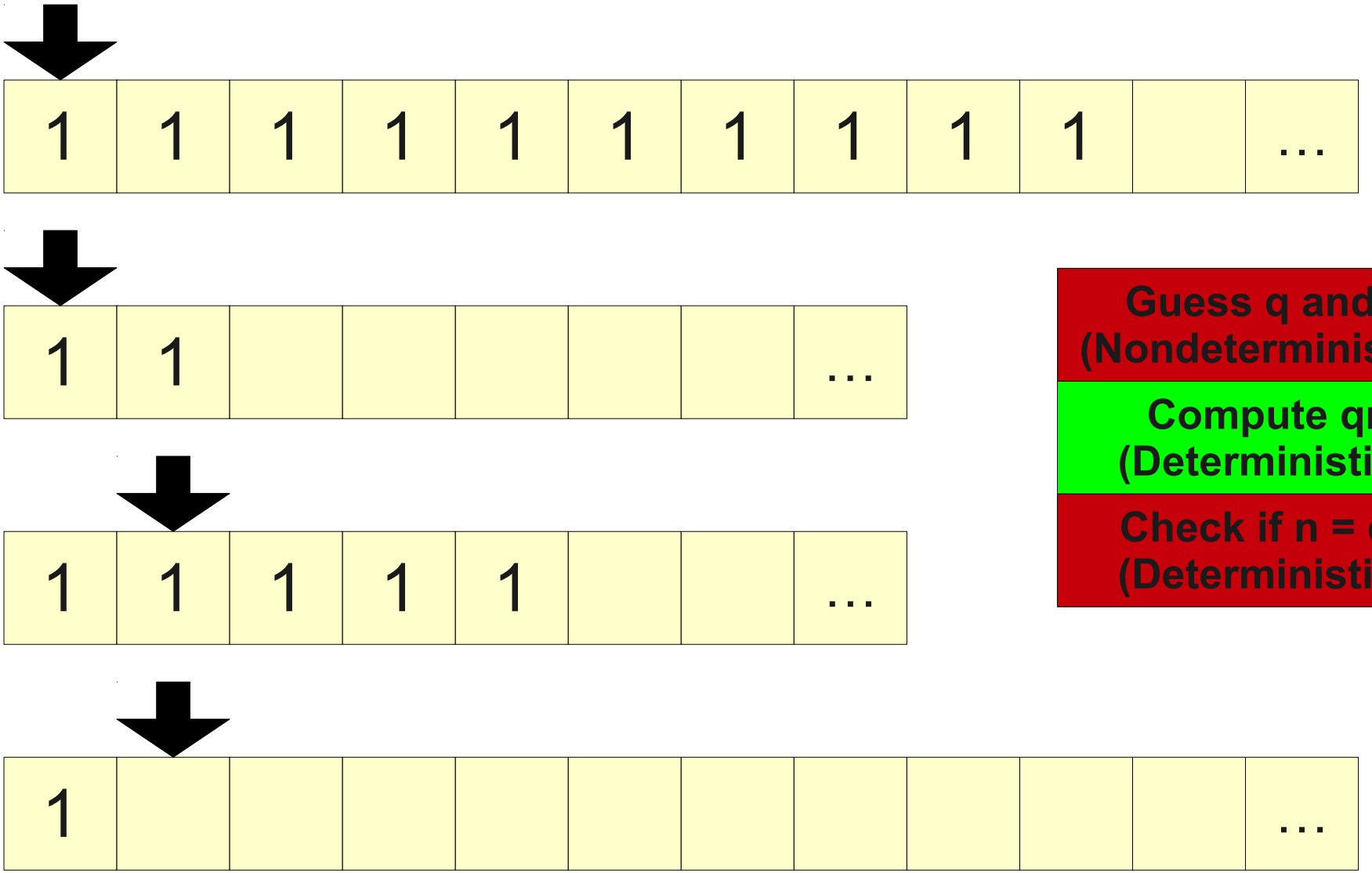


Guess q and r (Nondeterministic)

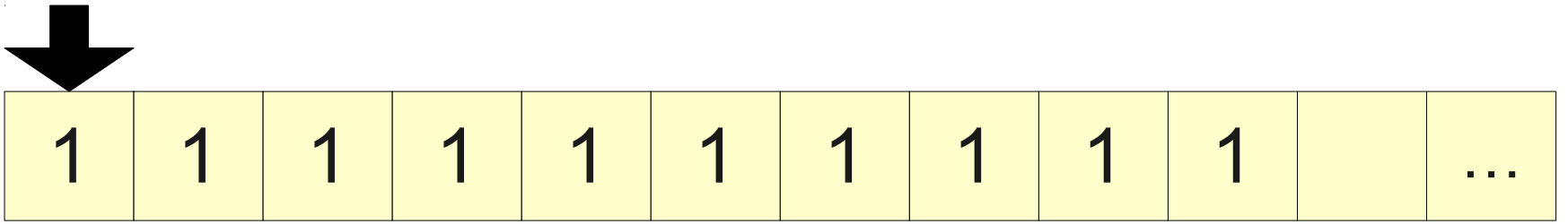
Compute qr (Deterministic)

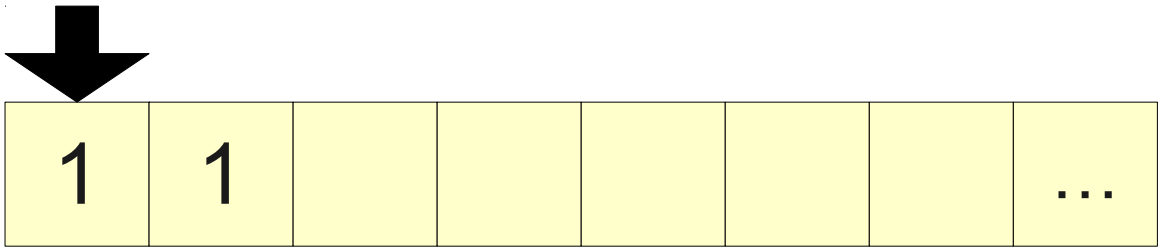
Check if n = qr (Deterministic)

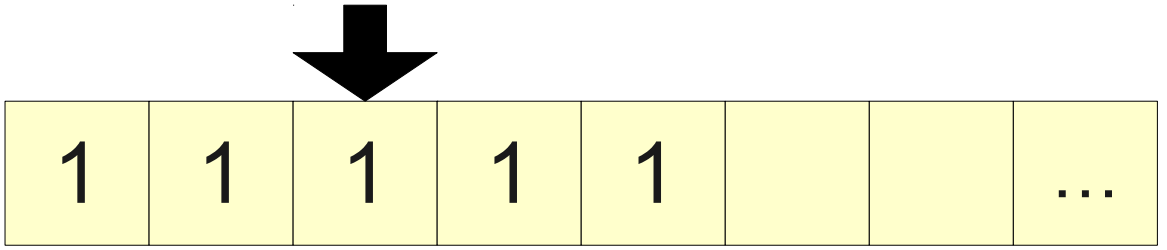
Nondeterministic Algorithms

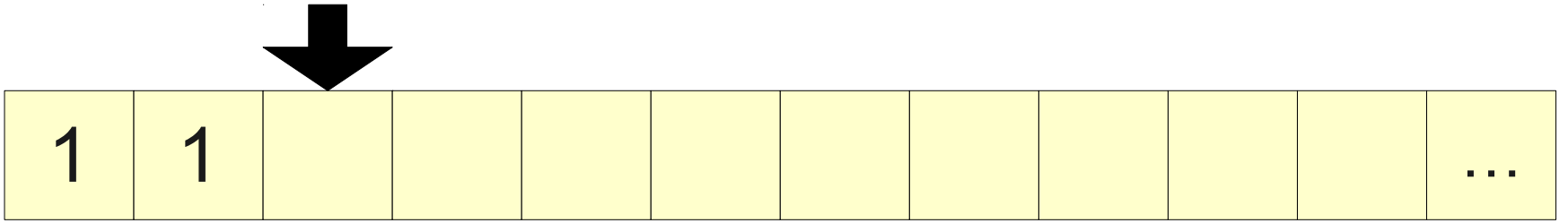


Nondeterministic Algorithms







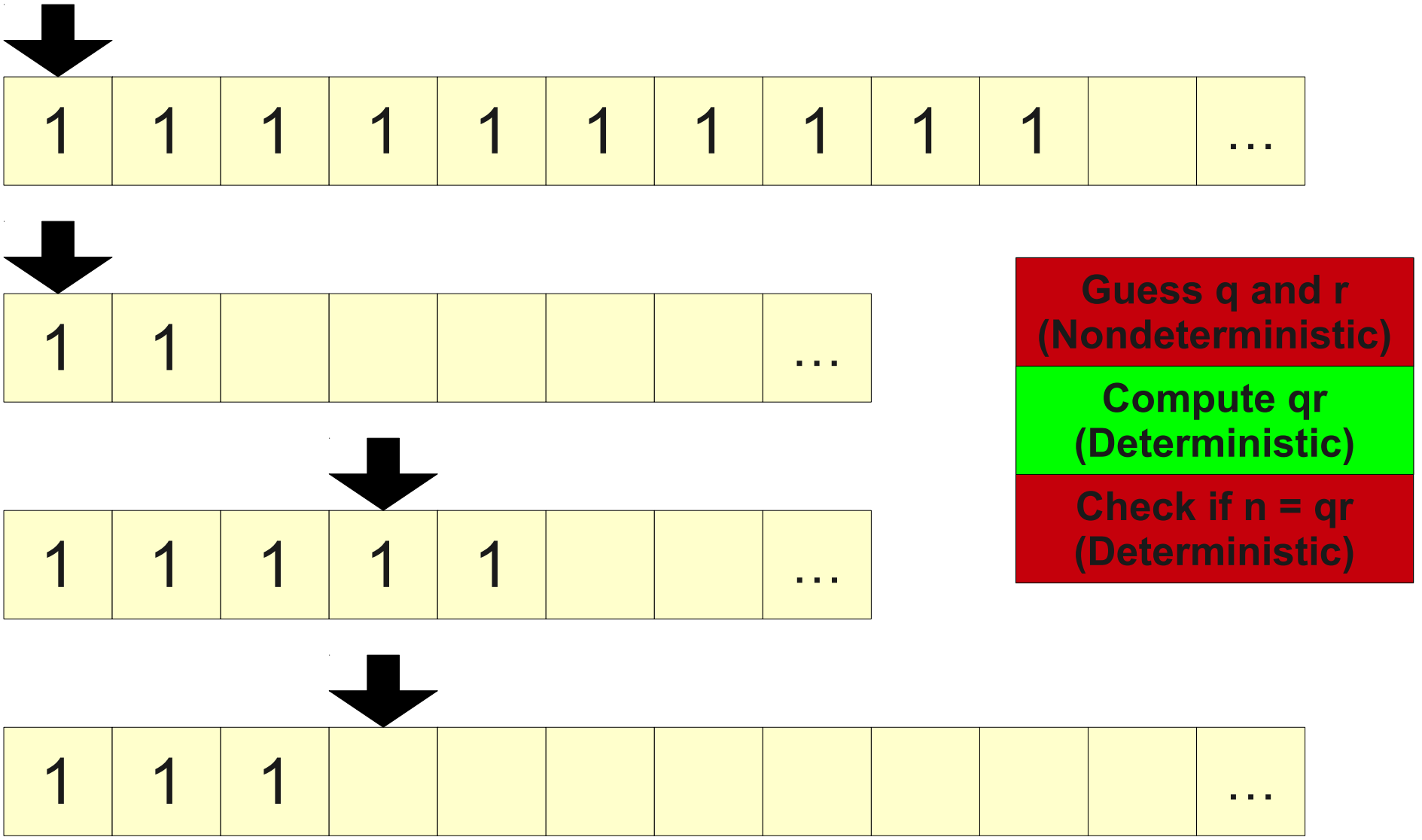


Guess q and r (Nondeterministic)

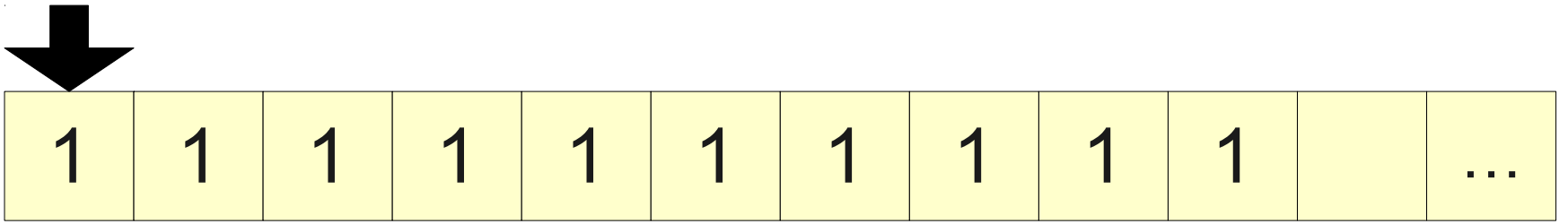
Compute qr (Deterministic)

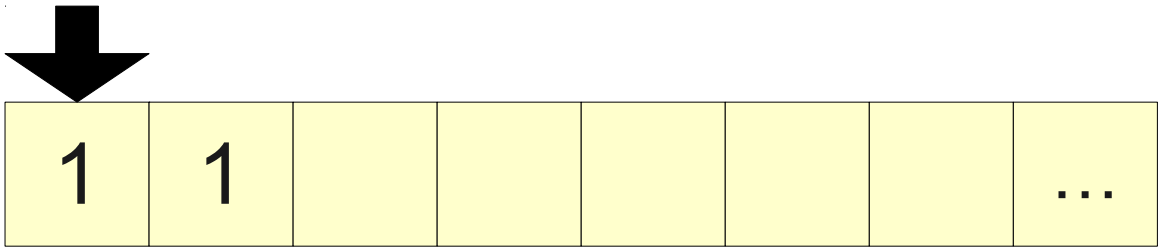
Check if n = qr (Deterministic)

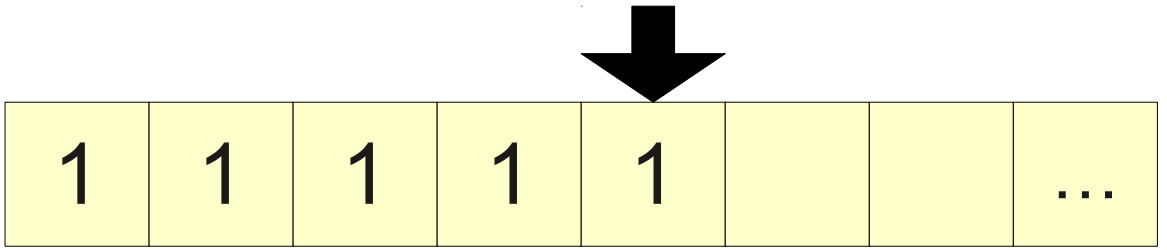
Nondeterministic Algorithms

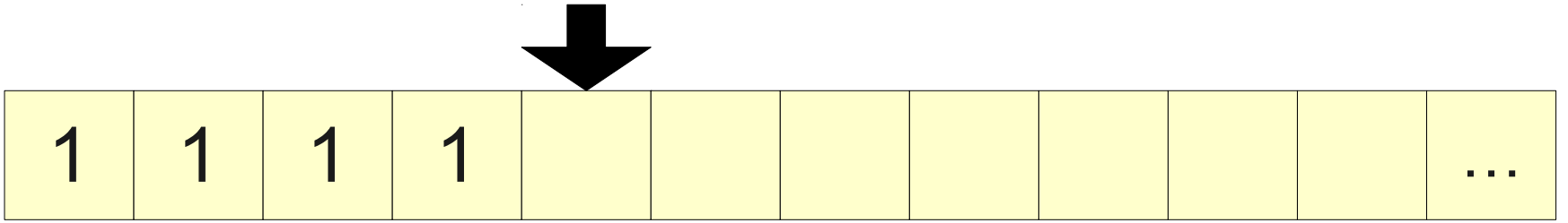


Nondeterministic Algorithms







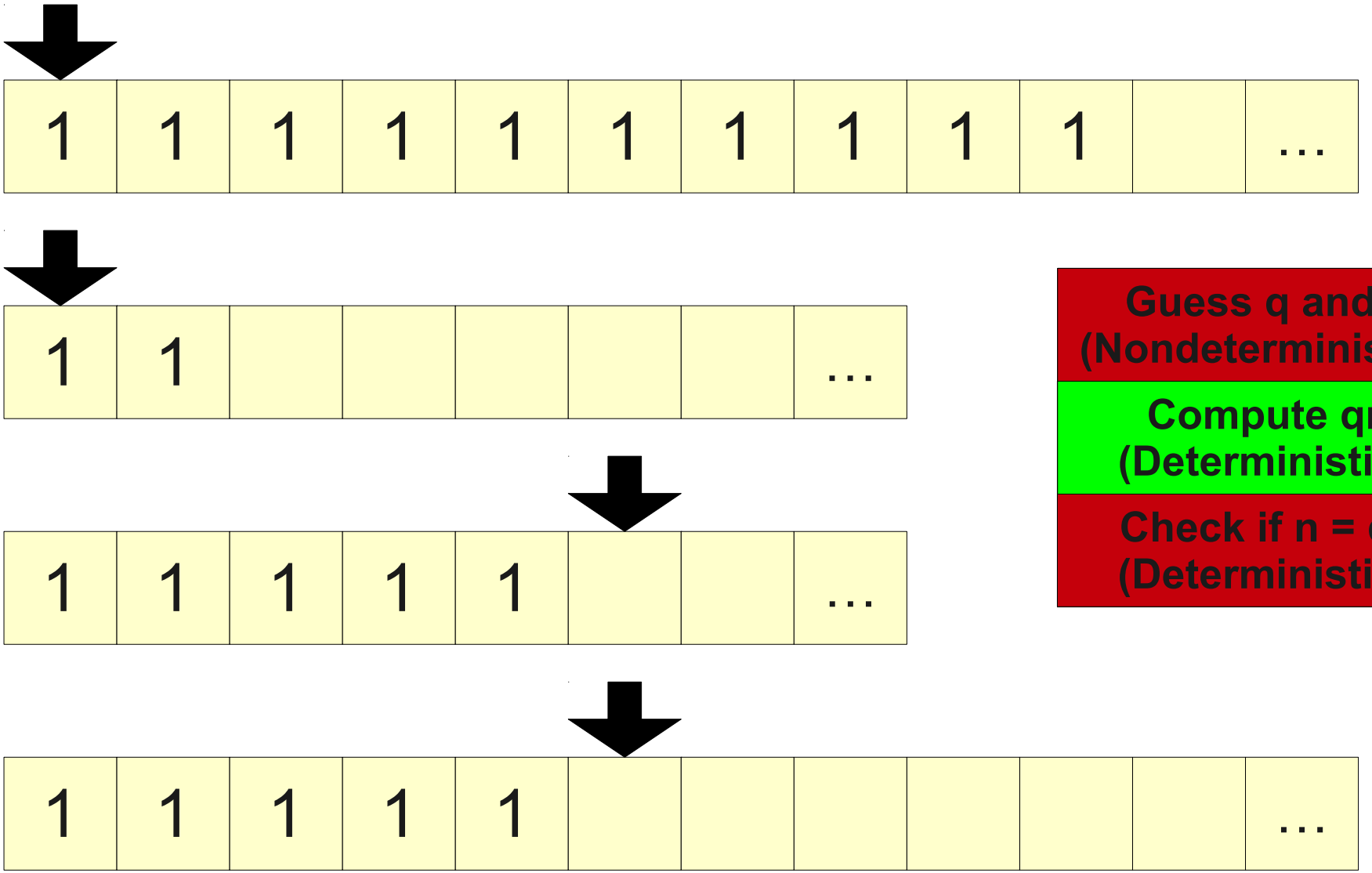


Guess q and r (Nondeterministic)

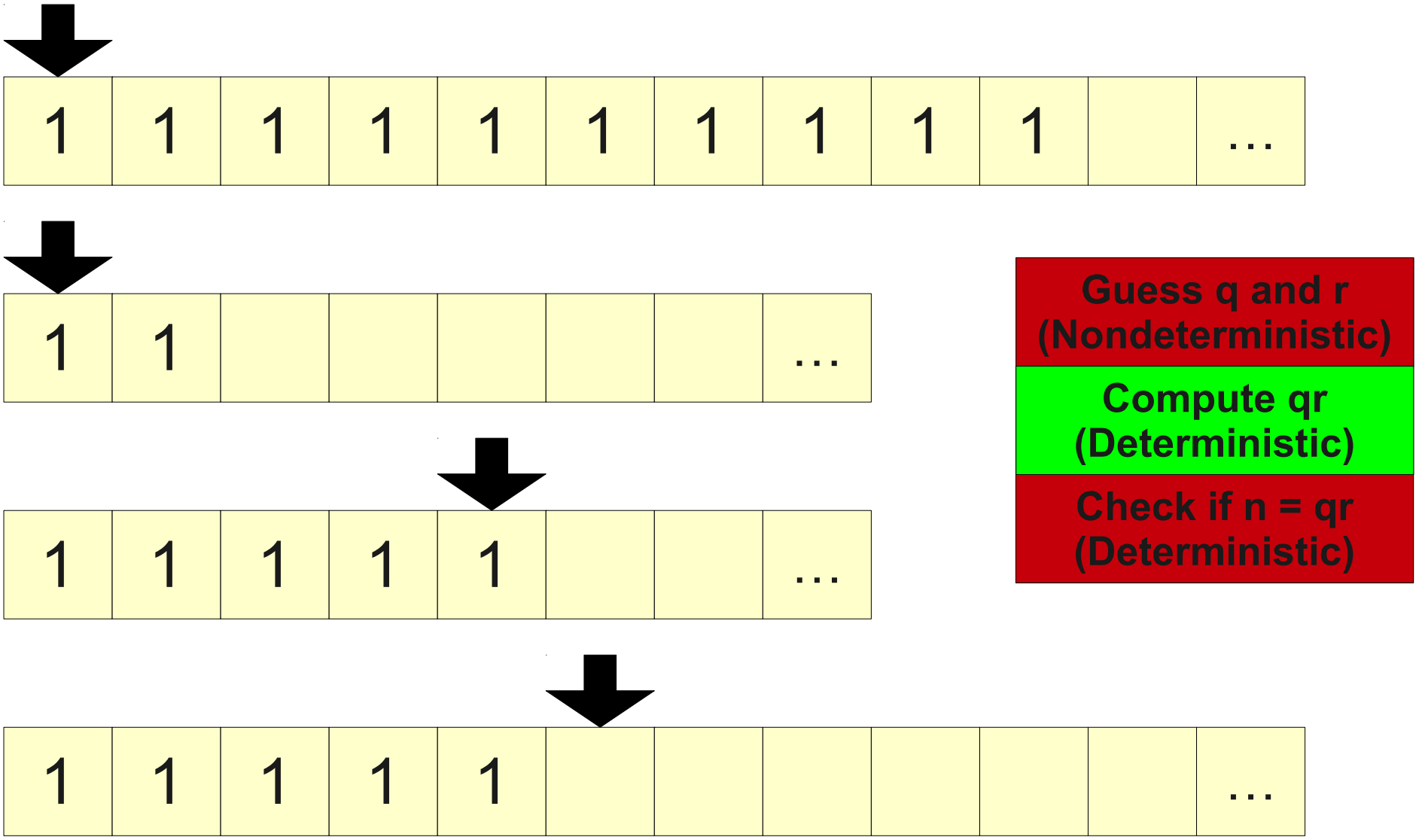
Compute qr (Deterministic)

Check if n = qr (Deterministic)

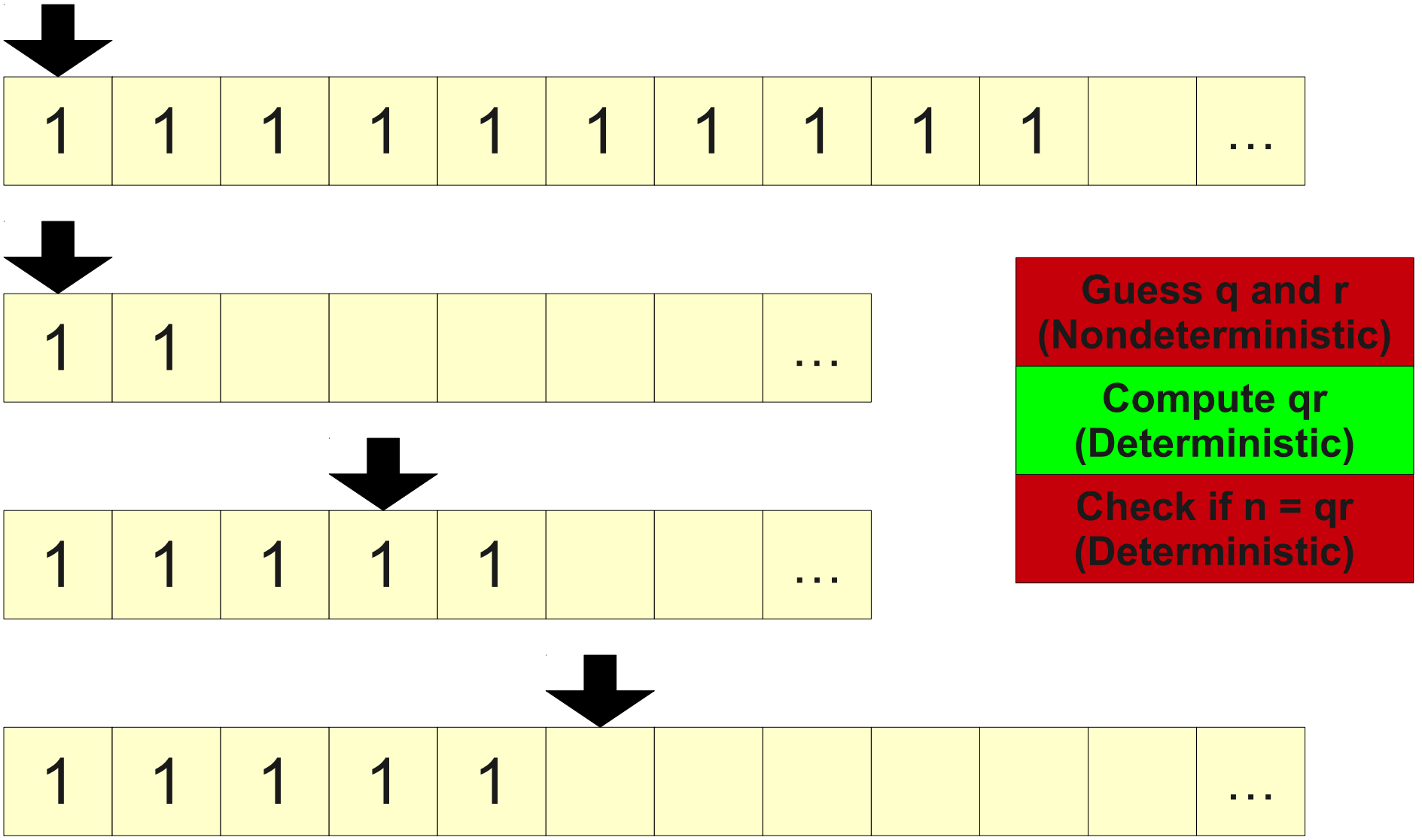
Nondeterministic Algorithms



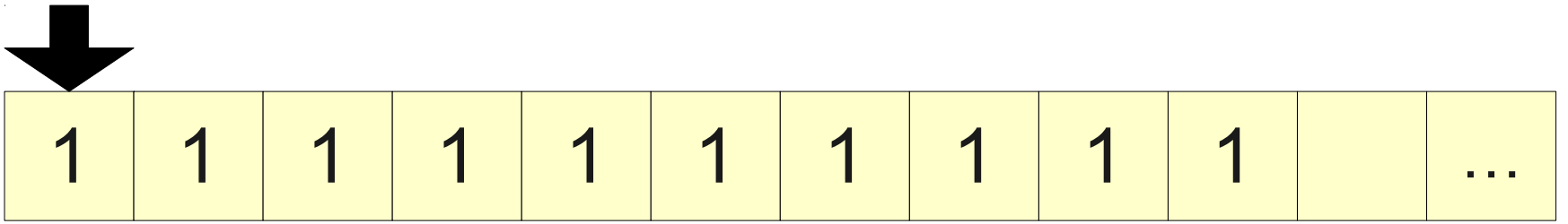
Nondeterministic Algorithms

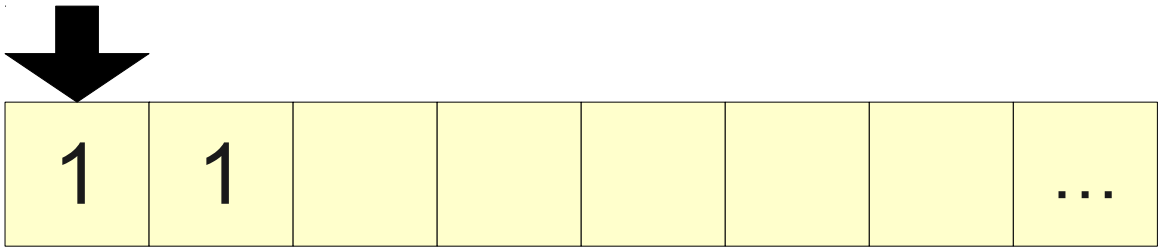


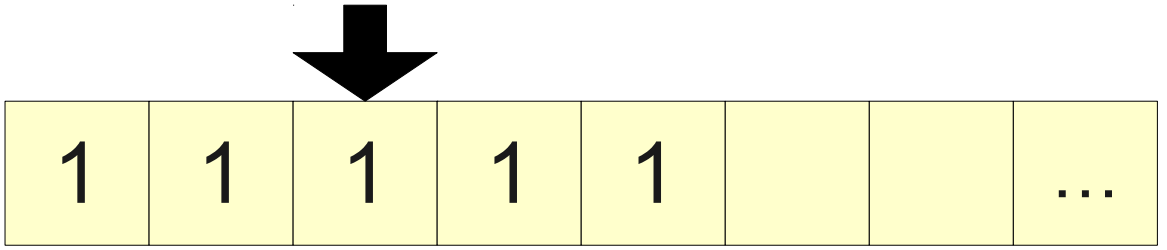
Nondeterministic Algorithms

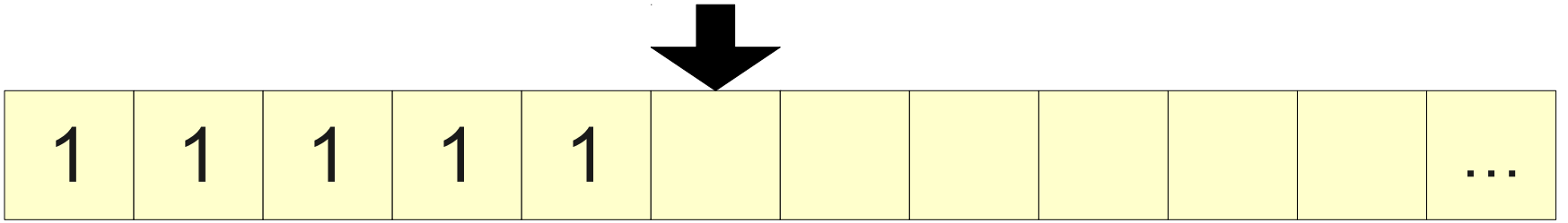


Nondeterministic Algorithms







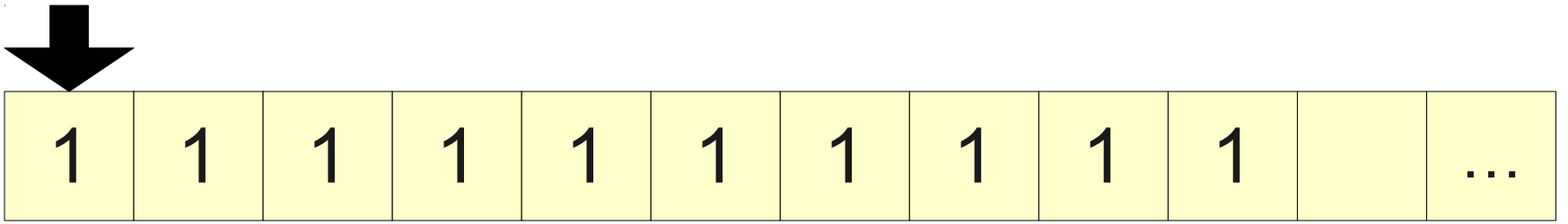


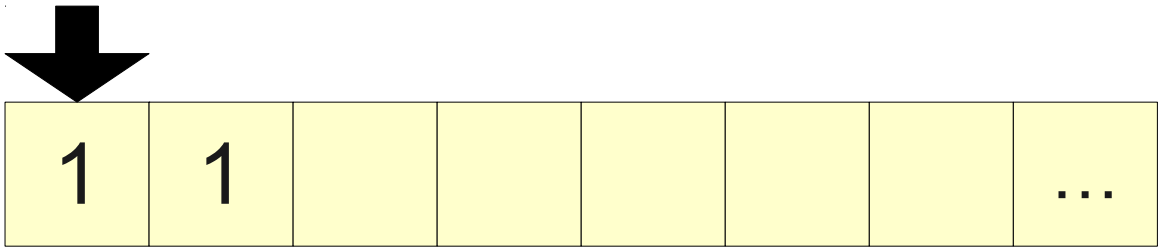
Guess q and r (Nondeterministic)

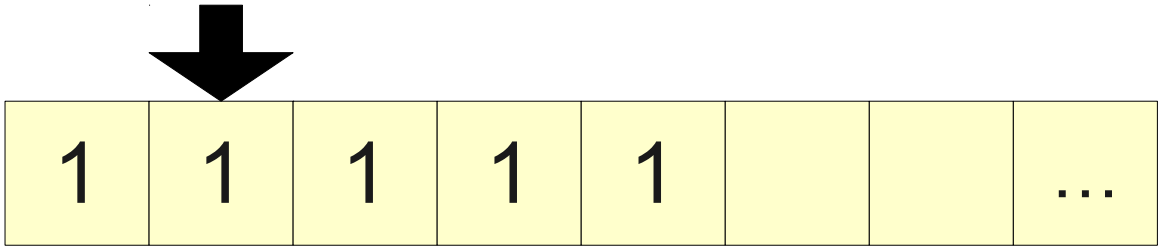
Compute qr (Deterministic)

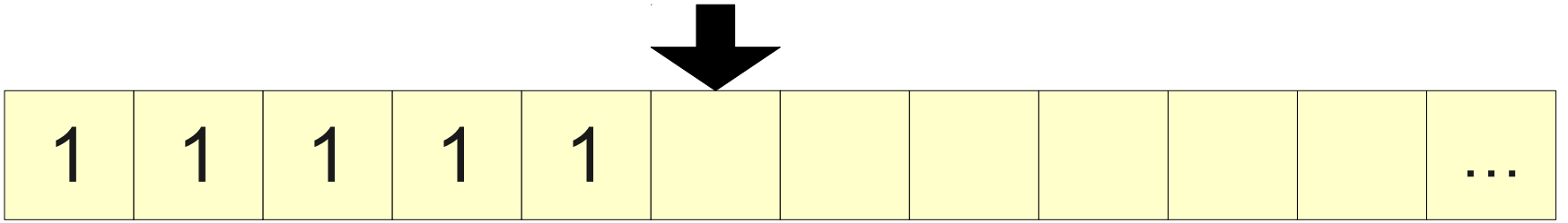
Check if n = qr (Deterministic)

