R and **RE** Languages

Announcements

- Problem Set 6 due now.
- Problem Set 7 out, due next Friday, November 16th.
  - Play around with Turing machines, the **R** and **RE** languages, and their limits.
  - Some problems require Monday's lecture; if you want to get a jump on that, look at Sipser, Chapter 4.
  - Late days don't cross Thanksgiving break.
  - **No checkpoint**, even though the syllabus says there's one.

Important Ideas for Today

- The material from today will lay the groundwork for the next few weeks.
- Key concepts:
  - **High-level specifications**
  - **R and RE languages.**
  - **Encodings.**
  - **Universal machines.**
  - **Nondeterministic Turing machines.**
- There is a ***lot*** of material today; *please do not hesitate to ask questions!*

High-Level Descriptions

The Church-Turing Thesis

- The Church-Turing thesis states that all effective models of computation are equivalent to or weaker than a Turing machine.
- As as a result, we can start to be less precise with our TM descriptions.

High-Level Descriptions

- A **high-level description** of a Turing machine is a description of the form

  M = "On input x:
  Do something with x."

- Example:

  M = "On input x:

  Repeat the following:

  If $|x| \leq 1$, accept.

  If the first and last symbols of x aren't the same, reject.

  Remove the first and last characters of x."

High-Level Descriptions

- A **high-level description** of a Turing machine is a description of the form

  M = "On input x:
  Do something with x."

- Example:

  M = "On input x:

  Construct y, the reverse of x.

  If $x = y$, accept.

  Otherwise, reject."

High-Level Descriptions

- A **high-level description** of a Turing machine is a description of the form

  M = "On input x:
  Do something with x."

- Example:

  M = "On input x:

  If x is a palindrome, accept.

  Otherwise, reject."

High-Level Descriptions

- A **high-level description** of a Turing machine is a description of the form

  M = "On input x:
  Do something with x."

- Example:

  M = "On input x:

  Check that x has the form $\mathtt{0}^n\mathtt{1}^m\mathtt{2}^p$.

  If not, reject.

  If $nm = p$, accept.

  Otherwise, reject."

High-Level Descriptions

- A **high-level description** of a Turing machine is a description of the form

  M = "On input x:
  Do something with x."

- Example:

  M = "On input x:

  Check that x has the form $p?t$,
  where $p, t \in \{0, 1\}^*$.

  If not, reject.

  If so, if p is a substring of t, accept.

  Otherwise, reject."

Formatted Input

- Many languages require the input to be in some particular format.
  - (Think about *ADD* or *SEARCH* from the problem sets).
- We can encode this directly into our TMs:

  M = "On input p`?`t, where p, $t \in$ {`0`, `1`}*

  If p is a substring of t, accept.

  Otherwise, reject."
- Machines of this form implicitly reject any inputs that don't have the right format.

Formatted Input

- Many languages require the input to be in some particular format.
  - (Think about *ADD* or *SEARCH* from the problem sets).
- We can encode this directly into our TMs:

  M = "On input $\mathtt{0}^m\mathtt{1}^n\mathtt{2}^p$:

  If $mn = p$, accept.

  Otherwise, reject."
- Machines of this form implicitly reject any inputs that don't have the right format.

What's Allowed?

- General rule of thumb:

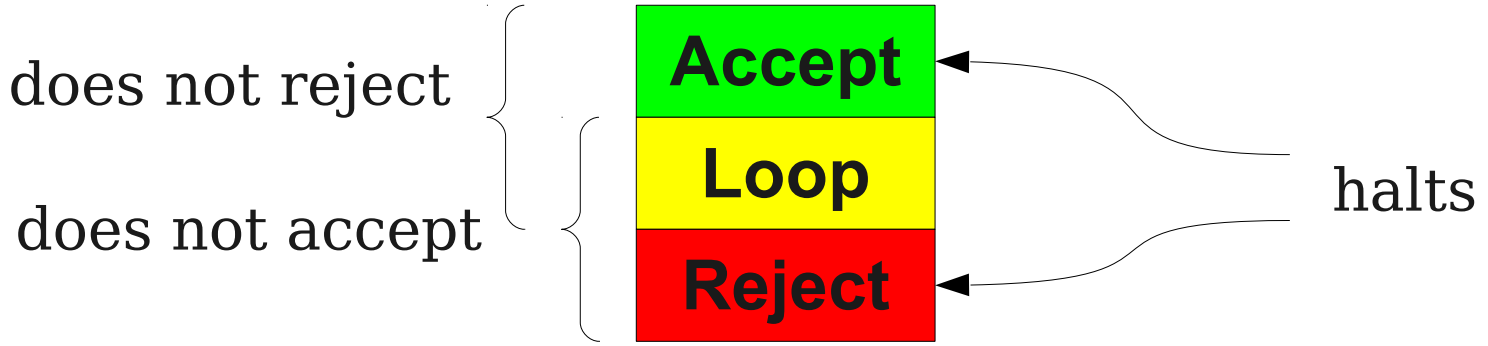
You can include *anything* in a high-level description, as long as you could write a computer program for it.

- A few exceptions: no user input, no randomness, etc.
- This is a consequence of the Church-Turing thesis.

R and **RE** Languages

Some Important Terminology

- A TM **accepts** a string *w* if it enters an accept state.
- A TM **rejects** a string *w* if it enters a reject state.
- A TM **loops infinitely** (or just **loops**) on a string *w* if neither of these happens.
- A TM **does not accept** a string *w* if it either rejects *w* or loops infinitely on *w*.
- A TM **does not reject** a string *w* if it either accepts *w* or loops infinitely on *w*.
- A TM **halts** if it accepts or rejects.



Recall: Language of a TM

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

$$\mathscr{L}(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

- For any $w \in \mathscr{L}(M)$, M accepts w.
- For any $w \notin \mathscr{L}(M)$, M does 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.
- Notation: **RE** is the set of all recognizable languages.

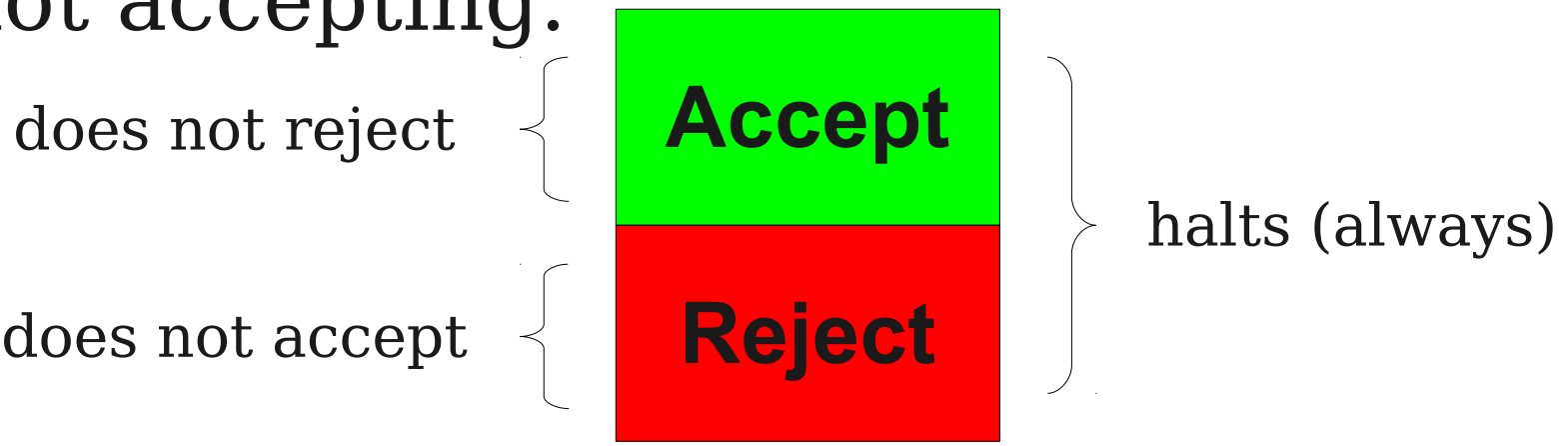
$L \in \mathbf{RE}$ iff L is recognizable

Why “Recognizable?”

- Given TM M with language $\mathscr{L}(M)$, running M on a string w will not necessarily tell you whether $w \in \mathscr{L}(M)$.
- 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 $w \in \mathscr{L}(M)$, then the machine can confirm this (it eventually accepts).
- The machine can't *decide* whether or not $w \in \mathscr{L}(M)$, 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 $\mathscr{L}(M) = L$.
  - These languages are also sometimes called **recursive**.
- Given a decider M, you *can* learn whether or not a string $w \in \mathscr{L}(M)$.
  - Run M on w.
  - Although it make take a staggeringly long time, M will eventually accept or reject w.
- The set **R** is the set of all decidable languages.

$w \in \mathbf{R}$ iff w is decidable

Why **R** Matters

- If a language is in **R**, there is an algorithm that can decide membership in that language.
  - Run the decider and see what it says.
- If there is an algorithm that can decide membership in a language, that language is in **R**.
  - By the Church-Turing thesis, any effective model of computation is equivalent in power to a Turing machine.
  - Thus if there is *any* algorithm for deciding membership in the language, there must be a decider for it.
  - Thus the language is in **R**.
- **A language is in R iff there is an algorithm for deciding membership in that language.**

$\mathbf{R} \stackrel{?}{=} \mathbf{RE}$

- Every decider is a Turing machine, but not every Turing machine is a decider.
- Thus $\mathbf{R} \subseteq \mathbf{RE}$.
- Hugely important theoretical question:

Is $\mathbf{R} = \mathbf{RE}$?

- That is, if we can *verify* that a string is in a language, can we *decide* whether that string is in the language?
- We will need to build up some more formalisms to answer this question.

Encodings

An Interesting Observation

- **WB** programs give us a way to check whether a string is contained within a given **RE** 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.
- It's always possible to encode a string in any alphabet using just two symbols.
  - This is how real computers work.

01000100	01001001	01001011	01000100	01001001	01001011
D	I	K	D	I	K

Notation for Encodings

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

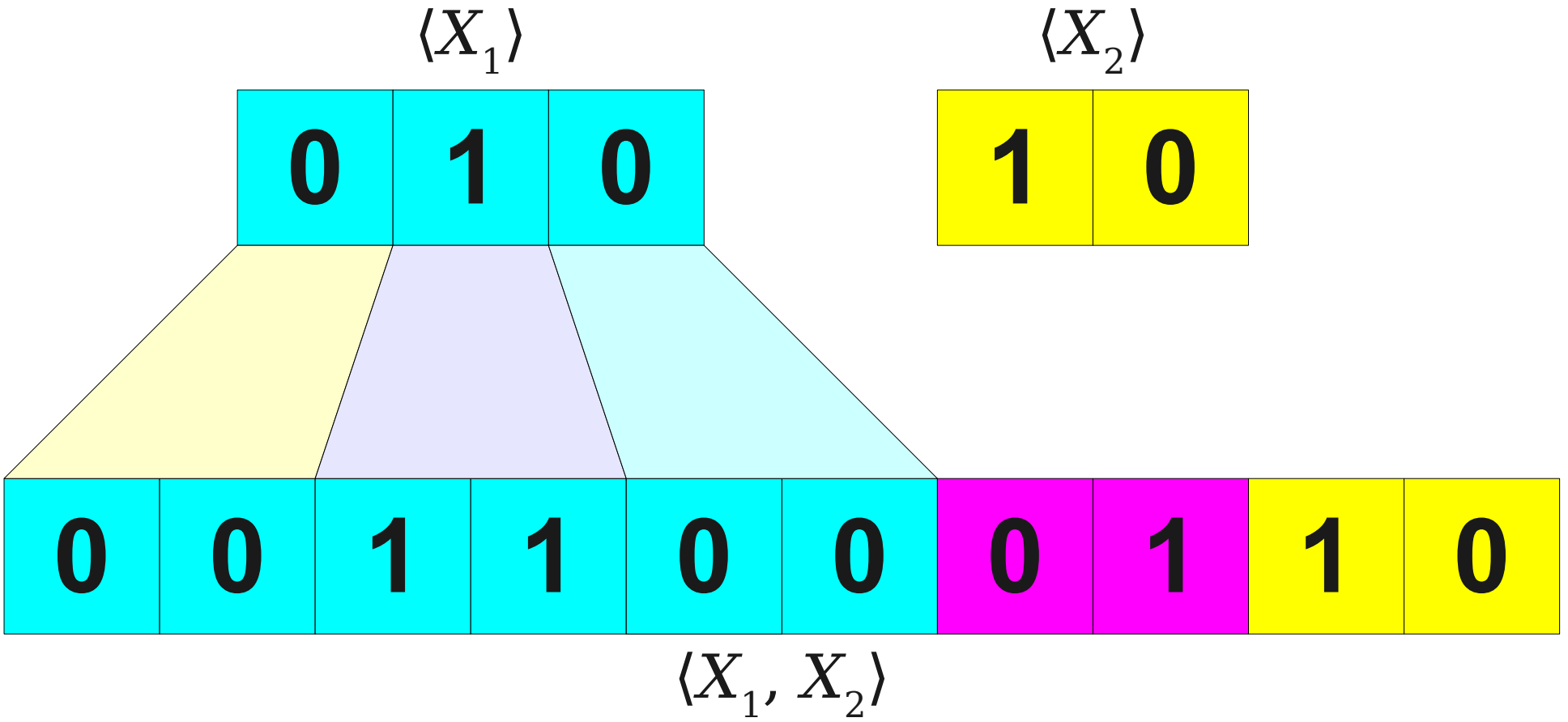
Encoding Other Automata

- 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 Turing machines.
  - A Turing machine and a string.
  - A graph and a path in the graph.
  - A rock and a hard place.
  - Guns and Roses.
  - Repeal and Replace.
  - “I just met you” and “this is crazy.”
- 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 M is a Turing machine and w is a string, then $\langle M, w \rangle$ is an encoding of that program and that string.
  - If G is a context-free grammar and P is a PDA, then $\langle G, P \rangle$ is an encoding of that grammar and that PDA.

We can now encode TMs as strings.

TMs can accept strings as input.

What can we do with this knowledge?

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.
- As a high-level description:

  U_{TM} = "On input $\langle M, w \rangle$, where M is a TM and $w \in \Sigma^*$

  Run M on w.

  If M accepts w, accept.

  If M rejects w, reject."

If M loops on w, then U_{TM} loops as well. This is usually omitted from the description.

Sketch of the Universal **WB** Program

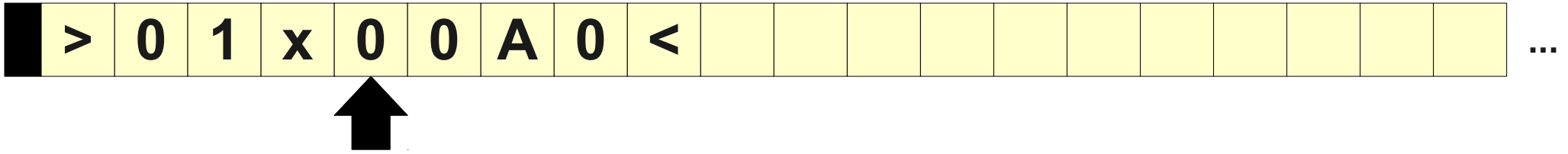
- It is a bit easier to understand the universal **WB** program than the universal TM.
- We will write the universal **WB** program in **WB6** (finite variables, multiple tracks, multiple tapes, multiple stacks) for simplicity, and can then “compile” it down into normal **WB**.

Sketch of the Universal **WB** Program

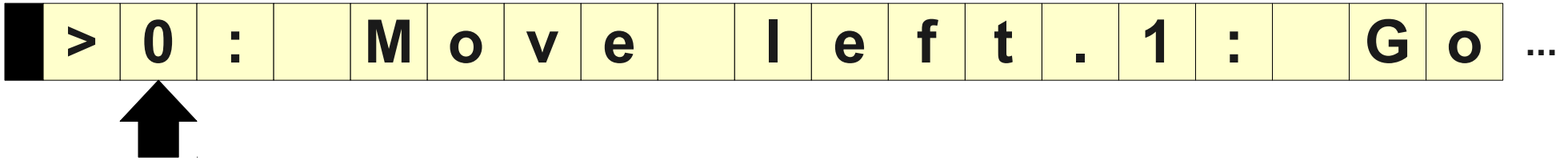
- To run a **WB** program, at each point in time we need to track three pieces of information:
  - The contents of the tape.
  - The location of the read head.
  - The index of the current instruction.
- Our program will enter a loop and repeatedly do the following:
  - Find the next instruction to execute.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

Sketch of the Universal **WB** Program

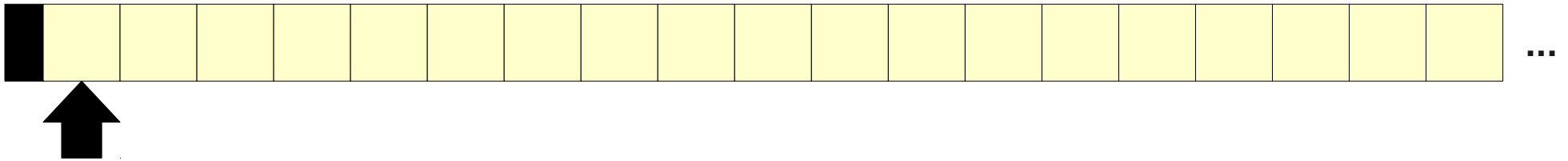
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

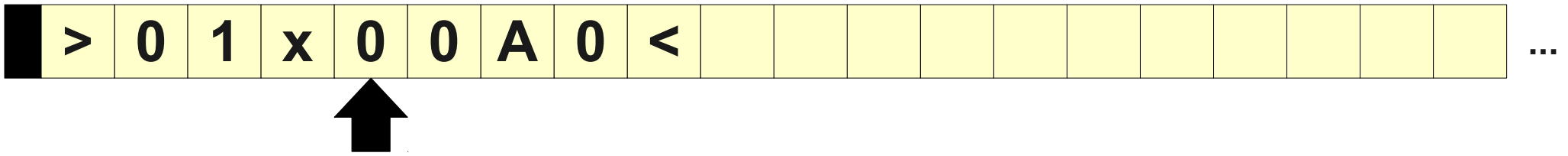


Variables for intermediate storage.

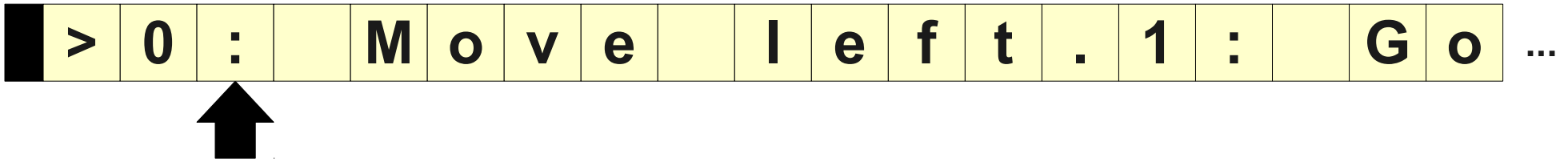
Instr **Letter**

Sketch of the Universal **WB** Program

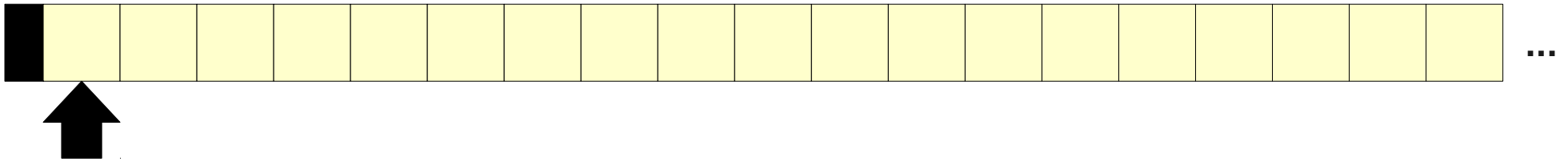
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

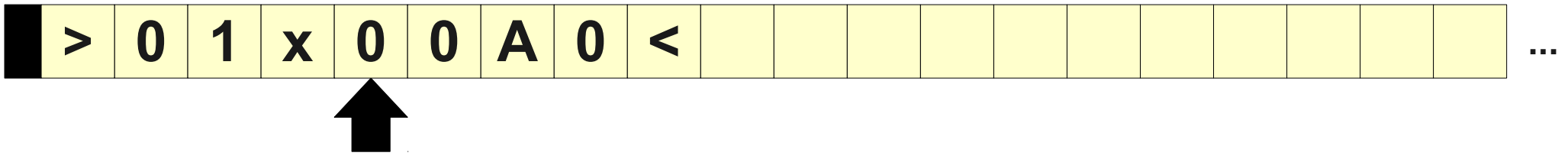


Variables for intermediate storage.

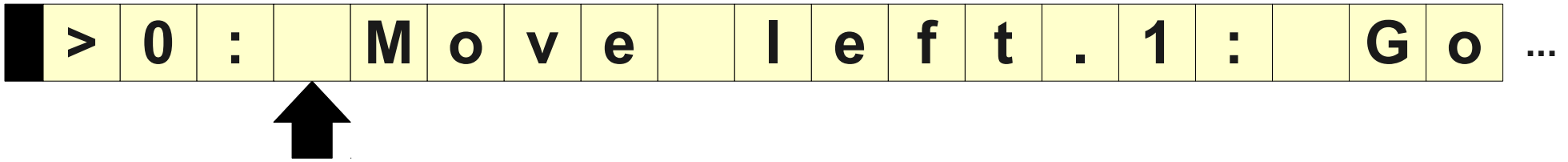
Instr **Letter**

Sketch of the Universal **WB** Program

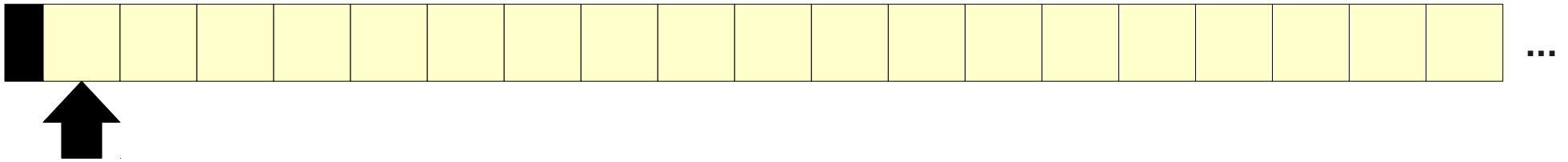
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

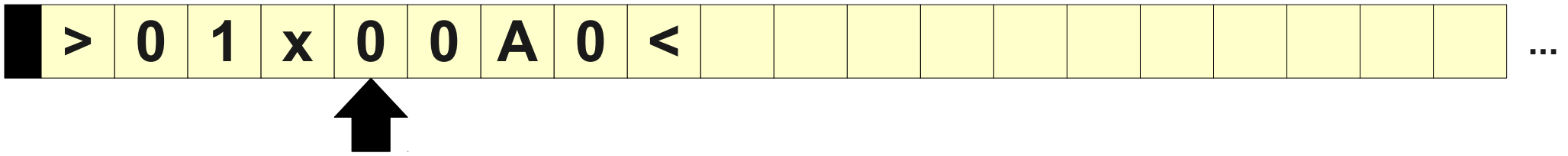


Variables for intermediate storage.

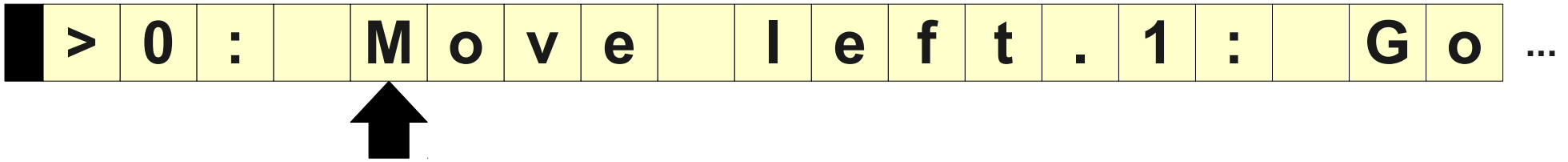
Instr **Letter**

Sketch of the Universal **WB** Program

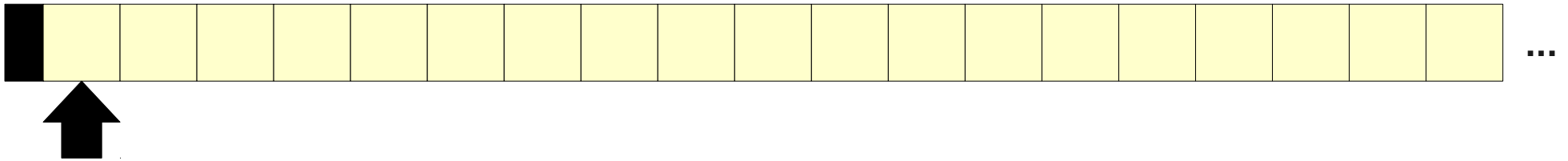
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

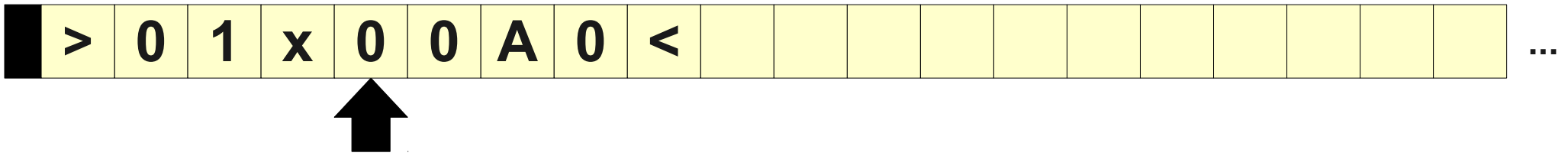


Variables for intermediate storage.

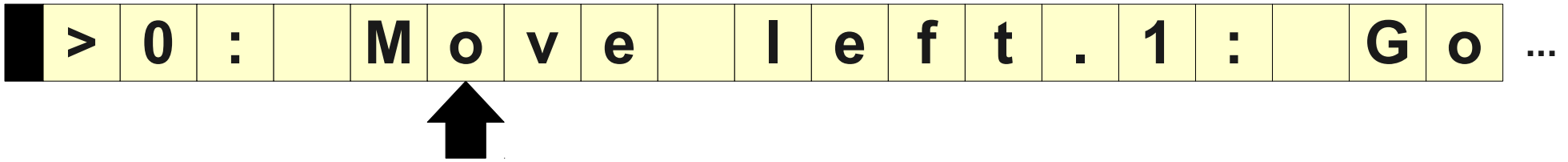
Instr **Letter**

Sketch of the Universal **WB** Program

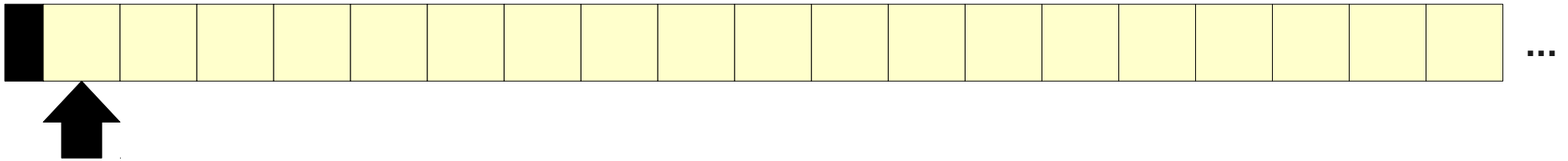
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

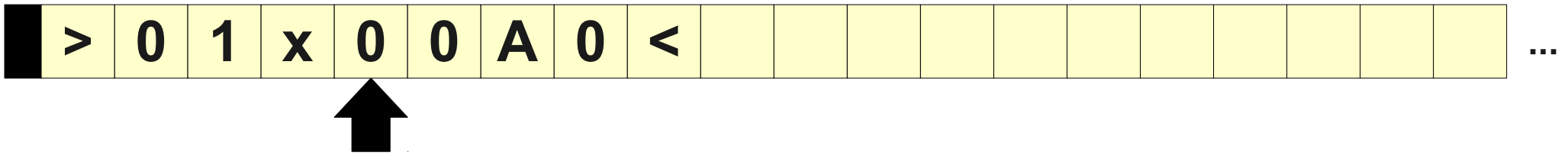


Variables for intermediate storage.

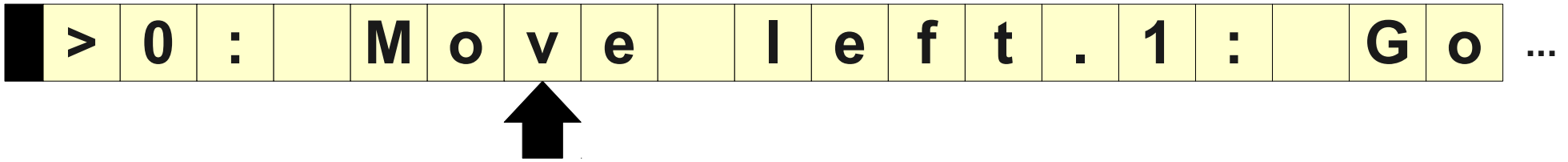
Instr **Letter**

Sketch of the Universal **WB** Program

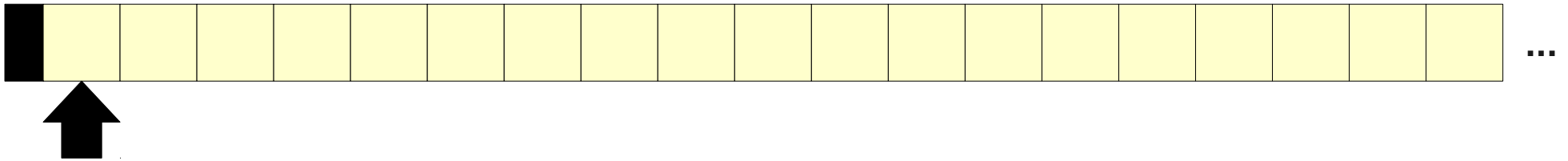
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

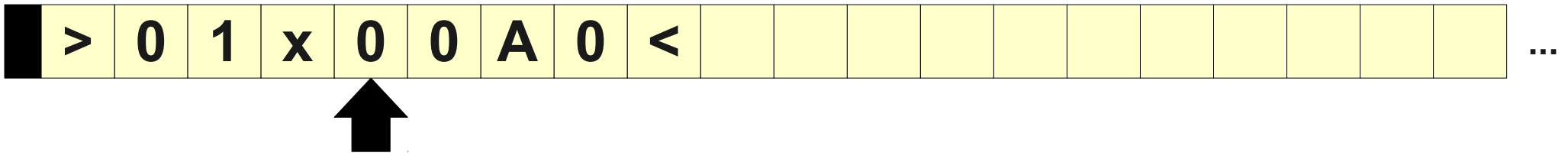


Variables for intermediate storage.

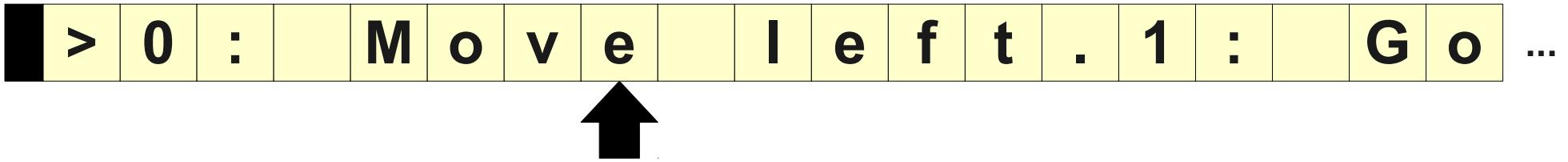
Instr **Letter**

Sketch of the Universal **WB** Program

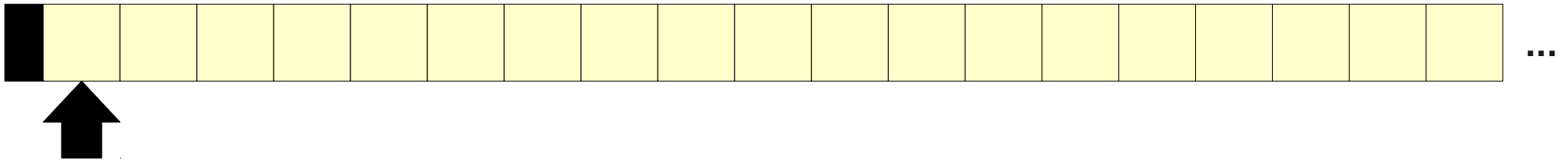
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

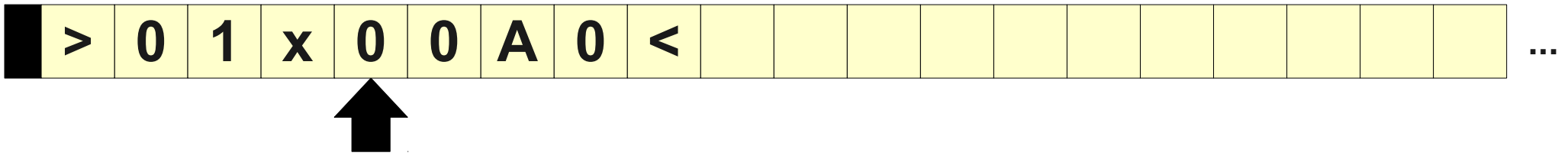


Variables for intermediate storage.

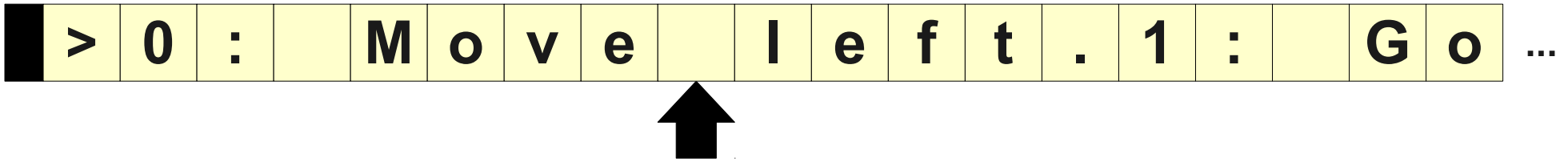
Instr **Letter**

Sketch of the Universal **WB** Program

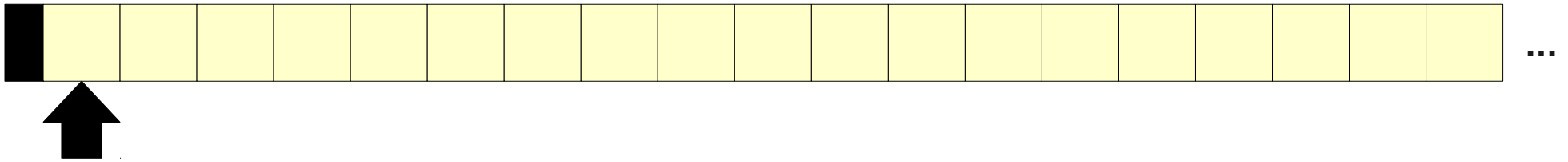
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

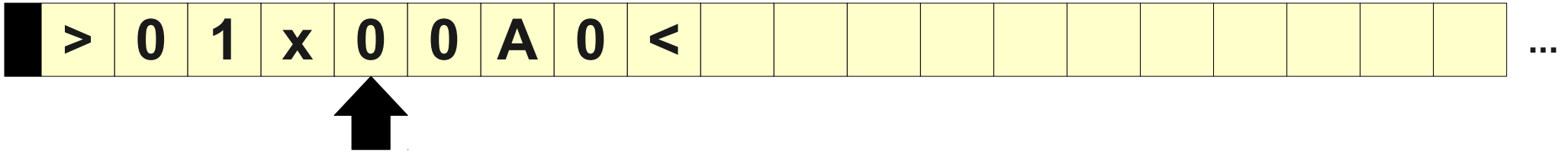


Variables for intermediate storage.

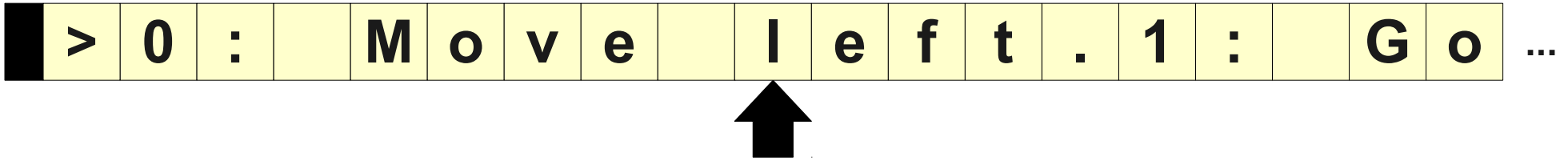
Instr **Letter**

Sketch of the Universal **WB** Program

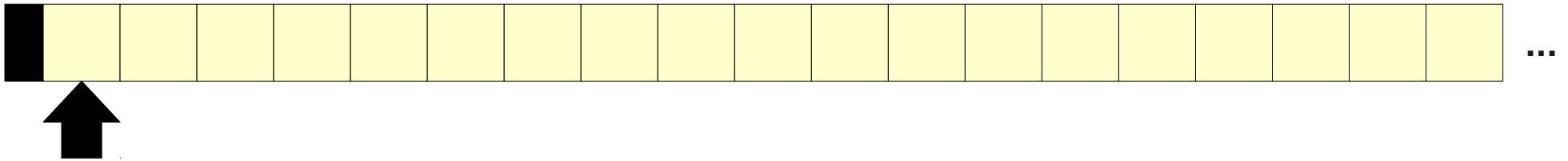
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

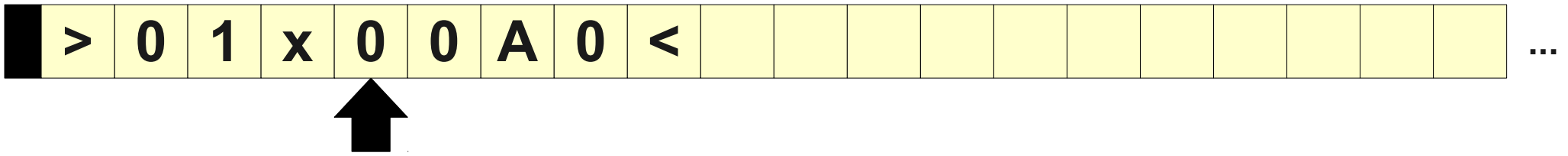


Variables for intermediate storage.

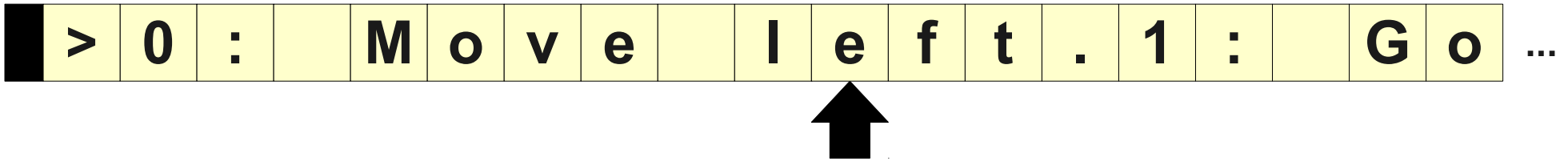
Instr **Letter**

Sketch of the Universal **WB** Program

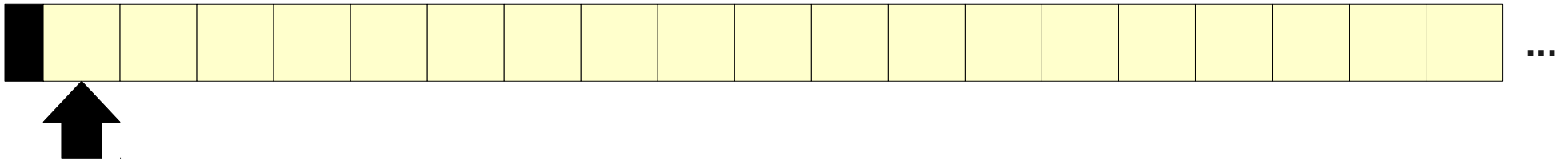
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

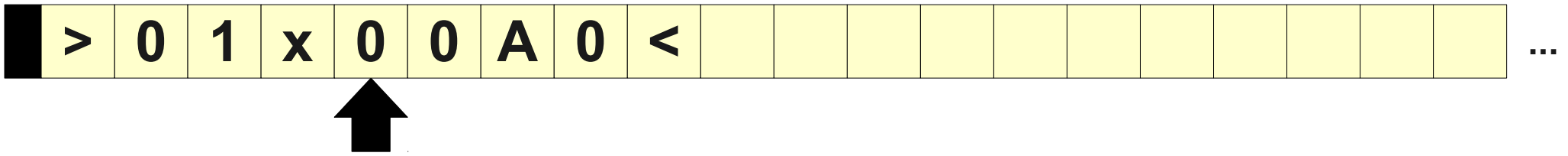


Variables for intermediate storage.