Nondeterministic Algorithms







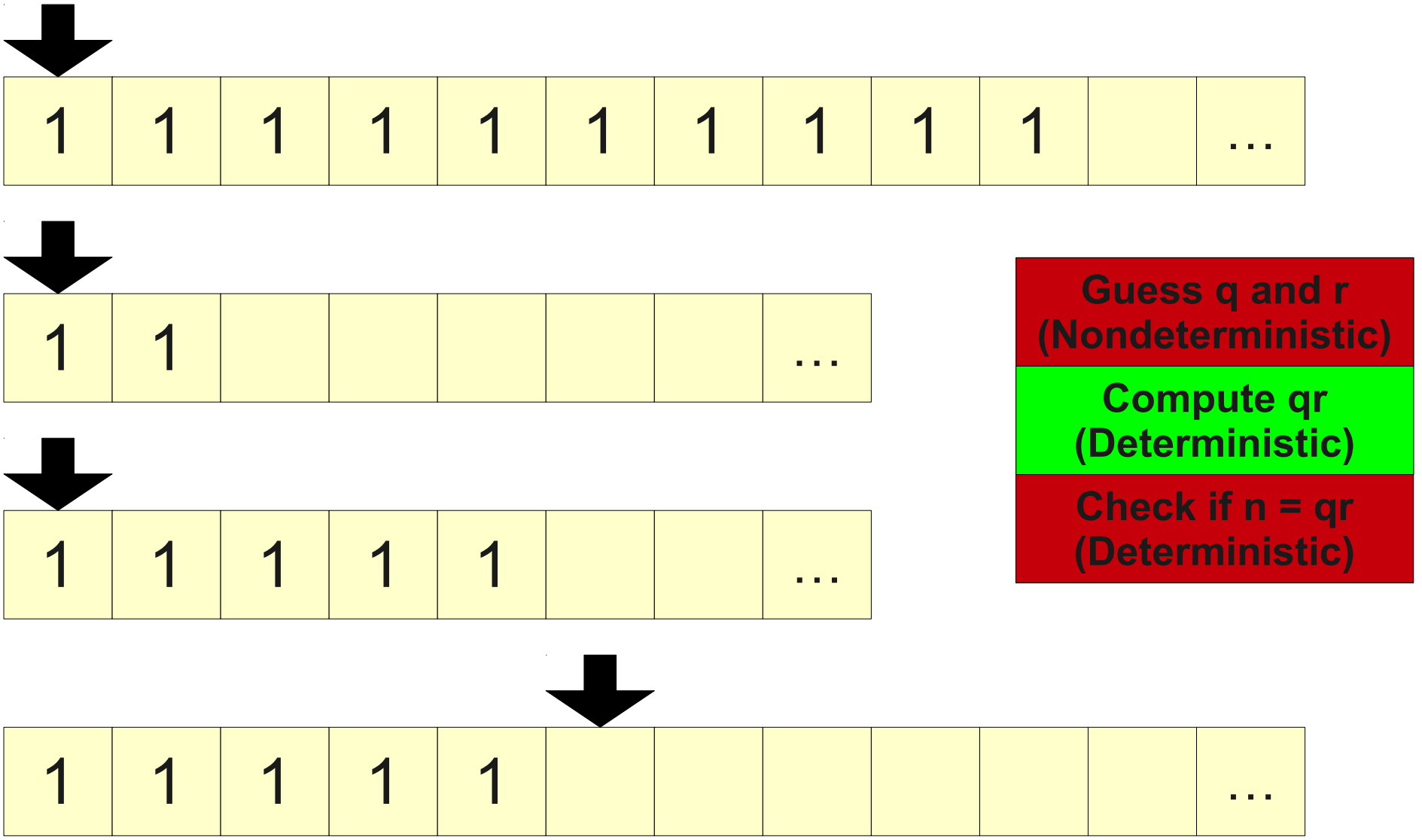


Guess q and r (Nondeterministic)

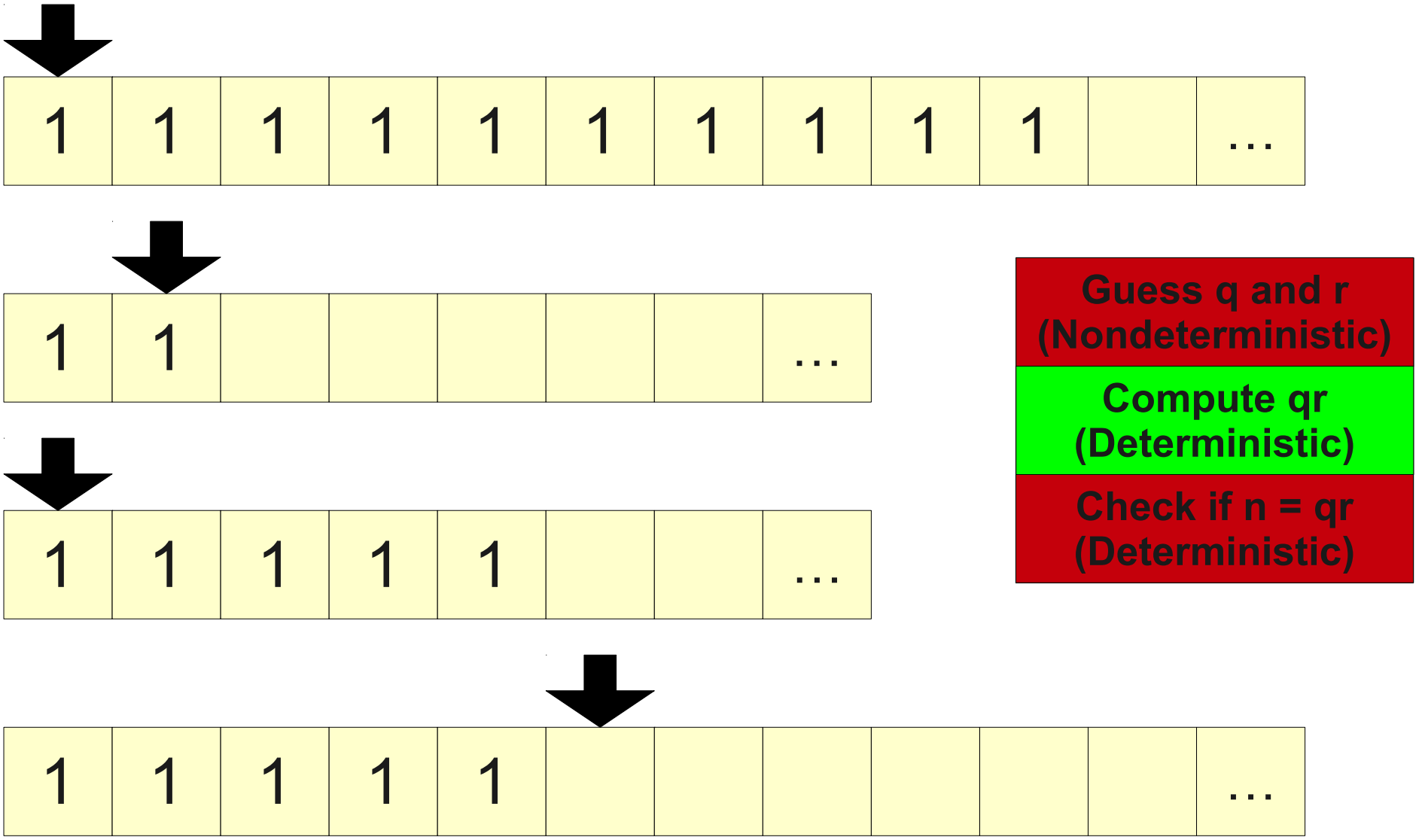
Compute qr (Deterministic)

Check if n = qr (Deterministic)

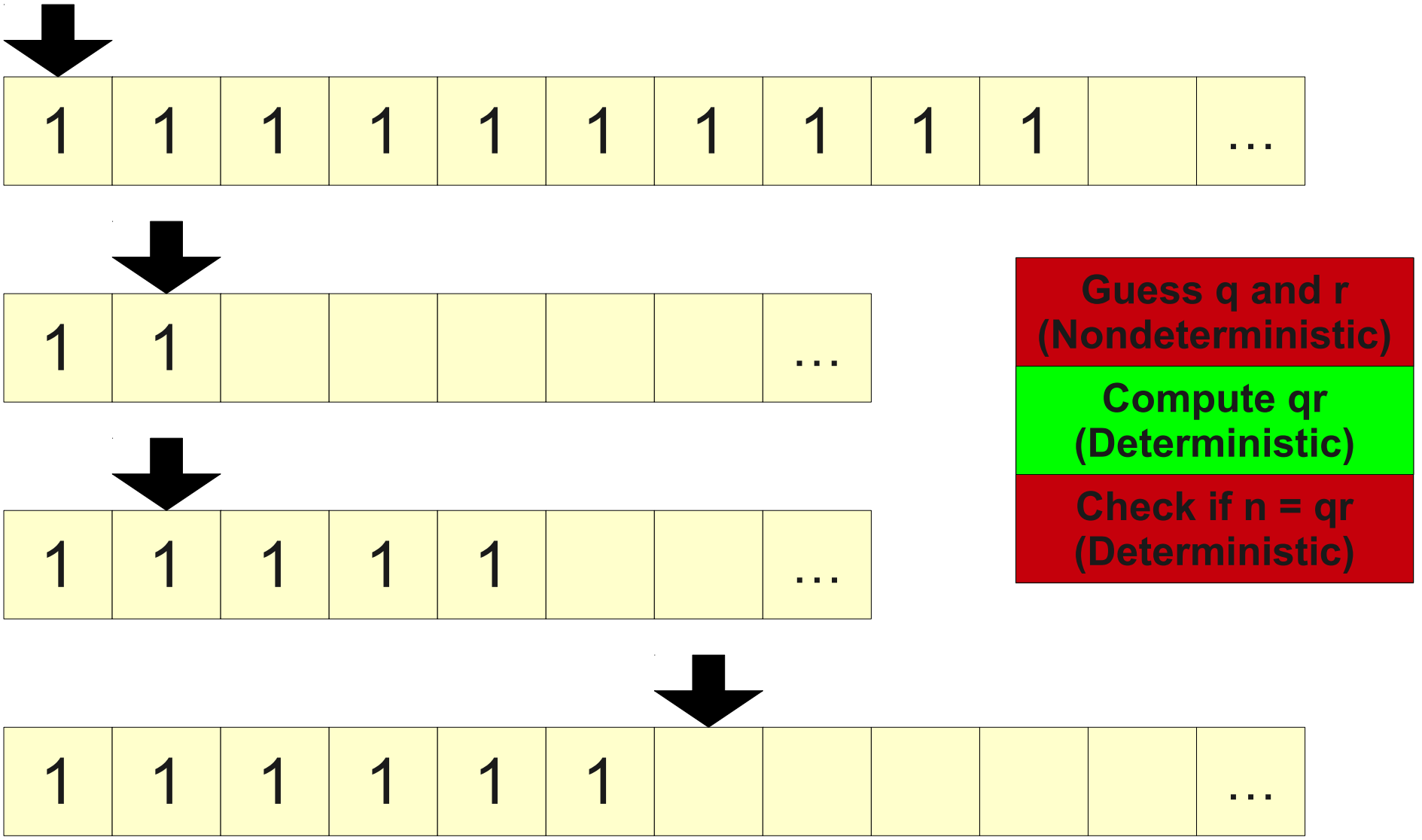
Nondeterministic Algorithms



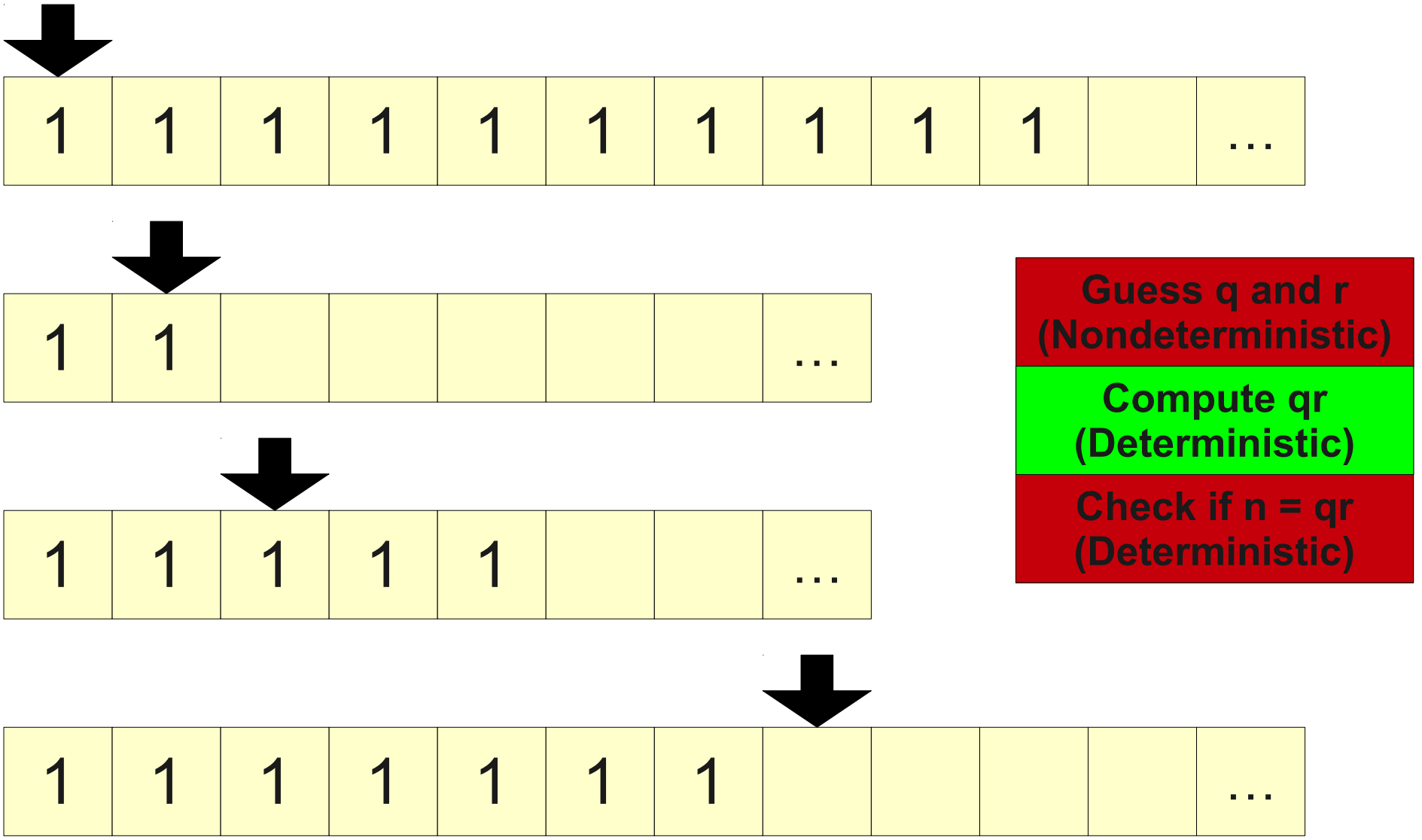
Nondeterministic Algorithms



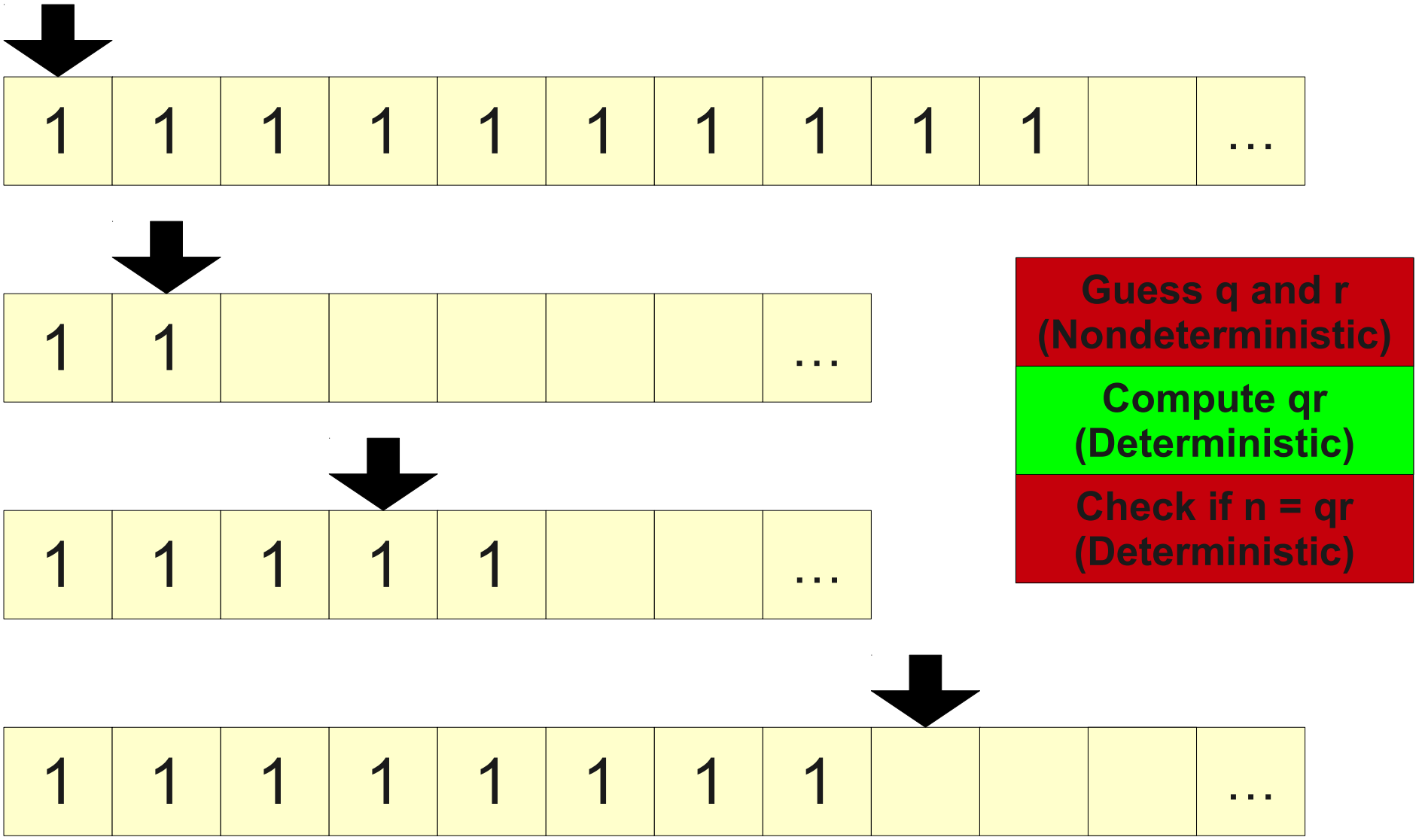
Nondeterministic Algorithms



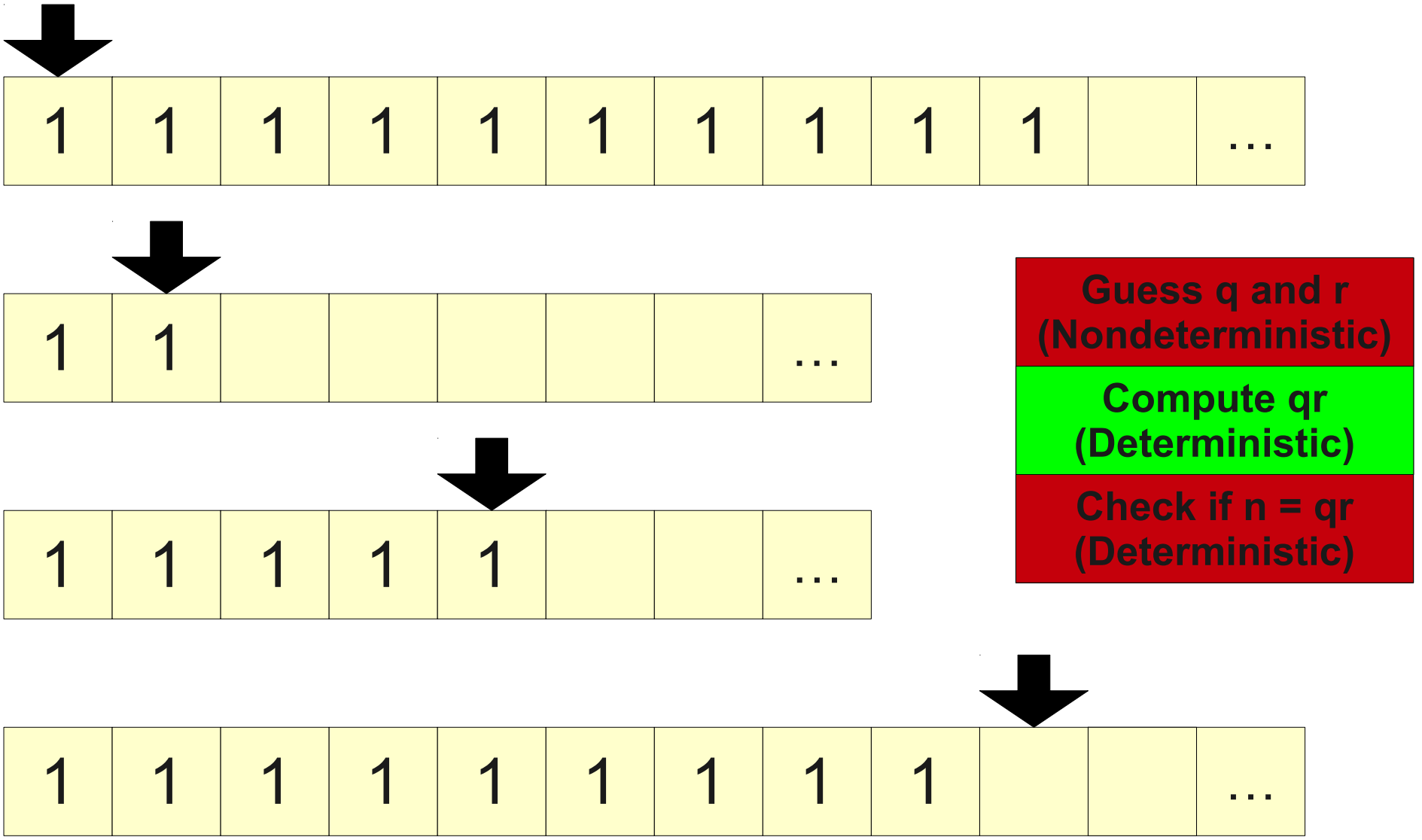
Nondeterministic Algorithms



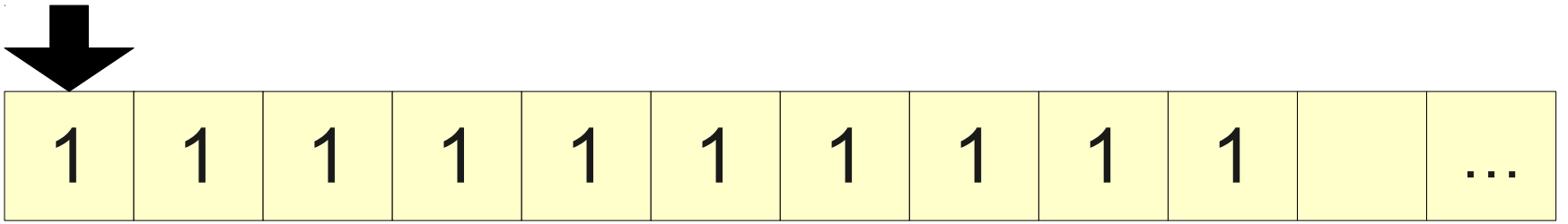
Nondeterministic Algorithms

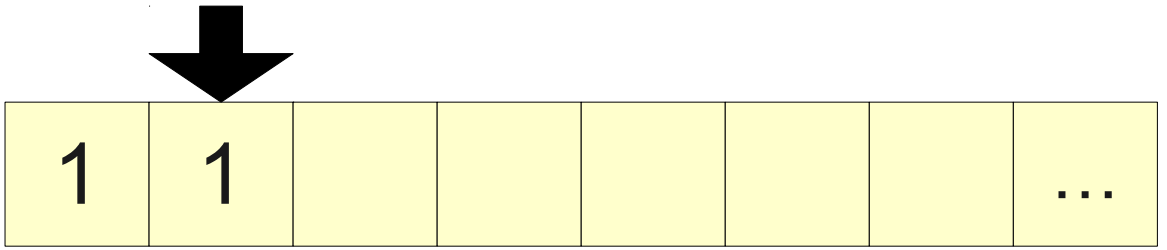


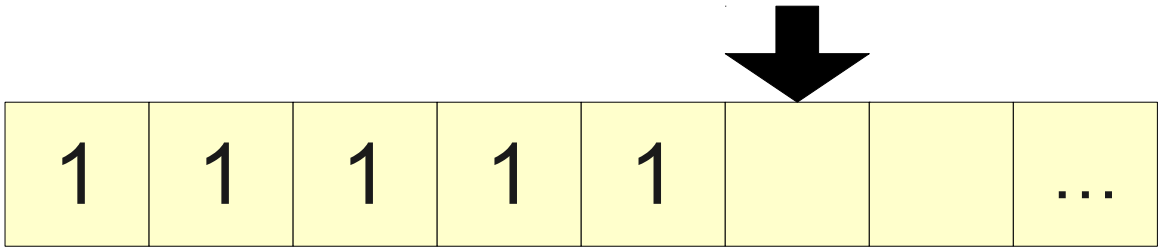
Nondeterministic Algorithms

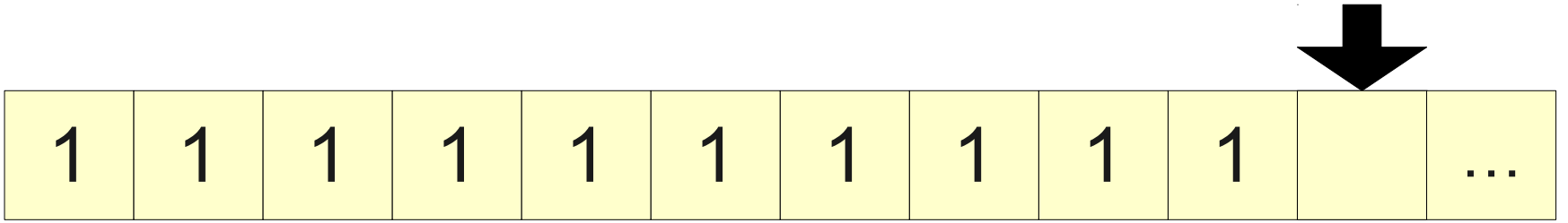


Nondeterministic Algorithms







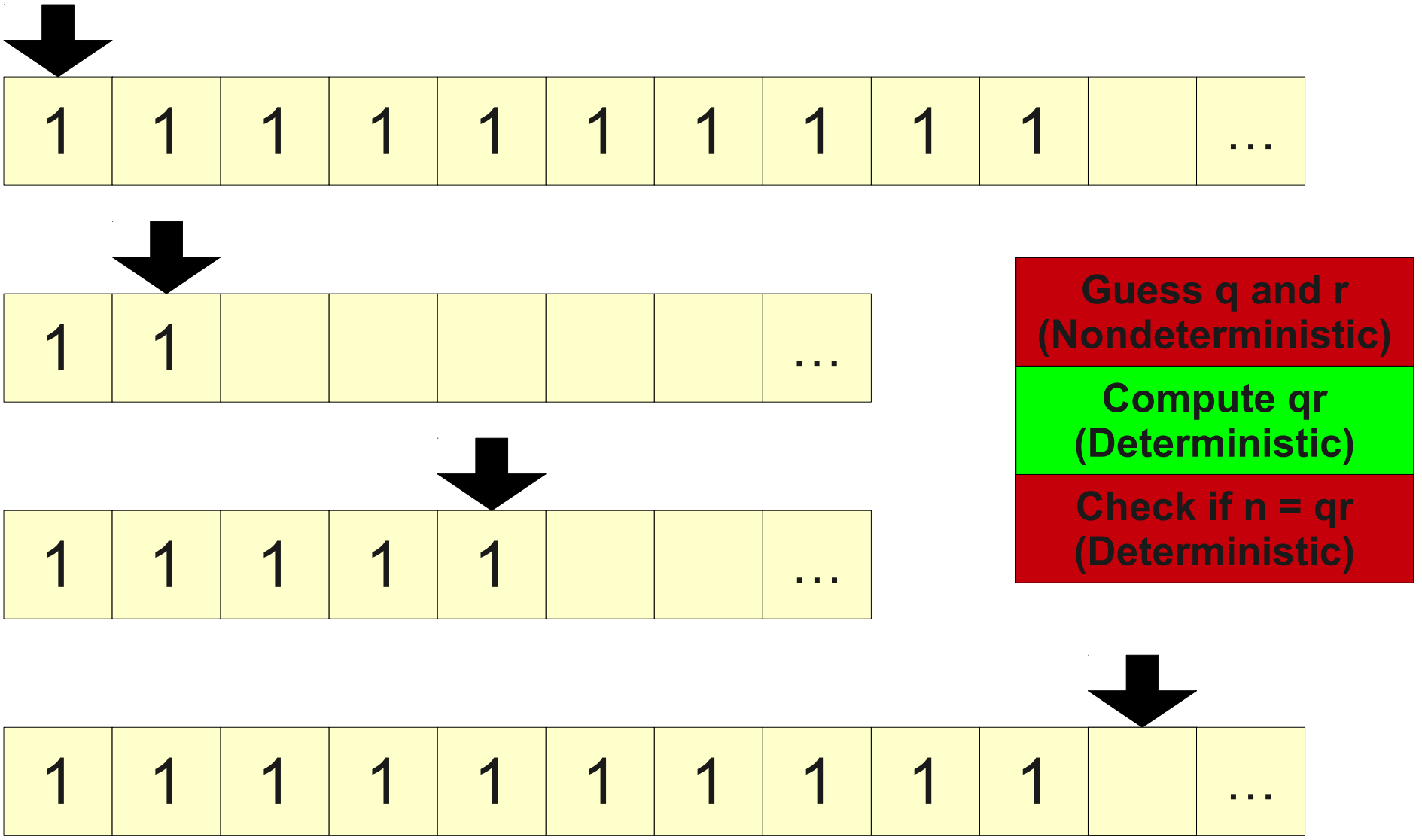


Guess q and r (Nondeterministic)

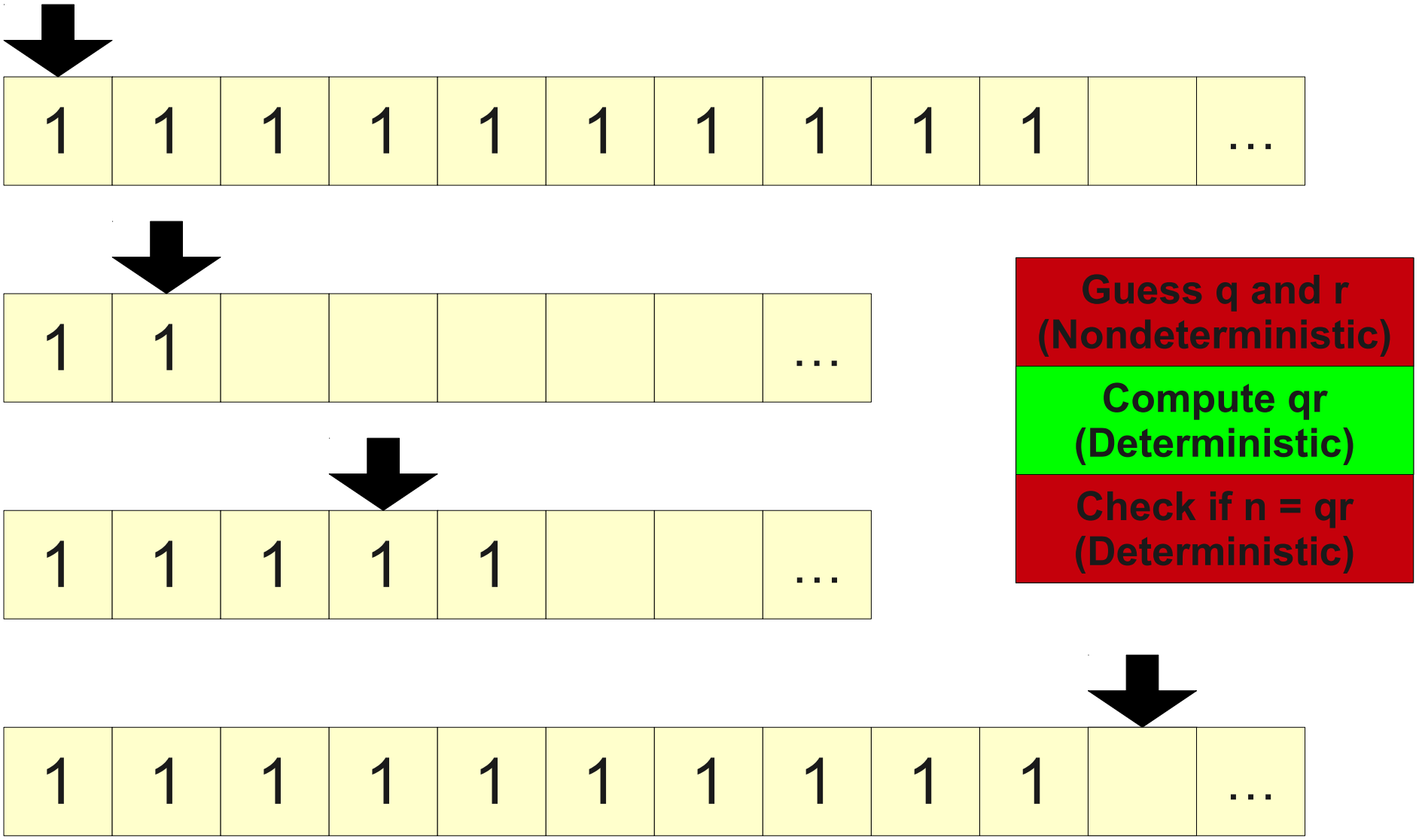
Compute qr (Deterministic)

Check if n = qr (Deterministic)

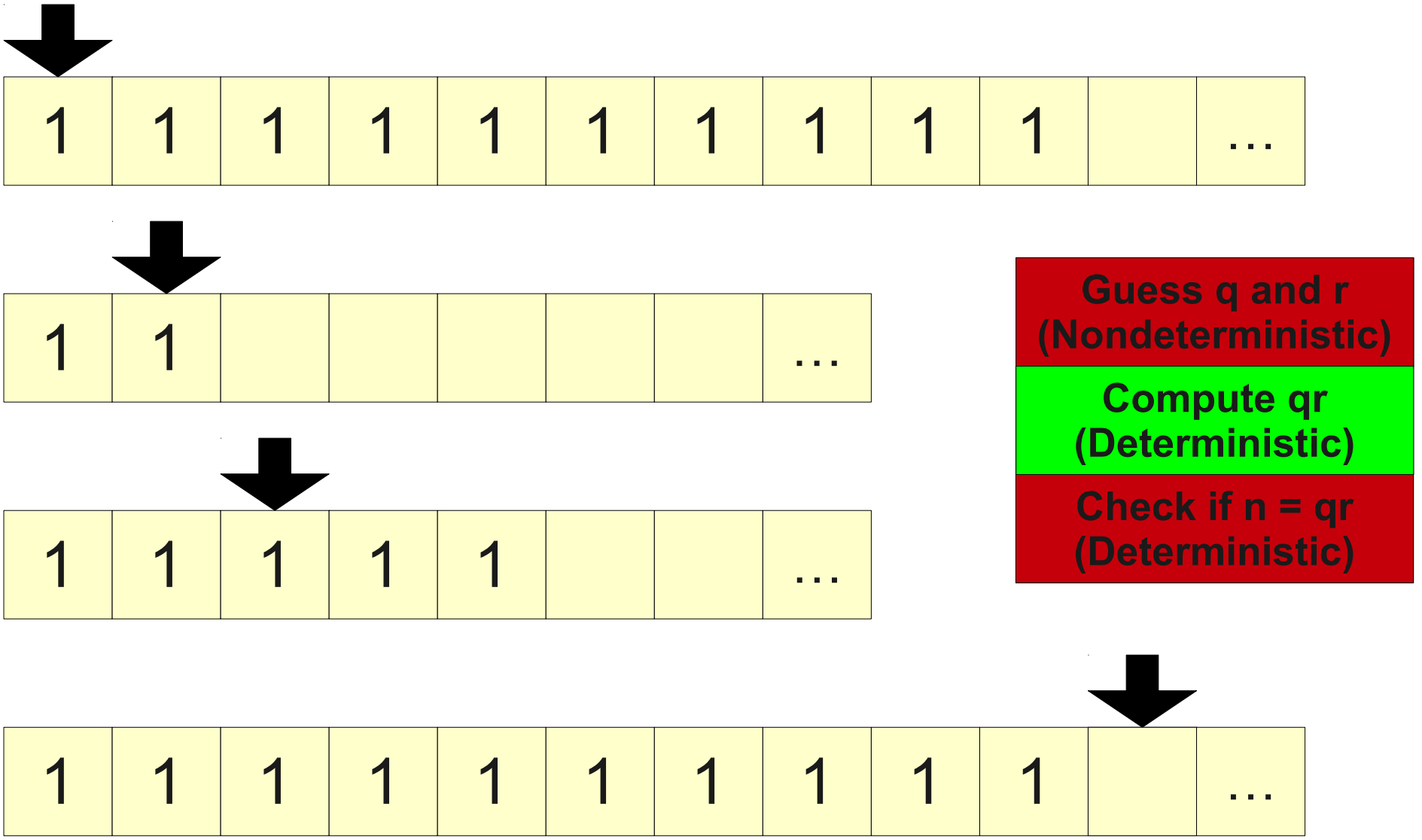
Nondeterministic Algorithms



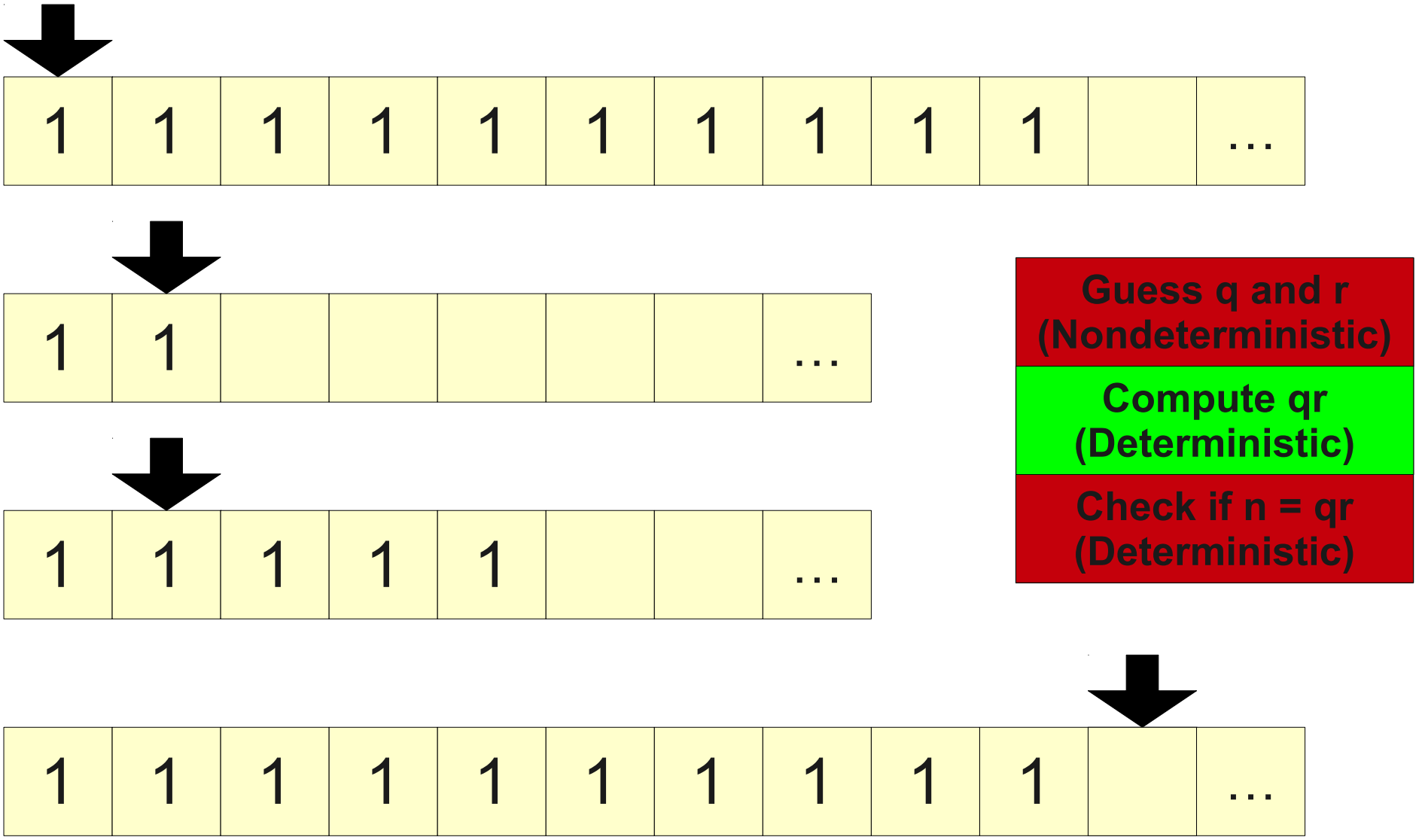
Nondeterministic Algorithms



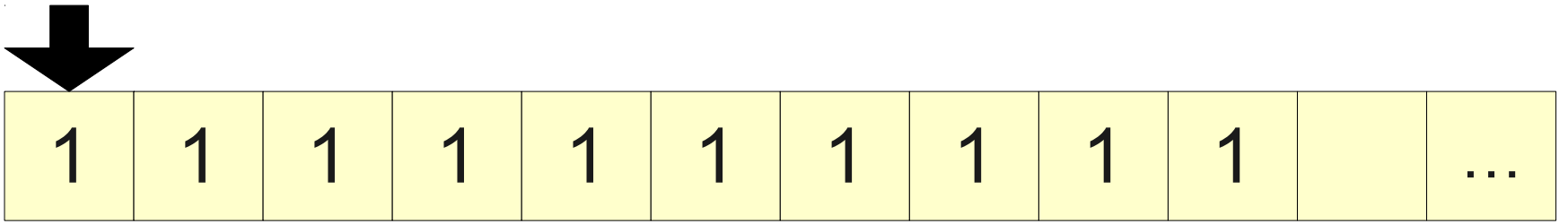
Nondeterministic Algorithms

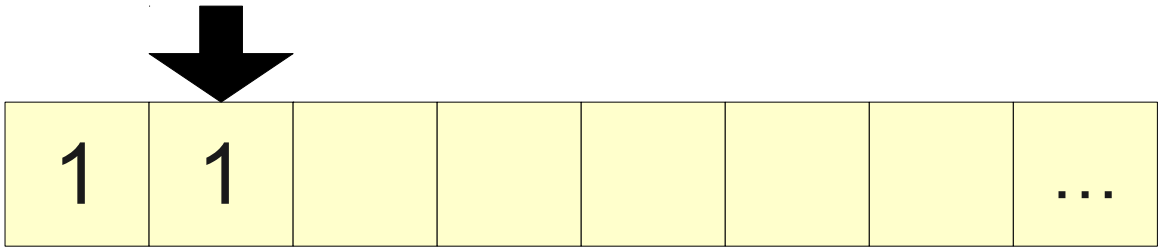


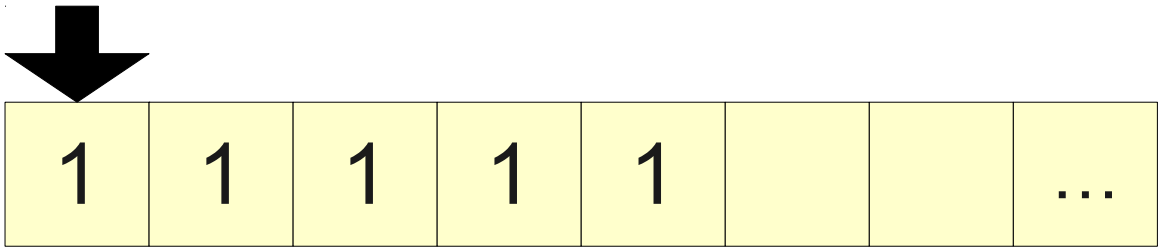
Nondeterministic Algorithms

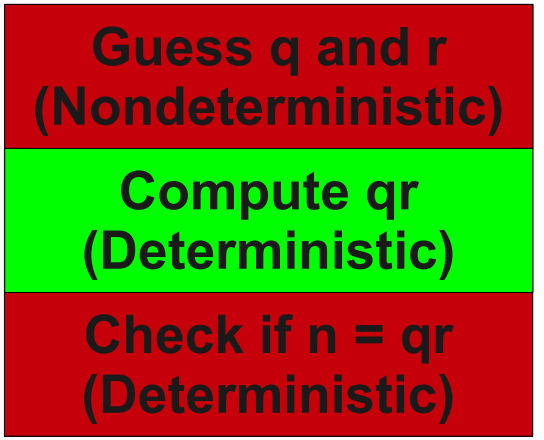


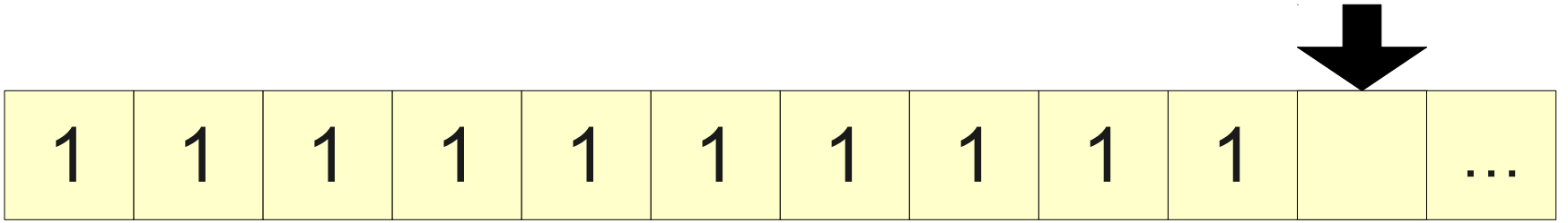
Nondeterministic Algorithms



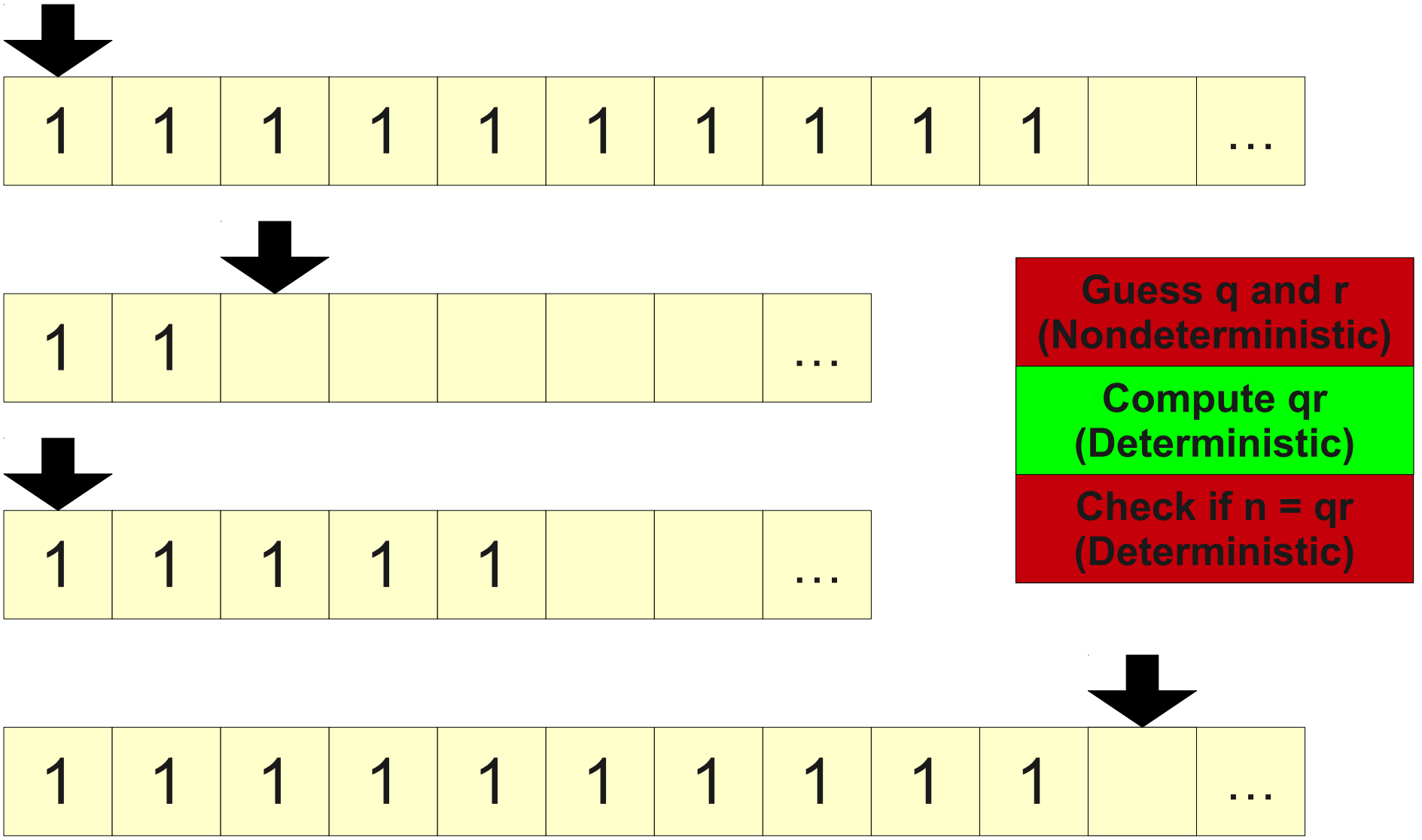




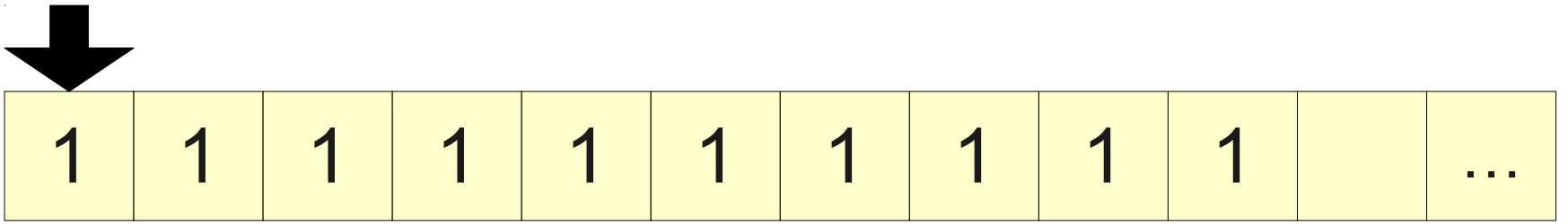


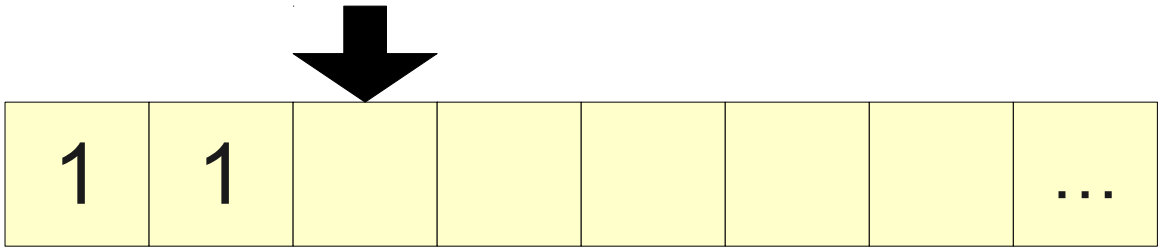


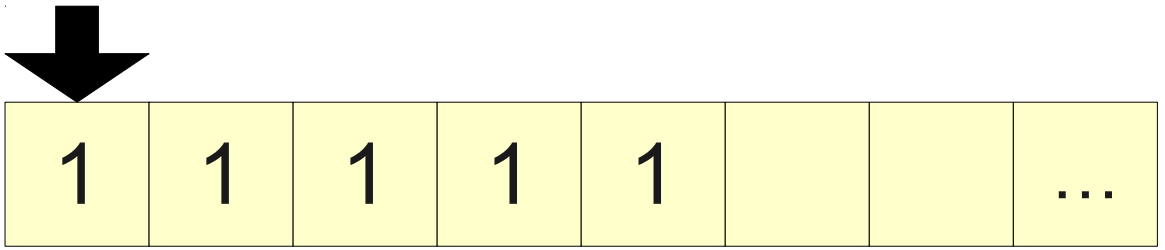
Nondeterministic Algorithms

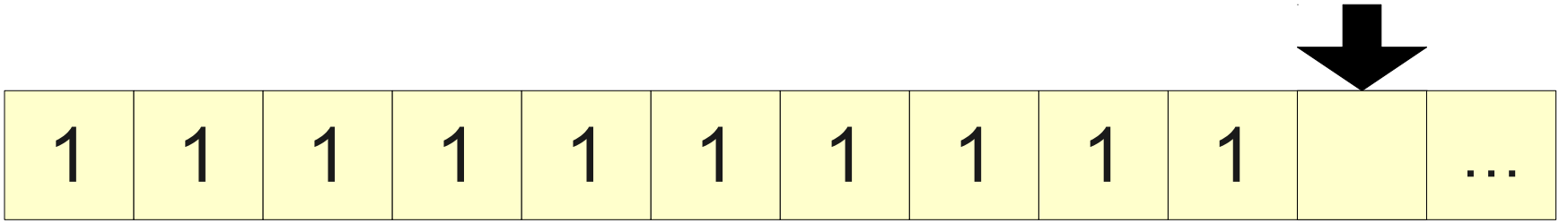


Nondeterministic Algorithms







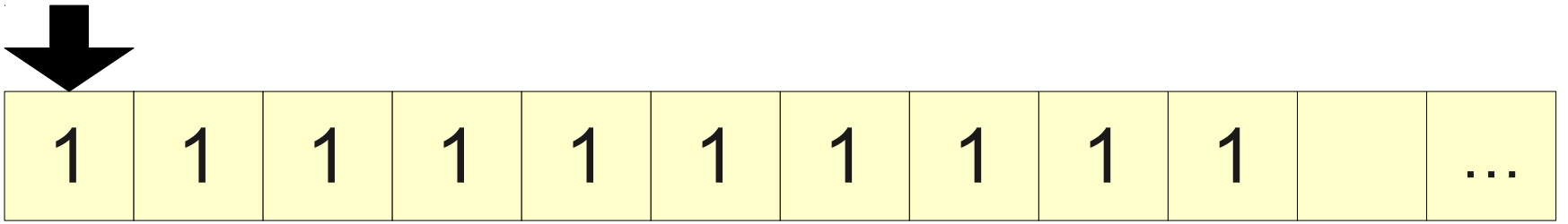


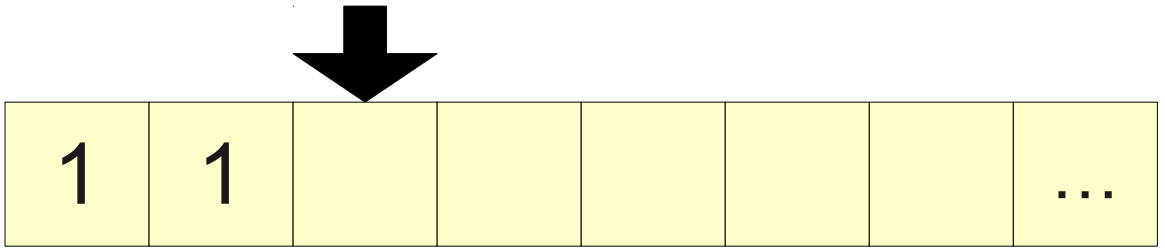
Guess q and r (Nondeterministic)

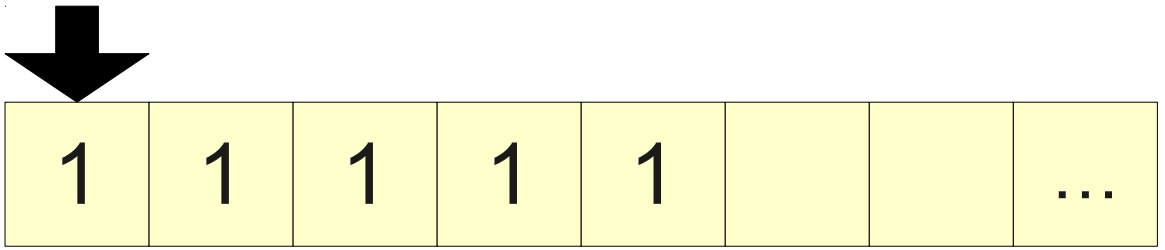
Compute qr (Deterministic)

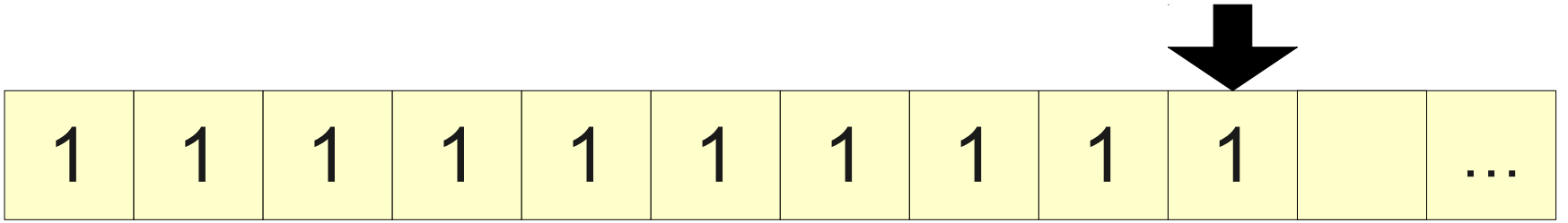
Check if n = qr (Deterministic)

Nondeterministic Algorithms







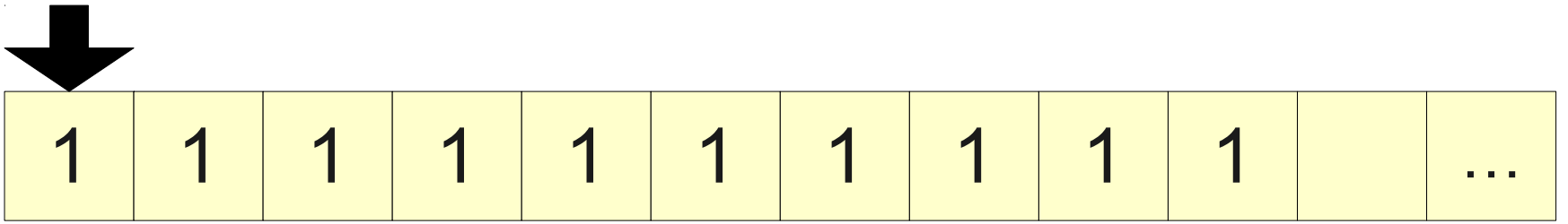


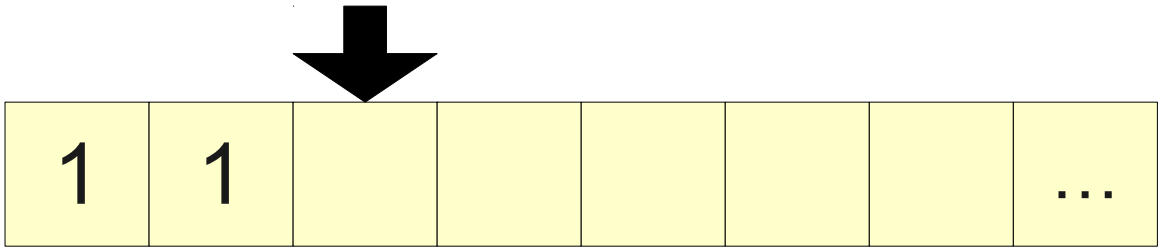
Guess q and r (Nondeterministic)

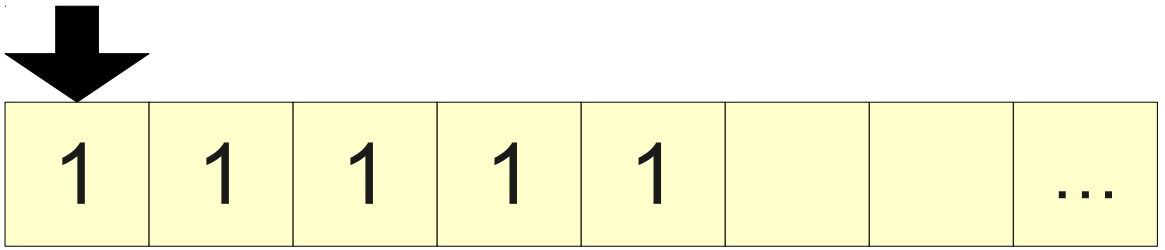
Compute qr (Deterministic)

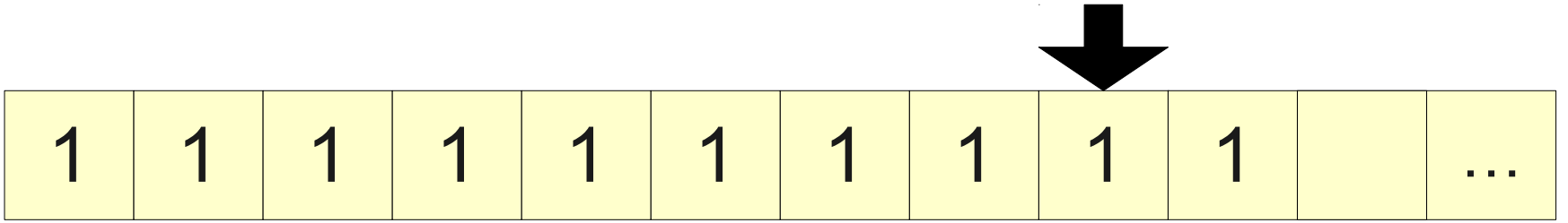
Check if n = qr (Deterministic)

Nondeterministic Algorithms







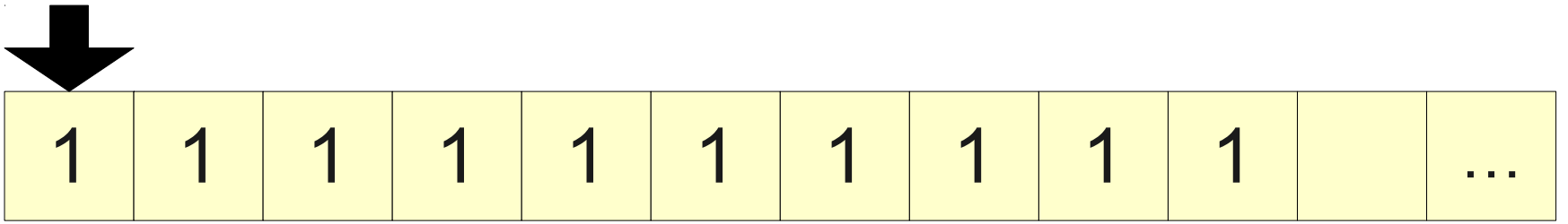


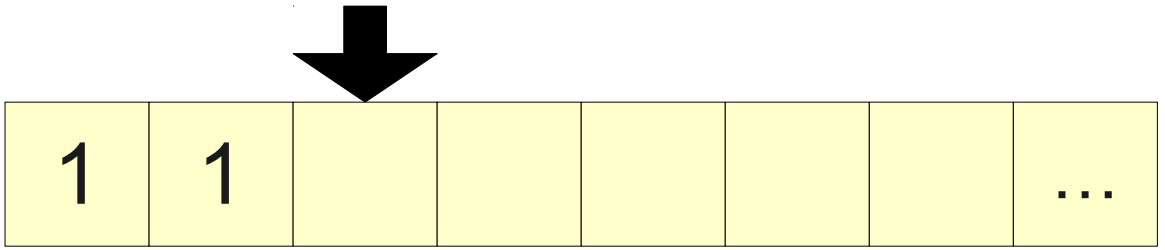
Guess q and r (Nondeterministic)

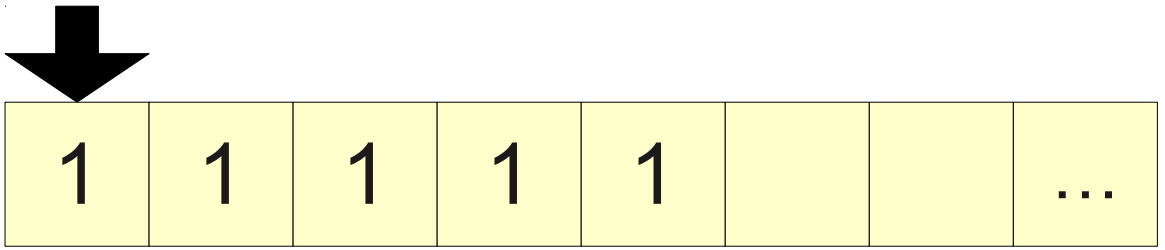
Compute qr (Deterministic)

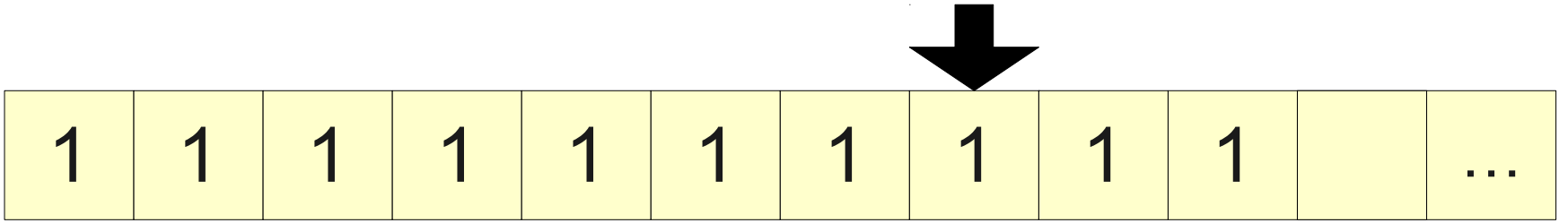
Check if n = qr (Deterministic)

Nondeterministic Algorithms







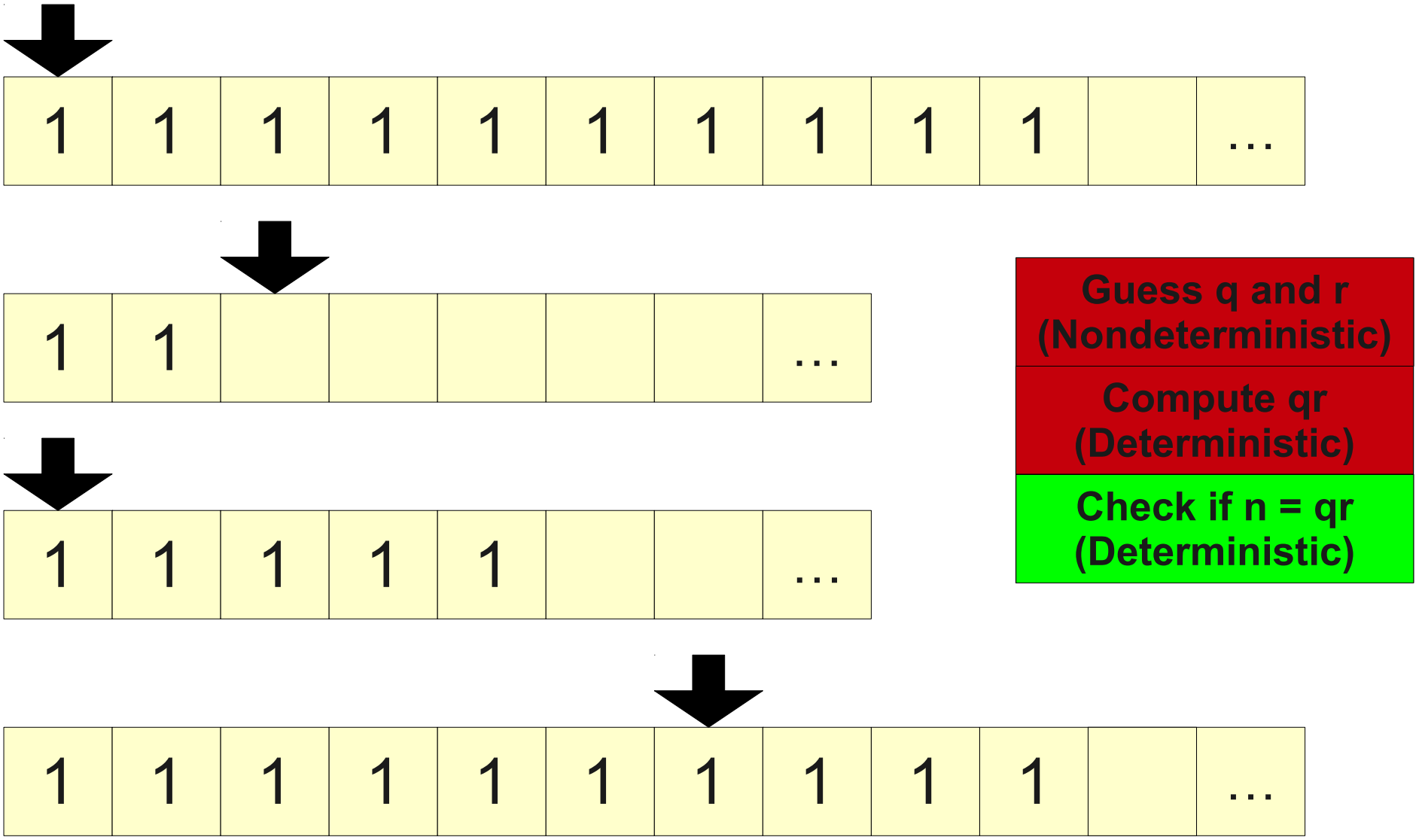


Guess q and r (Nondeterministic)

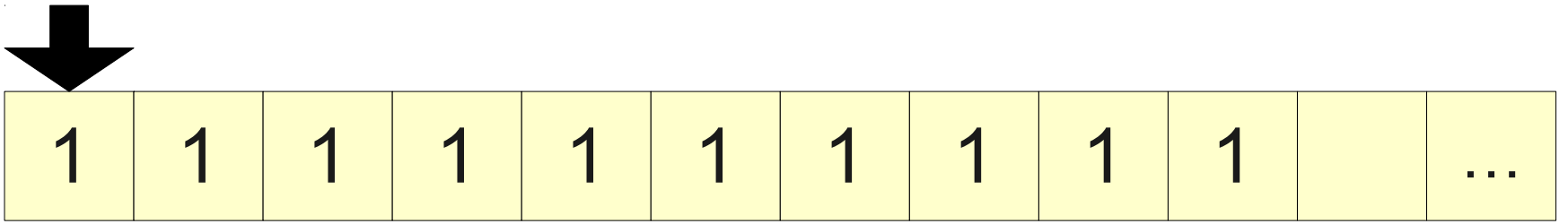
Compute qr (Deterministic)

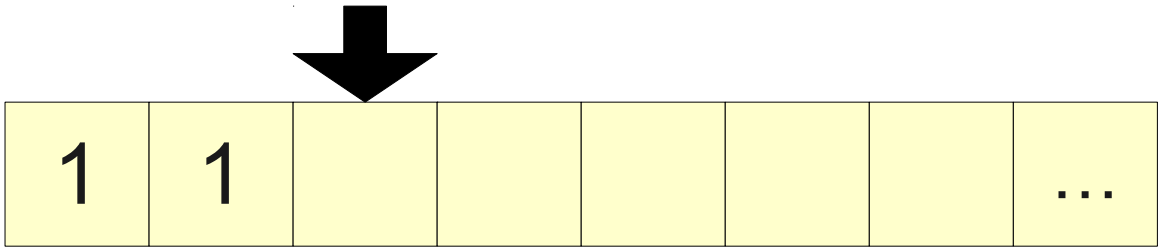
Check if n = qr (Deterministic)