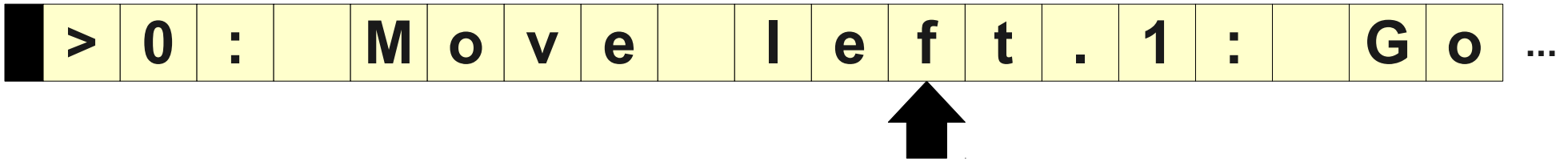
Instr **Letter**

Sketch of the Universal **WB** Program

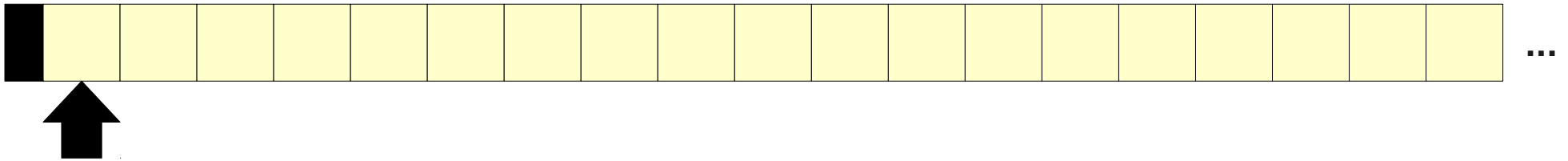
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

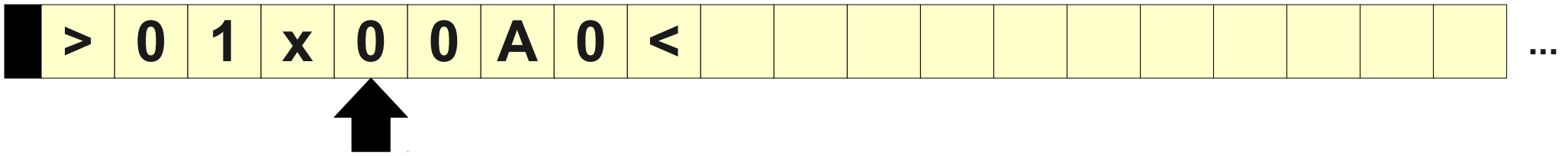


Variables for intermediate storage.

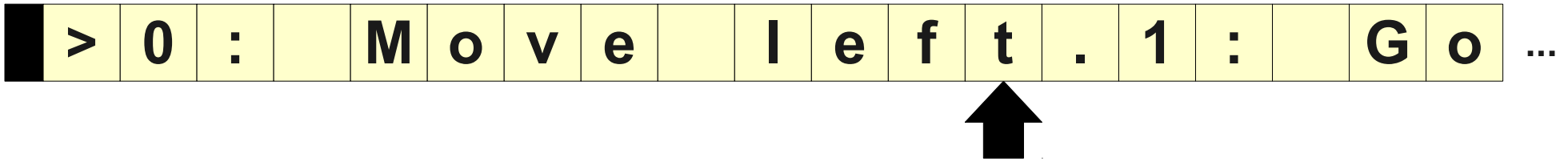
Instr **Letter**

Sketch of the Universal **WB** Program

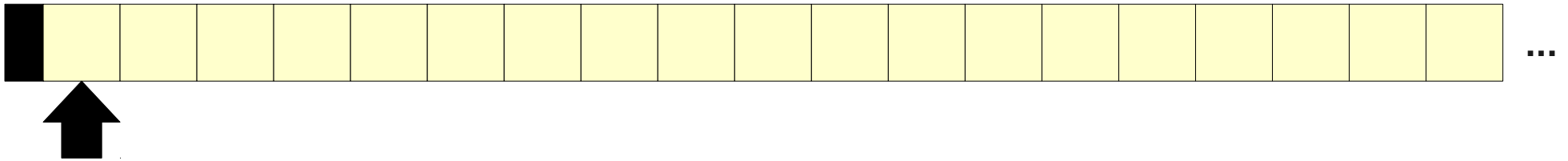
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

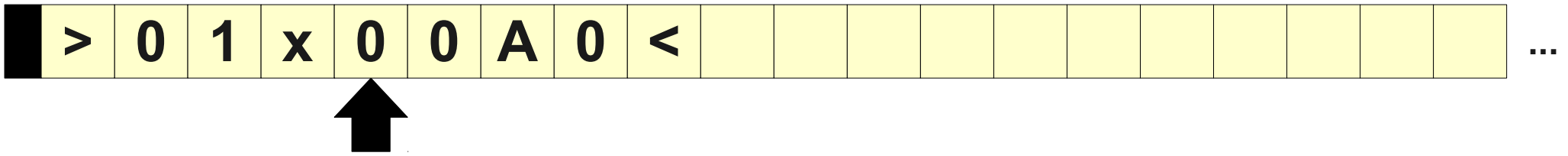


Variables for intermediate storage.

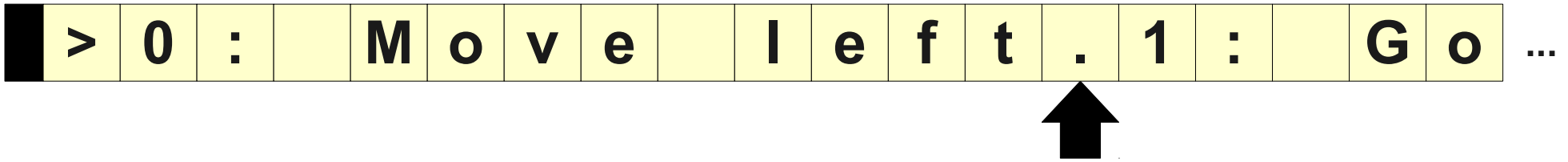
Instr **Letter**

Sketch of the Universal **WB** Program

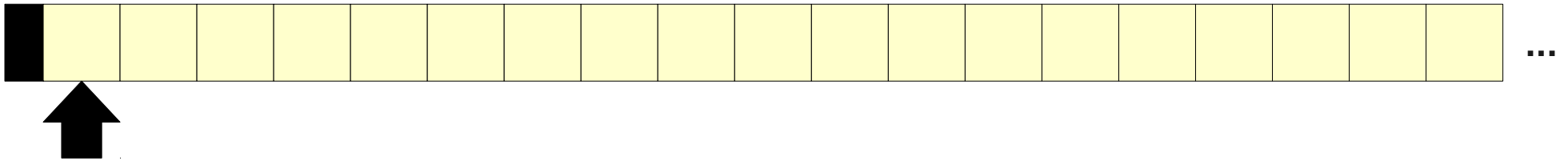
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

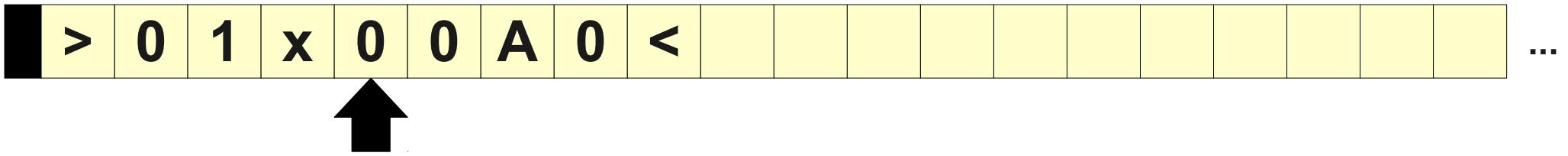


Variables for intermediate storage.

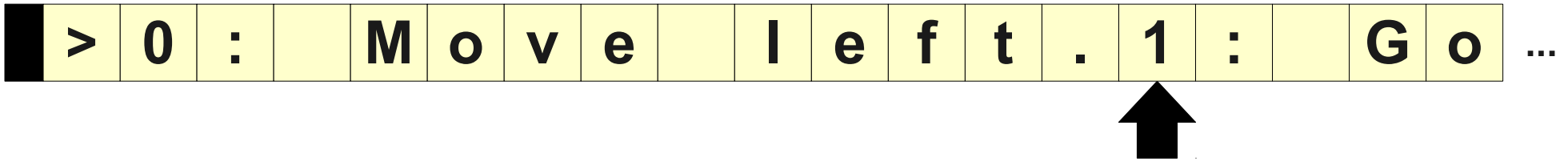
Instr **Letter**

Sketch of the Universal **WB** Program

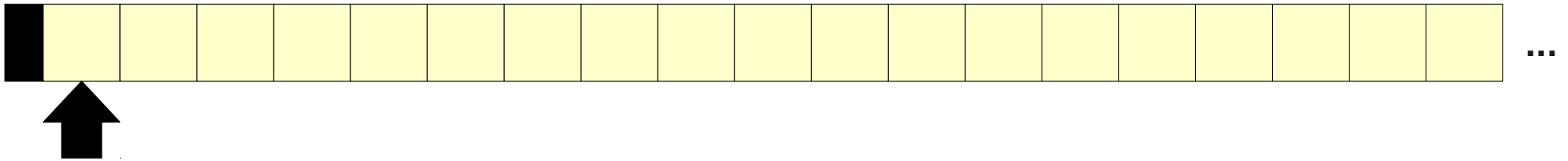
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

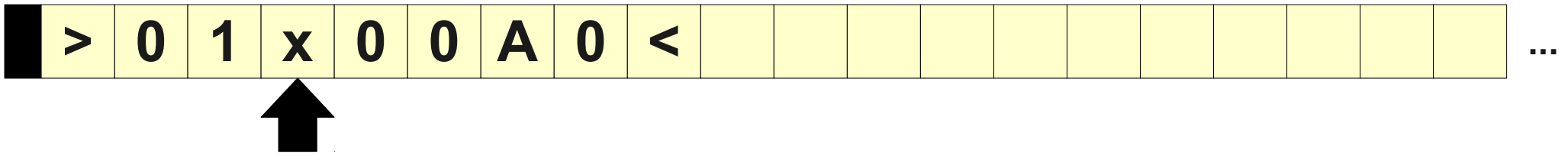


Variables for intermediate storage.

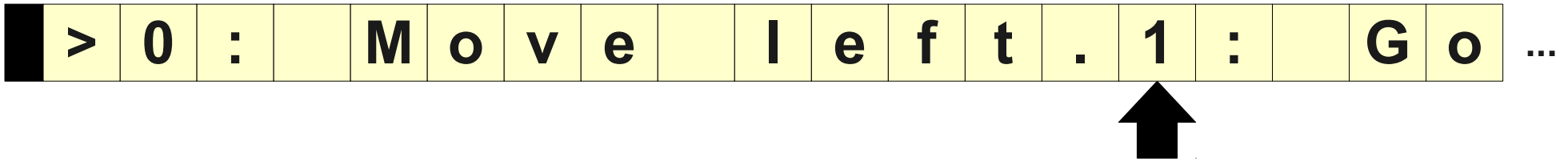
Instr **ML** **Letter**

Sketch of the Universal **WB** Program

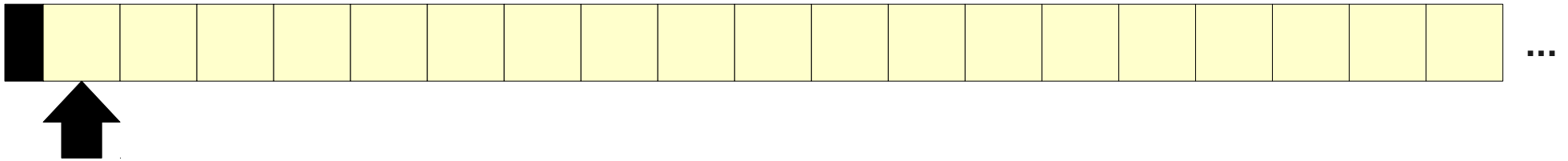
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

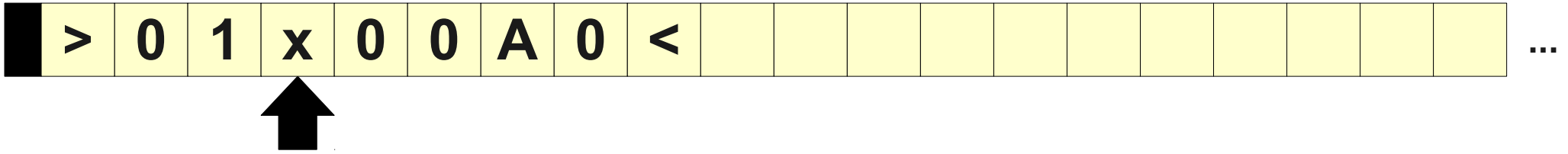


Variables for intermediate storage.

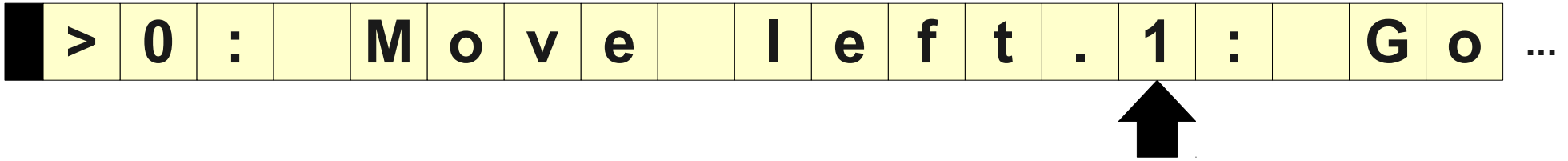
Instr **ML** **Letter**

Sketch of the Universal **WB** Program

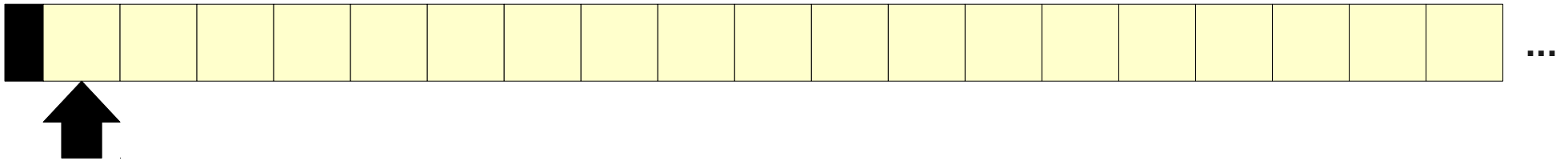
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

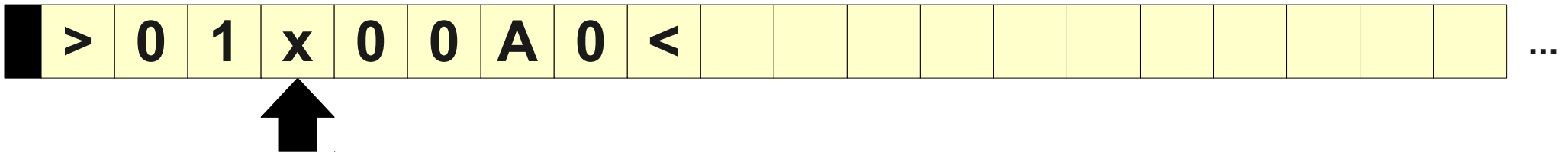


Variables for intermediate storage.

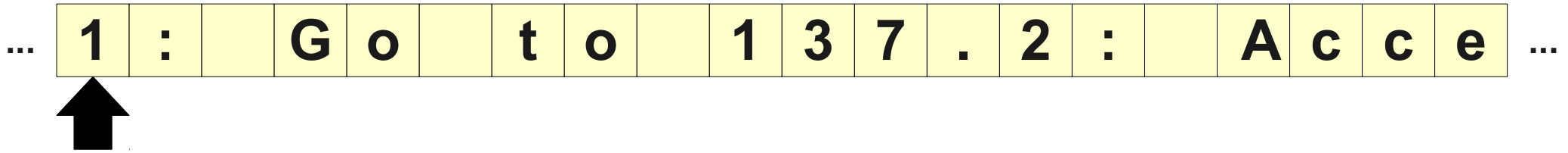
Instr **Letter**

Sketch of the Universal **WB** Program

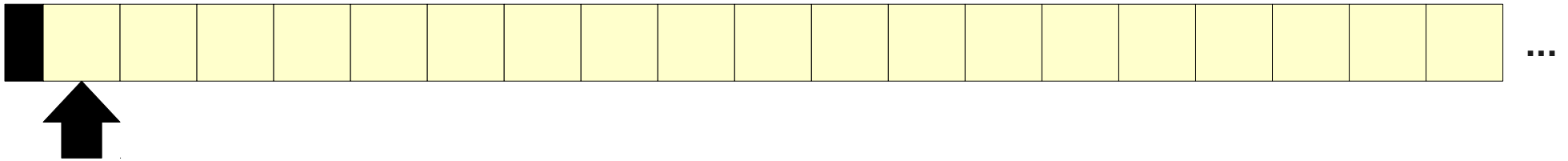
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

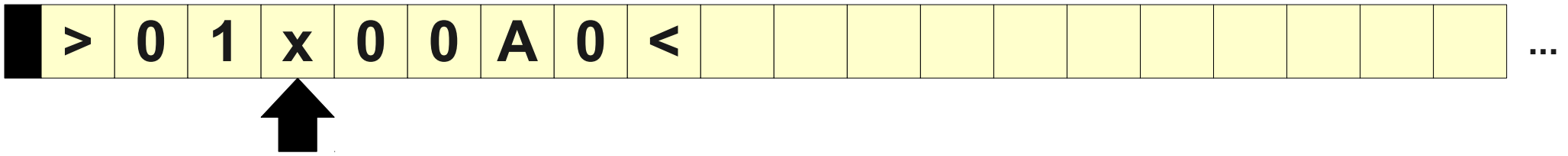


Variables for intermediate storage.

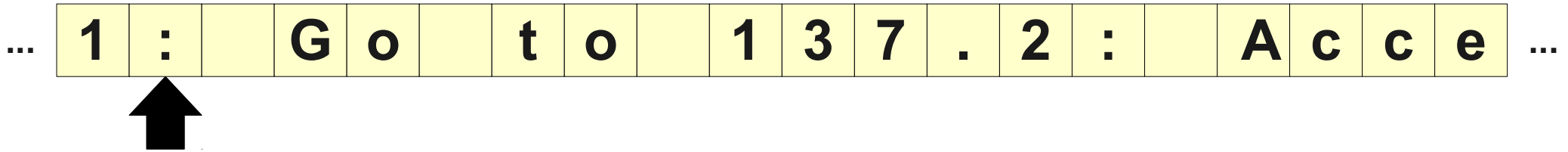
Instr **Letter**

Sketch of the Universal **WB** Program

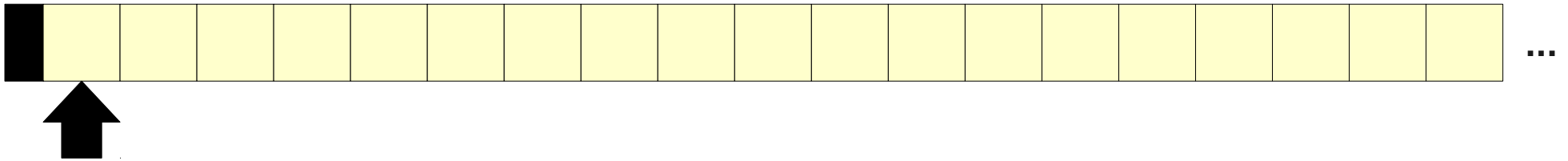
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

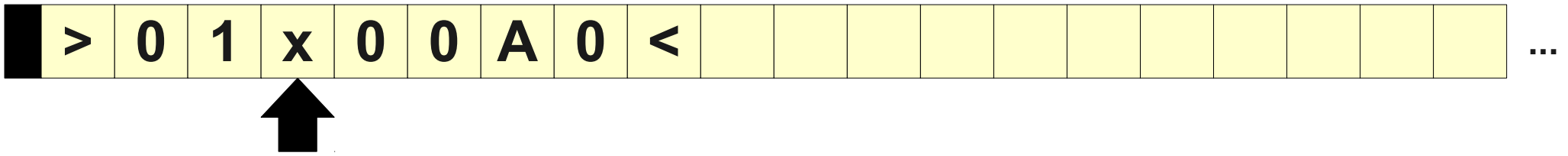


Variables for intermediate storage.

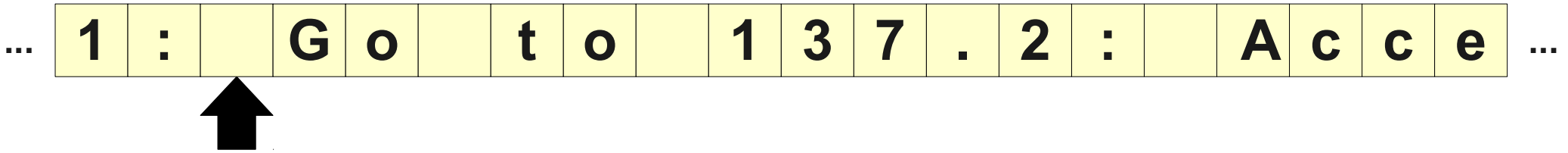
Instr **Letter**

Sketch of the Universal **WB** Program

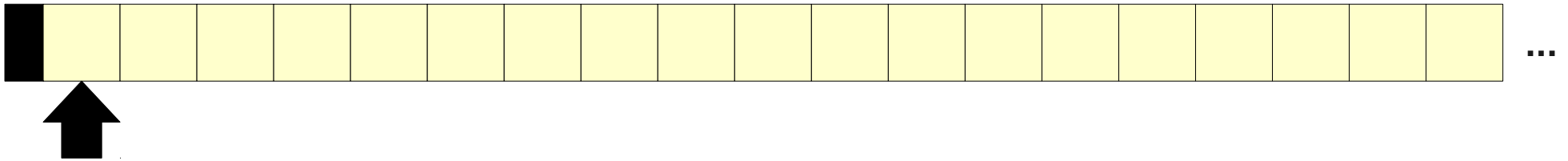
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

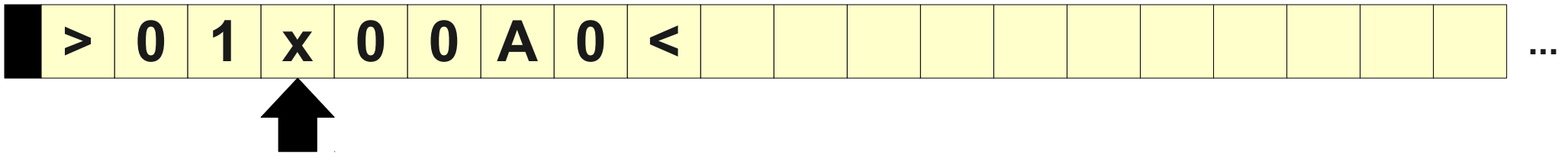


Variables for intermediate storage.