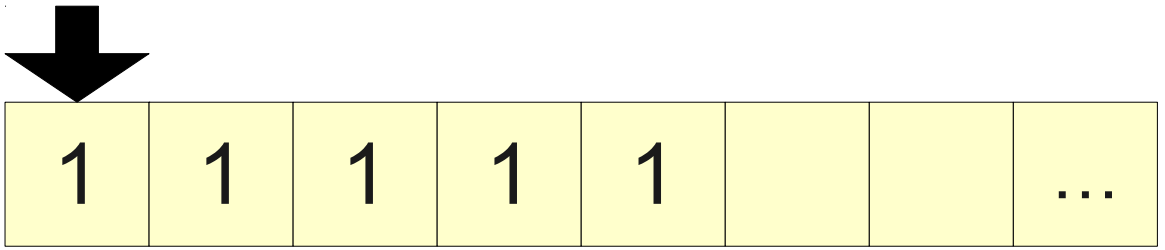
Nondeterministic Algorithms

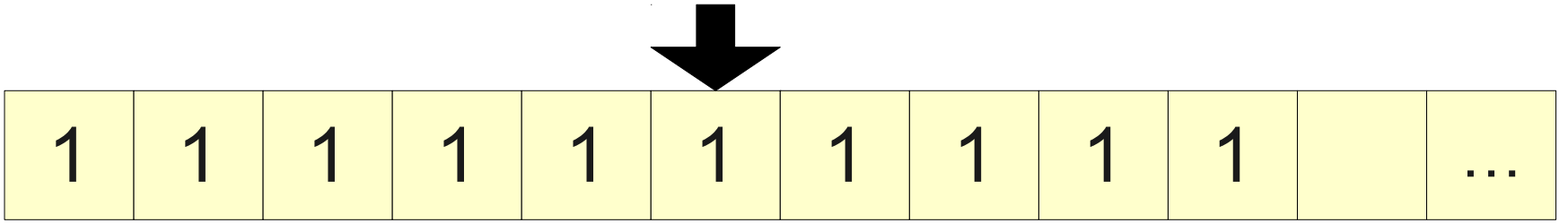


Nondeterministic Algorithms







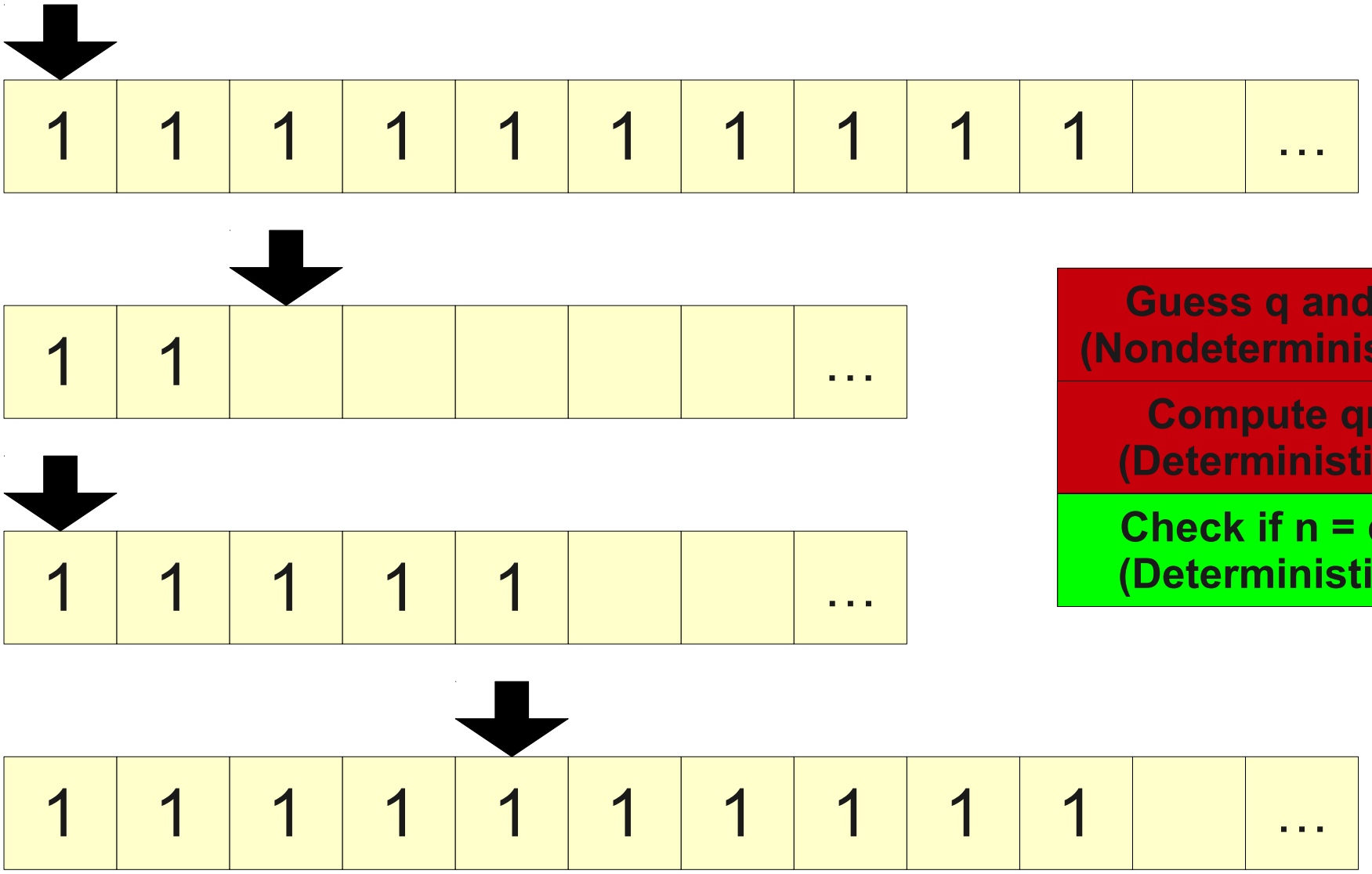


Guess q and r (Nondeterministic)

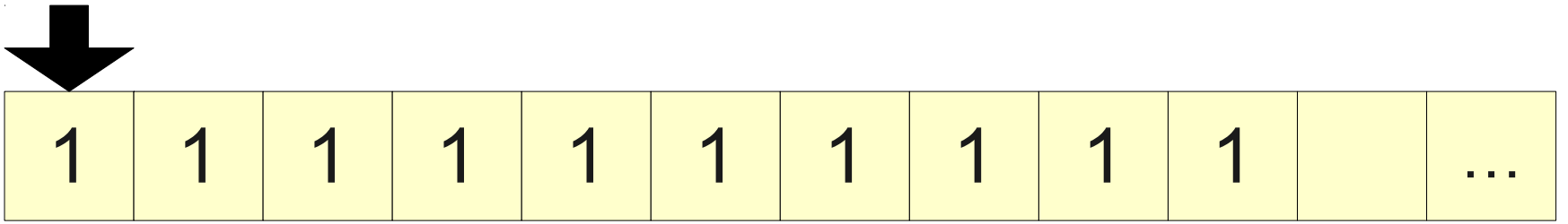
Compute qr (Deterministic)

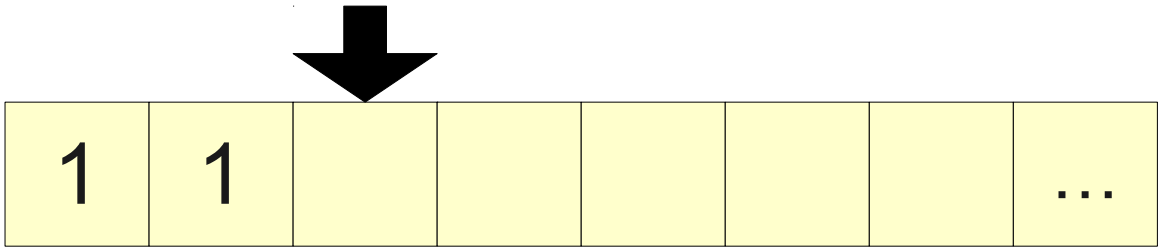
Check if n = qr (Deterministic)

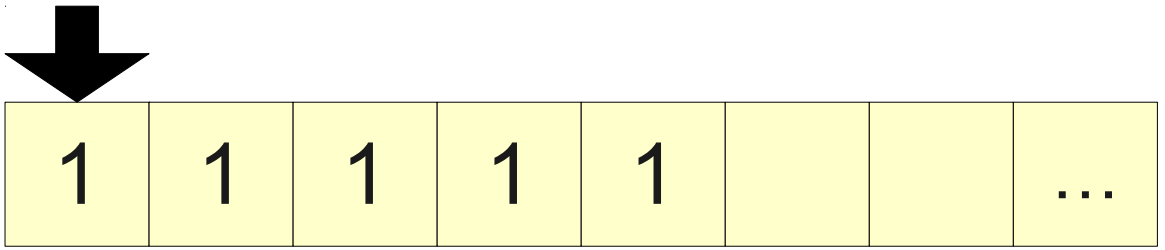
Nondeterministic Algorithms

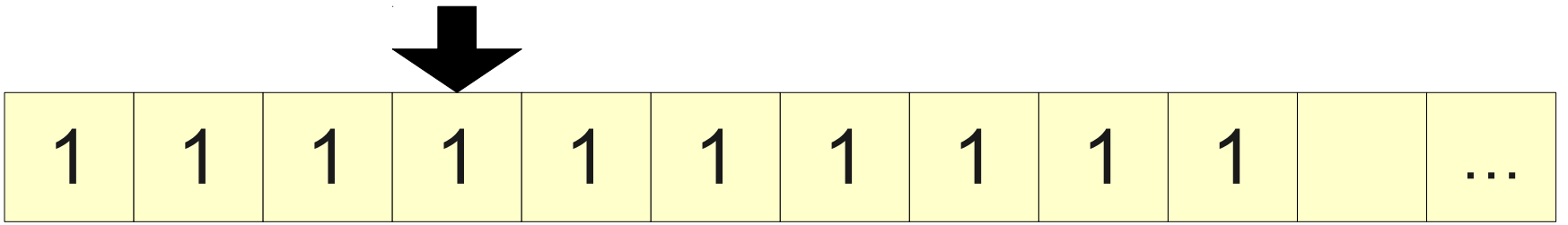


Nondeterministic Algorithms







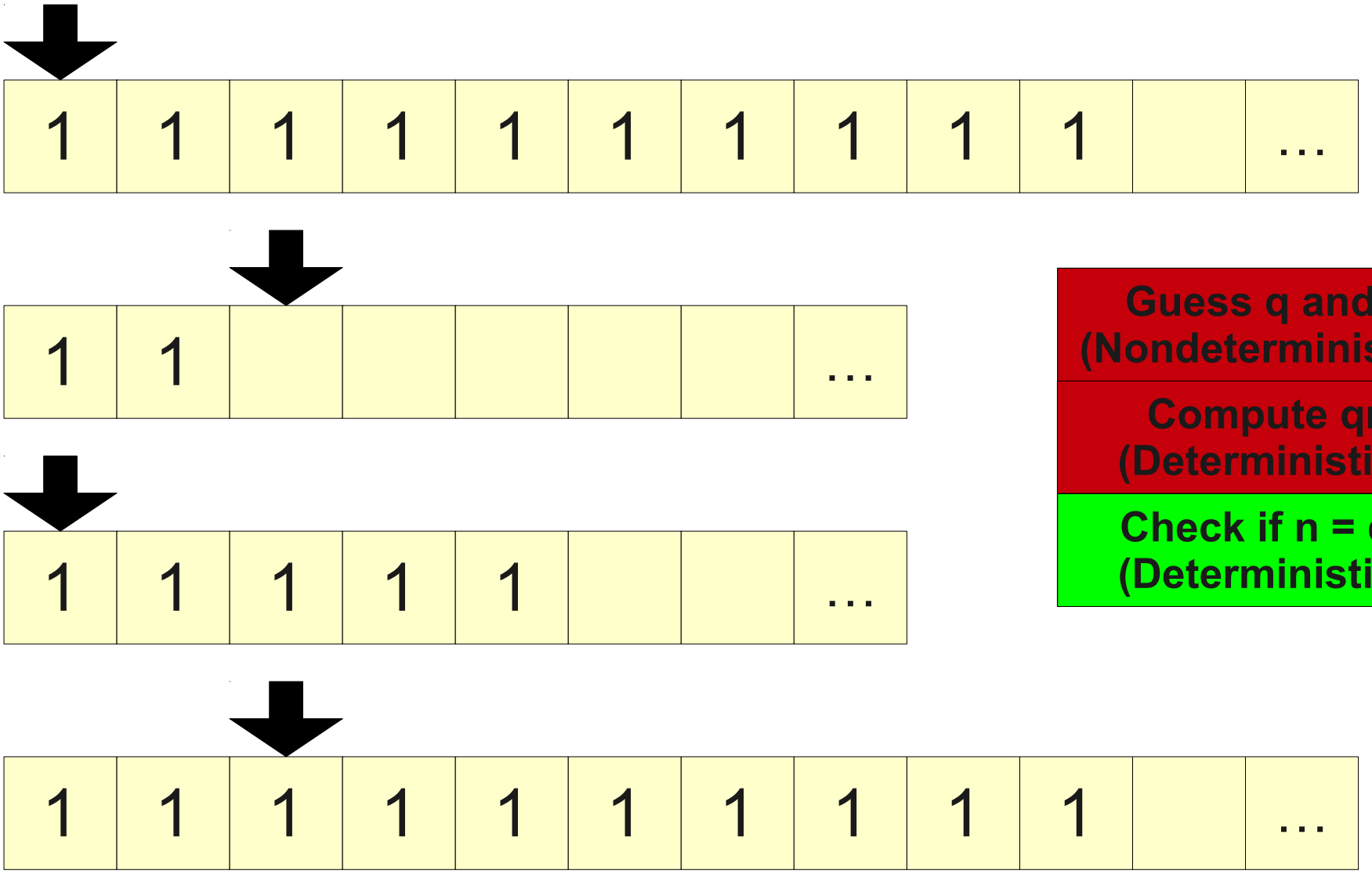


Guess q and r (Nondeterministic)

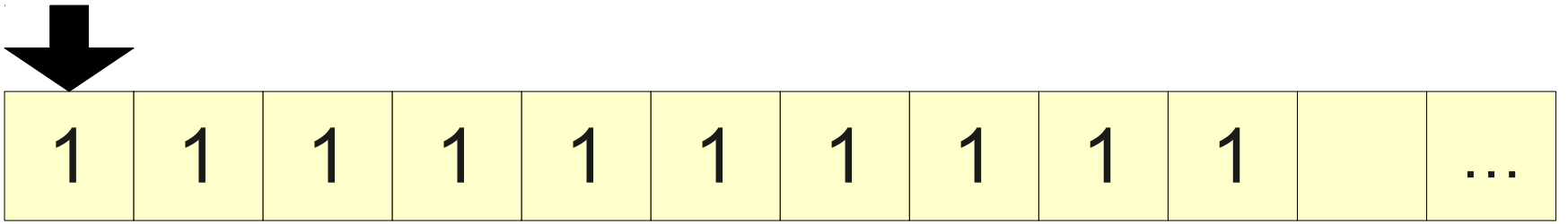
Compute qr (Deterministic)

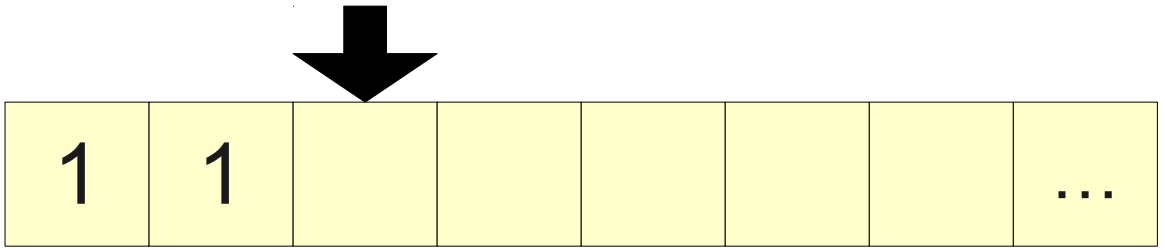
Check if n = qr (Deterministic)

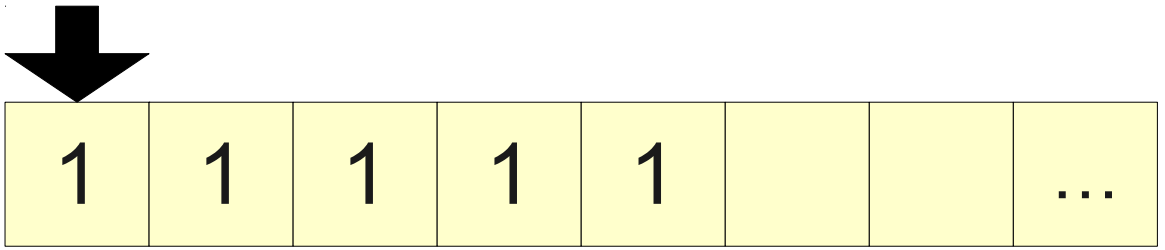
Nondeterministic Algorithms

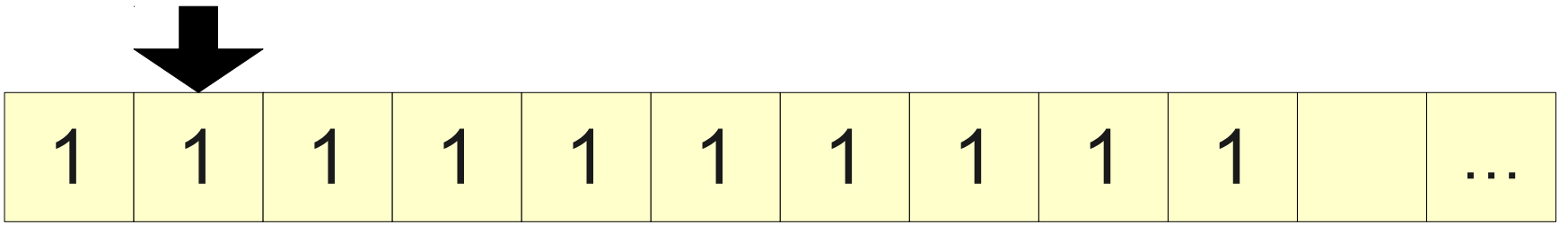


Nondeterministic Algorithms







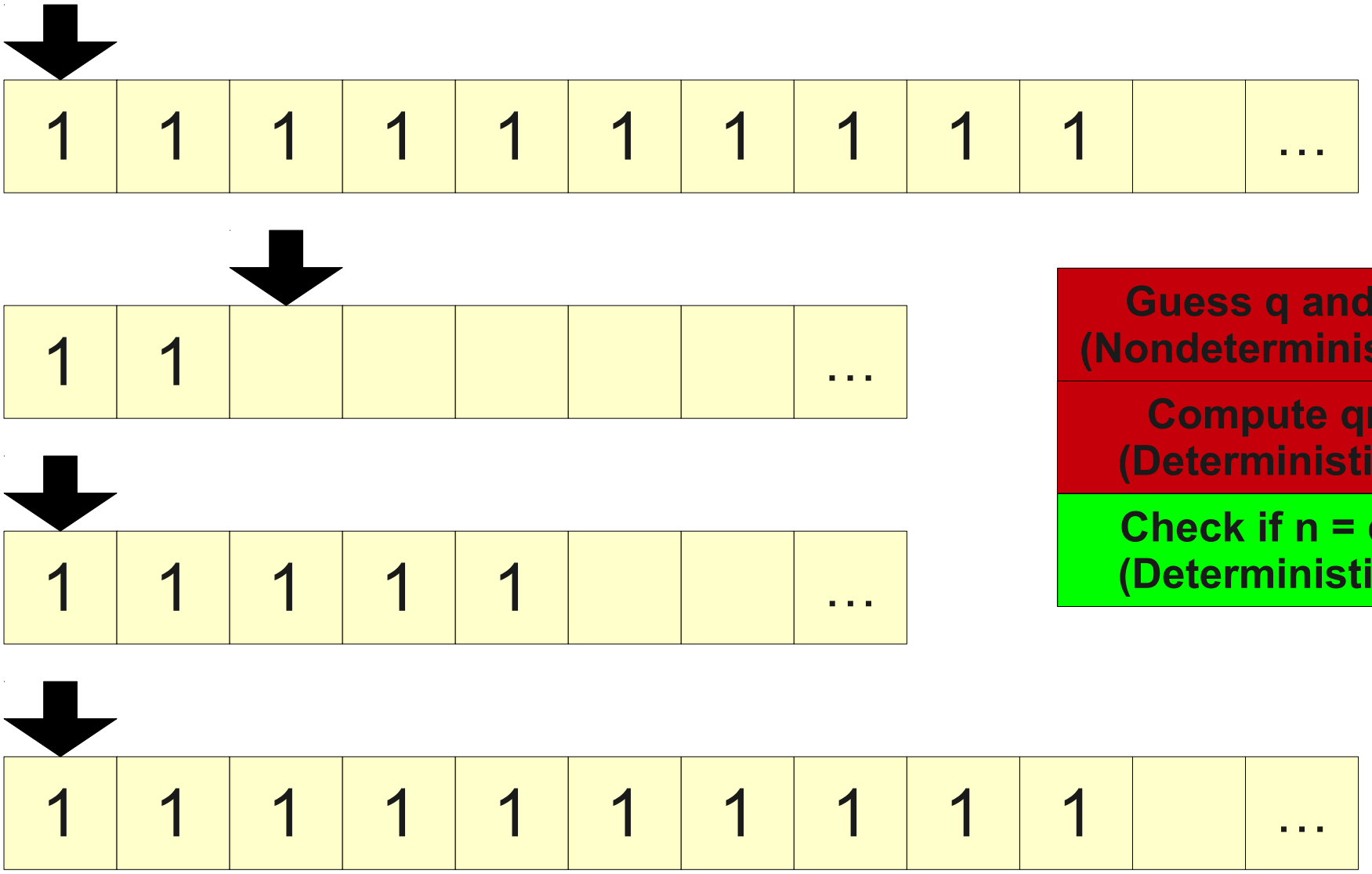


Guess q and r (Nondeterministic)

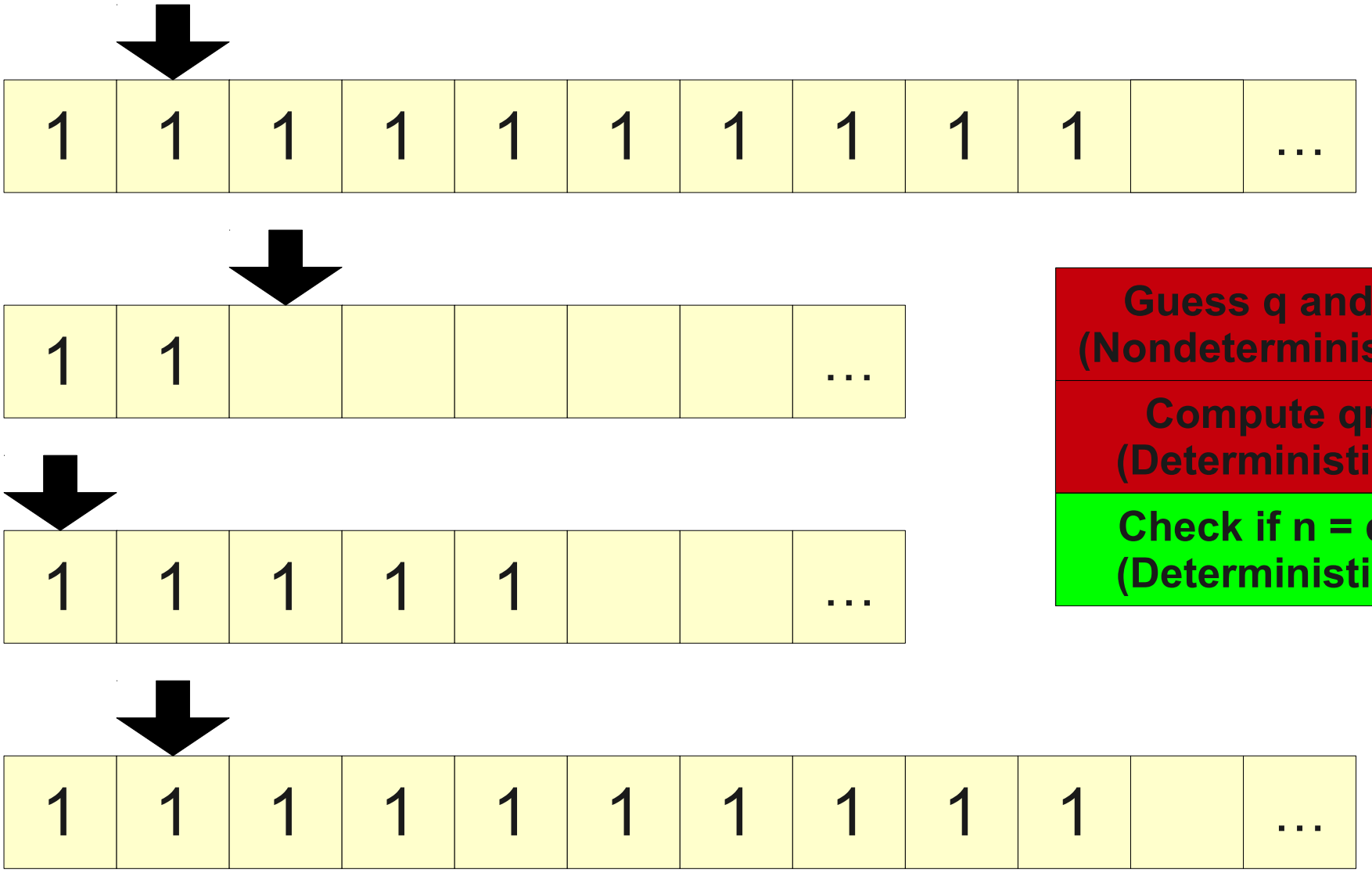
Compute qr (Deterministic)

Check if n = qr (Deterministic)