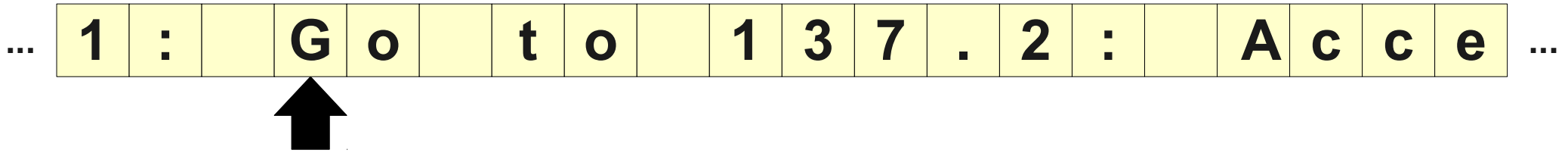
Instr **Letter**

Sketch of the Universal **WB** Program

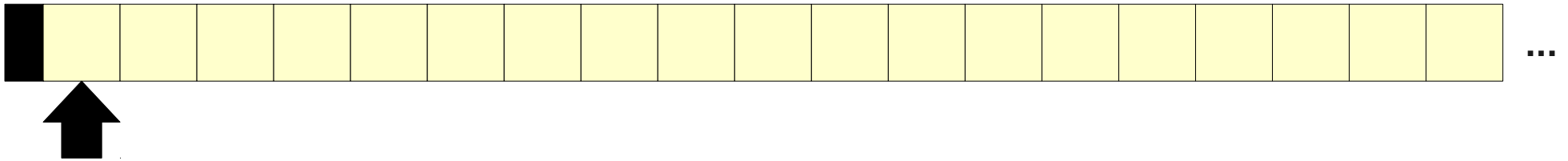
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

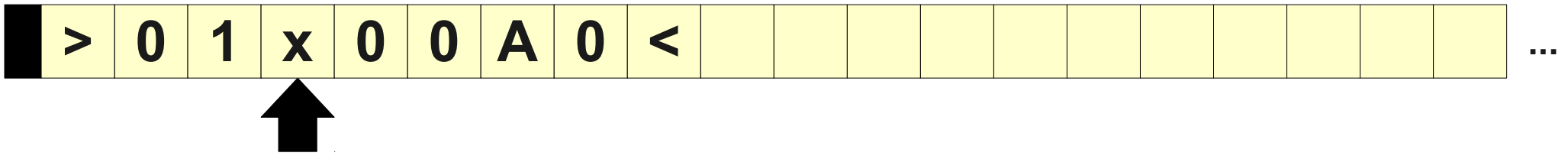


Variables for intermediate storage.

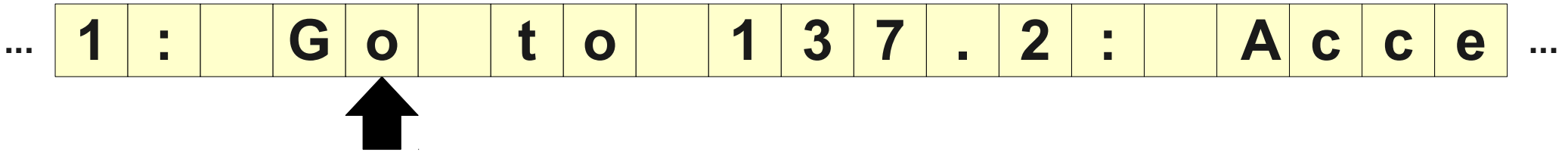
Instr **Letter**

Sketch of the Universal **WB** Program

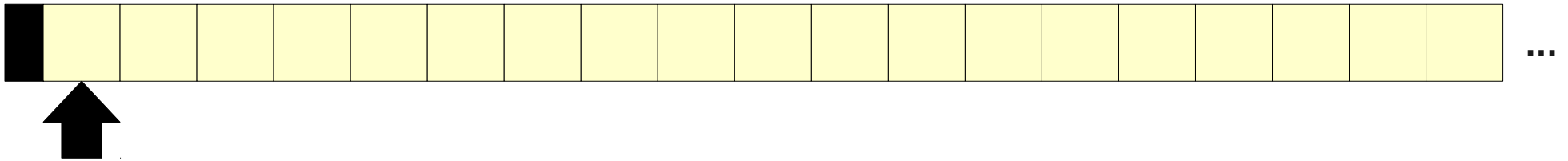
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

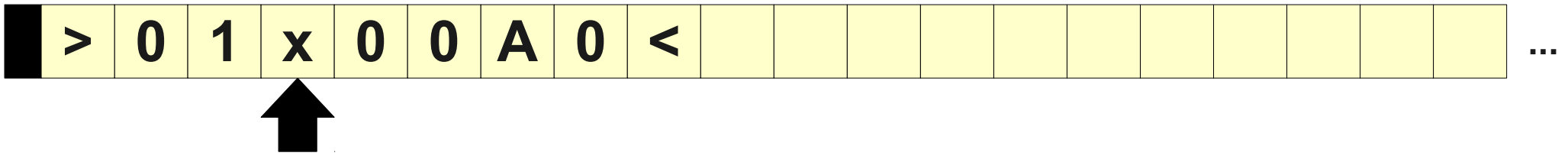


Variables for intermediate storage.

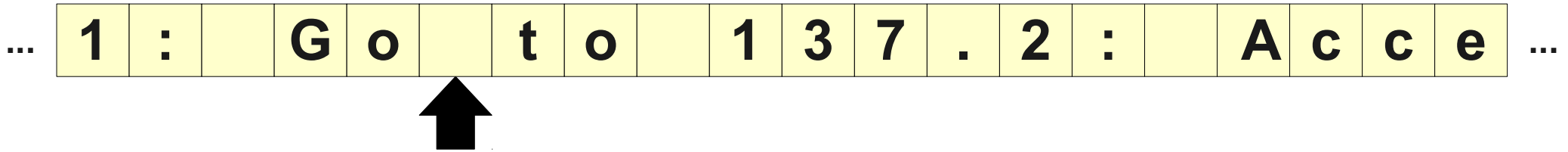
Instr **Letter**

Sketch of the Universal **WB** Program

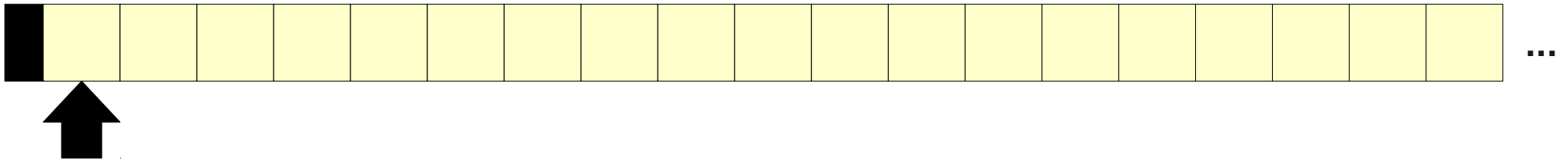
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

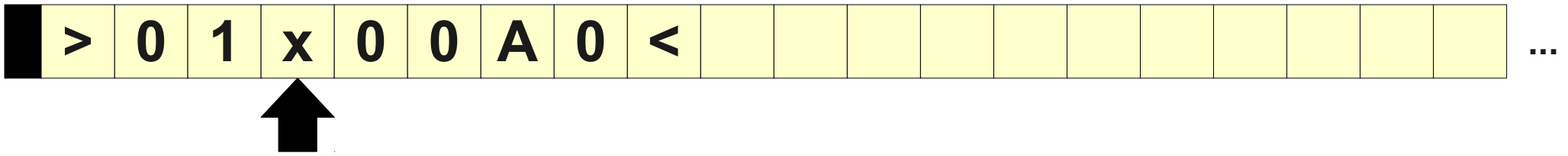


Variables for intermediate storage.

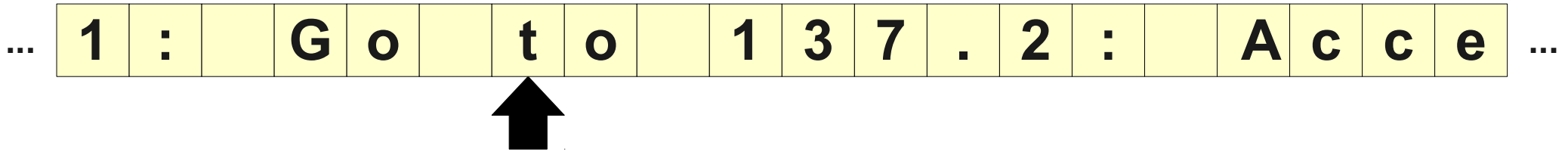
Instr **Letter**

Sketch of the Universal **WB** Program

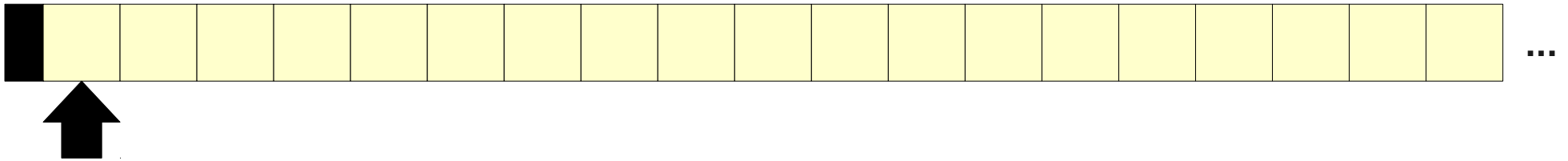
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

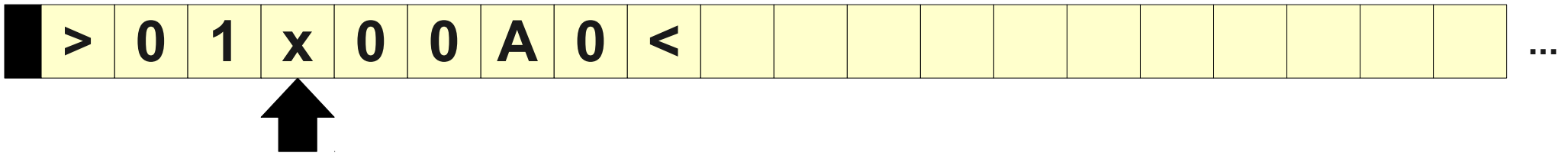


Variables for intermediate storage.

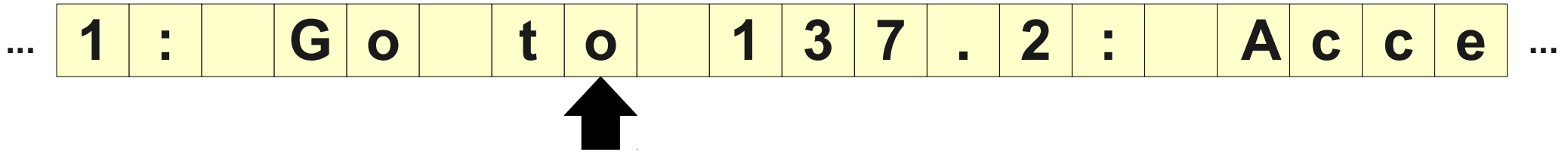
Instr **Letter**

Sketch of the Universal **WB** Program

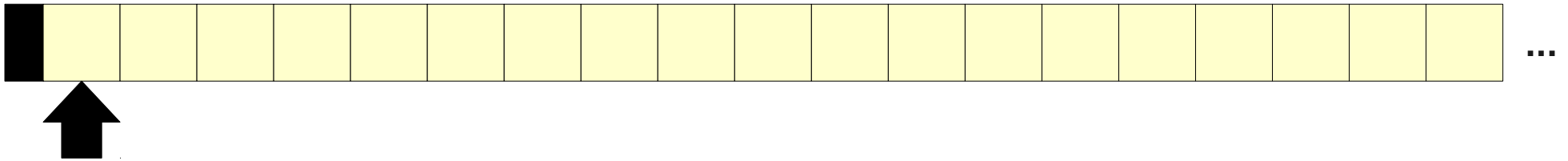
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

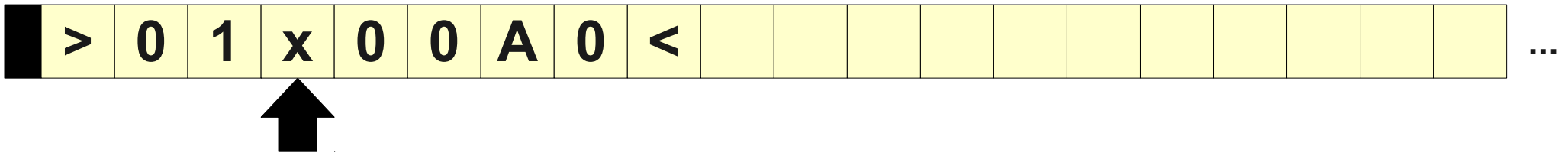


Variables for intermediate storage.

Instr **Letter**

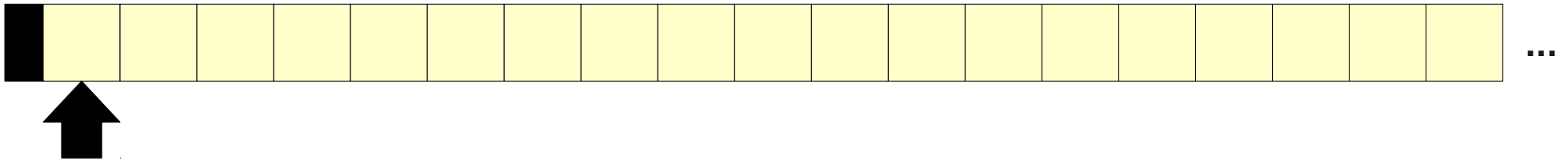
Sketch of the Universal **WB** Program

Simulated tape of the program being executed.



Program tape holding the program being executed.

Scratch tape for intermediate computation.

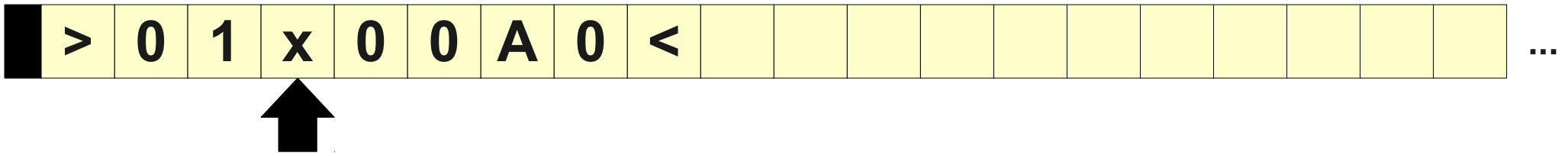


Variables for intermediate storage.

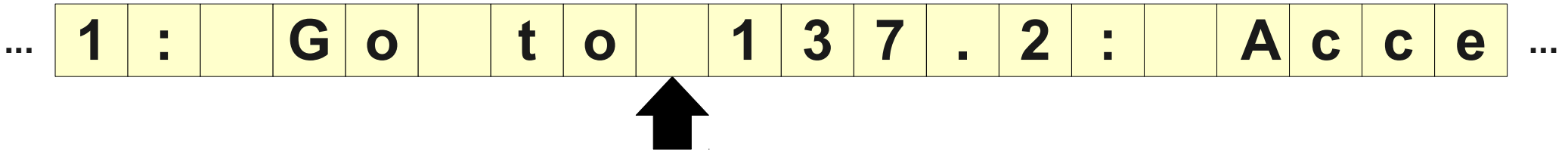
Instr **Letter**

Sketch of the Universal **WB** Program

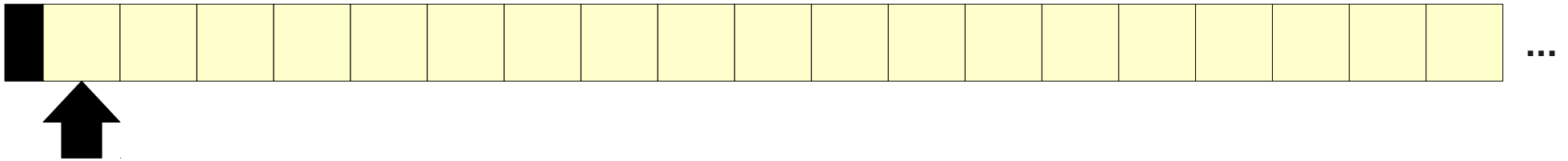
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

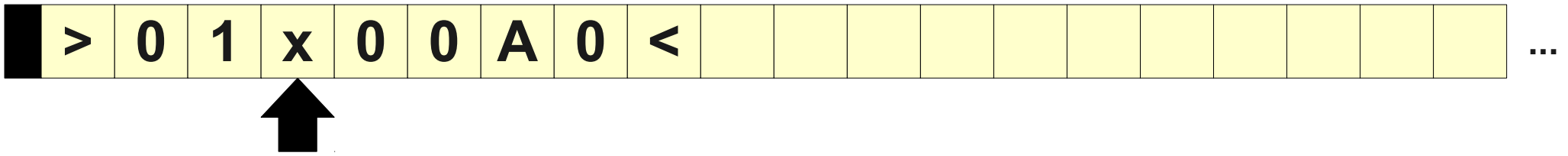


Variables for intermediate storage.

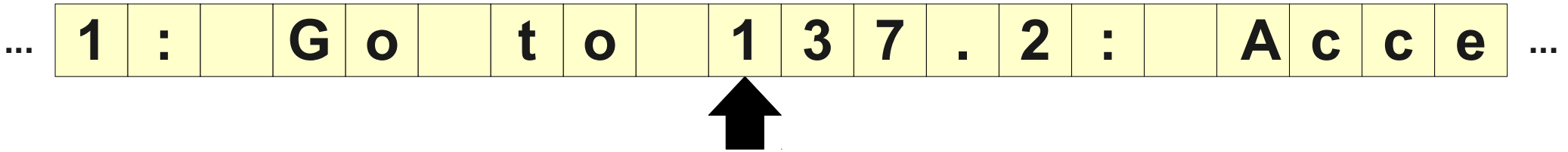
Instr **GoTo** **Letter**

Sketch of the Universal **WB** Program

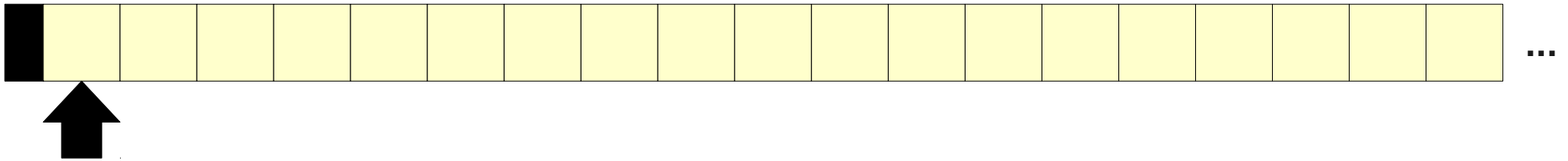
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

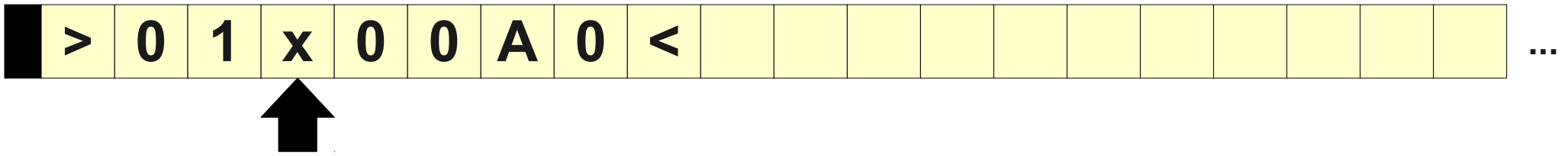


Variables for intermediate storage.

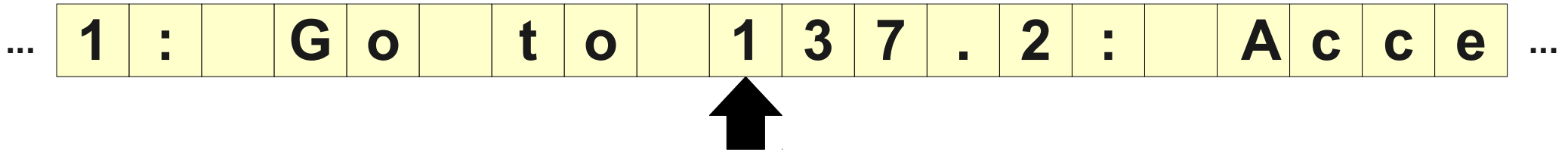
Instr **GoTo** **Letter**

Sketch of the Universal **WB** Program

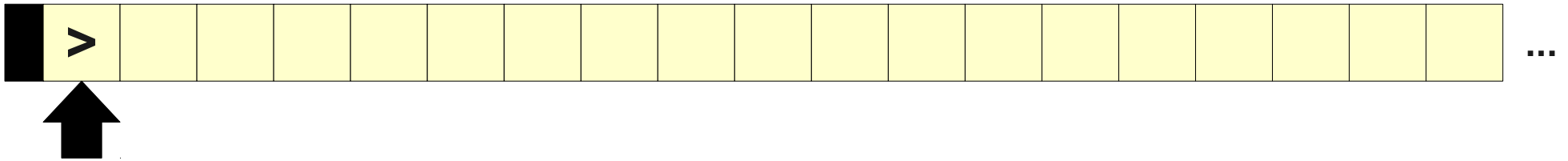
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

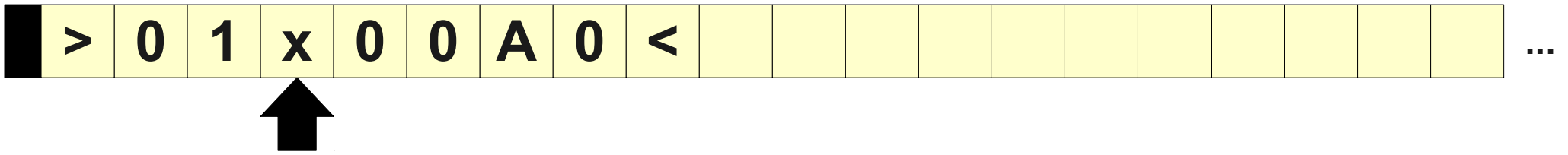


Variables for intermediate storage.

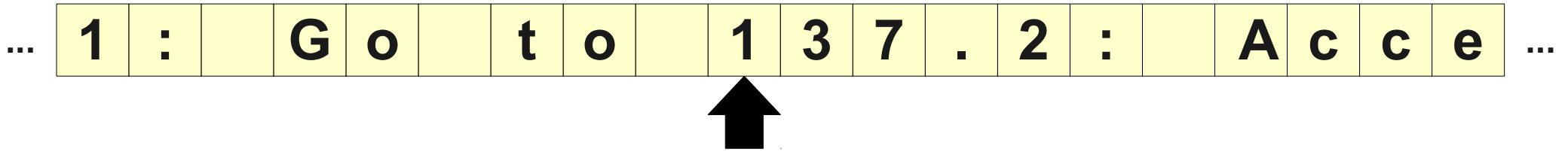
Instr **GoTo** **Letter**

Sketch of the Universal **WB** Program

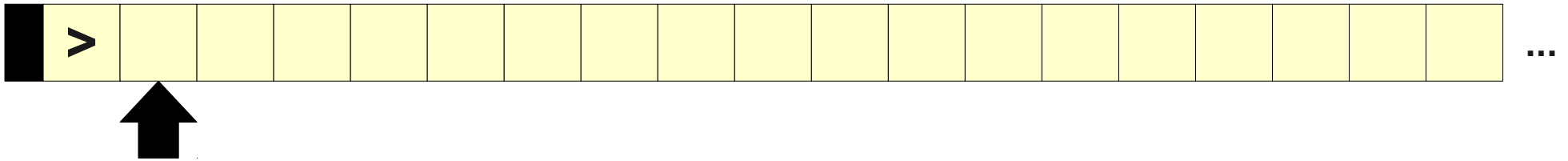
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

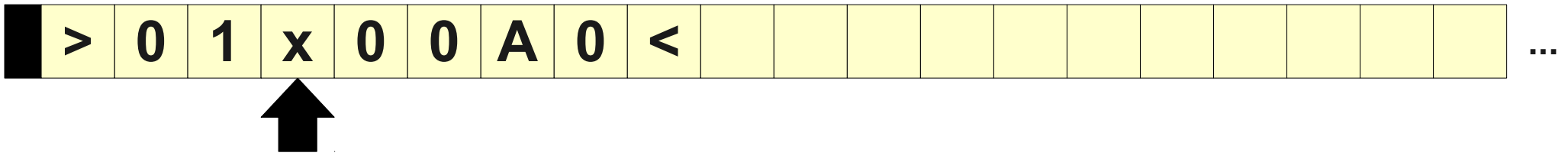


Variables for intermediate storage.

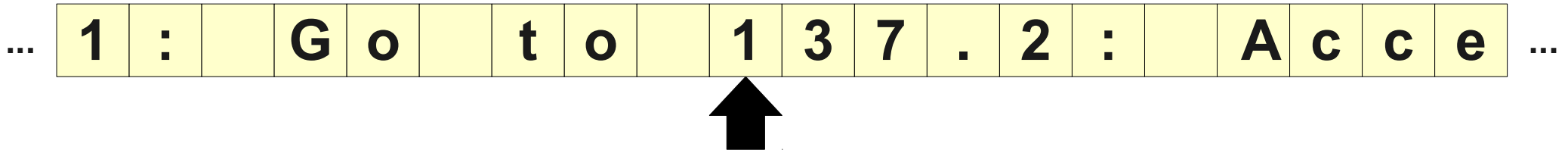
Instr **GoTo** **Letter**

Sketch of the Universal **WB** Program

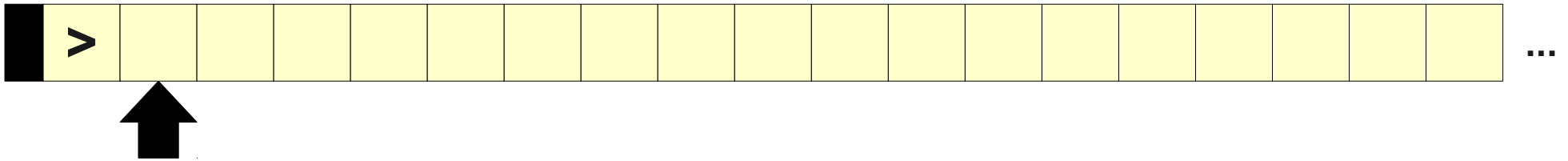
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

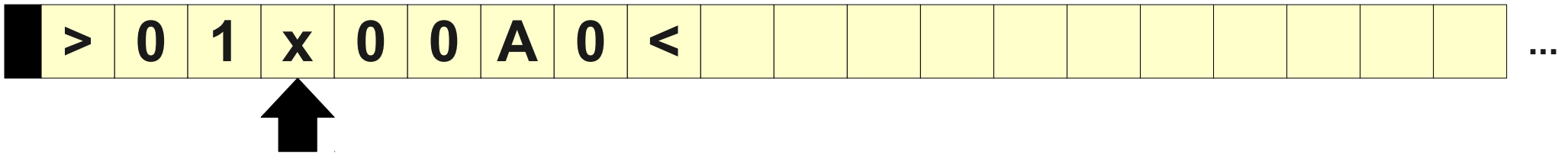


Variables for intermediate storage.

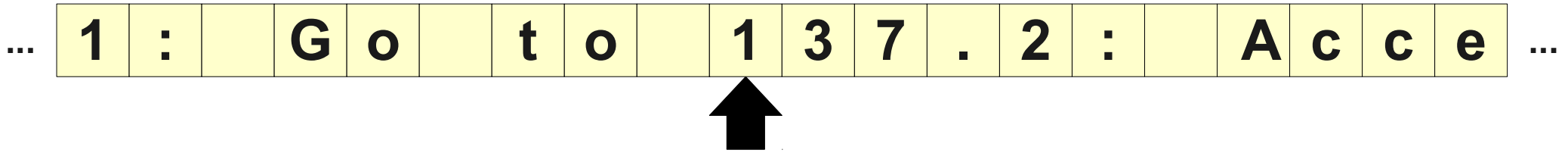
Instr GoTo **Letter** 1

Sketch of the Universal **WB** Program

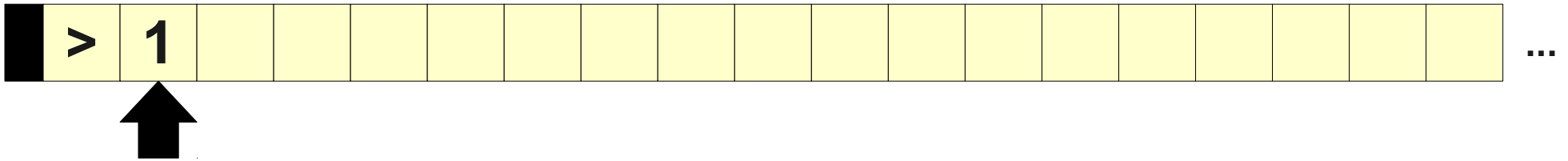
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

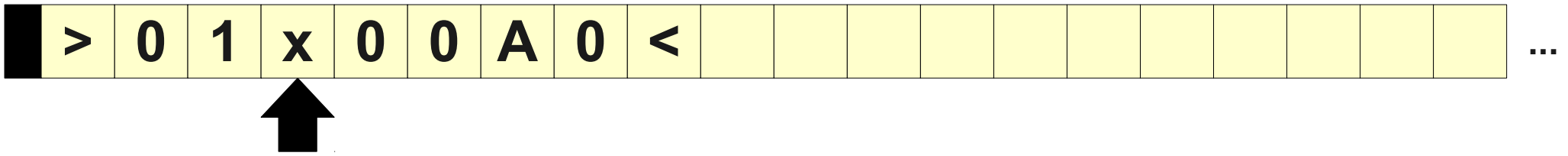


Variables for intermediate storage.

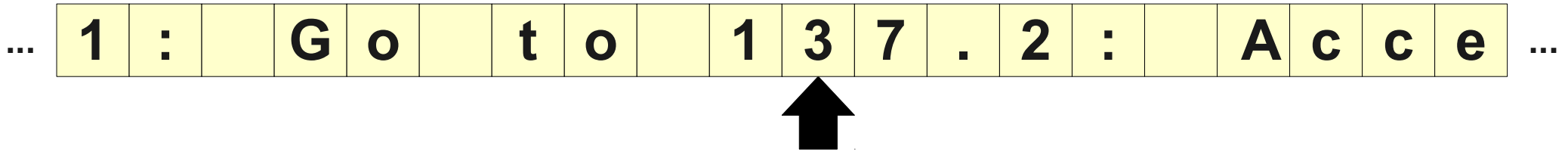
Instr **GoTo** **Letter** **1**

Sketch of the Universal **WB** Program

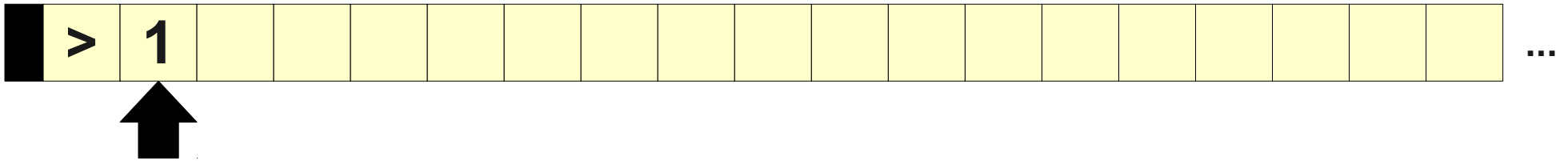
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

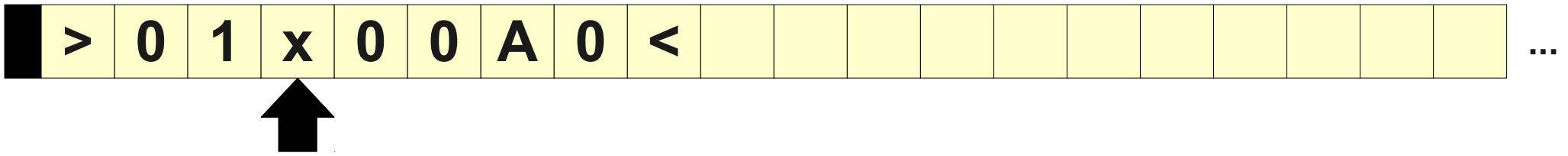


Variables for intermediate storage.

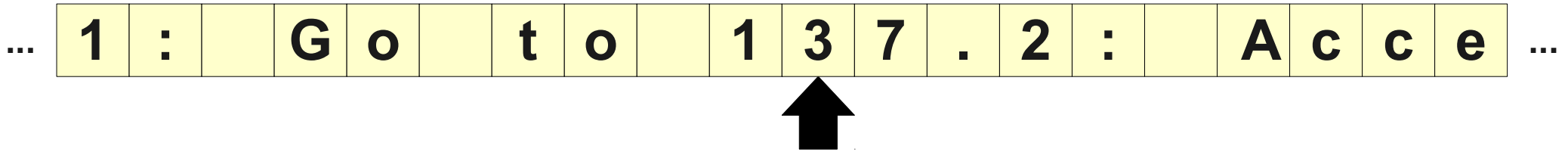
Instr GoTo **Letter** 1

Sketch of the Universal **WB** Program

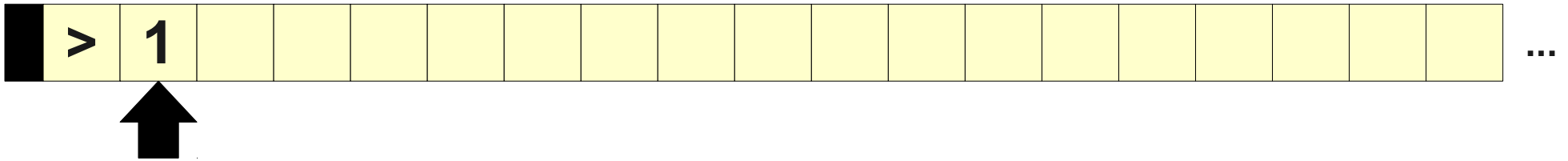
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

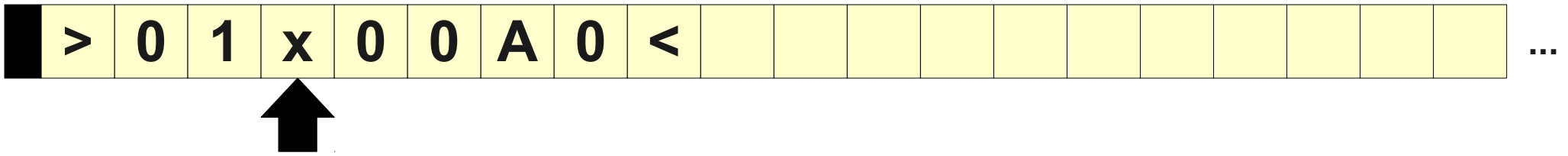


Variables for intermediate storage.

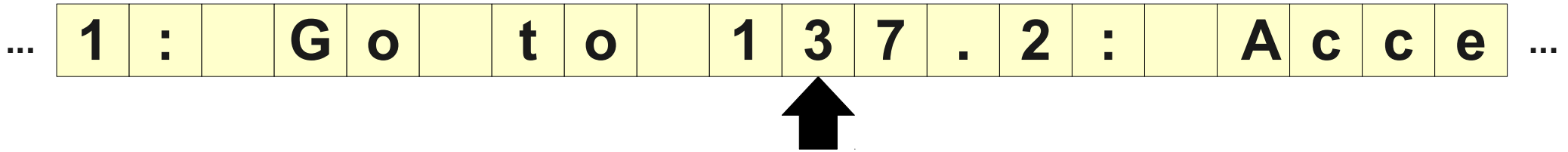
Instr GoTo **Letter** 3

Sketch of the Universal **WB** Program

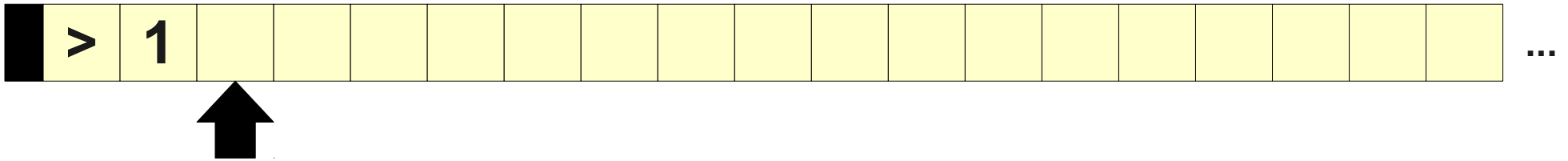
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

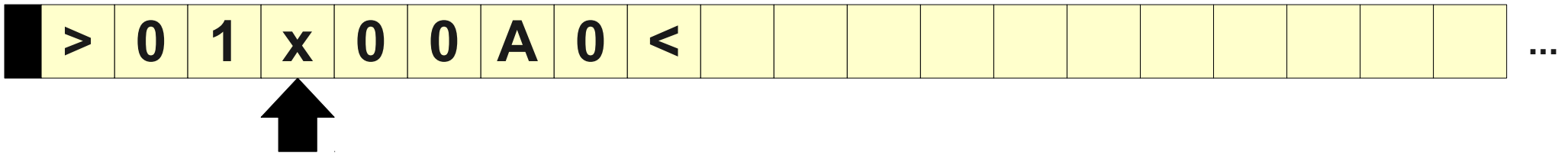


Variables for intermediate storage.

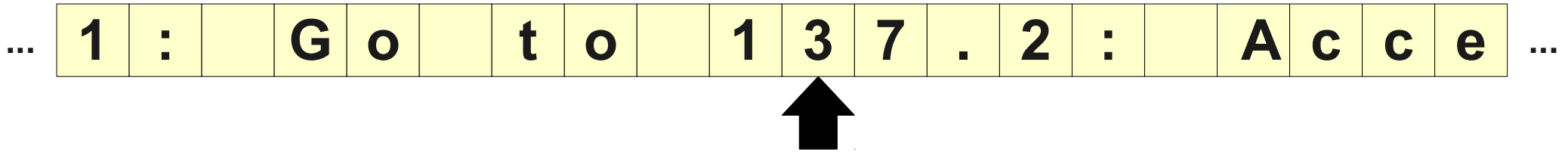
Instr **GoTo** **Letter** **3**

Sketch of the Universal **WB** Program

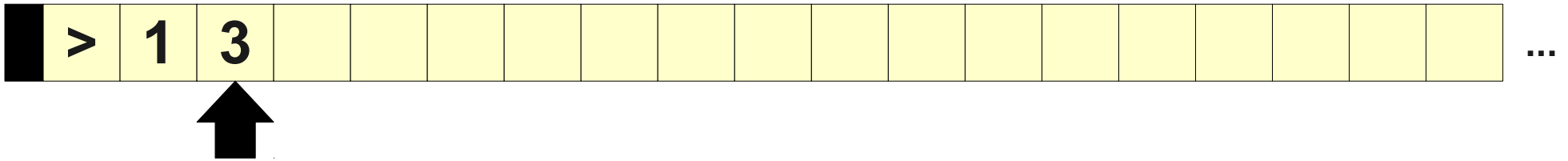
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

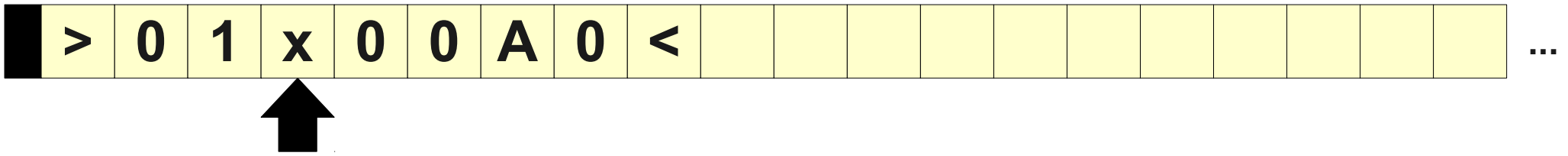


Variables for intermediate storage.