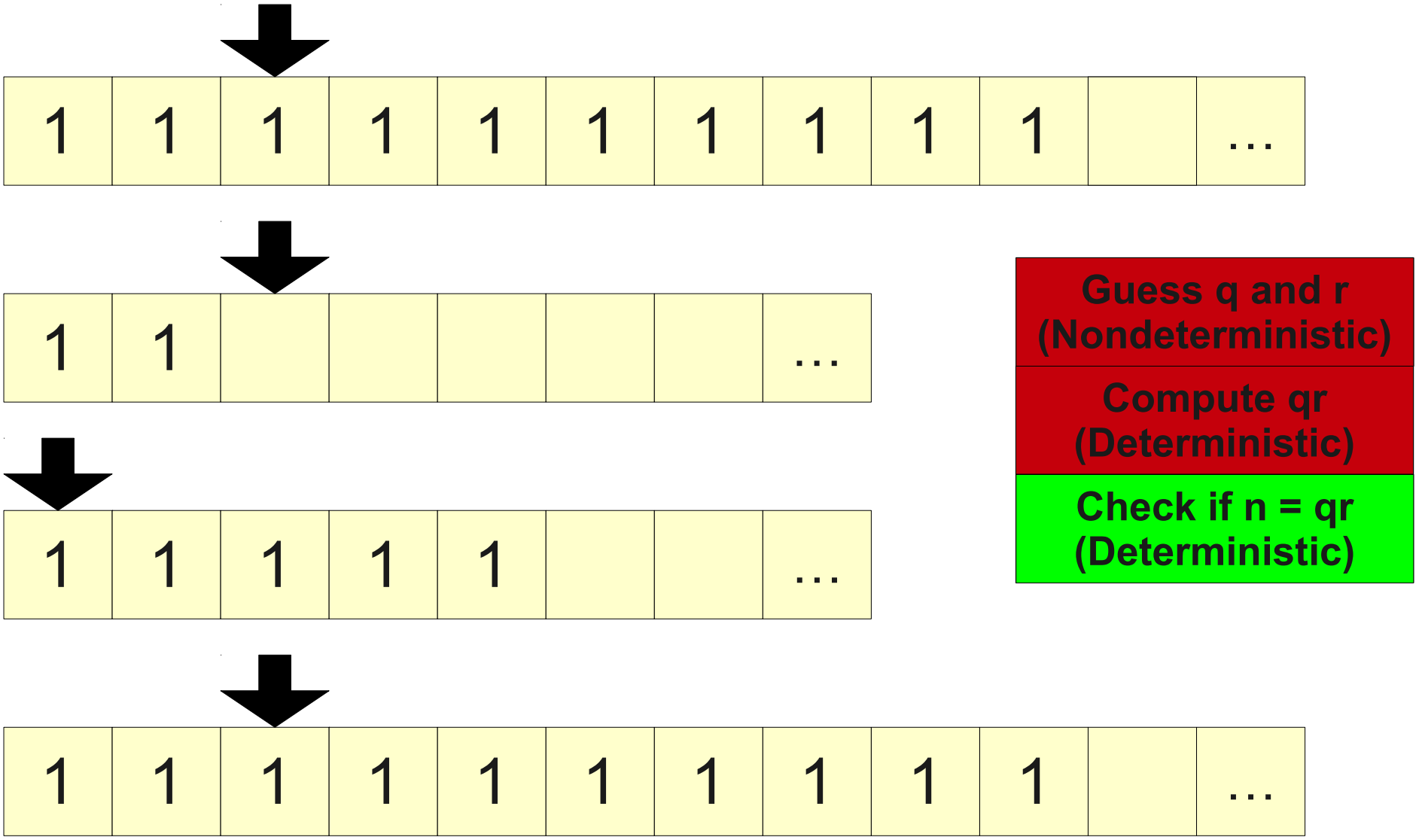
Nondeterministic Algorithms



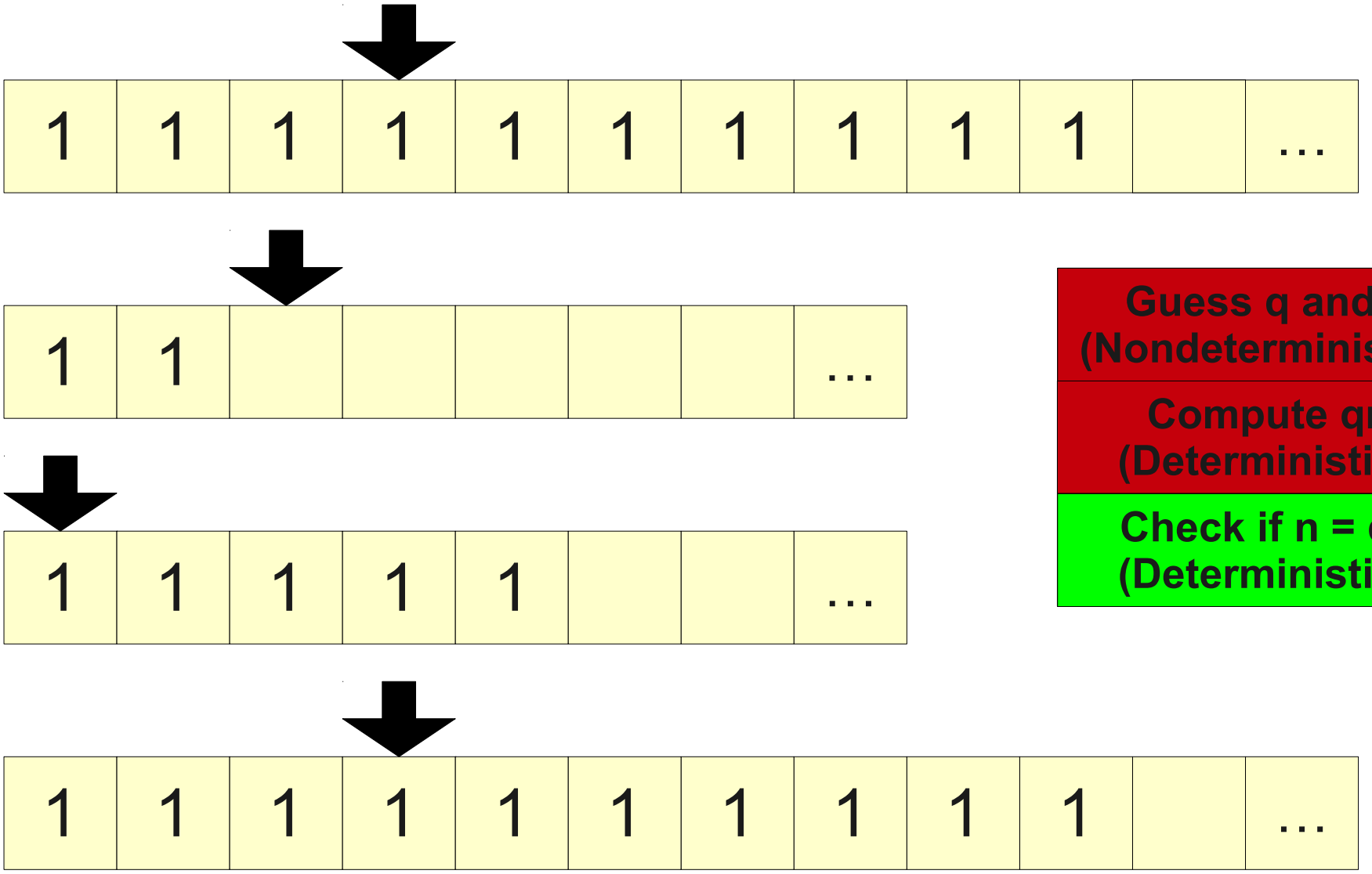
Nondeterministic Algorithms



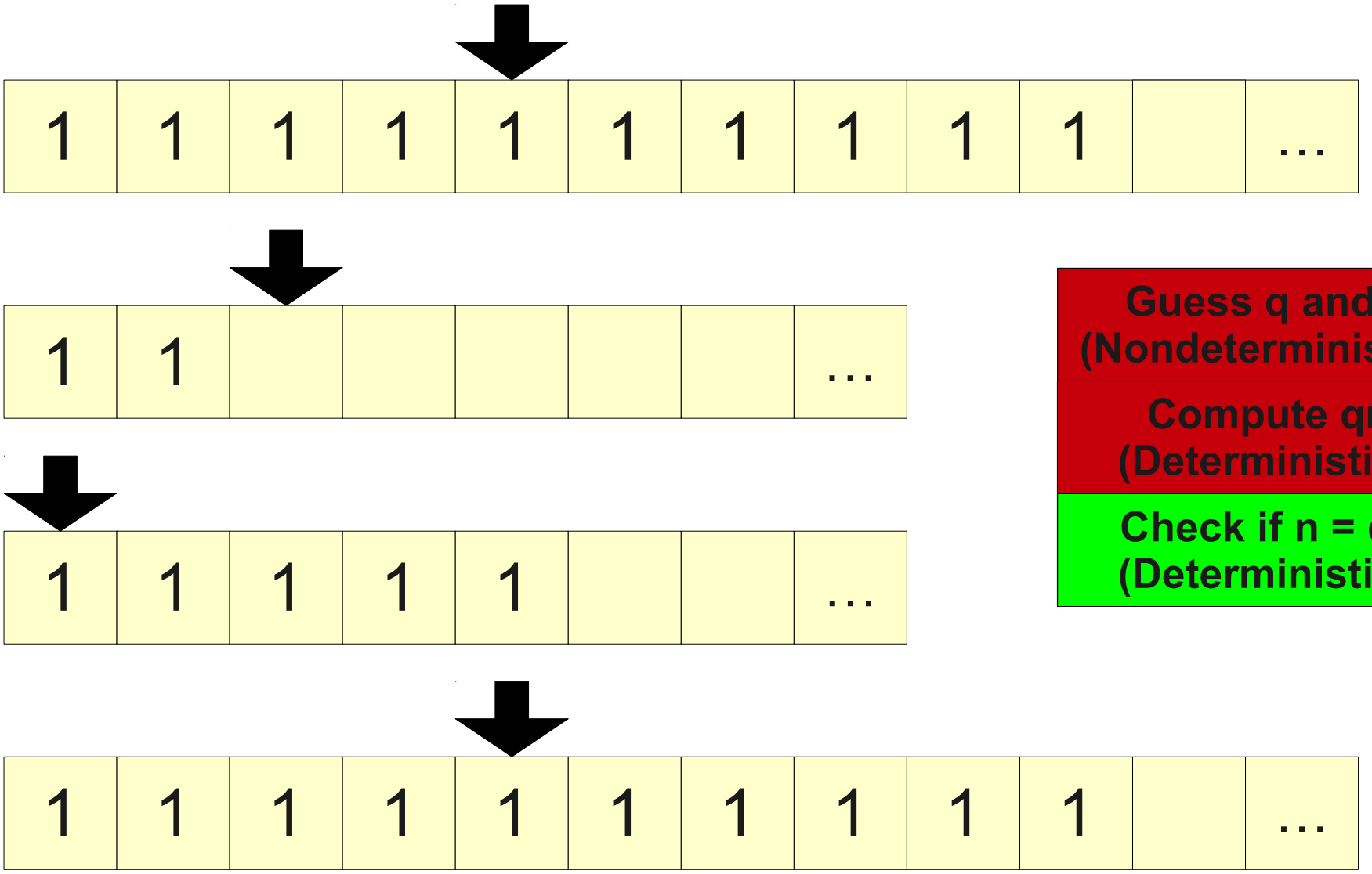
Nondeterministic Algorithms



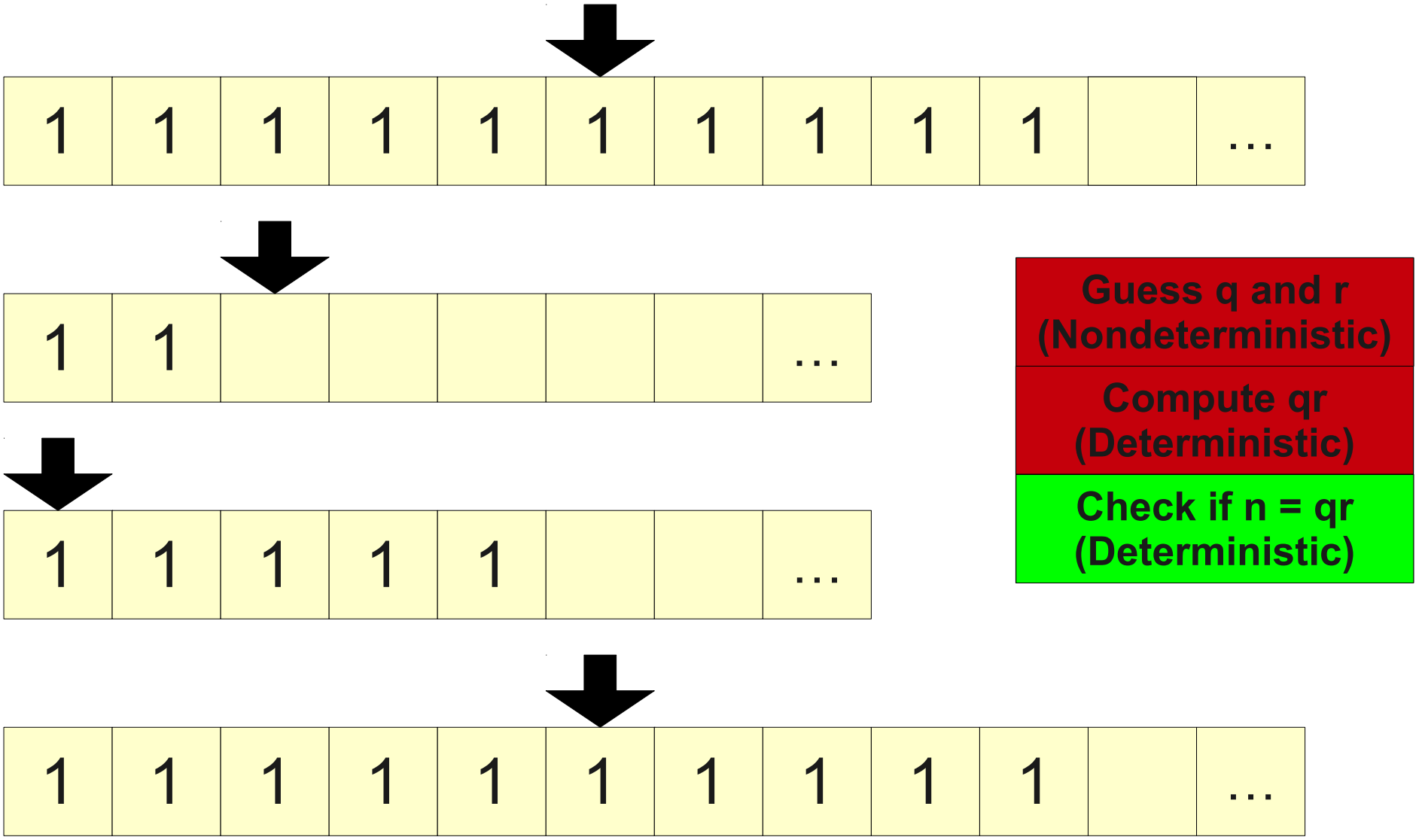
Nondeterministic Algorithms



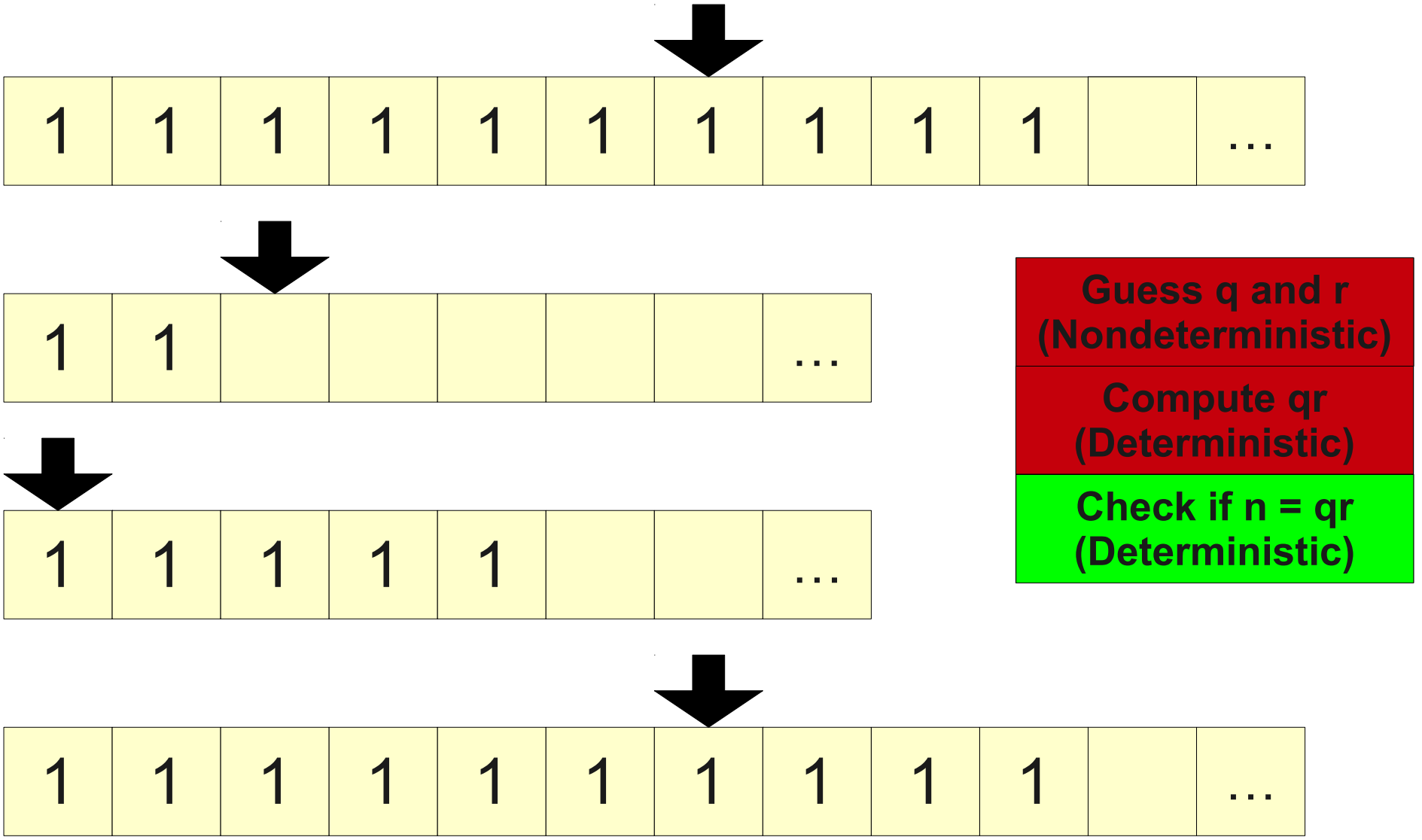
Nondeterministic Algorithms



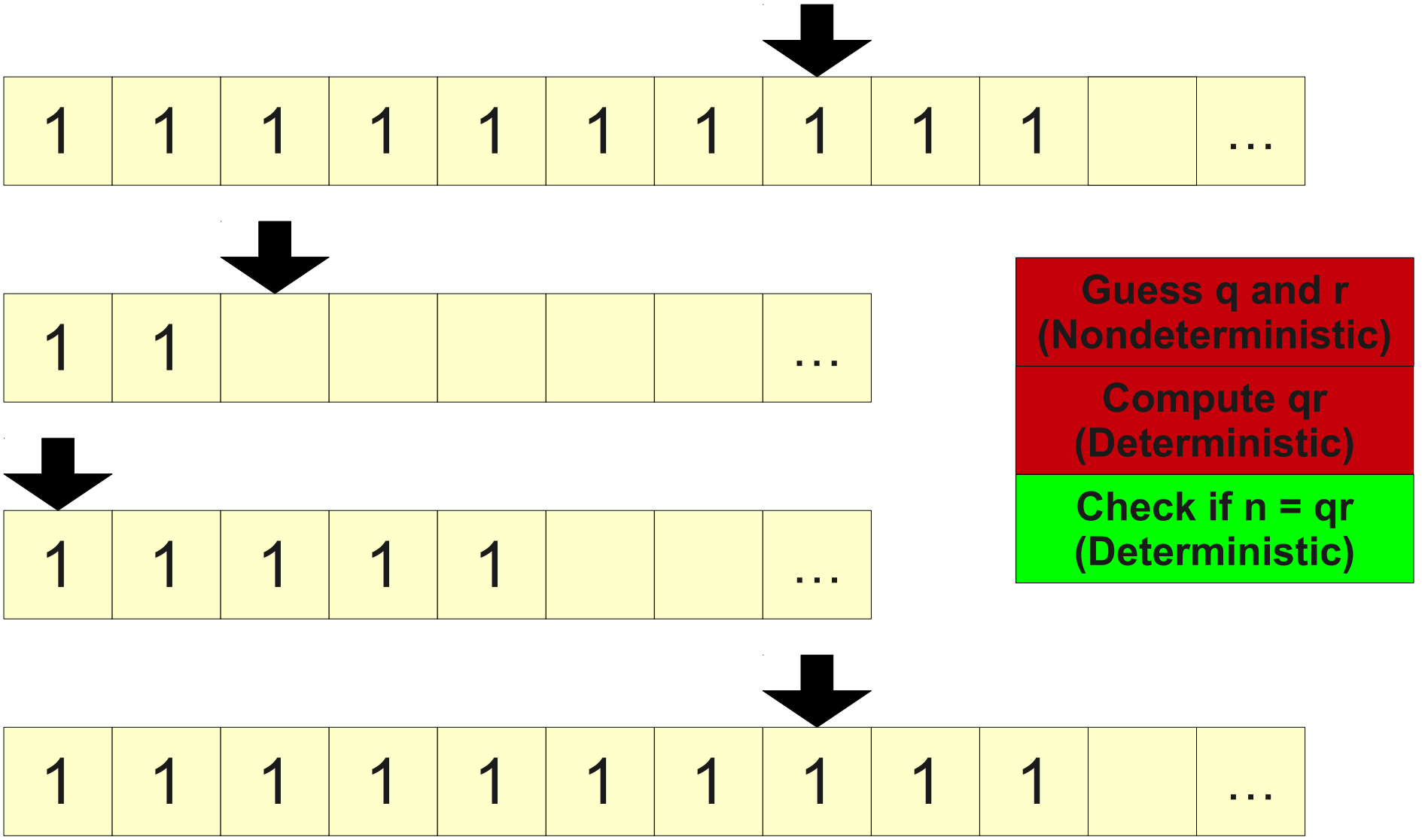
Nondeterministic Algorithms



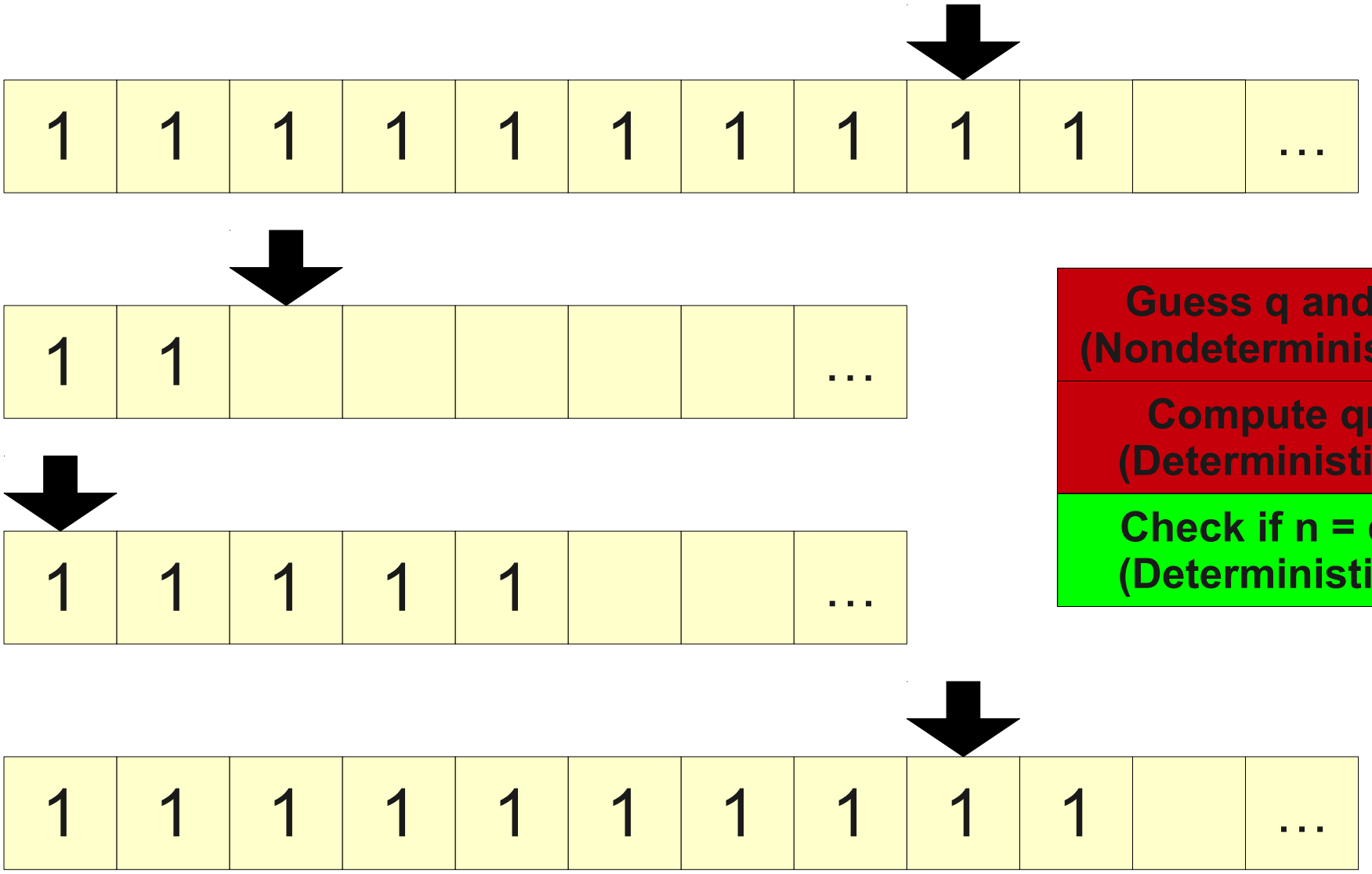
Nondeterministic Algorithms



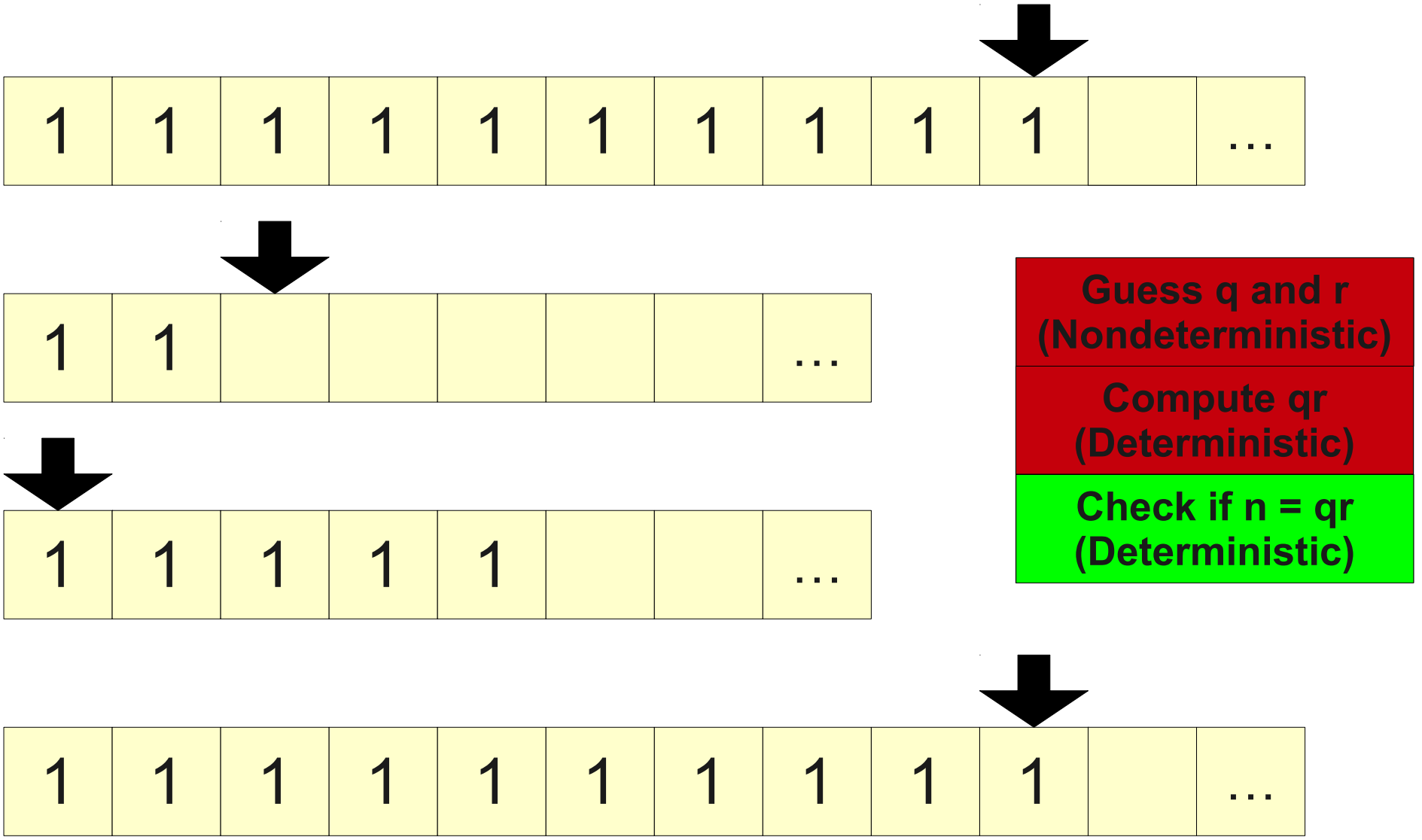
Nondeterministic Algorithms



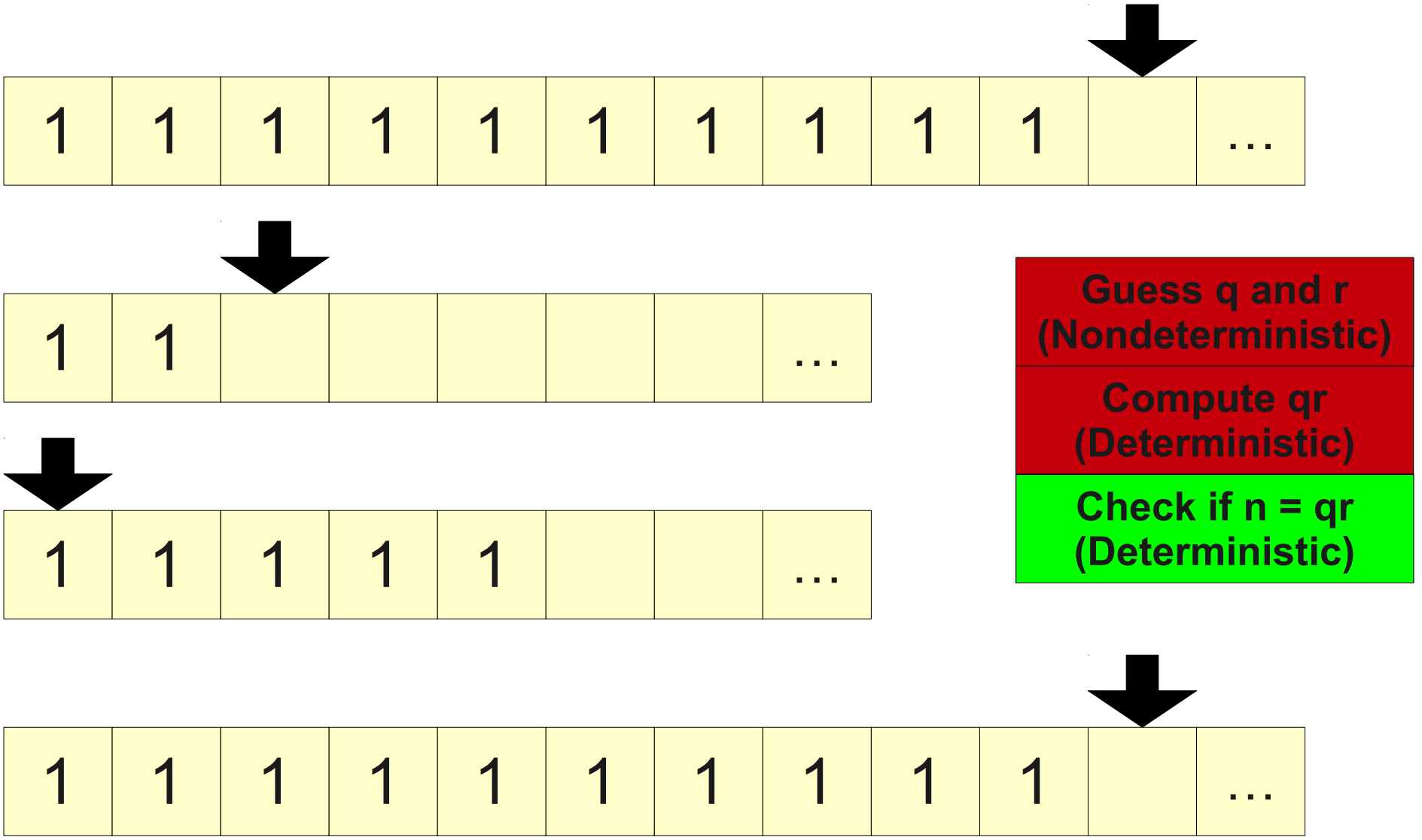
Nondeterministic Algorithms



Nondeterministic Algorithms



Nondeterministic Algorithms



A Nondeterministic Programming Language

- Just as we can have nondeterministic TMs, we can build a nondeterministic variant of **WB**.
- Define **NDWB** to be the language **WB** (the simplest version) with the addition of nondeterministic choice.
- The line of code

$$\texttt{N : \{ } \mathit{line}_1\texttt{, } \mathit{line}_2\texttt{, } \ldots\texttt{, } \mathit{line}_m \texttt{ \}}$$

- Means "nondeterministically choose to execute one of line_1, line_2, …, line_m."

A Simple **NDWB** Program

- Suppose we want to build an NDWB program for the following language over Σ = { `0`, `1` }:

  { *w* | *w* contains `101` as a substring. }

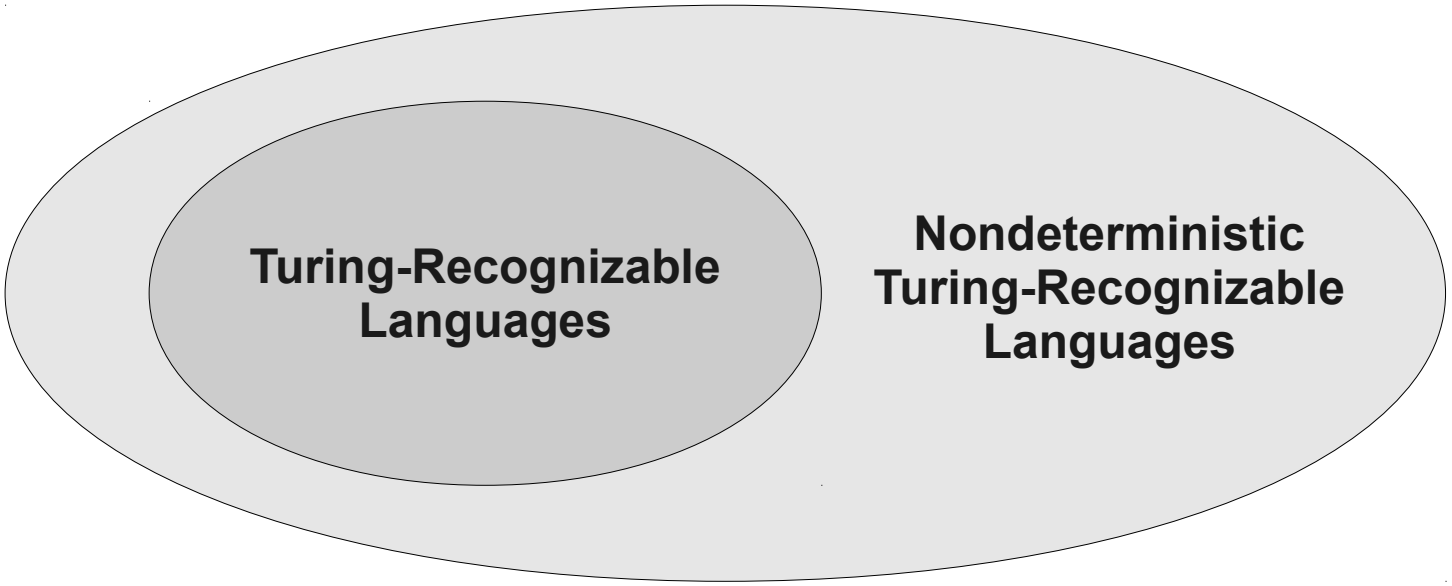
- This is not particularly hard to do deterministically (it's a regular language), but it's interesting to see how we might do this nondeterministically.

A Simple **NDWB** Program

```
// Start
 0: If reading B, go to Rej.
 1: If reading 0, go to Next.
 2: {
    If reading 1, go to Next.
    If reading 1, go to Match.
 }

// Next:
 3: Move right.
 4: Go to Start.

// Match
 8: Move right.
 9: If reading 0, go to 11
10: Reject.
11: Move right.
12: If reading 1, go to 14
13: Go to Rej.
14: Accept.

// Rej:
15: Reject.
```

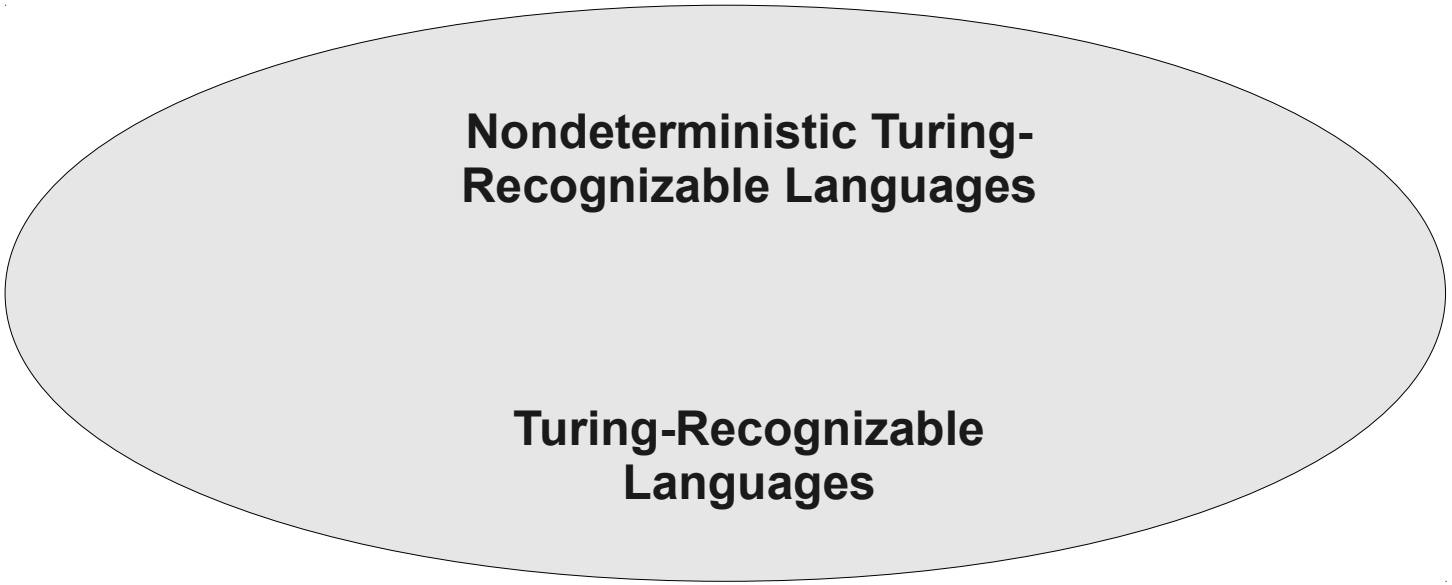
The Power of Nondeterminism

- In the case of finite automata, we saw that the DFA and NFA had equivalent power.
- In the case of pushdown automata, we saw that the DPDA was strictly weaker than the NPDA.
- What is the relative power of TMs and NTMs?



The Power of Nondeterminism

- In the case of finite automata, we saw that the DFA and NFA had equivalent power.
- In the case of pushdown automata, we saw that the DPDA was strictly weaker than the NPDA.
- What is the relative power of TMs and NTMs?

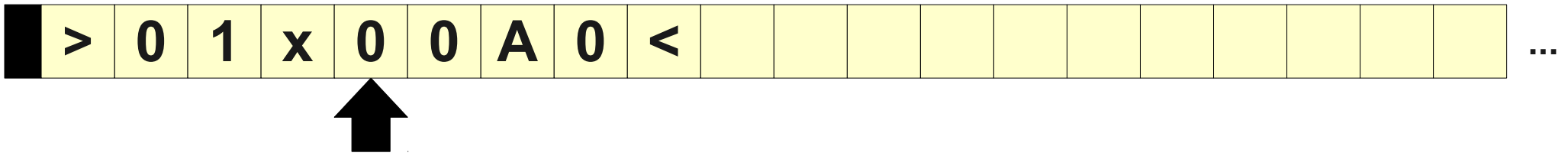


A Rather Remarkable Theorem

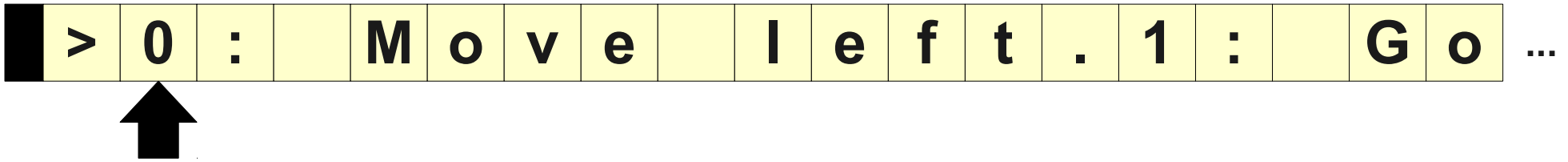
- **Theorem:** A language is recursively enumerable iff it is accepted by a nondeterministic Turing machine.
- **Theorem:** A language is recursively enumerable iff there is an **NDWB** program for it.
- How is this possible?

Sketch of the Universal **WB** Program

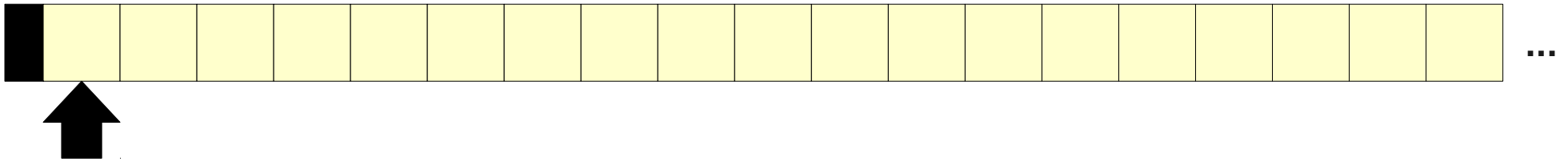
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

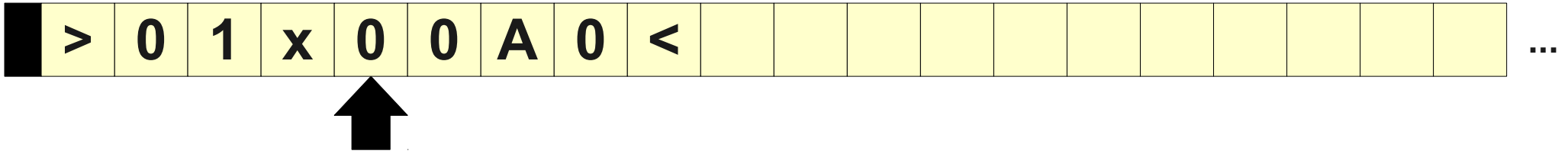


Variables for intermediate storage.

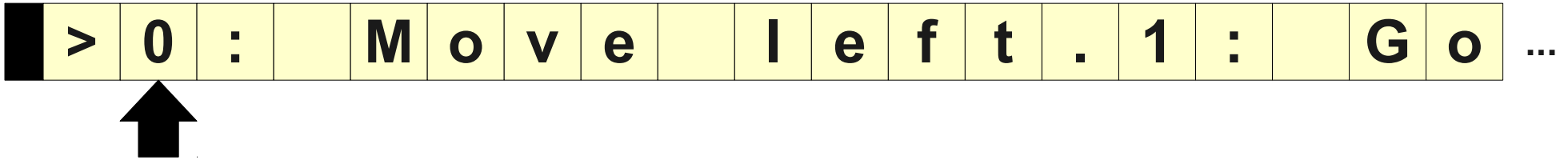
Instr **Letter**

Sketch of the Universal **WB** Program

Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

Variables for intermediate storage.