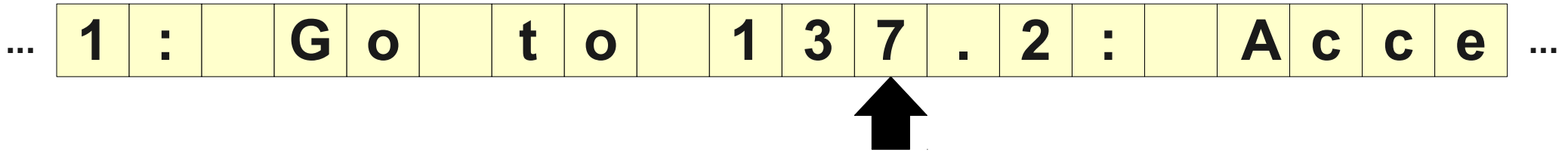
Instr GoTo **Letter** 3

Sketch of the Universal **WB** Program

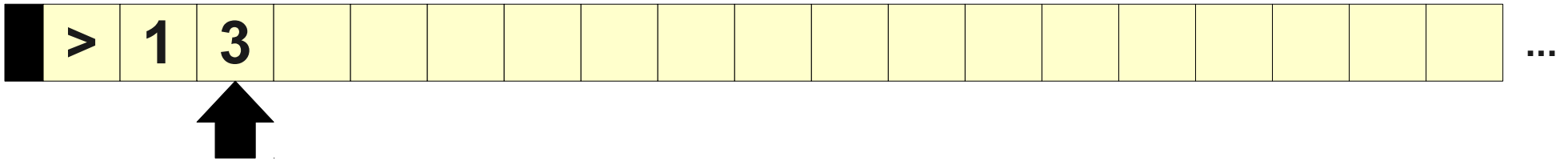
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

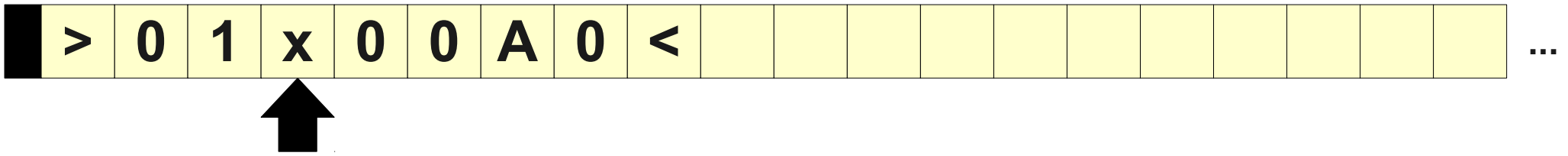


Variables for intermediate storage.

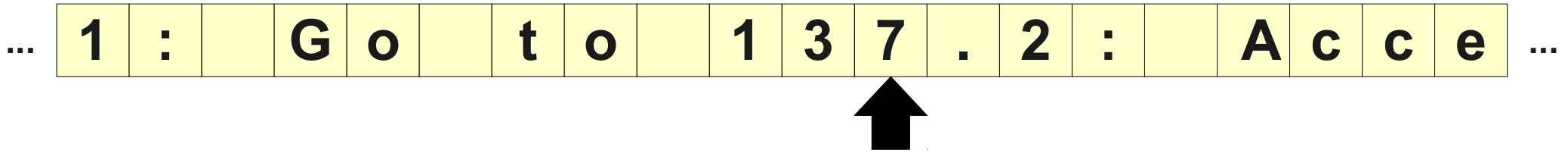
Instr **GoTo** **Letter** **3**

Sketch of the Universal **WB** Program

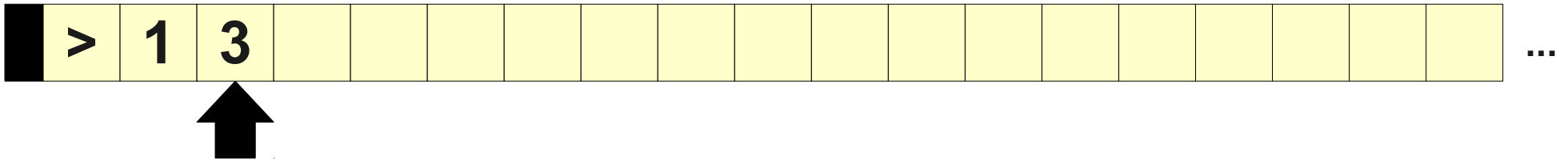
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

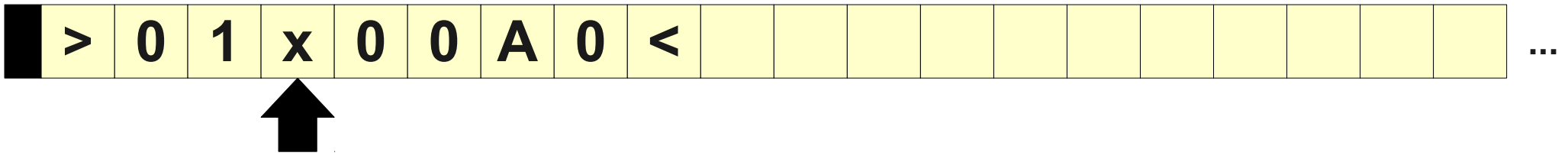


Variables for intermediate storage.

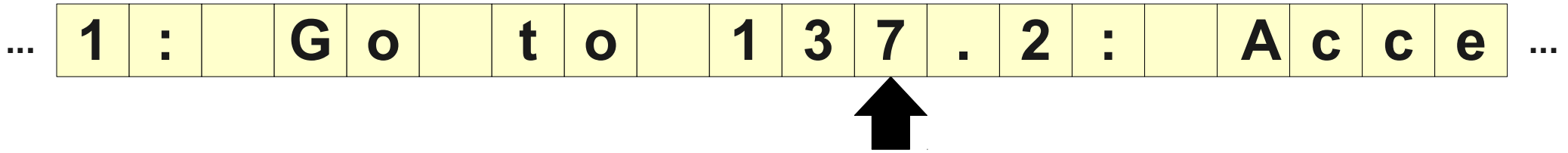
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

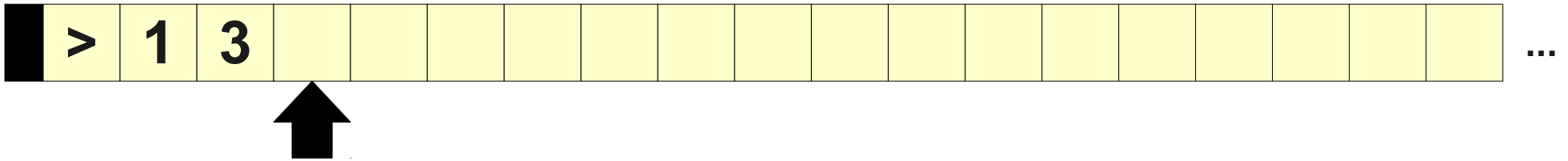
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

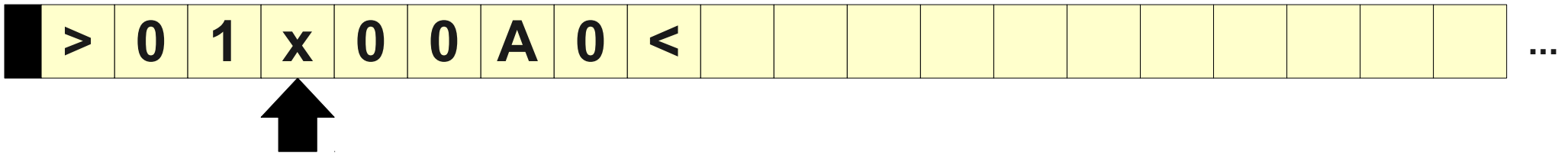


Variables for intermediate storage.

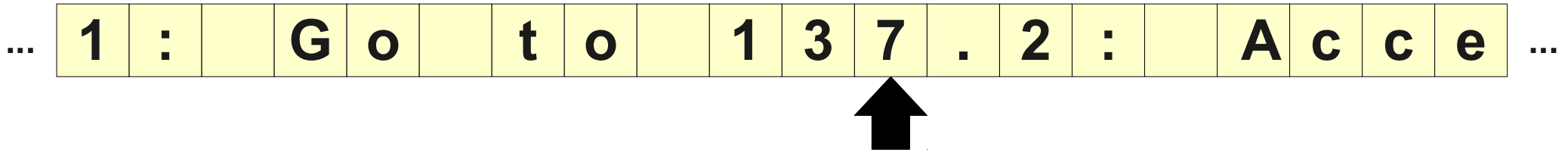
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

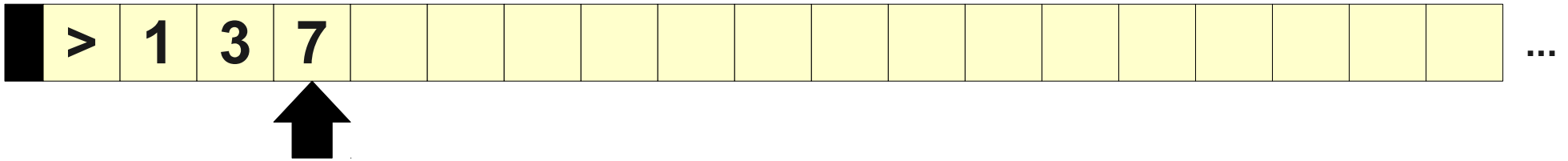
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

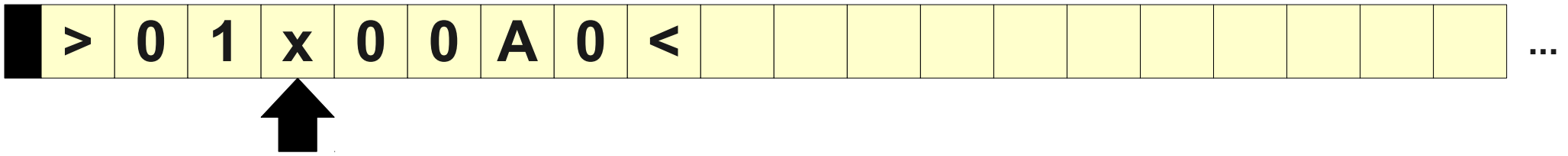


Variables for intermediate storage.

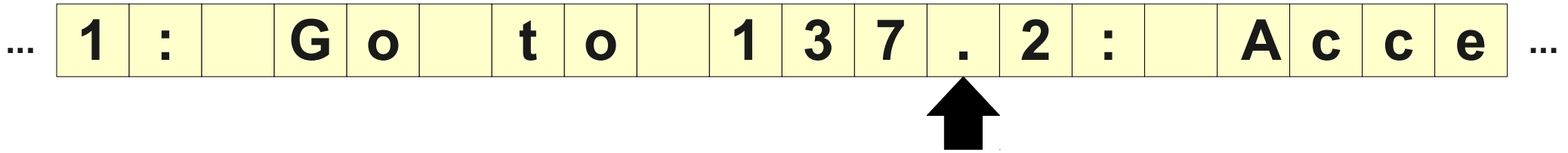
Instr GoTo **Letter** 7

Sketch of the Universal **WB** Program

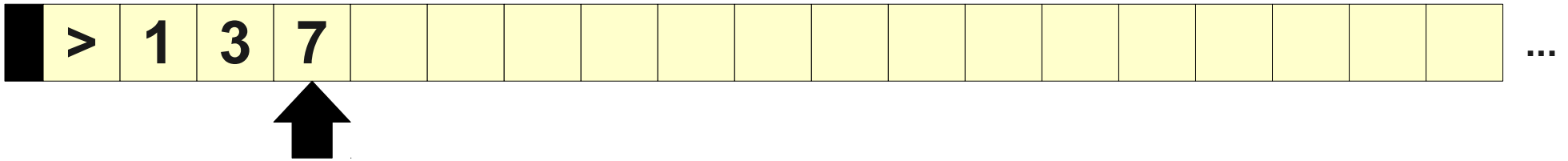
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

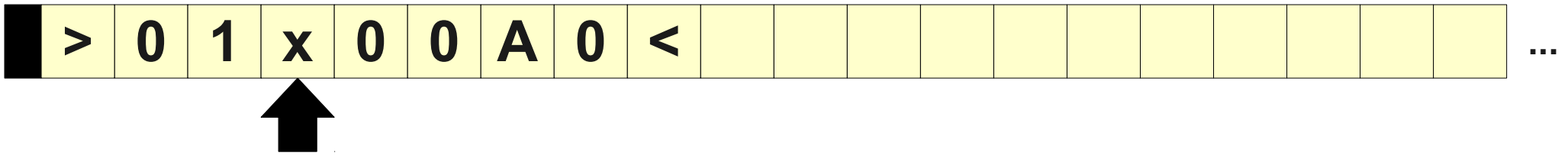


Variables for intermediate storage.

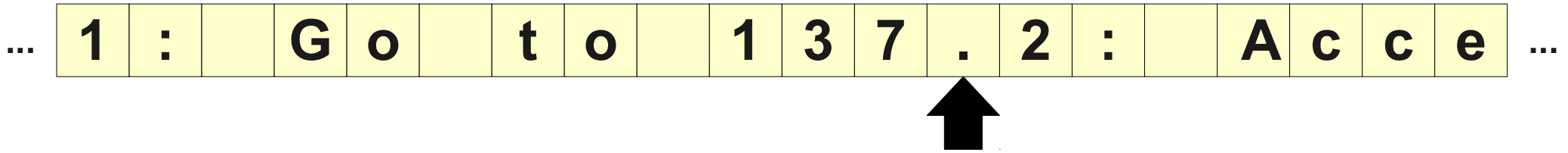
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

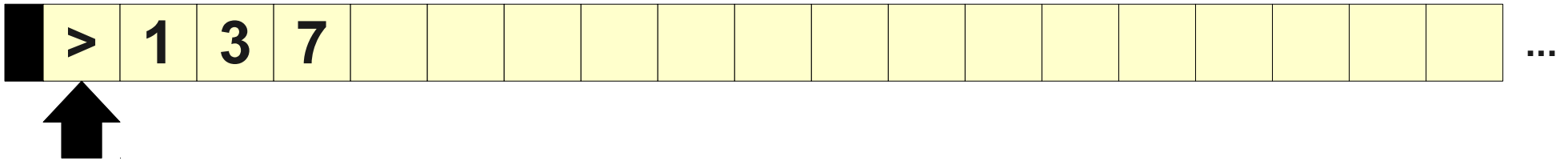
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

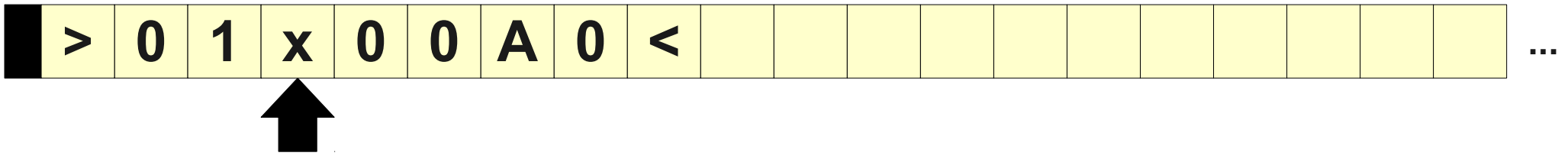


Variables for intermediate storage.

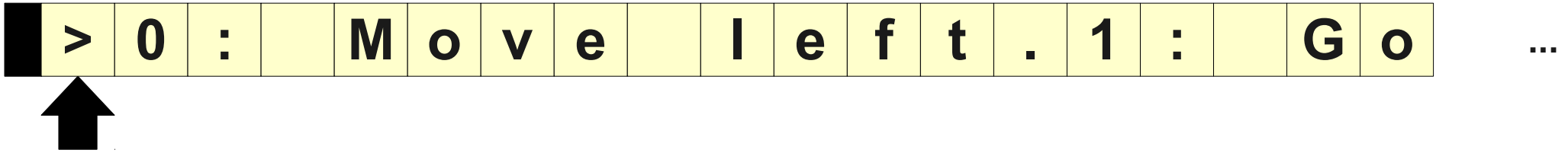
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

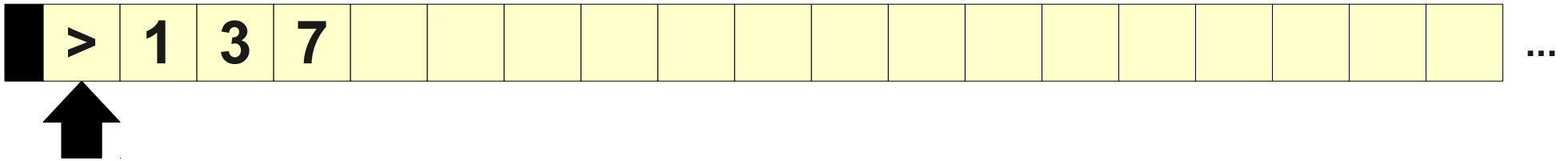
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

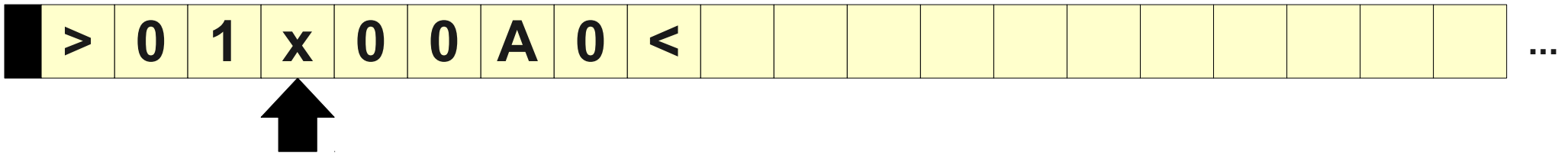


Variables for intermediate storage.

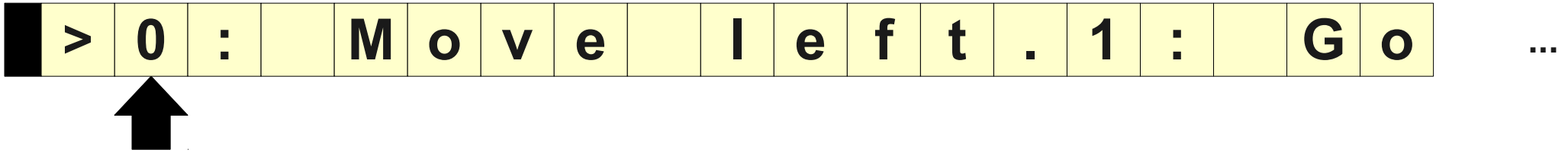
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

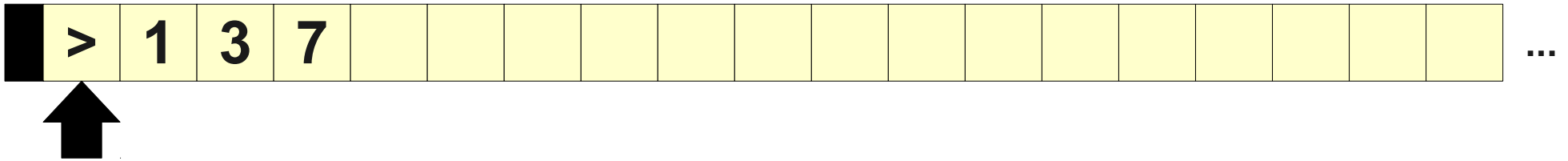
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

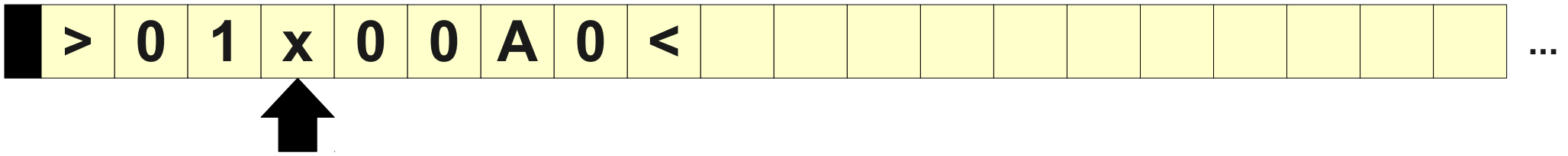


Variables for intermediate storage.

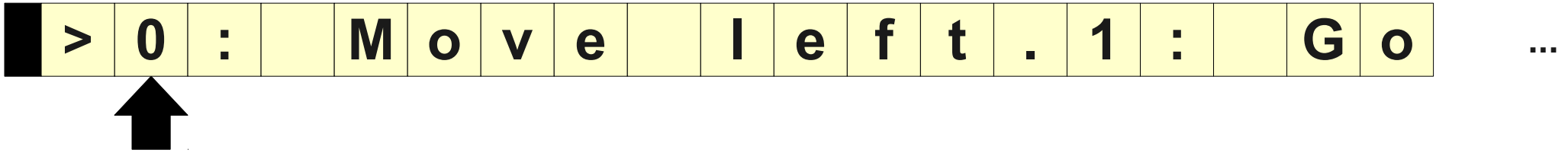
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

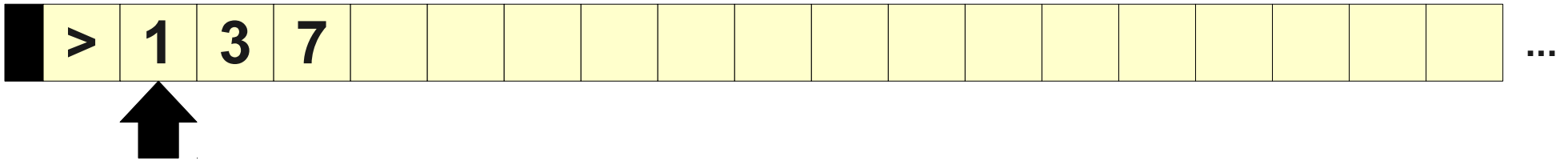
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

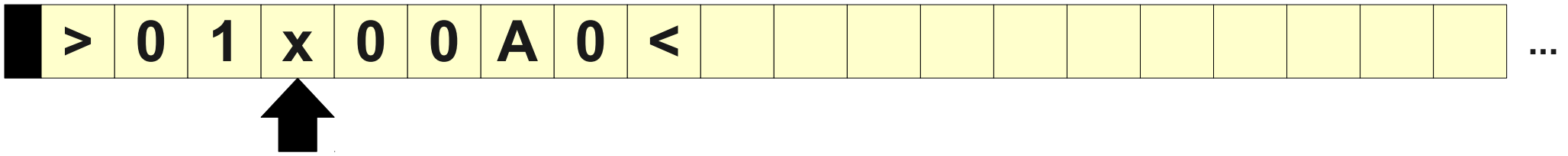


Variables for intermediate storage.