Instr **Letter**

Instantaneous Descriptions

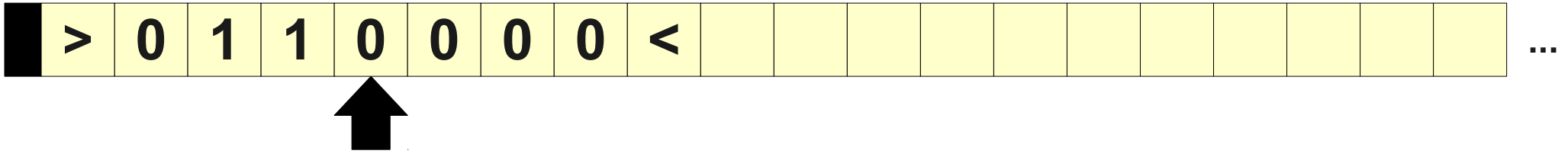
- An **instantaneous description** (or **ID**) of the execution of a program (TM, **WB** program, etc.) is a string encoding of a snapshot of that program at one instant in time.
- For Turing machines, it contains
  - The contents of the tape,
  - Where the tape head is, and
  - What state the machine is in.
- For **WB** programs, it contains
  - The contents of the tape,
  - Where the tape head is, and
  - What line of code is next to be executed.

IDs and Universal Machines

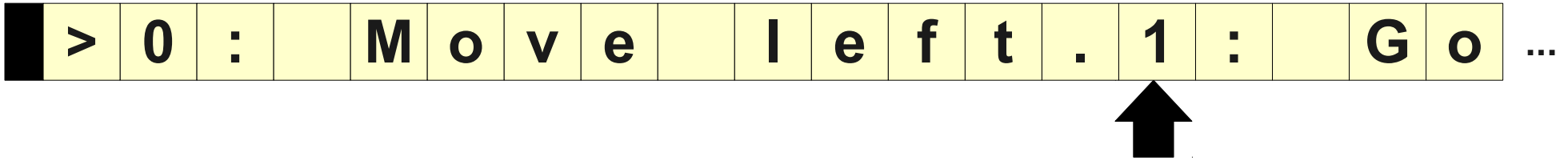
- There is a close connection between an ID and universal machines.
- The universal machines U_{TM} and U_{WB} work by repeatedly reading the ID of the machine being simulated, then executing one step of that machine.

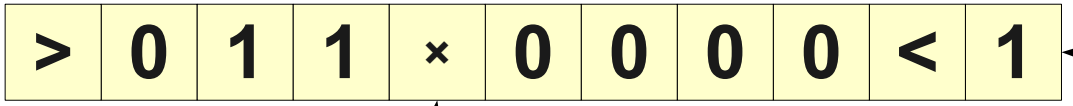
An ID for **WB** Programs

Simulated tape of the program being executed.



Program tape holding the program being executed.





This means "the tape head is under the next symbol."

We write the line number at the end of the ID.

Manipulating IDs

- Because IDs are strings (just like machine or program encodings), we can perform all sorts of operations on them:
  - Copy them to other tapes for later use.
  - Inspect them to see the state of the machine at any instant in time.
  - Transform them to represent making changes to the program as it is running.

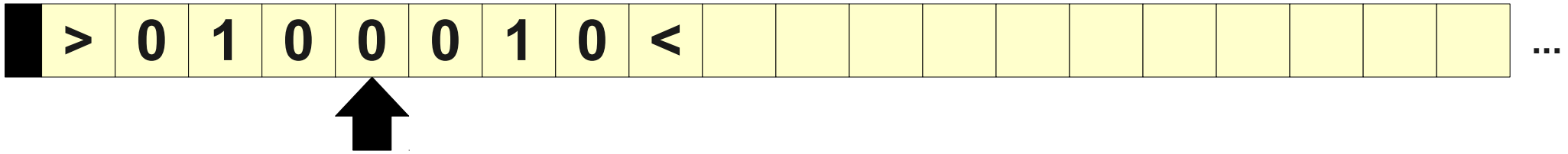
The Key Idea

- We can store IDs of all of the different branches of the NTM and execute them in a breadth-first search.
- This does a breadth-first walk of the “tree computation” interpretation of nondeterministic computation.
- If there is an accepting computation, eventually we will find it (though it might take an enormously long time to do so!)

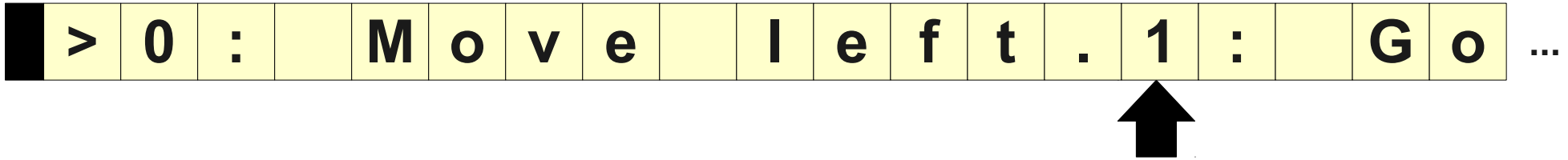
Simulating an NTM

- Given an NTM *M*, we can simulate it with a deterministic TM as follows:
- "On input *w*:
  - Put the initial ID of *M* running on *w* into a separate tape.
  - While that tape contains IDs:
    - Copy the first ID to a work tape.
    - If this ID is in an accepting state, accept.
    - If the ID is not in a rejecting state:
      - For all possible next steps, use the universal TM to simulate the NTM making that choice, then append the resulting ID to the other tape.
  - Reject."

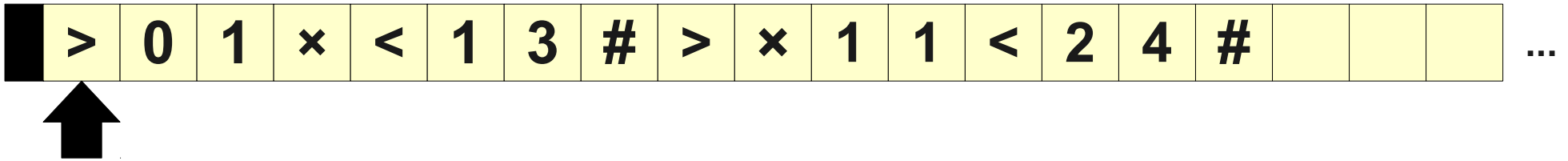
Simulated tape of the program being executed.



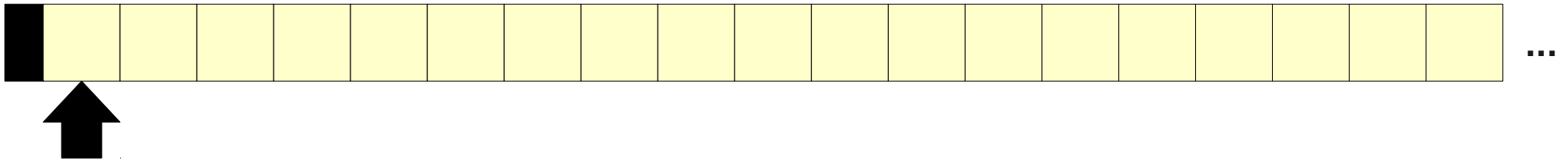
Program tape holding the program being executed.



Stored IDs



Scratch tape for intermediate computation.



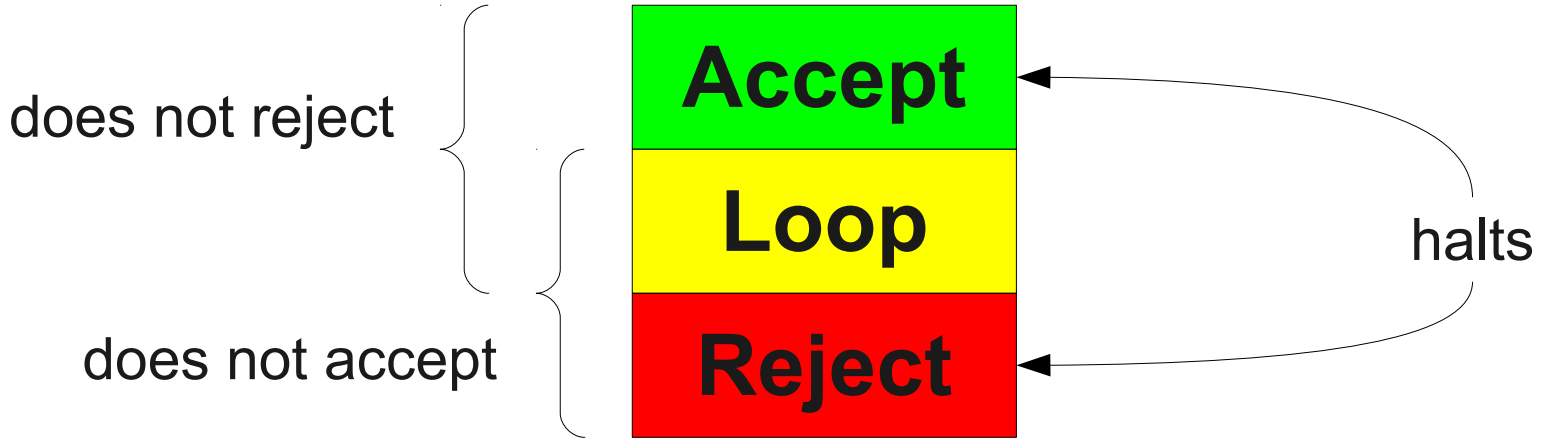
Variables for intermediate storage.

Instr **Letter**

Decidability and Recognizability

Some Important Terminology

- A TM/**WB** program **accepts** a string *w* if it enters an accept state or executes the **`Accept`** command.
- A TM/**WB** program **rejects** a string *w* if it enters a reject state or executes the **`Reject`** command.
- A TM/**WB** program **loops infinitely** on a string *w* if neither of these happens.
- A TM/**WB** program **does not accept** a string *w* if it either rejects *w* or loops infinitely on *w*.
- A TM/**WB** program **does not reject** a string *w* if it either accepts *w* or loops infinitely on *w*.
- A TM/**WB** program **halts** if it accepts or rejects.



Recall: Language of a TM

- The language of a Turing machine M, denoted $\mathcal{L}(M)$, is the set of all strings that M accepts:

$$\mathcal{L}(M) = \{ w \mid M \text{ accepts } w \}$$

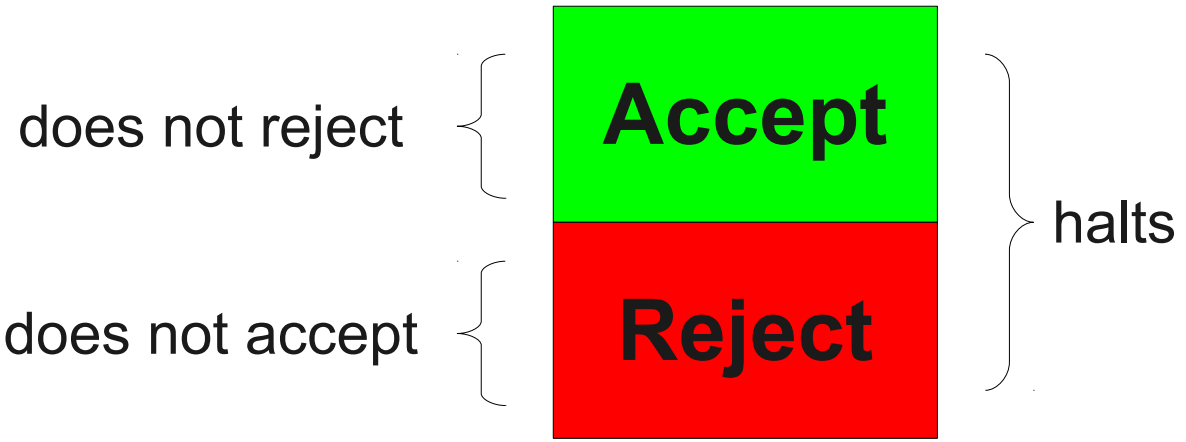
- For any $w \in \mathcal{L}(M)$, M will accept w.
- For any $w \notin \mathcal{L}(M)$, M will not accept w.
  - It might loop forever, or it might explicitly reject.
- A language is called **recognizable** or **recursively enumerable** if it is the language of some TM.

Why “Recognizable?”

- Given TM *M* with language $\mathcal{L}(M)$, running *M* on a string *w* will not necessarily tell you whether $w \in \mathcal{L}(M)$.
- If the machine is running, you can't tell whether
  - It is eventually going to halt, but just needs more time, and
  - It is never going to halt.
- However, if you know for a fact that $w \in \mathcal{L}(M)$, then the machine can confirm this.
- The machine can't *decide* whether or not $w \in \mathcal{L}(M)$, but it can *recognize* strings that are in the language.
- We sometimes call a TM for a language *L* a **recognizer** for *L*.

Deciders

- Some Turing machines always halt; they never go into an infinite loop.
- Turing machines of this sort are called **deciders**.
- For deciders, accepting is the same as not rejecting and rejecting is the same as not accepting.



Decidable Languages

- A language *L* is called **decidable** iff there is a decider *M* such that $\mathcal{L}(M) = L$.
  - These languages are also sometimes called **recursive**.
- Given a decider *M*, you *can* learn whether or not a string $w \in \mathcal{L}(M)$.
  - Run *M* on *w*.
  - Although it make take a staggeringly long time, *M* will eventually accept or reject *w*.

Decidability vs. Recognizability

- All deciders are Turing machines, but not all Turing machines are deciders.
- As a result, we know that $R \subseteq RE$.
  - R is the set of all recursive (decidable) languages; RE is the set of all recursively enumerable (recognizable) languages.
- **Enormously important question**: **Is $R = RE$?**
- This is an enormously important theoretical question. We'll explore the answer next time...

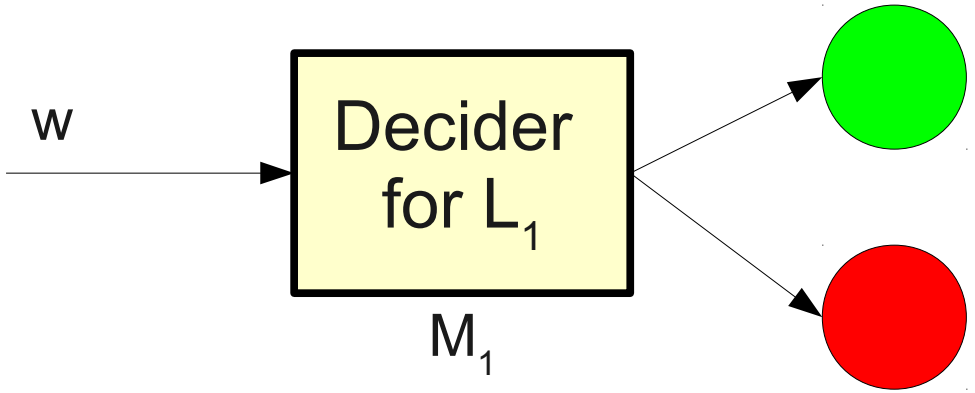
Properties of R and RE Languages

Closure under Intersection

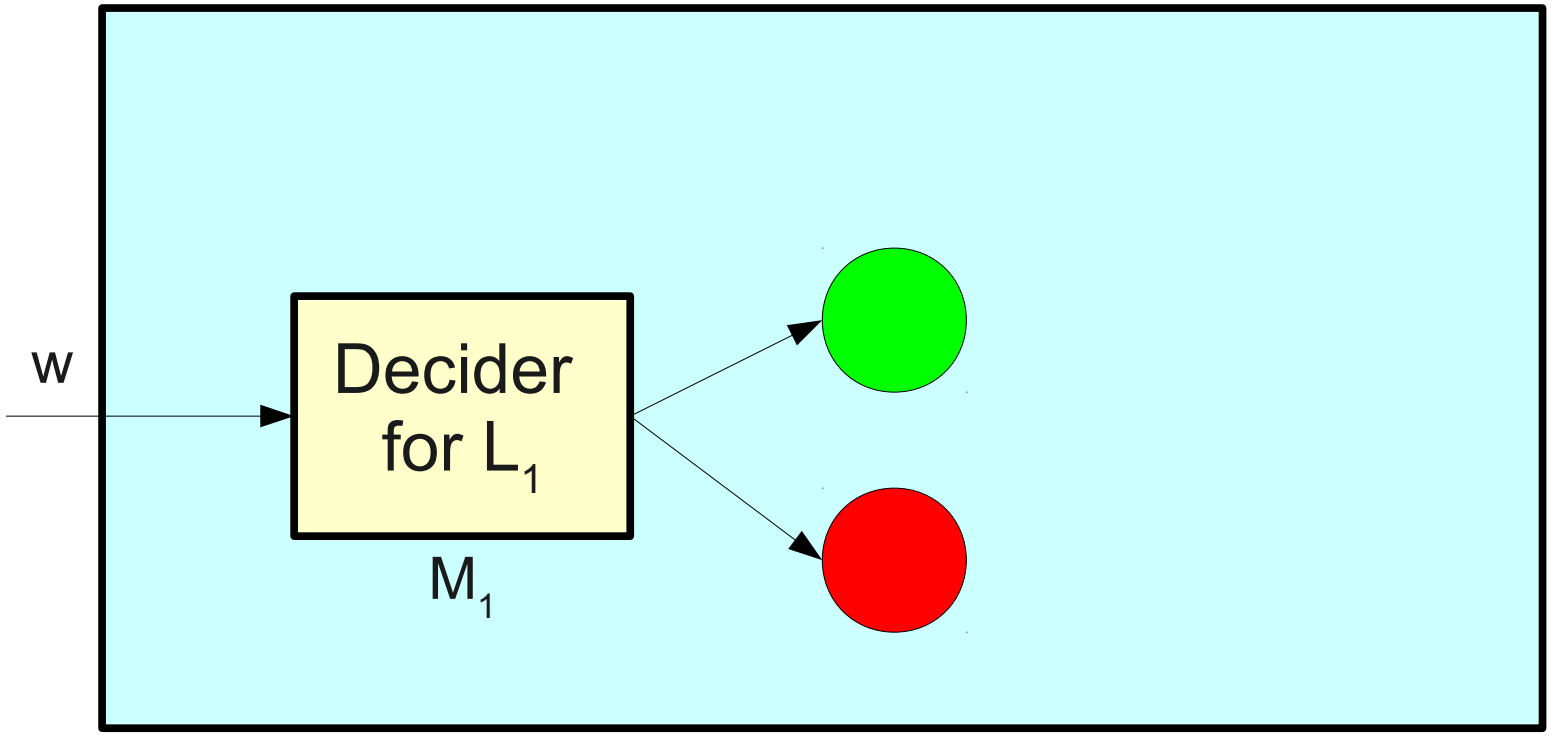
- The R and RE languages are closed intersection.
- If $L_1 \in R$ and $L_2 \in R$, then $L_1 \cap L_2 \in$ R.
- If $L_1 \in RE$ and $L_2 \in RE$, then $L_1 \cap L_2 \in RE$.
- How would we prove this?

R is Closed Under Intersection

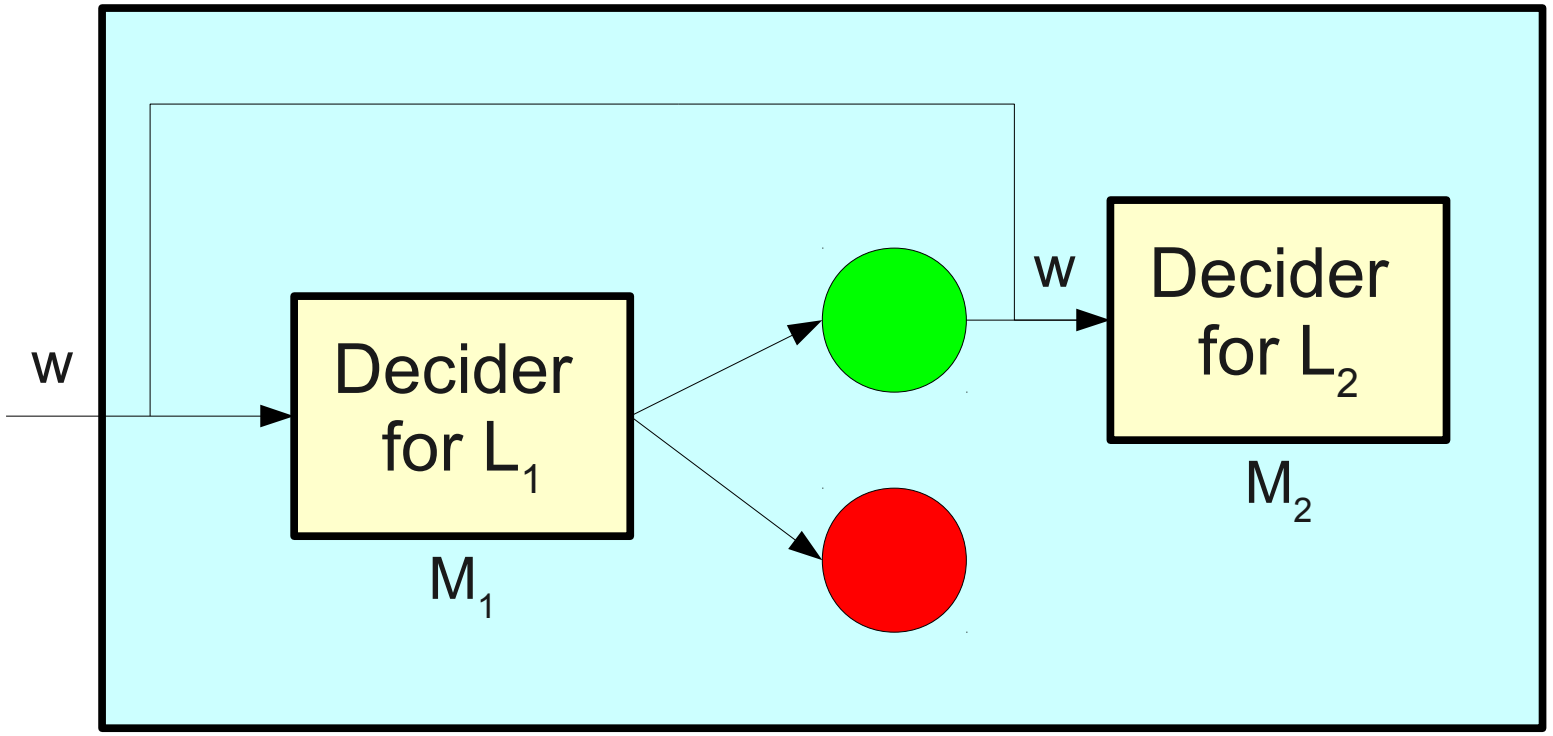
R is Closed Under Intersection



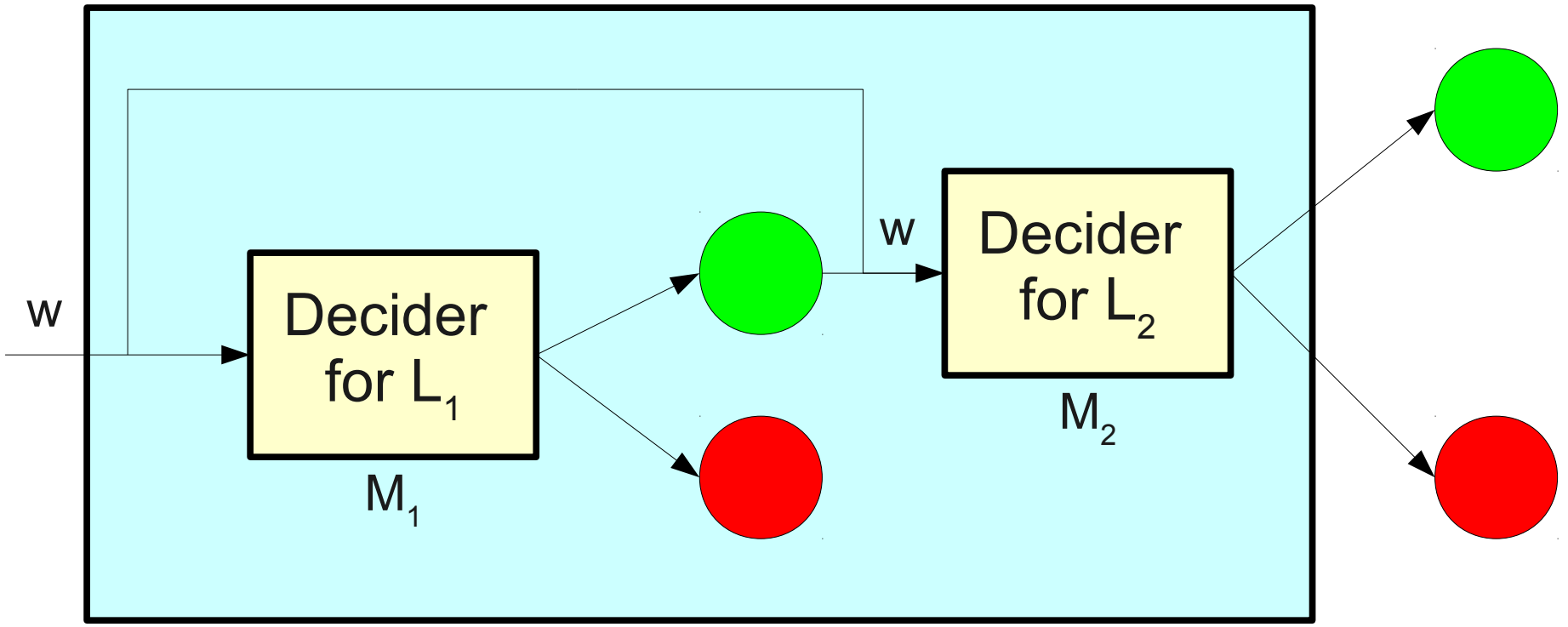
R is Closed Under Intersection



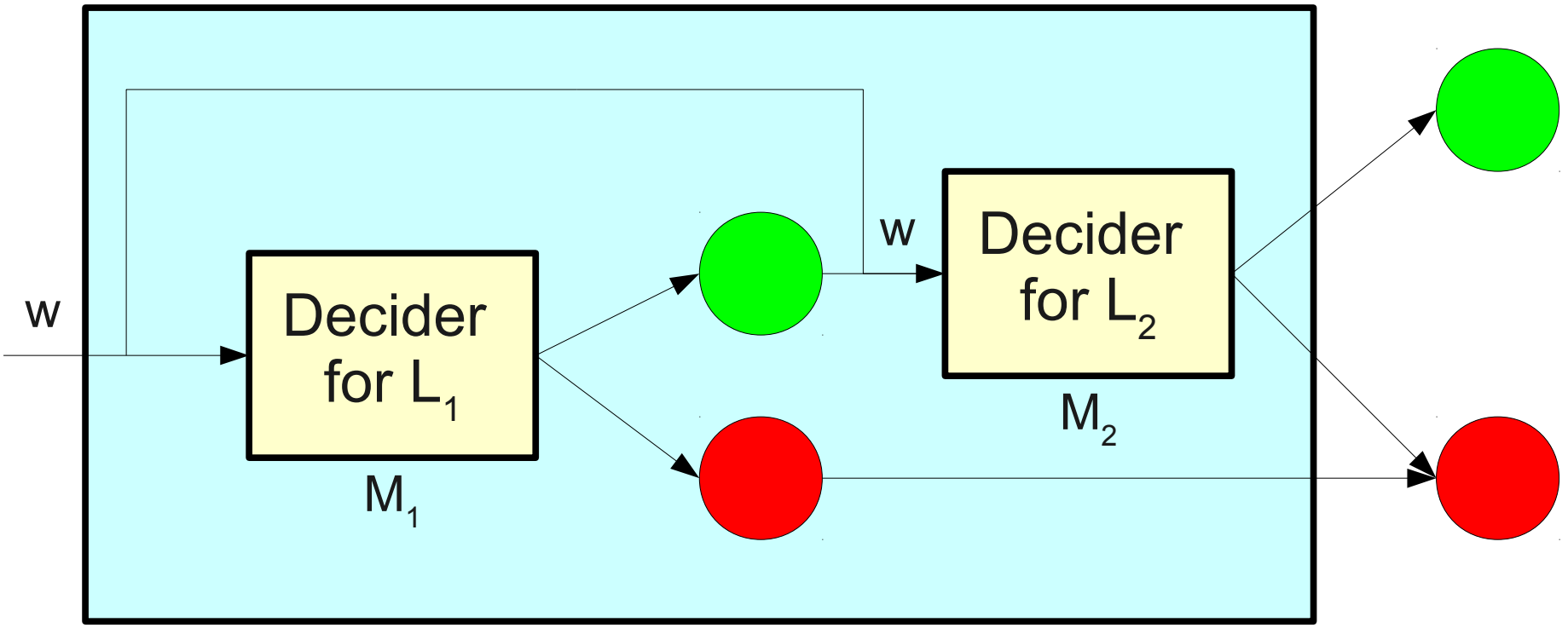
R is Closed Under Intersection



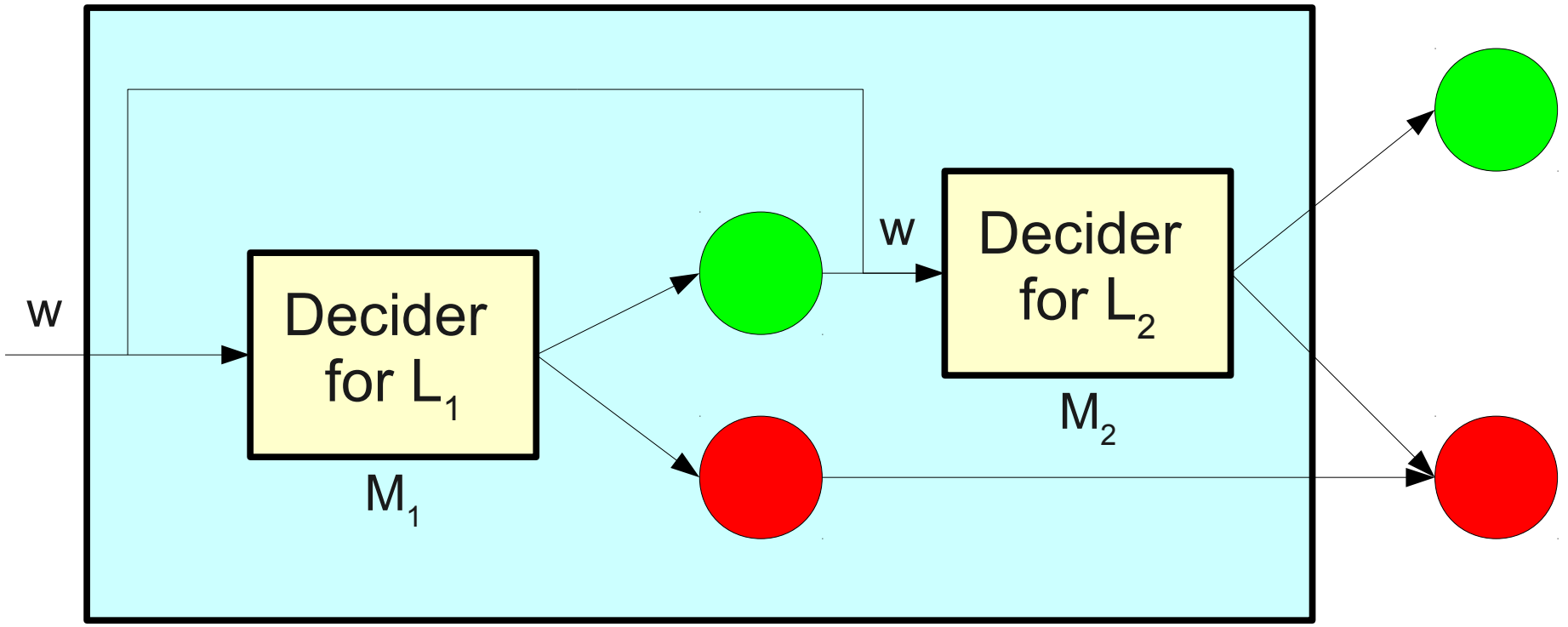
R is Closed Under Intersection



R is Closed Under Intersection

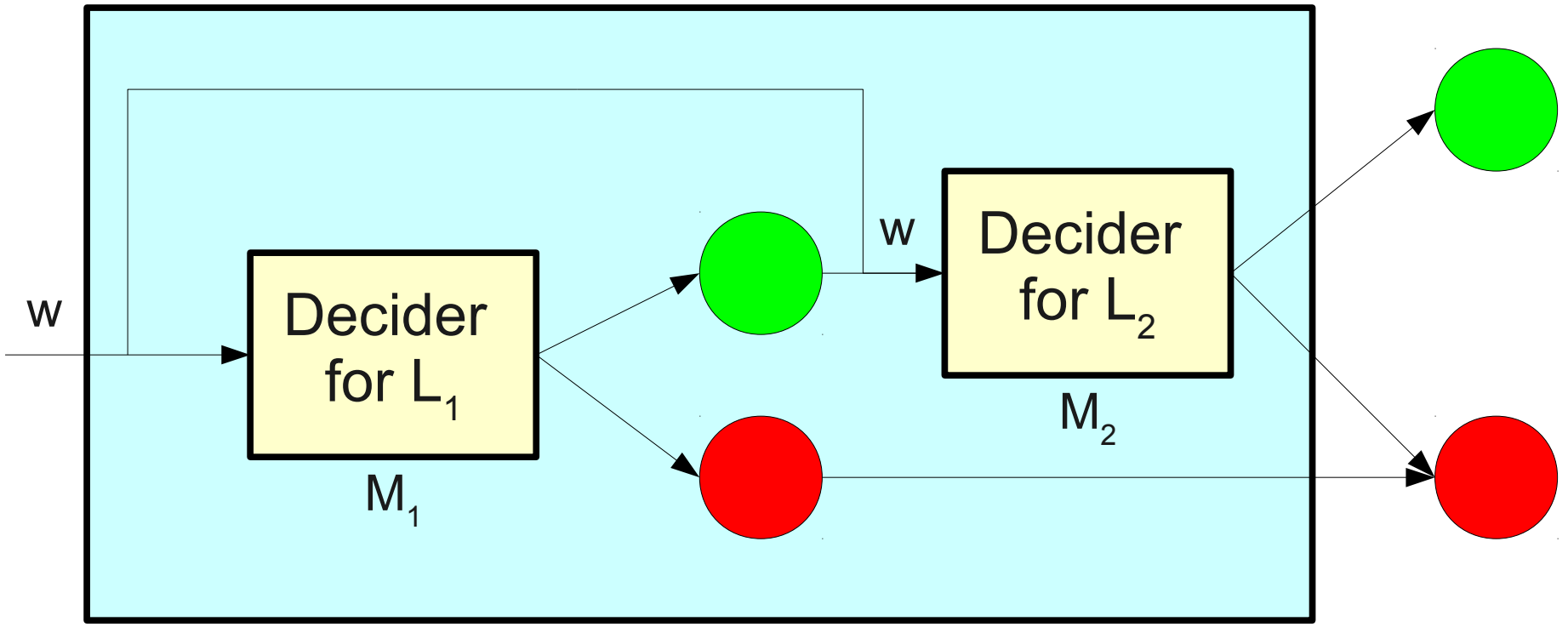


R is Closed Under Intersection



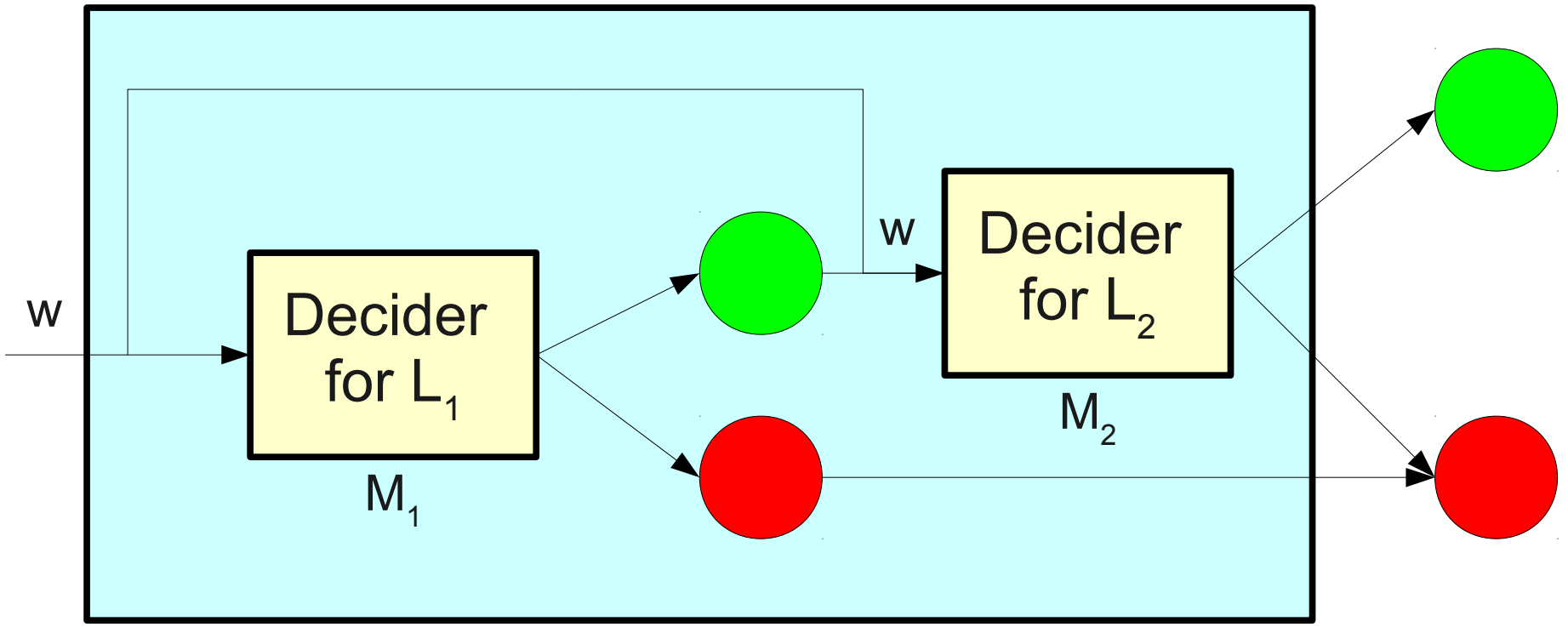
M' = “On input w:

R is Closed Under Intersection



M' = “On input w:
Run M_1 on w.

R is Closed Under Intersection

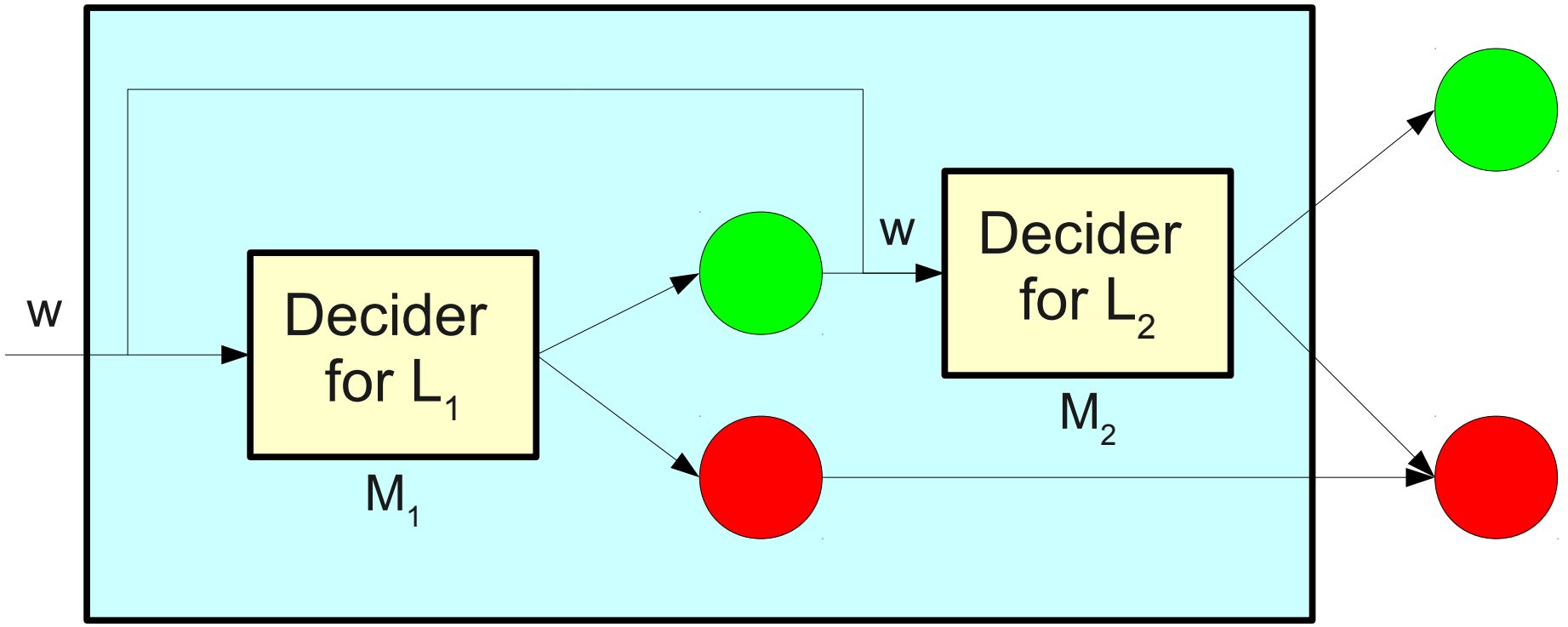


M' = “On input w:

Run M_1 on w.

If M_1 rejects, reject.

R is Closed Under Intersection



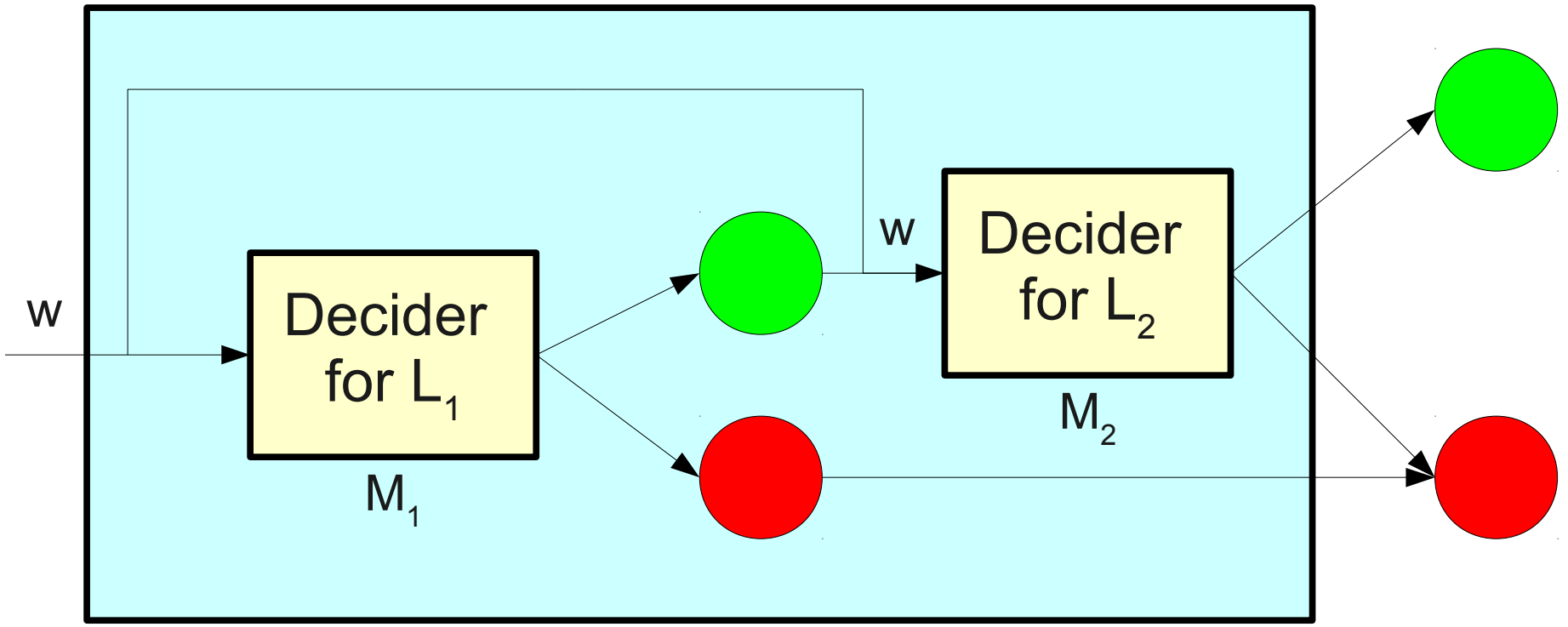
M' = “On input w:

Run M_1 on w.

If M_1 rejects, reject.

Otherwise, run M_2 on w.

R is Closed Under Intersection



M' = “On input w:

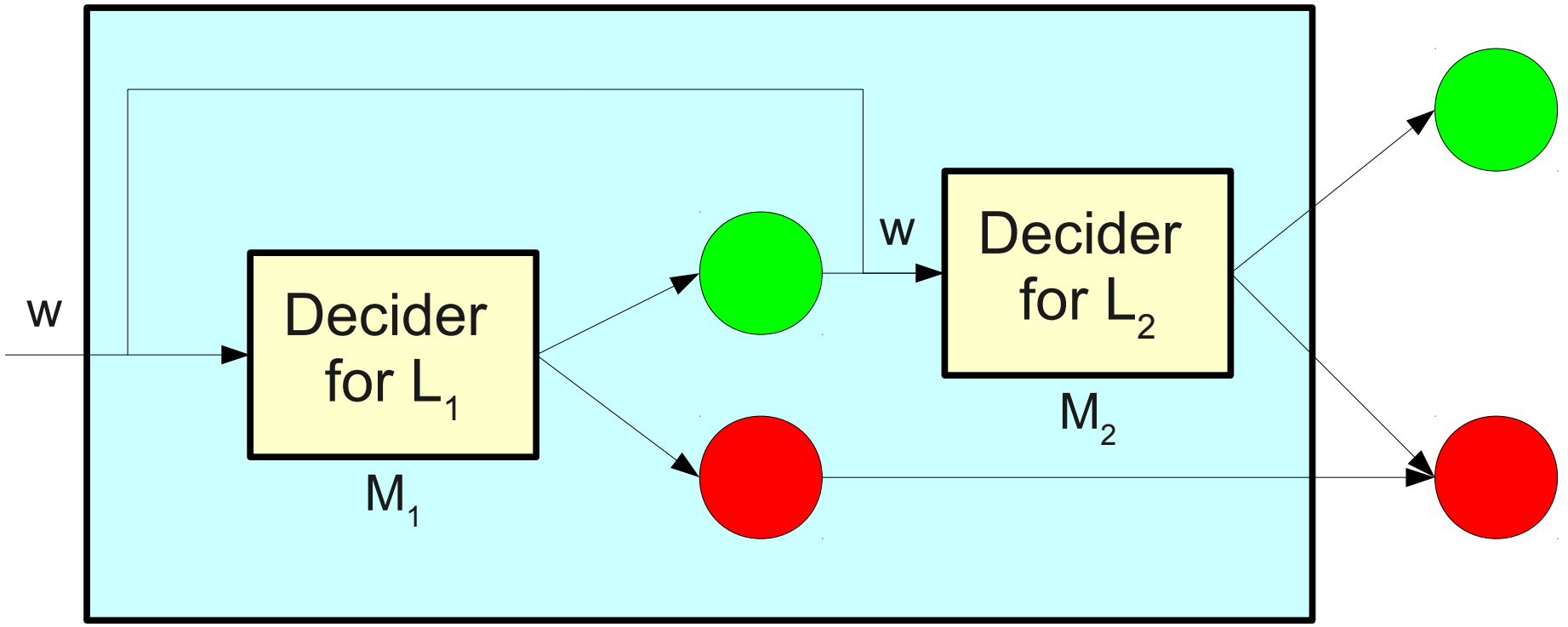
Run M_1 on w.

If M_1 rejects, reject.

Otherwise, run M_2 on w.

If M_2 accepts, accept.

R is Closed Under Intersection



M' = “On input w:

Run M_1 on w.

If M_1 rejects, reject.

Otherwise, run M_2 on w.

If M_2 accepts, accept.

Otherwise, reject.”

Theorem: R is closed under intersection.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = “On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject.”

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = “On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject.”

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$.

It is critical to prove both of these properties. If we don't show that M' is a decider, then we have proven that $L_1 \cap L_2$ is RE, but not necessarily R!

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = “On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject.”

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = “On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject.”

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts. If M_1 rejects, M' rejects.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = “On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject.”

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts. If M_1 rejects, M' rejects. Otherwise, M' runs M_2, and since M_2 is a decider, it always halts.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = “On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject.”

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts. If M_1 rejects, M' rejects. Otherwise, M' runs M_2, and since M_2 is a decider, it always halts. Then, if M_2 accepts, M' accepts, and if M_2 rejects, M' rejects.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = “On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject.”

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts. If M_1 rejects, M' rejects. Otherwise, M' runs M_2, and since M_2 is a decider, it always halts. Then, if M_2 accepts, M' accepts, and if M_2 rejects, M' rejects. In each case, M' halts on any input w, so M' is a decider.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts. If M_1 rejects, M' rejects. Otherwise, M' runs M_2, and since M_2 is a decider, it always halts. Then, if M_2 accepts, M' accepts, and if M_2 rejects, M' rejects. In each case, M' halts on any input w, so M' is a decider.

To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts. If M_1 rejects, M' rejects. Otherwise, M' runs M_2, and since M_2 is a decider, it always halts. Then, if M_2 accepts, M' accepts, and if M_2 rejects, M' rejects. In each case, M' halts on any input w, so M' is a decider.

To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w.

Remember that the language of a machine is the set of strings it accepts. To reason about the language of a TM, we only need to reason about what strings it accepts.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts. If M_1 rejects, M' rejects. Otherwise, M' runs M_2, and since M_2 is a decider, it always halts. Then, if M_2 accepts, M' accepts, and if M_2 rejects, M' rejects. In each case, M' halts on any input w, so M' is a decider.

To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w. M_1 accepts w iff $w \in L_1$ and M_2 accepts w iff $w \in L_2$.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts. If M_1 rejects, M' rejects. Otherwise, M' runs M_2, and since M_2 is a decider, it always halts. Then, if M_2 accepts, M' accepts, and if M_2 rejects, M' rejects. In each case, M' halts on any input w, so M' is a decider.

To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w. M_1 accepts w iff $w \in L_1$ and M_2 accepts w iff $w \in L_2$. Thus M' accepts w iff $w \in L_1$ and $w \in L_2$, so M' accepts w iff $w \in L_1 \cap L_2$.

Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts. If M_1 rejects, M' rejects. Otherwise, M' runs M_2, and since M_2 is a decider, it always halts. Then, if M_2 accepts, M' accepts, and if M_2 rejects, M' rejects. In each case, M' halts on any input w, so M' is a decider.

To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w. M_1 accepts w iff $w \in L_1$ and M_2 accepts w iff $w \in L_2$. Thus M' accepts w iff $w \in L_1$ and $w \in L_2$, so M' accepts w iff $w \in L_1 \cap L_2$. Thus $\mathcal{L}(M') = L_1 \cap L_2$.

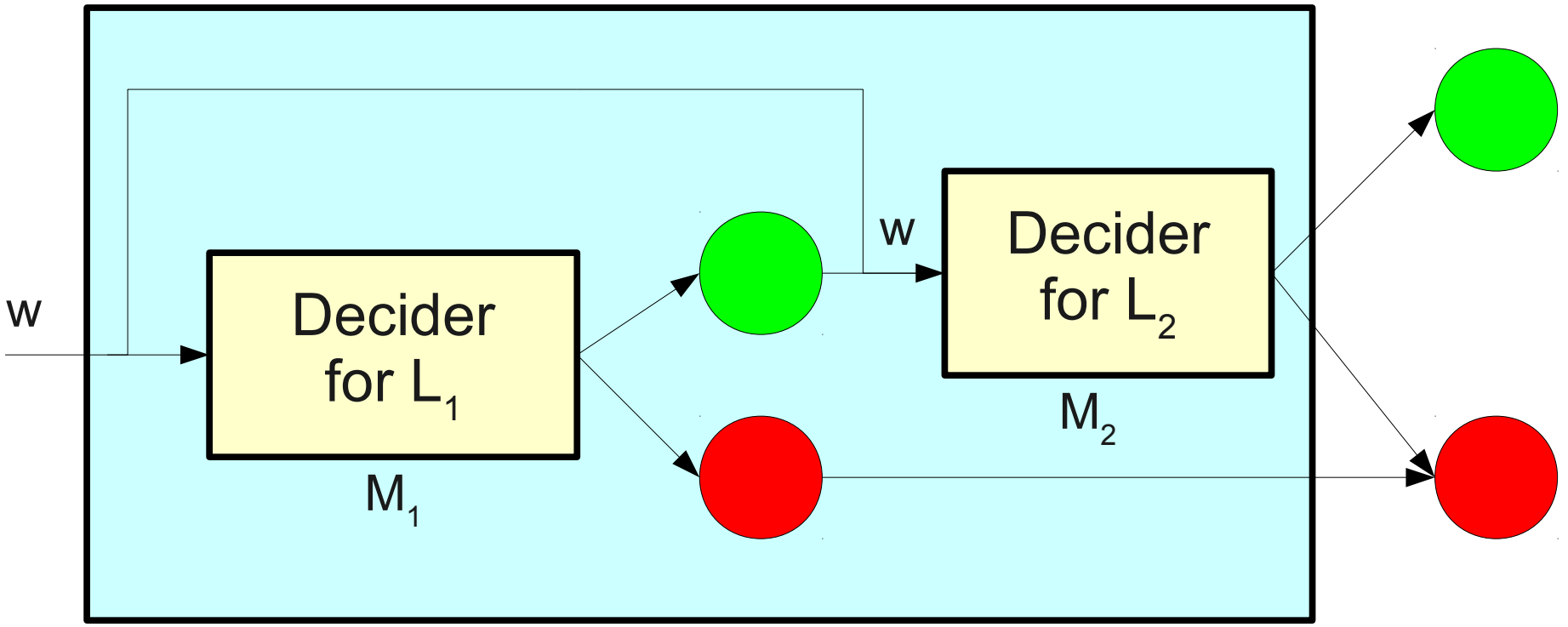
Theorem: R is closed under intersection.
Proof: Suppose L_1 and L_2 are recursive, and let M_1 and M_2 be deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a decider, then show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that M' is a decider, note that since M_1 is a decider, when M_1 runs on w, M_1 always halts. If M_1 rejects, M' rejects. Otherwise, M' runs M_2, and since M_2 is a decider, it always halts. Then, if M_2 accepts, M' accepts, and if M_2 rejects, M' rejects. In each case, M' halts on any input w, so M' is a decider.

To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w. M_1 accepts w iff $w \in L_1$ and M_2 accepts w iff $w \in L_2$. Thus M' accepts w iff $w \in L_1$ and $w \in L_2$, so M' accepts w iff $w \in L_1 \cap L_2$. Thus $\mathcal{L}(M') = L_1 \cap L_2$. ■

R is Closed Under Intersection



M' = “On input w:

Run M_1 on w.

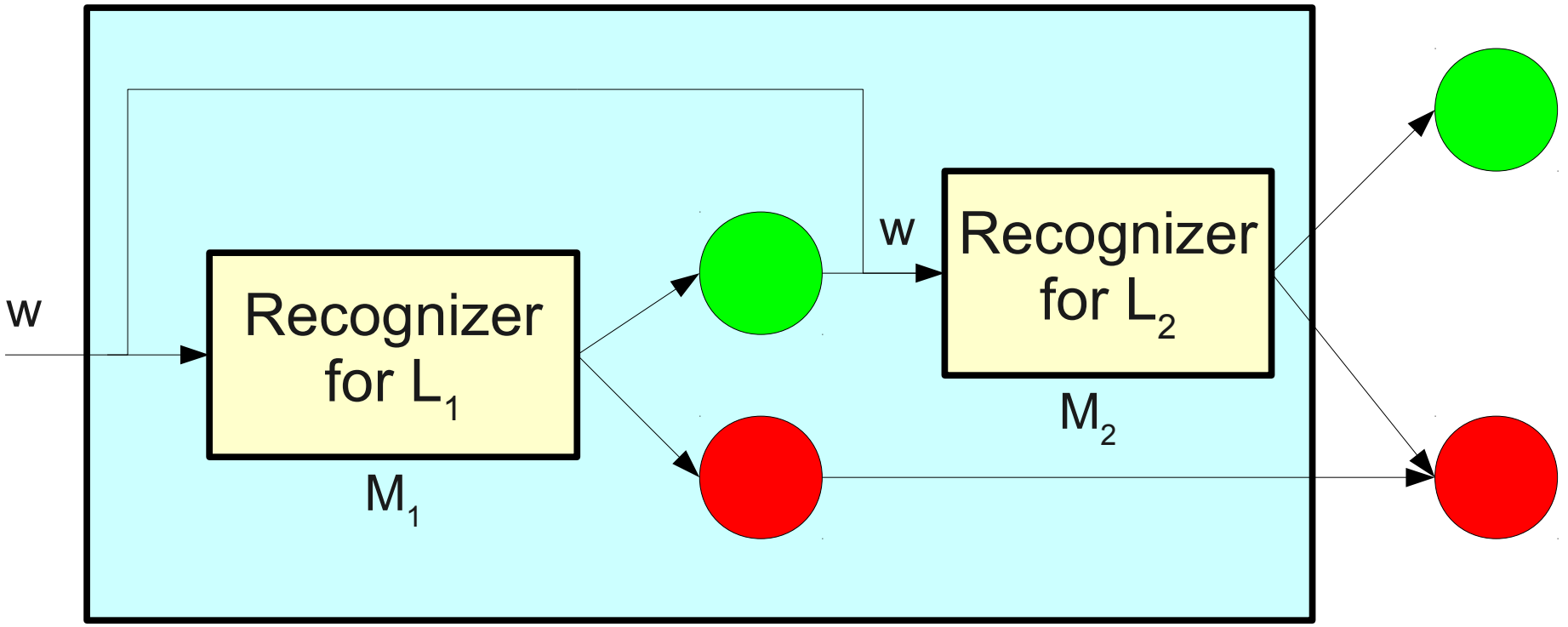
If M_1 rejects, reject.

Otherwise, run M_2 on w.

If M_2 accepts, accept.

Otherwise, reject.”

RE is Closed Under Intersection



M' = “On input w:

Run M_1 on w.

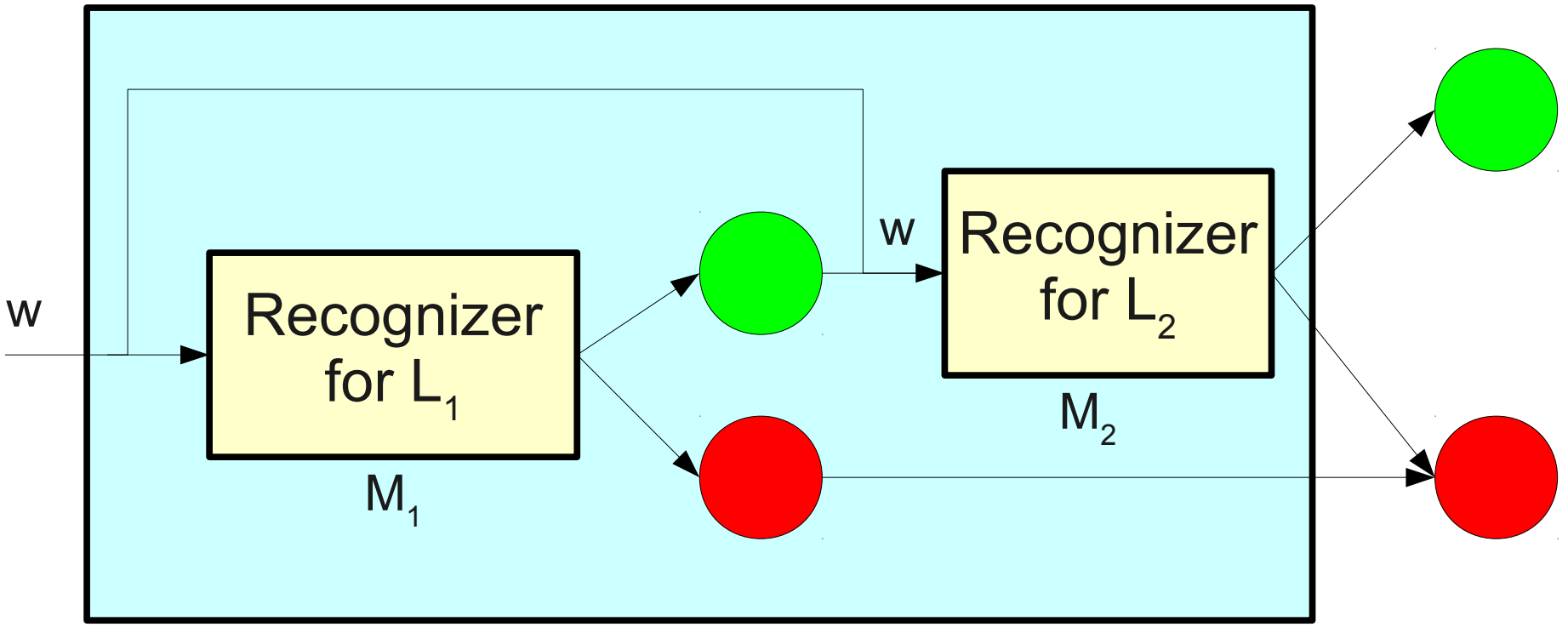
If M_1 rejects, reject.

Otherwise, run M_2 on w.

If M_2 accepts, accept.

Otherwise, reject.”

RE is Closed Under Intersection



M' = "On input w:

Run M_1 on w.

If M_1 rejects, reject.

Otherwise, run M_2 on w.

If M_2 accepts, accept.

Otherwise, reject."

Does this construction still work?

Theorem: RE is closed under intersection.

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively.

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M' as follows:

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = “On input w:

Run M_1 on w.

If M_1 rejects, reject.

Otherwise, run M_2 on w.

If M_2 accepts, accept.

Otherwise, reject.”

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = “On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject.”

We show that $\mathcal{L}(M') = L_1 \cap L_2$.

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that $\mathcal{L}(M') = L_1 \cap L_2$.

Note that there is no requirement that we prove that M' halts on all inputs. All we need to do is reason about what inputs it accepts.

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w.

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w.

We only care about what strings M' accepts, and can ignore the strings it rejects or loops infinitely on.

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w. M_1 accepts w iff $w \in L_1$ and M_2 accepts w iff $w \in L_2$.

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w. M_1 accepts w iff $w \in L_1$ and M_2 accepts w iff $w \in L_2$. Thus M' accepts w iff $w \in L_1$ and $w \in L_2$, so M' accepts w iff $w \in L_1 \cap L_2$.

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w. M_1 accepts w iff $w \in L_1$ and M_2 accepts w iff $w \in L_2$. Thus M' accepts w iff $w \in L_1$ and $w \in L_2$, so M' accepts w iff $w \in L_1 \cap L_2$. Thus $\mathcal{L}(M') = L_1 \cap L_2$.

Theorem: RE is closed under intersection.
Proof: Suppose L_1 and L_2 are RE, and let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:

Run M_1 on w.
If M_1 rejects, reject.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that $\mathcal{L}(M') = L_1 \cap L_2$. To see that $\mathcal{L}(M') = L_1 \cap L_2$, note that M' accepts w iff both M_1 accepts w and M_2 accepts w. M_1 accepts w iff $w \in L_1$ and M_2 accepts w iff $w \in L_2$. Thus M' accepts w iff $w \in L_1$ and $w \in L_2$, so M' accepts w iff $w \in L_1 \cap L_2$. Thus $\mathcal{L}(M') = L_1 \cap L_2$. ■

Proving Closure Properties of R

- To show that R is closed under some operation:
  - Obtain deciders M_1, M_2, …, M_n for all of the original languages.
  - Construct a new TM M' based on M_1, M_2, …, M_n.
  - Prove that $\mathcal{L}$(M') is the correct language.
  - Prove that M' is a decider (that it halts on all inputs)

Proving Closure Properties of RE

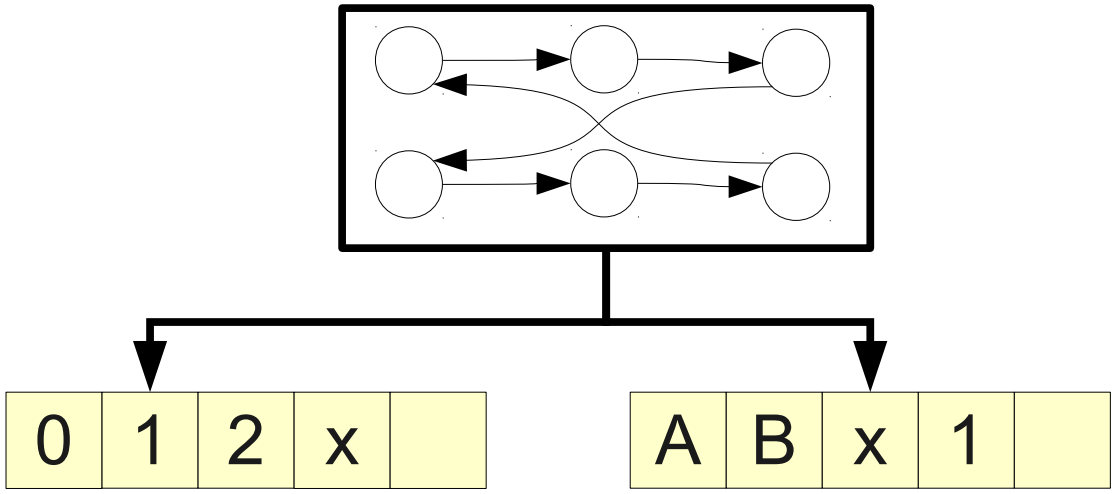
- To show that RE is closed under some operation:
  - Obtain **recognizers** M_1, M_2, …, M_n for all of the original languages.
  - Construct a new TM M' based on M_1, M_2, …, M_n.
  - Prove that $\mathcal{L}$(M') is the correct language.

An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.

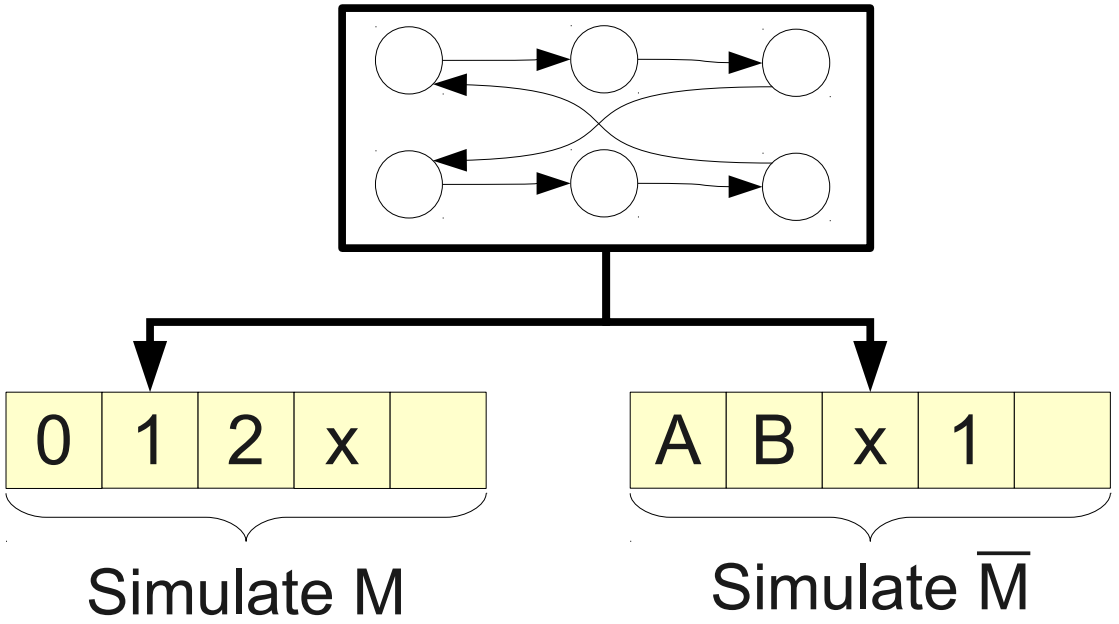
An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.



An Important Result

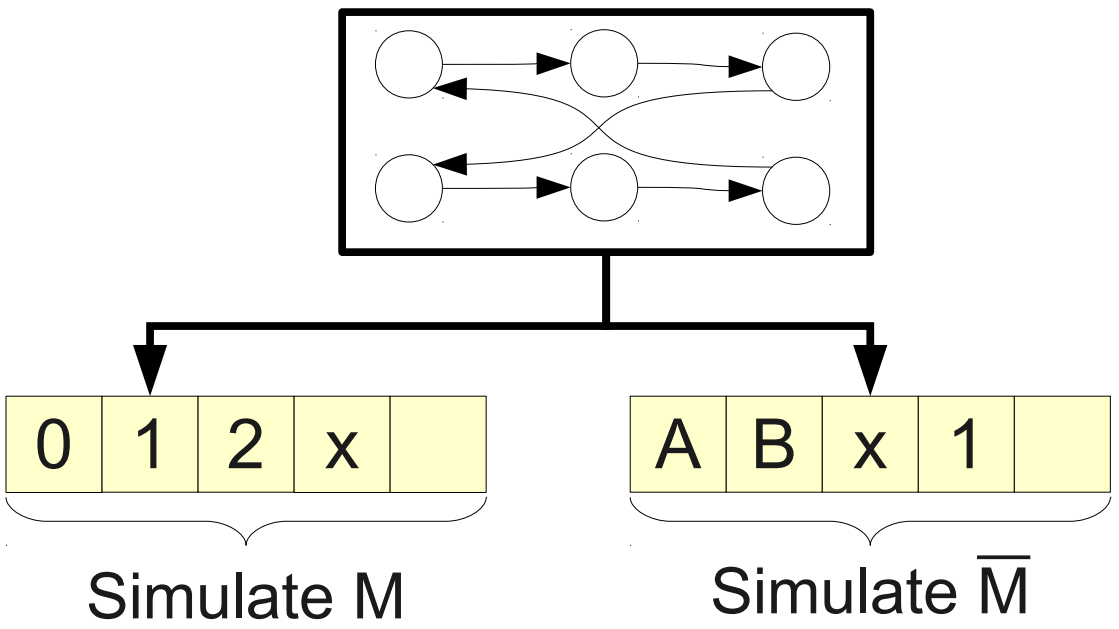
- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.



An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.

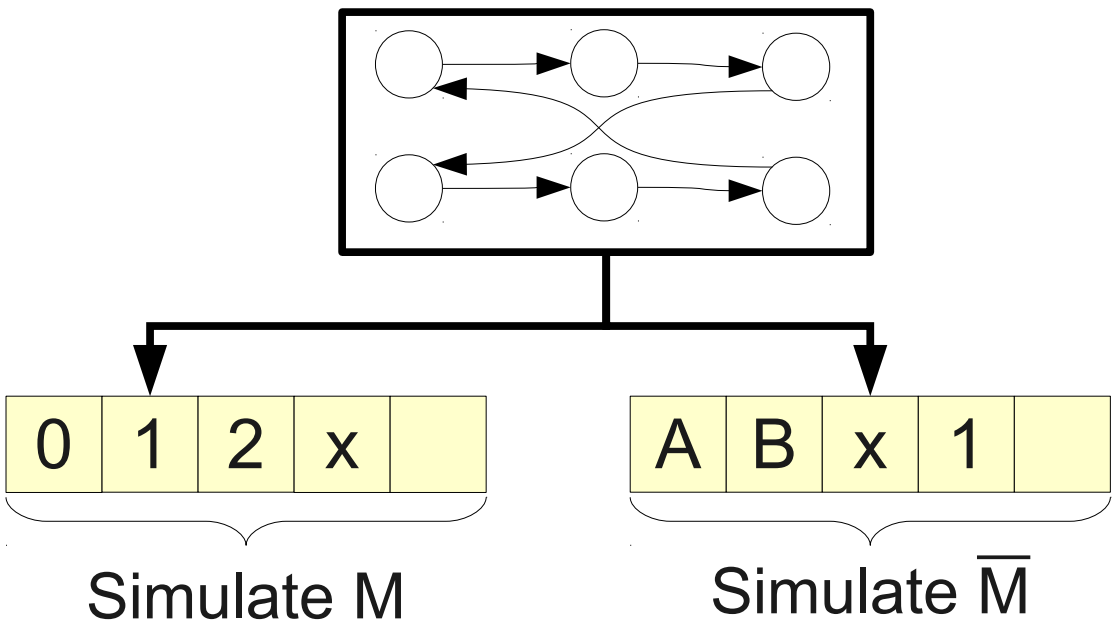
M' = “On input w,
Run M and $\overline{M}$ on w in parallel.
If M accepts, accept.
If $\overline{M}$ accepts, reject.”