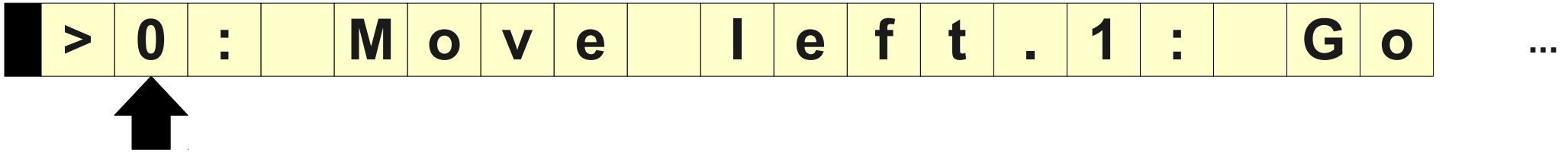
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

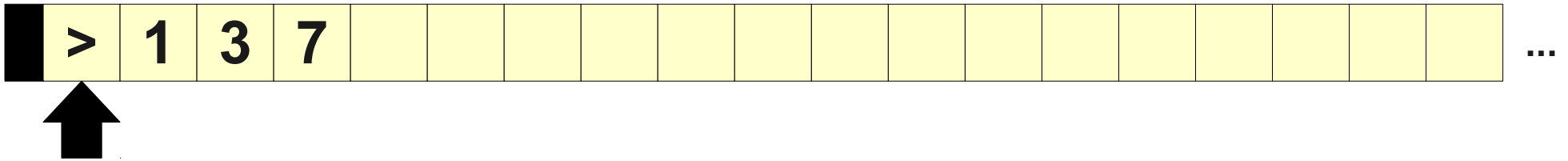
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

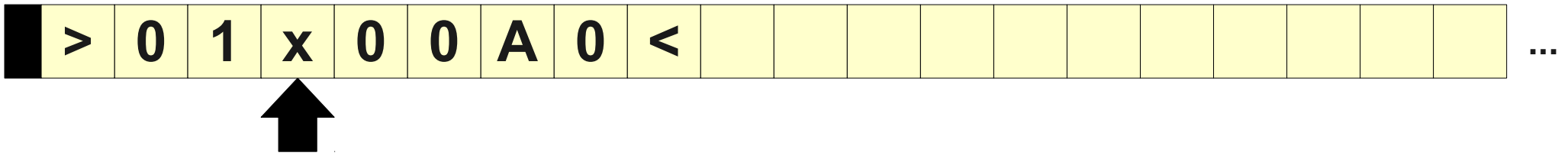


Variables for intermediate storage.

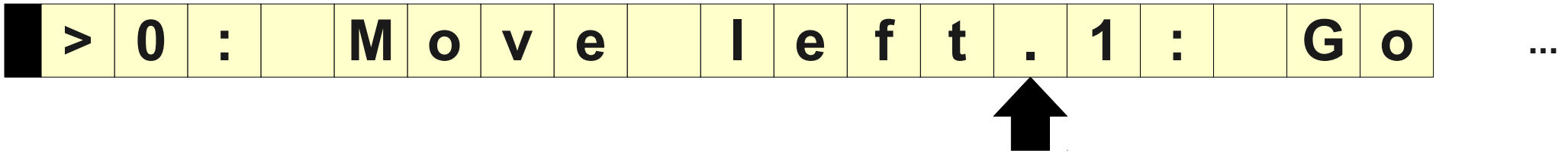
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

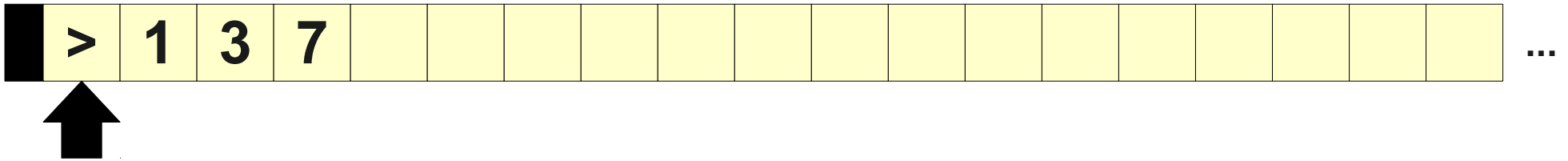
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

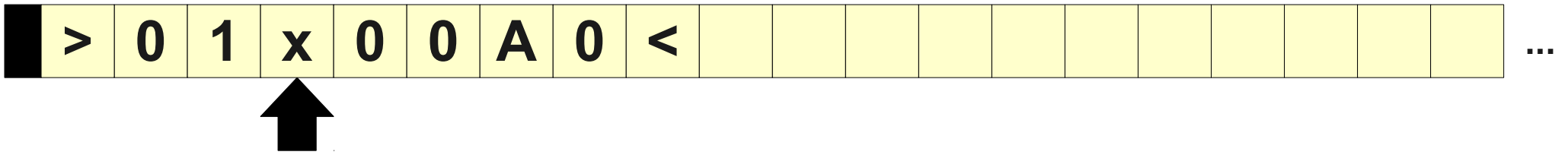


Variables for intermediate storage.

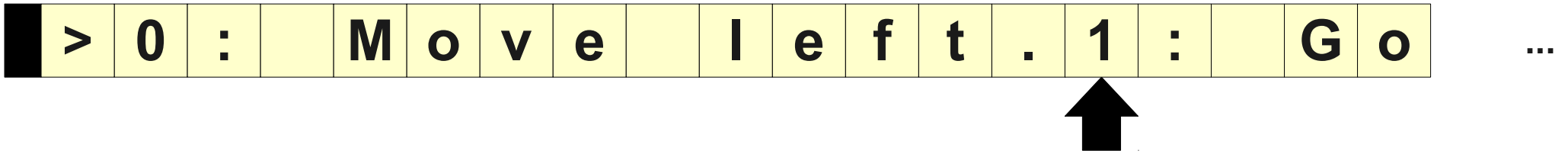
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

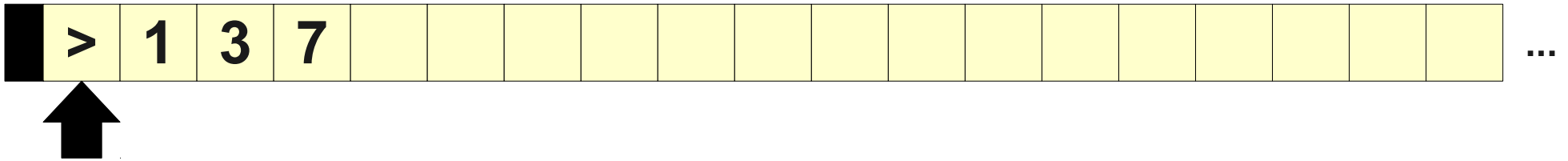
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

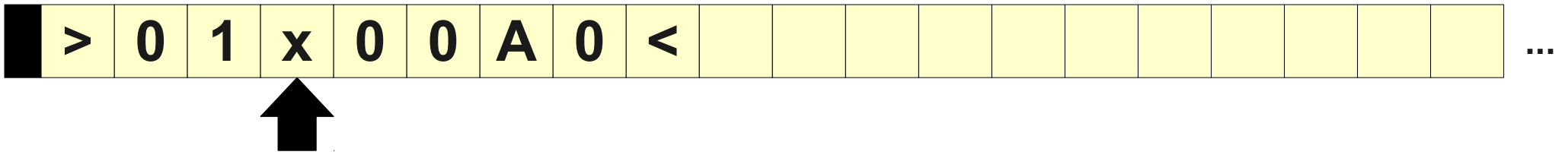


Variables for intermediate storage.

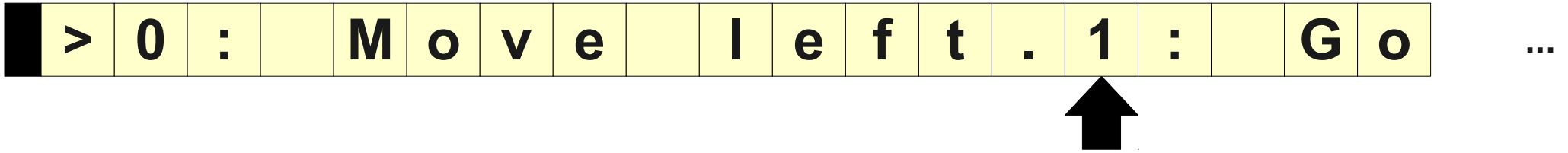
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

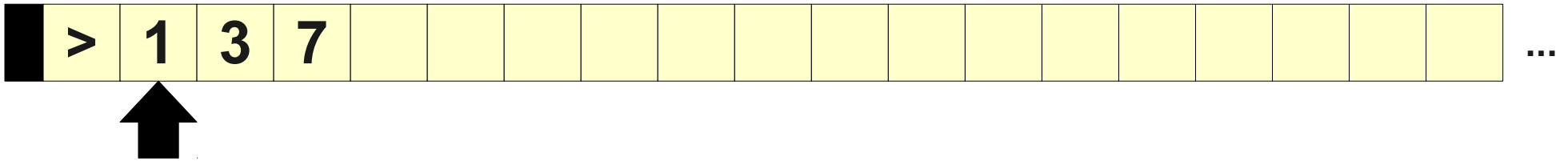
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

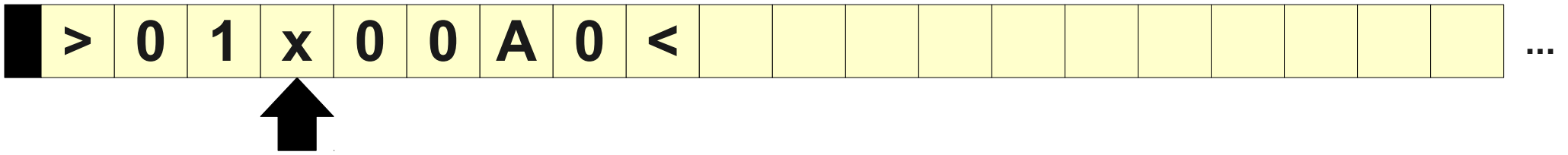


Variables for intermediate storage.

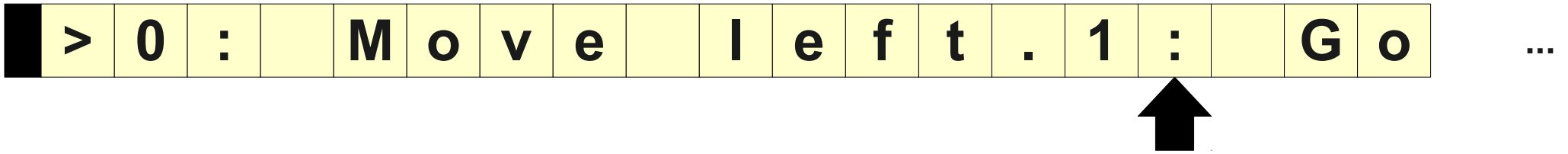
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

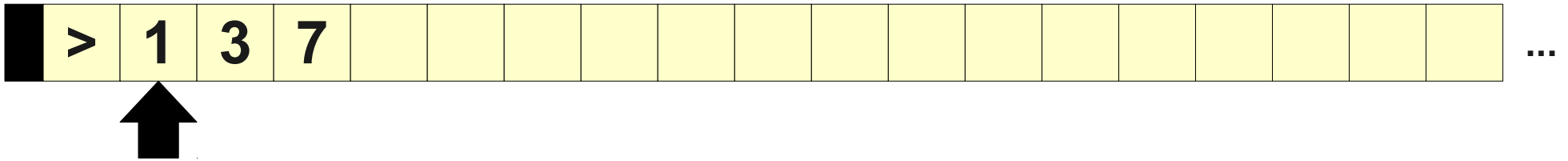
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

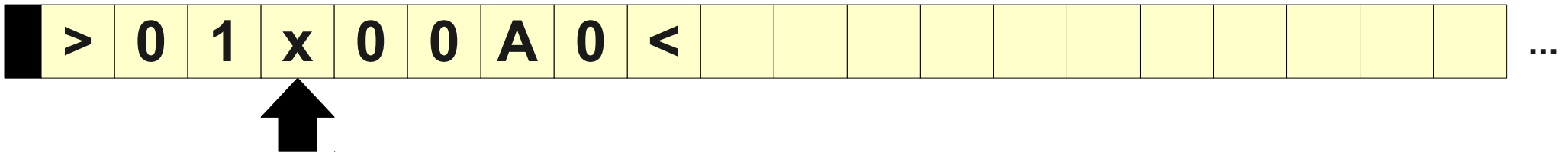


Variables for intermediate storage.

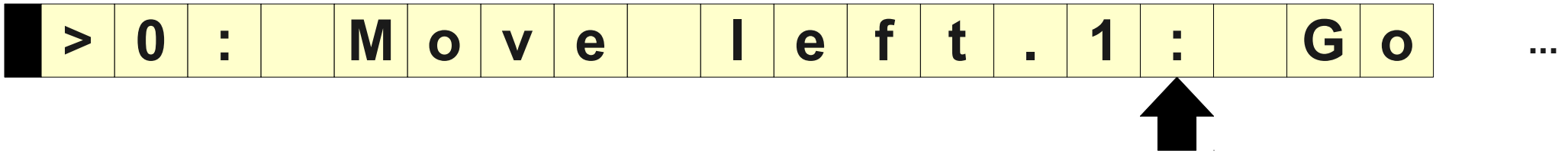
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

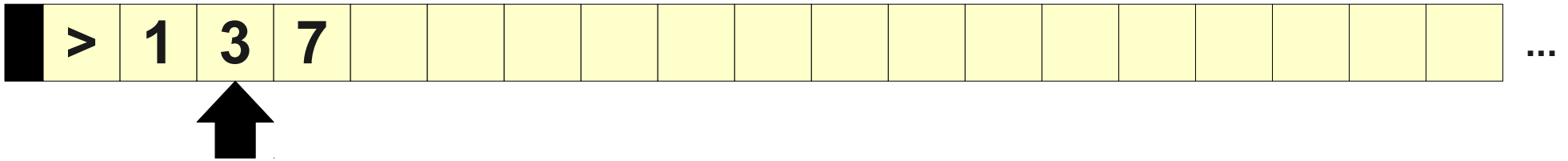
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

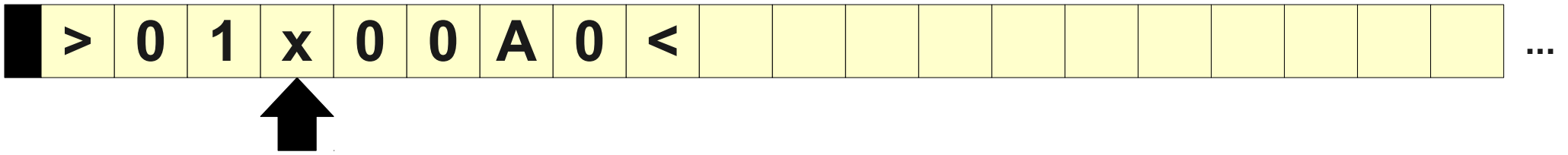


Variables for intermediate storage.

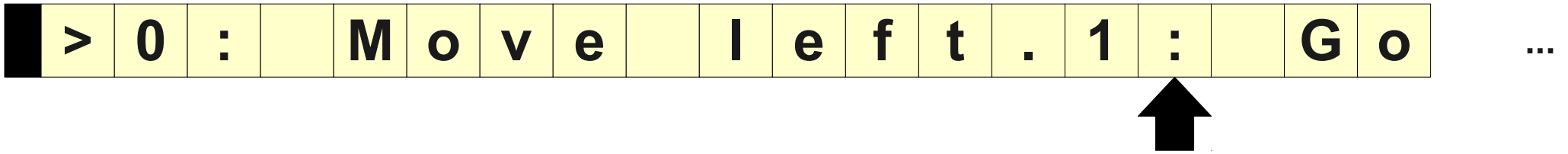
Instr **GoTo** **Letter** **7**

Sketch of the Universal **WB** Program

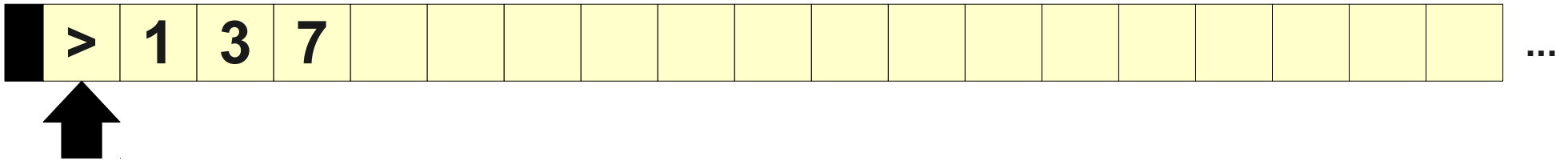
Simulated tape of the program being executed.



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 the universal TM U_{TM}?
- 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 $\mathbf{A_{TM}}$:

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

- Since $\mathscr{L}(U_{TM}) = A_{TM}$, we know A_{TM} is recursively enumerable.

The Language of U_{TM}

- From a formal language perspective, what is the language of a the universal TM U_{TM}?
- 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 $\mathbf{A_{TM}}$:

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

- Since $\mathscr{L}(U_{TM}) = A_{TM}$, we know A_{TM} is recursively enumerable.

Why This Matters (Philosophically)

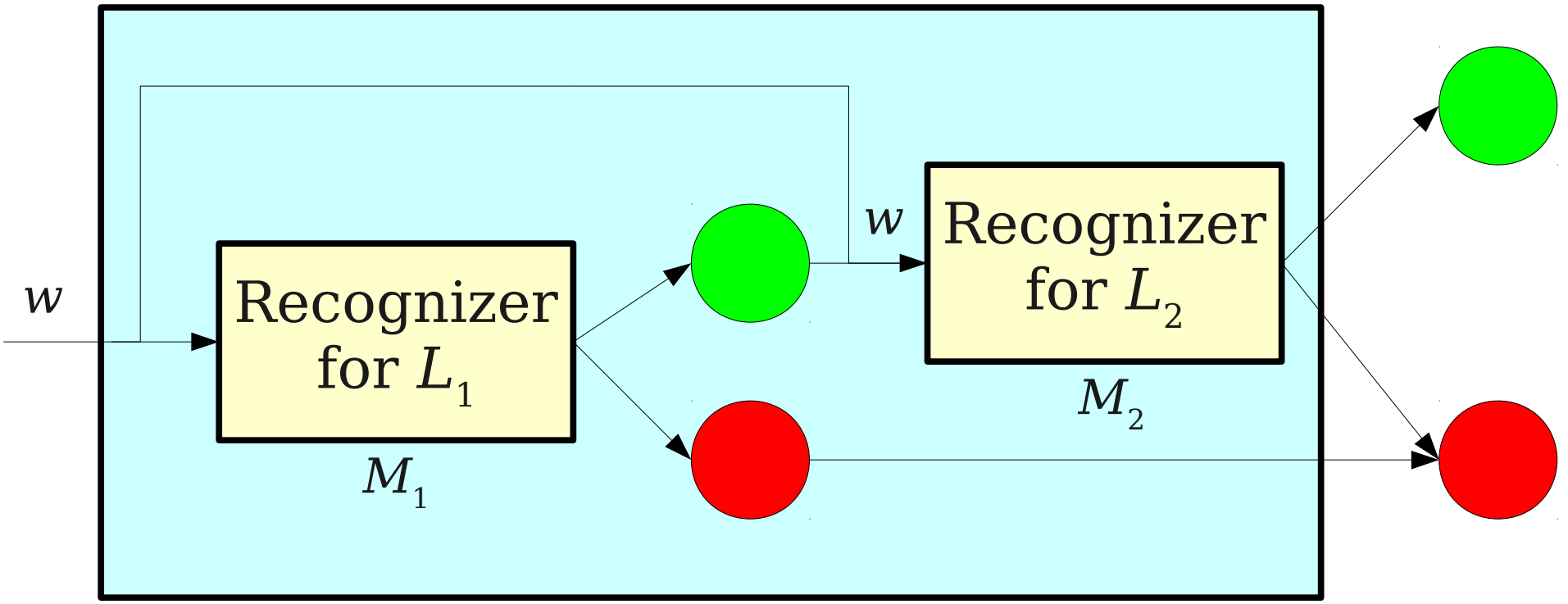
- **As a historically significant achievement.**
  - The universal Turing machine might be the very first “complicated” program ever designed for a computer.
  - Motivation for the “stored-program” model of computers.
- **As a justification for the Church-Turing thesis.**
  - All sufficiently powerful models of computation can simulate one another.