An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.

M' = “On input w,
Run M and $\overline{M}$ on w in parallel.
If M accepts, accept.
If $\overline{M}$ accepts, reject.”

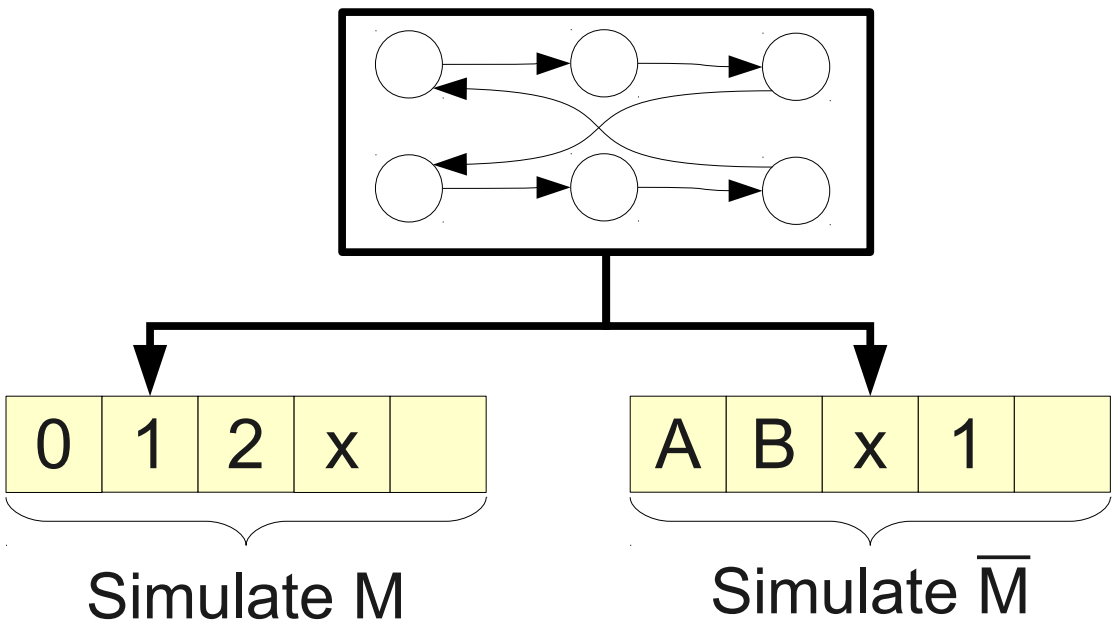


What happens if $w \in L$?

An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.

M' = “On input w,
Run M and $\overline{M}$ on w in parallel.
If **M accepts**, accept.
If $\overline{M}$ accepts, reject.”

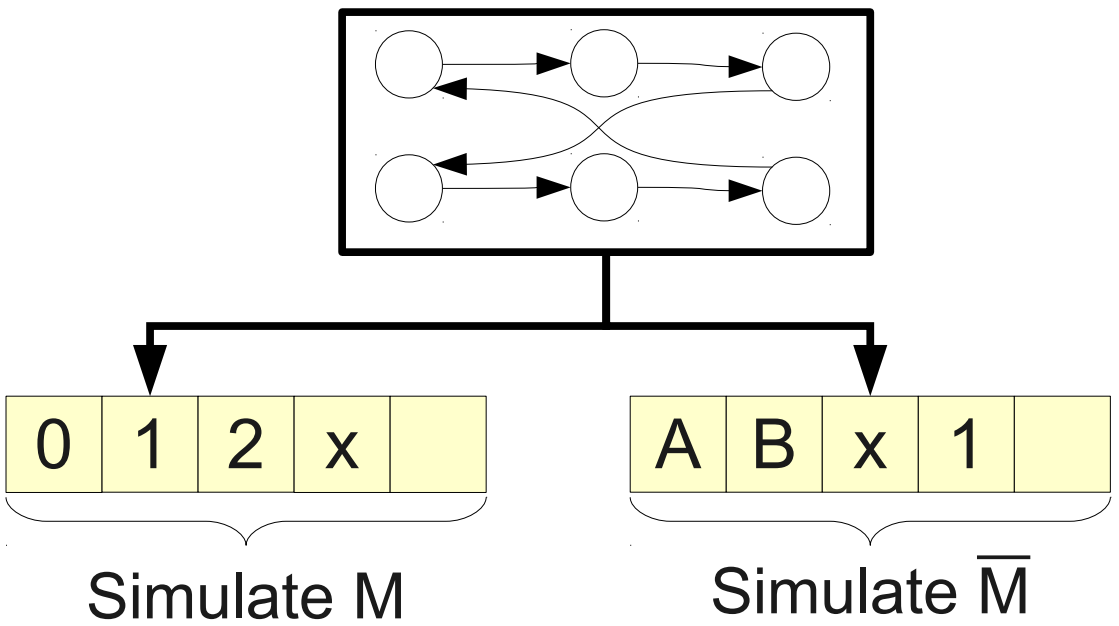


What happens if $w \in L$?

An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.

M' = “On input w,
Run M and $\overline{M}$ on w in parallel.
If M accepts, accept.
If $\overline{M}$ accepts, reject.”

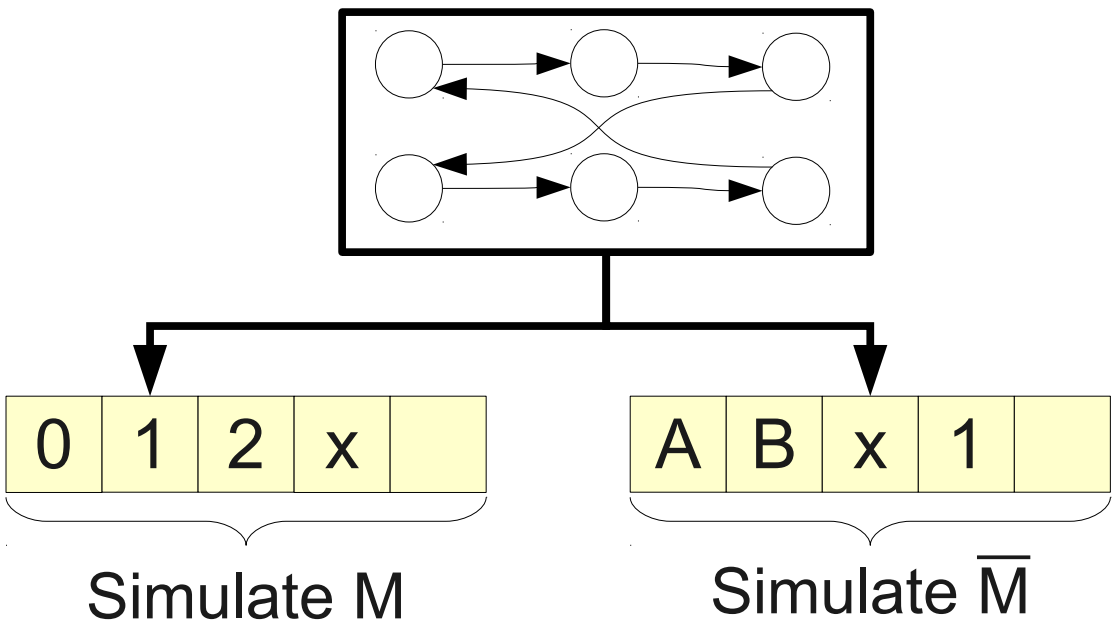


What happens if $w \in L$?

An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.

M' = “On input w,
Run M and $\overline{M}$ on w in parallel.
If M accepts, accept.
If $\overline{M}$ accepts, reject.”

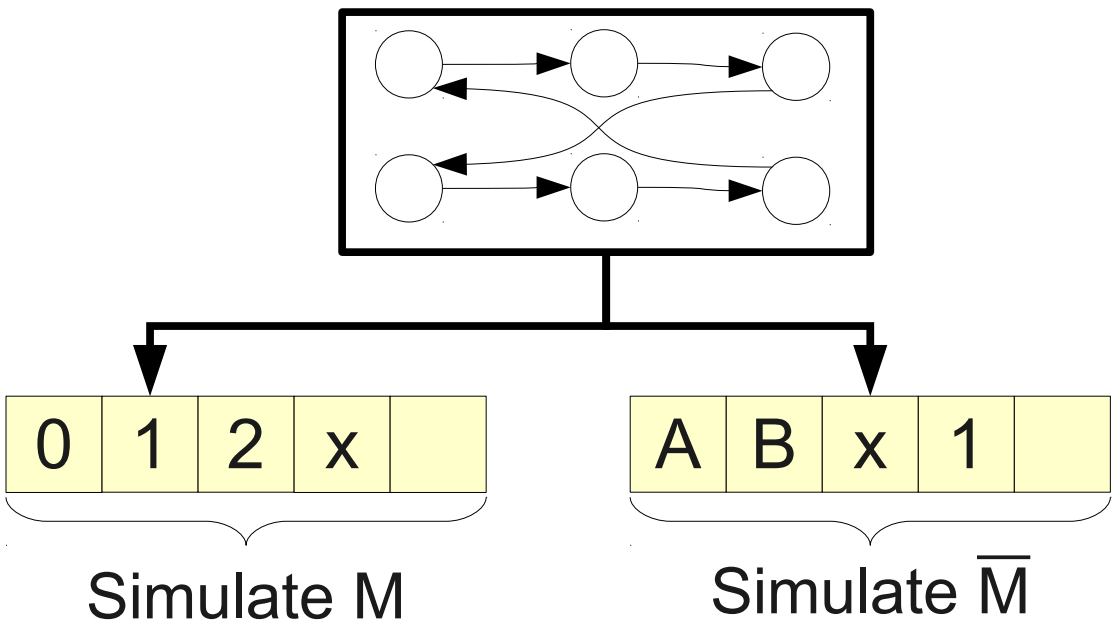


What happens if $w \notin L$?

An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.

M' = “On input w,
Run M and $\overline{M}$ on w in parallel.
If M accepts, accept.
If $\overline{M}$ accepts, reject.”

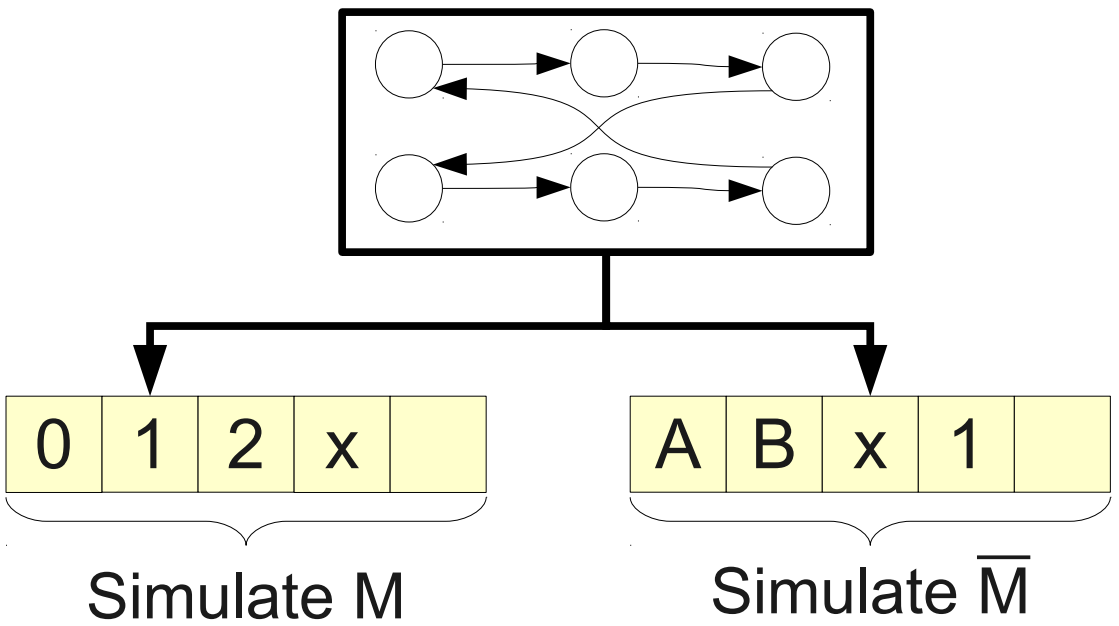


What happens if $w \in \overline{L}$?

An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.

M' = "On input w,
Run M and $\overline{M}$ on w in parallel.
If M accepts, accept.
If **$\overline{M}$ accepts**, reject."

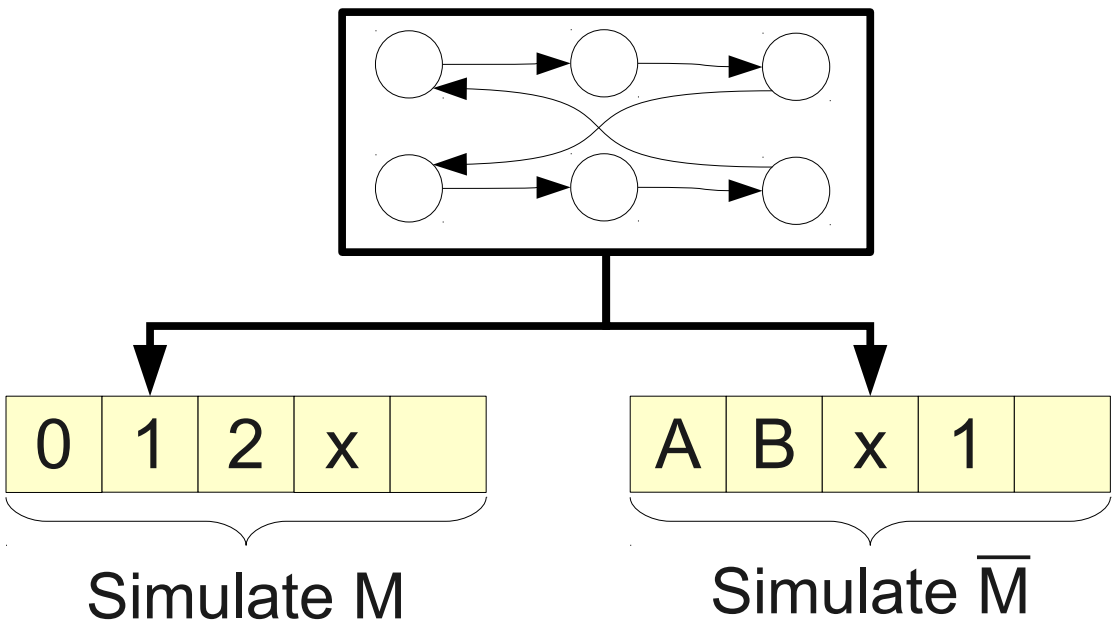


What happens if $w \in \overline{L}$?

An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.

M' = "On input w,
Run M and $\overline{M}$ on w in parallel.
If M accepts, accept.
If $\overline{M}$ accepts, reject."

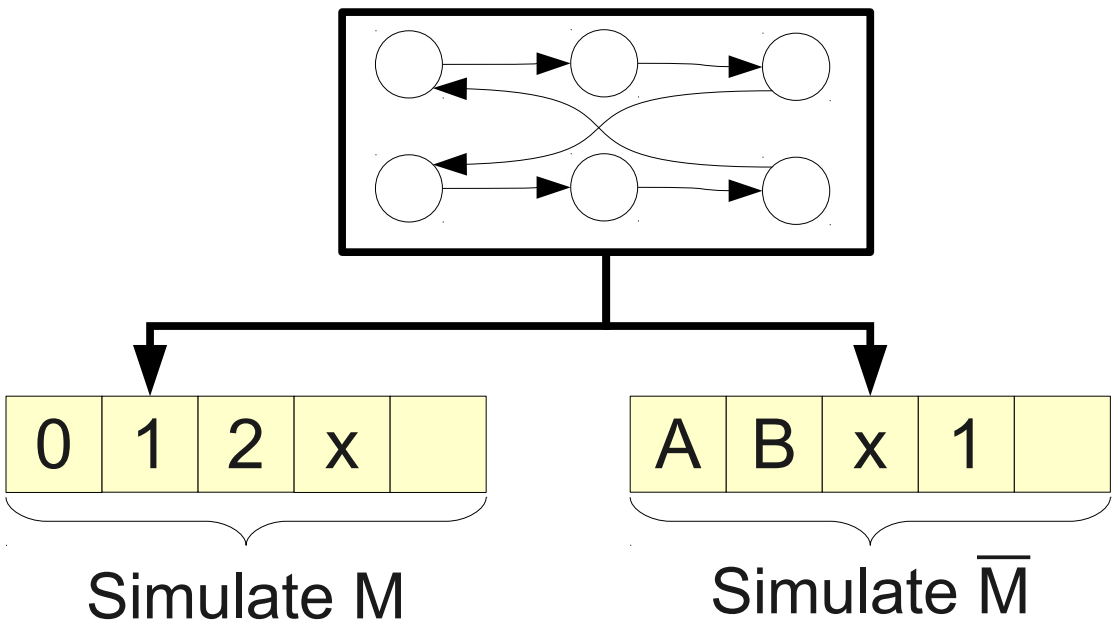


What happens if $w \in \overline{L}$?

An Important Result

- Suppose that L and $\overline{L}$ are RE languages.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.

M' = "On input w,
Run M and $\overline{M}$ on w in parallel.
If M accepts, accept.
If $\overline{M}$ accepts, reject."



M' is a decider!

Theorem: If L and $\overline{L}$ are RE, then L is recursive.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$.
Consider the TM M' defined as follows:

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$.
Consider the TM M' defined as follows:

M' = “On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject.”

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$.
Consider the TM M' defined as follows:

M' = “On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject.”

We prove that $\mathcal{L}$(M') = L and that M' is a decider.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$.
Consider the TM M' defined as follows:

M' = “On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject.”

We prove that $\mathcal{L}$(M') = L and that M' is a decider. To see that M' is a decider, consider any string w.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$.
Consider the TM M' defined as follows:

M' = “On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject.”

We prove that $\mathcal{L}$(M') = L and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$.
Consider the TM M' defined as follows:

M' = “On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject.”

We prove that $\mathcal{L}$(M') = L and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$.
Consider the TM M' defined as follows:

M' = “On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject.”

We prove that $\mathcal{L}$(M') = L and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$. Thus either $w \in L$ or $w \in \overline{L}$, but not both.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$.
Consider the TM M' defined as follows:

M' = “On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject.”

We prove that $\mathcal{L}$(M') = L and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$. Thus either $w \in L$ or $w \in \overline{L}$, but not both. If $w \in L$, then M will eventually accept w and M' halts.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$. Consider the TM M' defined as follows:

M' = “On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject.”

We prove that $\mathcal{L}$(M') = L and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$. Thus either $w \in L$ or $w \in \overline{L}$, but not both. If $w \in L$, then M will eventually accept w and M' halts. If $w \in \overline{L}$, then $\overline{M}$ will eventually accept w and M' halts.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$.
Consider the TM M' defined as follows:

M' = “On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject.”

We prove that $\mathcal{L}$(M') = L and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$. Thus either $w \in L$ or $w \in \overline{L}$, but not both. If $w \in L$, then M will eventually accept w and M' halts. If $w \in \overline{L}$, then $\overline{M}$ will eventually accept w and M' halts. Thus M' halts on all inputs.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$. Consider the TM M' defined as follows:

M' = "On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject."

We prove that $\mathcal{L}$(M') = L and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$. Thus either $w \in L$ or $w \in \overline{L}$, but not both. If $w \in L$, then M will eventually accept w and M' halts. If $w \in \overline{L}$, then $\overline{M}$ will eventually accept w and M' halts. Thus M' halts on all inputs.

To see that $\mathcal{L}$(M') = L, note that M' accepts w iff M accepts w and $\overline{M}$ does not accept w.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$. Consider the TM M' defined as follows:

M' = "On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject."

We prove that $\mathcal{L}(M') = L$ and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$. Thus either $w \in L$ or $w \in \overline{L}$, but not both. If $w \in L$, then M will eventually accept w and M' halts. If $w \in \overline{L}$, then $\overline{M}$ will eventually accept w and M' halts. Thus M' halts on all inputs.

To see that $\mathcal{L}(M') = L$, note that M' accepts w iff M accepts w and $\overline{M}$ does not accept w.

We do have to consider this case - what if M would eventually accept w, but $\overline{M}$ accepts it first?

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$. Consider the TM M' defined as follows:

M' = "On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject."

We prove that $\mathcal{L}$(M') = L and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$. Thus either $w \in L$ or $w \in \overline{L}$, but not both. If $w \in L$, then M will eventually accept w and M' halts. If $w \in \overline{L}$, then $\overline{M}$ will eventually accept w and M' halts. Thus M' halts on all inputs.

To see that $\mathcal{L}$(M') = L, note that M' accepts w iff M accepts w and $\overline{M}$ does not accept w. Since M accepts w iff $w \in L$ and $\overline{M}$ accepts w iff $w \notin L$, if M accepts w, $\overline{M}$ does not accept w.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$. Consider the TM M' defined as follows:

M' = "On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject."

We prove that $\mathcal{L}$(M') = L and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$. Thus either $w \in L$ or $w \in \overline{L}$, but not both. If $w \in L$, then M will eventually accept w and M' halts. If $w \in \overline{L}$, then $\overline{M}$ will eventually accept w and M' halts. Thus M' halts on all inputs.

To see that $\mathcal{L}$(M') = L, note that M' accepts w iff M accepts w and $\overline{M}$ does not accept w. Since M accepts w iff $w \in L$ and $\overline{M}$ accepts w iff $w \notin L$, if M accepts w, $\overline{M}$ does not accept w. Thus M' accepts w iff M accepts w iff $w \in L$.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$. Consider the TM M' defined as follows:

M' = "On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject."

We prove that $\mathcal{L}(M') = L$ and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$. Thus either $w \in L$ or $w \in \overline{L}$, but not both. If $w \in L$, then M will eventually accept w and M' halts. If $w \in \overline{L}$, then $\overline{M}$ will eventually accept w and M' halts. Thus M' halts on all inputs.

To see that $\mathcal{L}(M') = L$, note that M' accepts w iff M accepts w and $\overline{M}$ does not accept w. Since M accepts w iff $w \in L$ and $\overline{M}$ accepts w iff $w \notin L$, if M accepts w, $\overline{M}$ does not accept w. Thus M' accepts w iff M accepts w iff $w \in L$. Thus $\mathcal{L}(M') = L$.

Theorem: If L and $\overline{L}$ are RE, then L is recursive.
Proof: Let M be a recognizer for L and $\overline{M}$ be a recognizer for $\overline{L}$. Consider the TM M' defined as follows:

M' = "On input w:
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject."

We prove that $\mathcal{L}(M') = L$ and that M' is a decider. To see that M' is a decider, consider any string w. Either $w \in L$ or $w \notin L$, but not both. If $w \notin L$, then $w \in \overline{L}$. Thus either $w \in L$ or $w \in \overline{L}$, but not both. If $w \in L$, then M will eventually accept w and M' halts. If $w \in \overline{L}$, then $\overline{M}$ will eventually accept w and M' halts. Thus M' halts on all inputs.

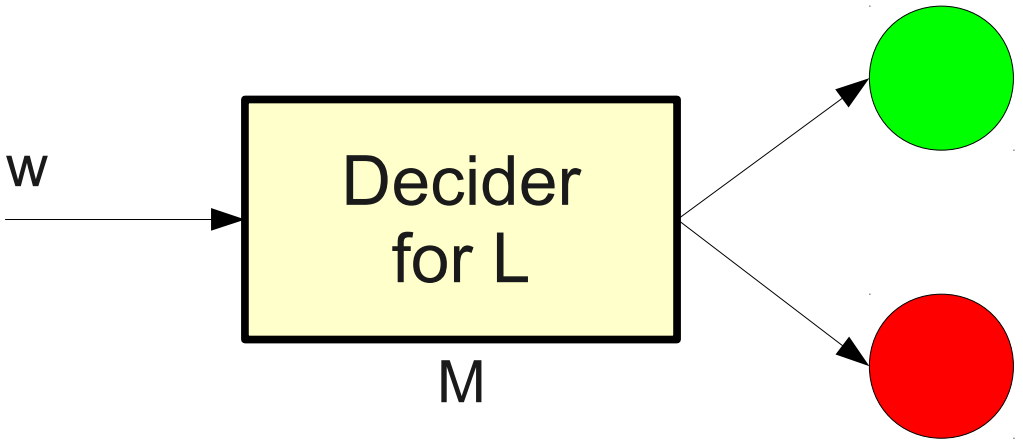
To see that $\mathcal{L}(M') = L$, note that M' accepts w iff M accepts w and $\overline{M}$ does not accept w. Since M accepts w iff $w \in L$ and $\overline{M}$ accepts w iff $w \notin L$, if M accepts w, $\overline{M}$ does not accept w. Thus M' accepts w iff M accepts w iff $w \in L$. Thus $\mathcal{L}(M') = L$. ■

R is Closed Under Complement

If L is recursive, then $\overline{L}$ is recursive as well.

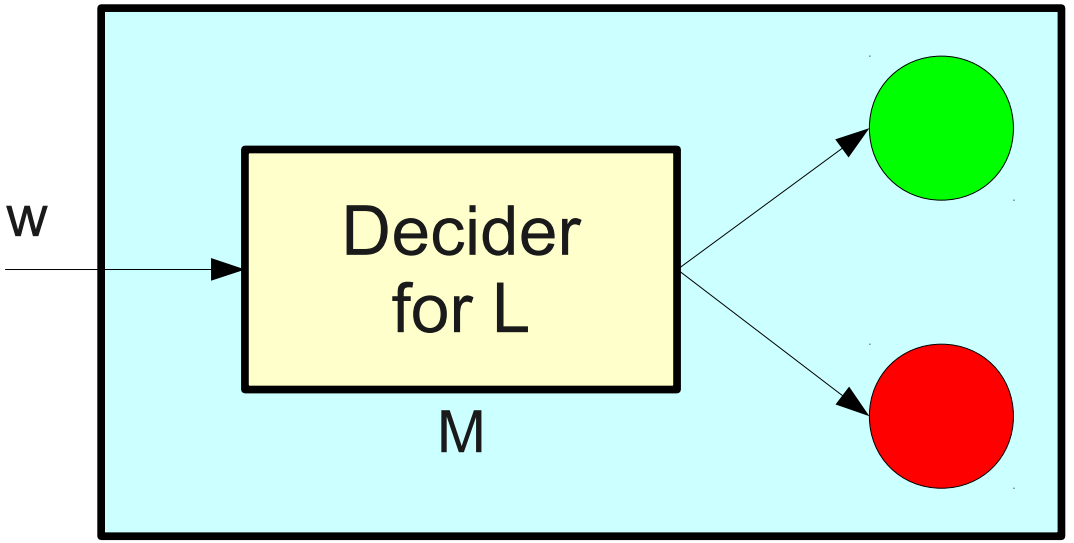
R is Closed Under Complement

If L is recursive, then $\overline{L}$ is recursive as well.



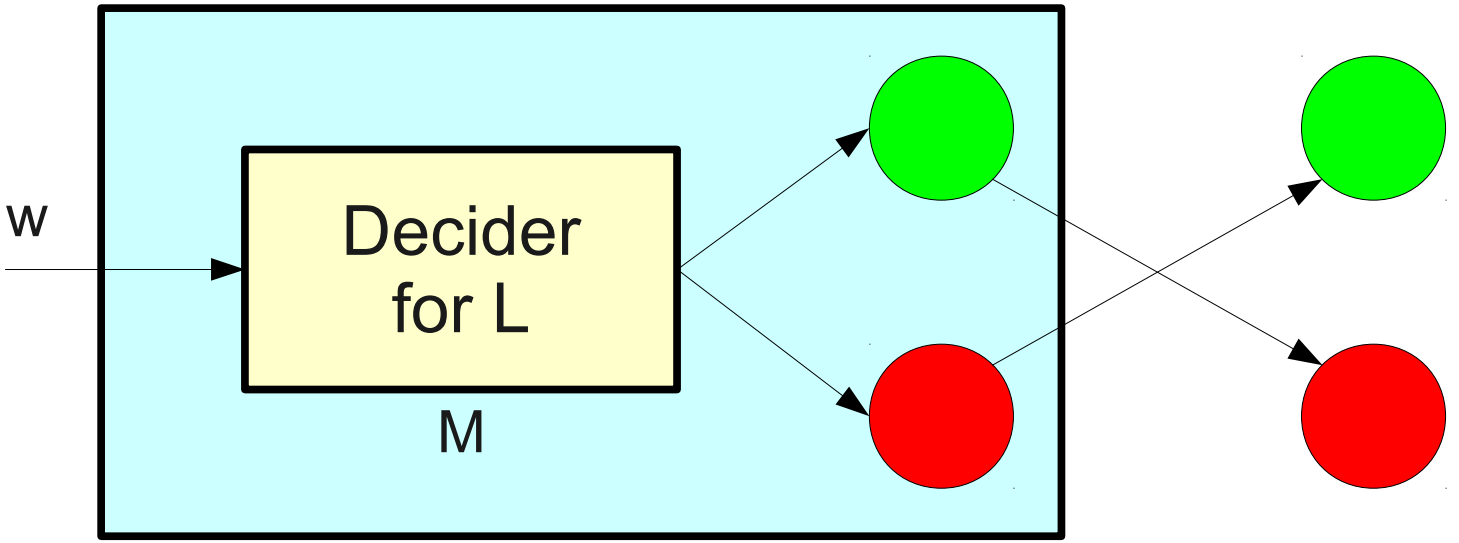
R is Closed Under Complement

If L is recursive, then $\overline{L}$ is recursive as well.



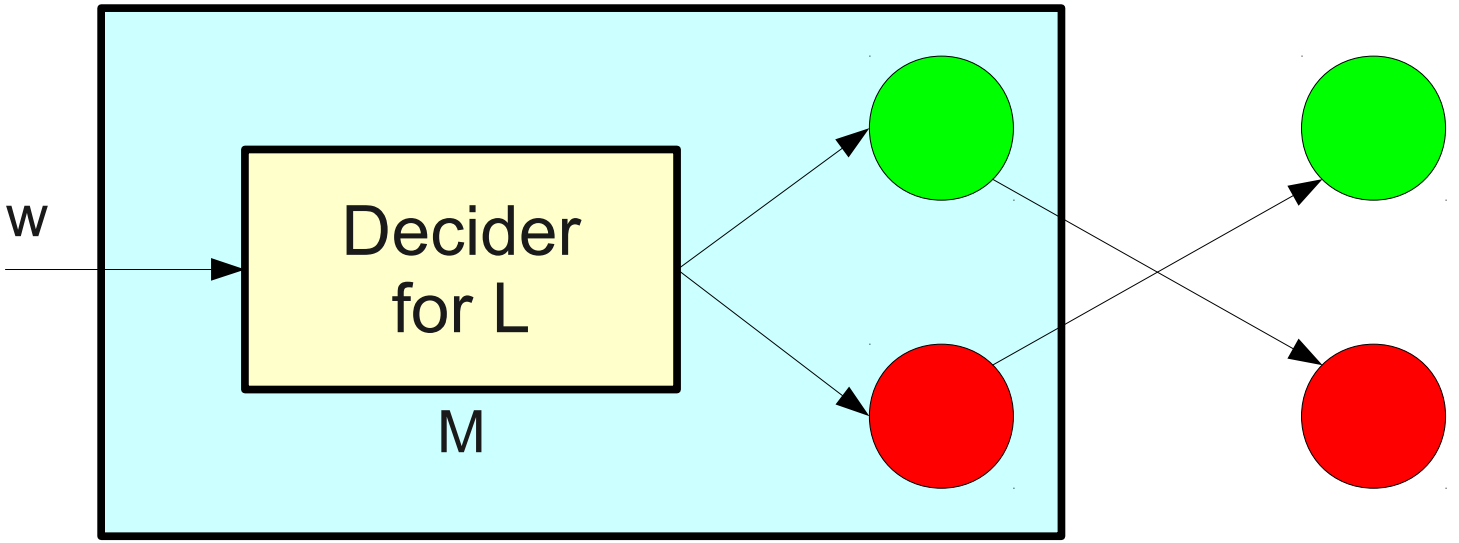
R is Closed Under Complement

If L is recursive, then $\overline{L}$ is recursive as well.



R is Closed Under Complement

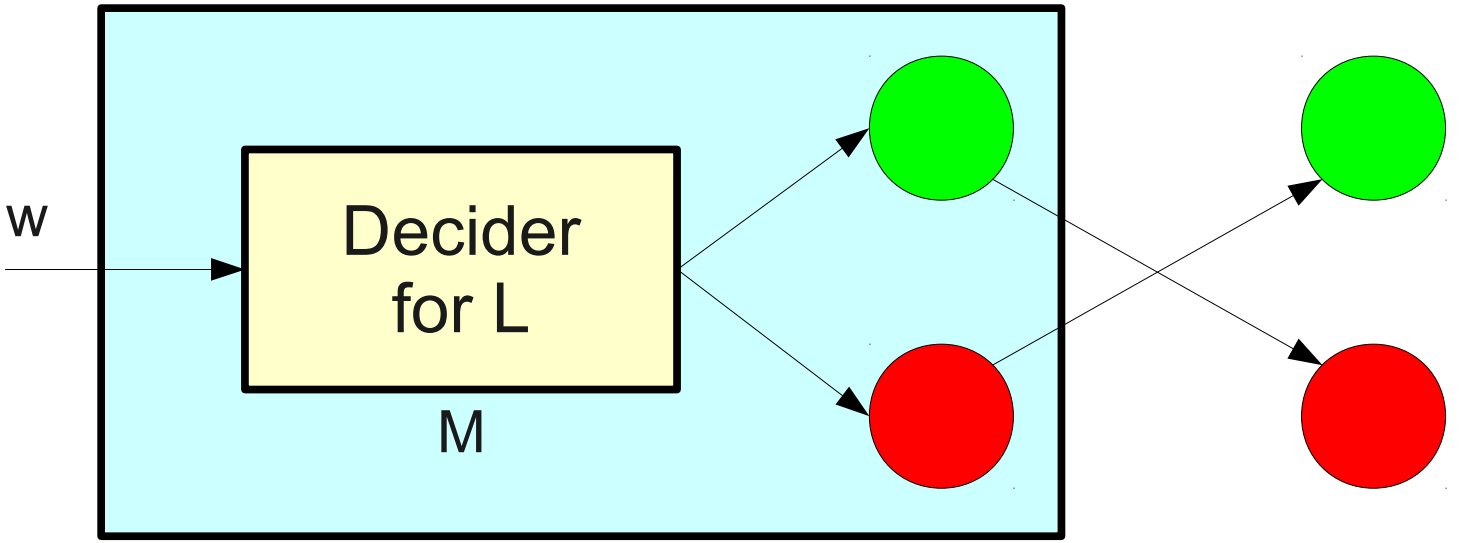
If L is recursive, then $\overline{L}$ is recursive as well.



M' = "On input *w*:
Run *M* on *w*.
If *M* accepts, reject.
If *M* rejects, accept."

R is Closed Under Complement

If L is recursive, then $\overline{L}$ is recursive as well.



M' = "On input *w*:
Run *M* on *w*.
If *M* accepts, reject.
If *M* rejects, accept."

Why doesn't this work for RE languages?

Next Time

- **Unsolvable Problems**
  - What languages are *not R* or *RE*?
  - What problems are provably impossible to solve?
  - Does *R* = *RE*?