Why This Matters (Practically)

- **Turing machines can run other Turing machines.**
  - A TM can use the universal TM as a subroutine to simulate the execution of other TMs.
  - This lets TMs base their behaviors on the behavior of other TMs.
- **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 Monday.


YO DAWG, I HEARD YOU LIKED TMS
SO I PUT A TM IN YOUR TM SO YOU CAN RUN WHILE YOU RUN
memegenerator.net

RE is Closed Under Intersection



M' = "On input w:

Run M_1 on w.

If M_1 accepts:

Run M_2 on w.

If M_2 accepts w, accept.

If M_2 rejects w, reject.

If M_1 rejects w, reject."

Same here.

If M_1 loops on w, we will never get past this line. When running a TM as a subroutine, make sure you remember to account for this!

Theorem: **RE** is closed under intersection.

Theorem: **RE** is closed under intersection.
Proof: Consider any L_1, $L_2 \in$ **RE**.

Theorem: **RE** is closed under intersection.

Proof: Consider any $L_1, L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

Theorem: **RE** is closed under intersection.

Proof: Consider any L_1, $L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

Let M_1 and M_2 be recognizers for L_1 and L_2, respectively.

Theorem: **RE** is closed under intersection.

Proof: Consider any L_1, $L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

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: Consider any L_1, $L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

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

Theorem: **RE** is closed under intersection.

Proof: Consider any $L_1, L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

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

We show that $\mathscr{L}(M) = L_1 \cap L_2$ by proving that M accepts w iff $w \in L_1 \cap L_2$.

Theorem: **RE** is closed under intersection.

Proof: Consider any $L_1, L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

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

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

Theorem: **RE** is closed under intersection.

Proof: Consider any L_1, $L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

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

We show that $\mathscr{L}(M) = L_1 \cap L_2$ by proving that M accepts w iff $w \in L_1 \cap L_2$. To see this, 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: Consider any L_1, $L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

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

We show that $\mathscr{L}(M) = L_1 \cap L_2$ by proving that M accepts w iff $w \in L_1 \cap L_2$. To see this, 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: Consider any L_1, $L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

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

We show that $\mathscr{L}(M) = L_1 \cap L_2$ by proving that M accepts w iff $w \in L_1 \cap L_2$. To see this, 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 $\mathscr{L}(M) = L_1 \cap L_2$.

Theorem: **RE** is closed under intersection.

Proof: Consider any L_1, $L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

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

We show that $\mathscr{L}(M) = L_1 \cap L_2$ by proving that M accepts w iff $w \in L_1 \cap L_2$. To see this, 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 $\mathscr{L}(M) = L_1 \cap L_2$. Since $\mathscr{L}(M) = L_1 \cap L_2$, we have that $L_1 \cap L_2 \in$ **RE**, as required.

Theorem: **RE** is closed under intersection.

Proof: Consider any L_1, $L_2 \in$ **RE**. We will prove that $L_1 \cap L_2 \in$ **RE** by constructing a TM M such that $\mathscr{L}(M) = L_1 \cap L_2$.

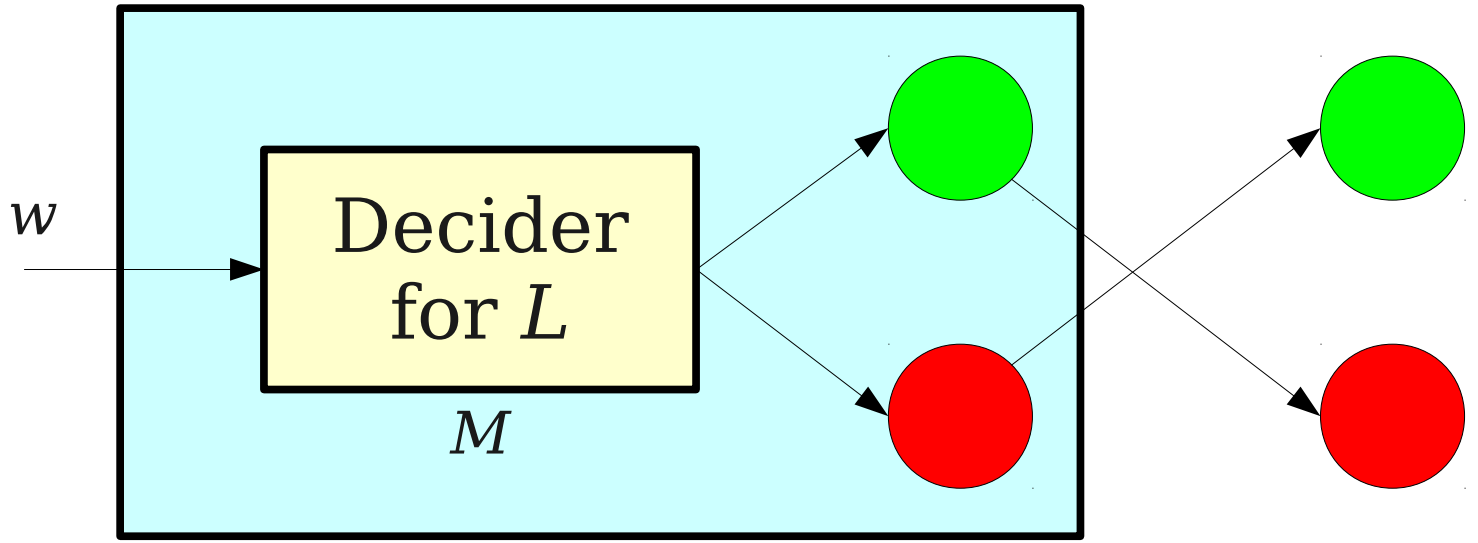
Let M_1 and M_2 be recognizers for L_1 and L_2, respectively. Then construct the machine M as follows:

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

We show that $\mathscr{L}(M) = L_1 \cap L_2$ by proving that M accepts w iff $w \in L_1 \cap L_2$. To see this, 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 $\mathscr{L}(M) = L_1 \cap L_2$. Since $\mathscr{L}(M) = L_1 \cap L_2$, we have that $L_1 \cap L_2 \in$ **RE**, as required. ■

R is Closed Under Complement

If $L \in \mathbf{R}$, then $\overline{L} \in \mathbf{R}$ as well.



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

Will this work if M is a **recognizer**, rather than a **decider**?

Theorem: **R** is closed under complementation.

Theorem: **R** is closed under complementation.
Proof: Consider any $L \in \mathbf{R}$.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

This is the standard way to show that a language is in **R**. Note that we aren't just building any arbitrary TM; it has to be a decider.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

There are two proofs required here, and they're separate from one another. Just showing one or the other isn't sufficient.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w. If M rejects w, M' accepts w.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w. If M rejects w, M' accepts w. Thus M' always accepts or rejects, so M' is a decider.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w. If M rejects w, M' accepts w. Thus M' always accepts or rejects, so M' is a decider.

To show that $\mathscr{L}(M') = \overline{L}$, we will prove that M' accepts w iff $w \in \overline{L}$.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w. If M rejects w, M' accepts w. Thus M' always accepts or rejects, so M' is a decider.

To show that $\mathscr{L}(M') = \overline{L}$, we will prove that M' accepts w iff $w \in \overline{L}$. Note that M' accepts w iff $w \in \Sigma^*$ and M rejects w.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w. If M rejects w, M' accepts w. Thus M' always accepts or rejects, so M' is a decider.

To show that $\mathscr{L}(M') = \overline{L}$, we will prove that M' accepts w iff $w \in \overline{L}$. Note that M' accepts w iff $w \in \Sigma^*$ and M rejects w. Since M is a decider, M rejects w iff M does not accept w.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w. If M rejects w, M' accepts w. Thus M' always accepts or rejects, so M' is a decider.

To show that $\mathscr{L}(M') = \overline{L}$, we will prove that M' accepts w iff $w \in \overline{L}$. Note that M' accepts w iff $w \in \Sigma^*$ and M rejects w. Since M is a decider, M rejects w iff M does not accept w. M does not accept w iff $w \notin \mathscr{L}(M)$.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w. If M rejects w, M' accepts w. Thus M' always accepts or rejects, so M' is a decider.

To show that $\mathscr{L}(M') = \overline{L}$, we will prove that M' accepts w iff $w \in \overline{L}$. Note that M' accepts w iff $w \in \Sigma^*$ and M rejects w. Since M is a decider, M rejects w iff M does not accept w. M does not accept w iff $w \notin \mathscr{L}(M)$. Thus M' accepts w iff $w \in \Sigma^*$ and $w \notin \mathscr{L}(M)$, so M' accepts w iff $w \in \overline{L}$.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w. If M rejects w, M' accepts w. Thus M' always accepts or rejects, so M' is a decider.

To show that $\mathscr{L}(M') = \overline{L}$, we will prove that M' accepts w iff $w \in \overline{L}$. Note that M' accepts w iff $w \in \Sigma^*$ and M rejects w. Since M is a decider, M rejects w iff M does not accept w. M does not accept w iff $w \notin \mathscr{L}(M)$. Thus M' accepts w iff $w \in \Sigma^*$ and $w \notin \mathscr{L}(M)$, so M' accepts w iff $w \in \overline{L}$. Therefore, $\mathscr{L}(M') = \overline{L}$.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w. If M rejects w, M' accepts w. Thus M' always accepts or rejects, so M' is a decider.

To show that $\mathscr{L}(M') = \overline{L}$, we will prove that M' accepts w iff $w \in \overline{L}$. Note that M' accepts w iff $w \in \Sigma^*$ and M rejects w. Since M is a decider, M rejects w iff M does not accept w. M does not accept w iff $w \notin \mathscr{L}(M)$. Thus M' accepts w iff $w \in \Sigma^*$ and $w \notin \mathscr{L}(M)$, so M' accepts w iff $w \in \overline{L}$. Therefore, $\mathscr{L}(M') = \overline{L}$.

Since M' is a decider with $\mathscr{L}(M') = \overline{L}$, we have $\overline{L} \in \mathbf{R}$, as required.

Theorem: **R** is closed under complementation.

Proof: Consider any $L \in \mathbf{R}$. We will prove that $\overline{L} \in \mathbf{R}$ by constructing a decider M' such that $\mathscr{L}(M') = \overline{L}$.

Let M be a decider for L. Then construct the machine M' as follows:

M' = "On input $w \in \Sigma^*$:
Run M on w.
If M accepts w, reject.
If M rejects w, accept."

We need to show that M' is a decider and that $\mathscr{L}(M') = \overline{L}$.

To show that M' is a decider, we will prove that it always halts. Consider what happens if we run M' on any input w. First, M' runs M on w. Since M is a decider, M either accepts w or rejects w. If M accepts w, M' rejects w. If M rejects w, M' accepts w. Thus M' always accepts or rejects, so M' is a decider.

To show that $\mathscr{L}(M') = \overline{L}$, we will prove that M' accepts w iff $w \in \overline{L}$. Note that M' accepts w iff $w \in \Sigma^*$ and M rejects w. Since M is a decider, M rejects w iff M does not accept w. M does not accept w iff $w \notin \mathscr{L}(M)$. Thus M' accepts w iff $w \in \Sigma^*$ and $w \notin \mathscr{L}(M)$, so M' accepts w iff $w \in \overline{L}$. Therefore, $\mathscr{L}(M') = \overline{L}$.

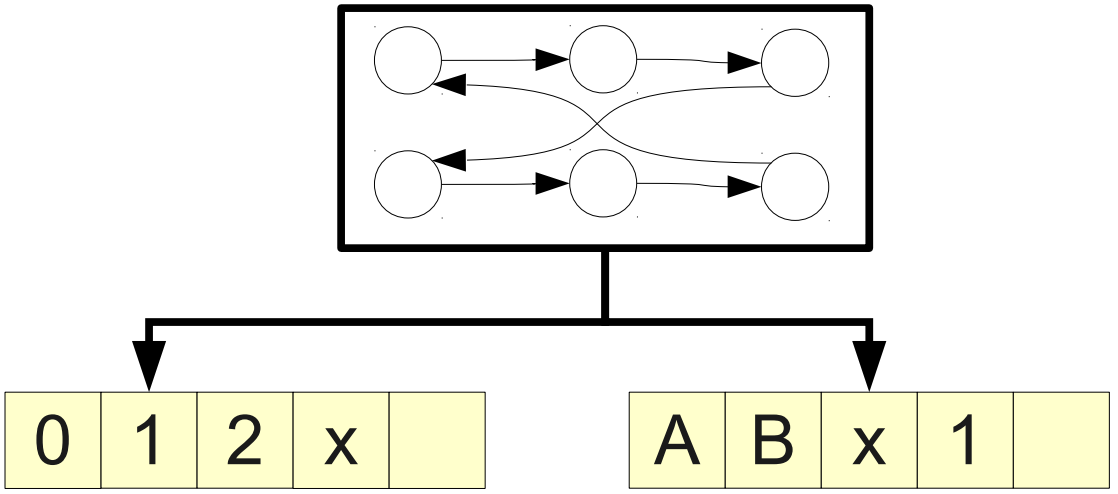
Since M' is a decider with $\mathscr{L}(M') = \overline{L}$, we have $\overline{L} \in \mathbf{R}$, as required. ■

An Important Result

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

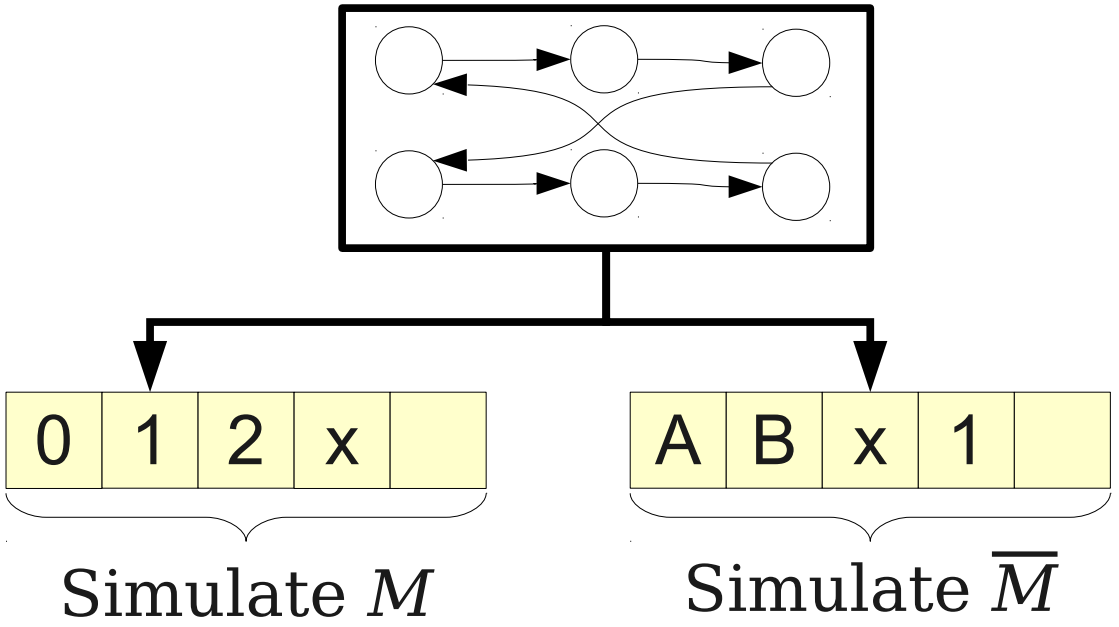
An Important Result

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



An Important Result

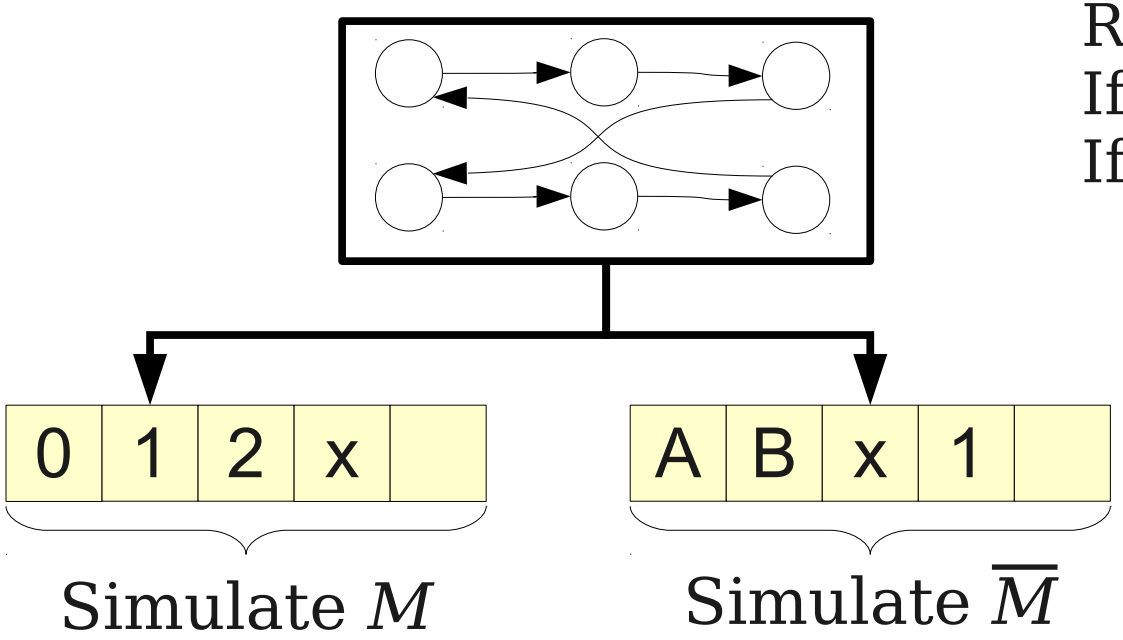
- Suppose that $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$.
- Let M and $\overline{M}$ be recognizers for L and $\overline{L}$, respectively.



An Important Result

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

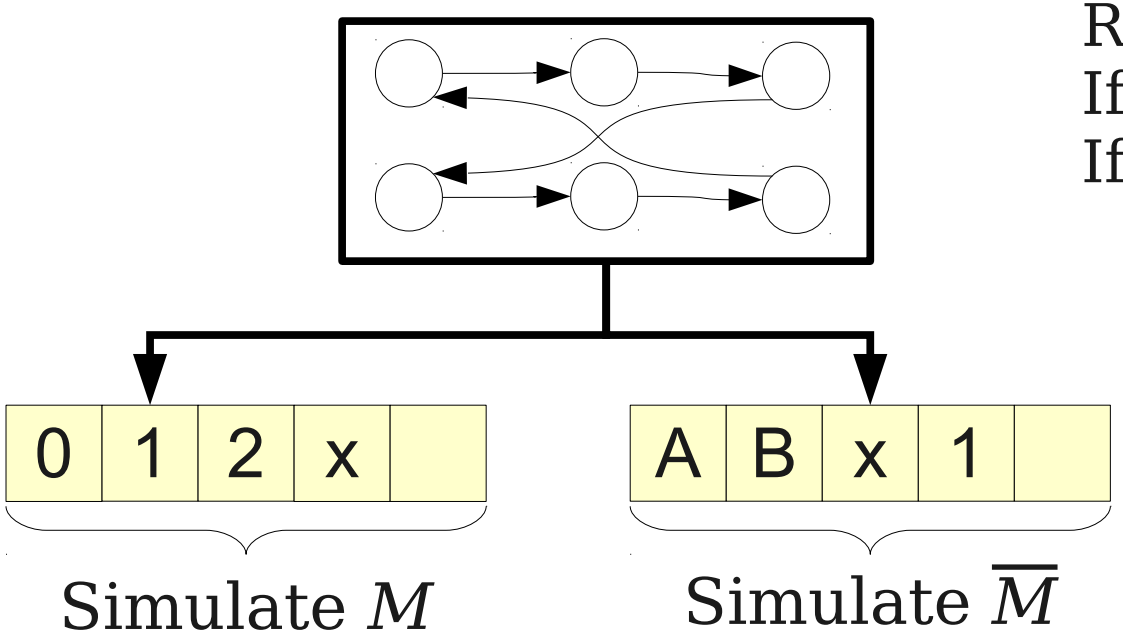
M' = "On input w,
Run M and $\overline{M}$ on w in parallel.
If M accepts w, accept.
If $\overline{M}$ accepts w, reject."



An Important Result

- Suppose that $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$.
- 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 w, accept.
If $\overline{M}$ accepts w, reject."

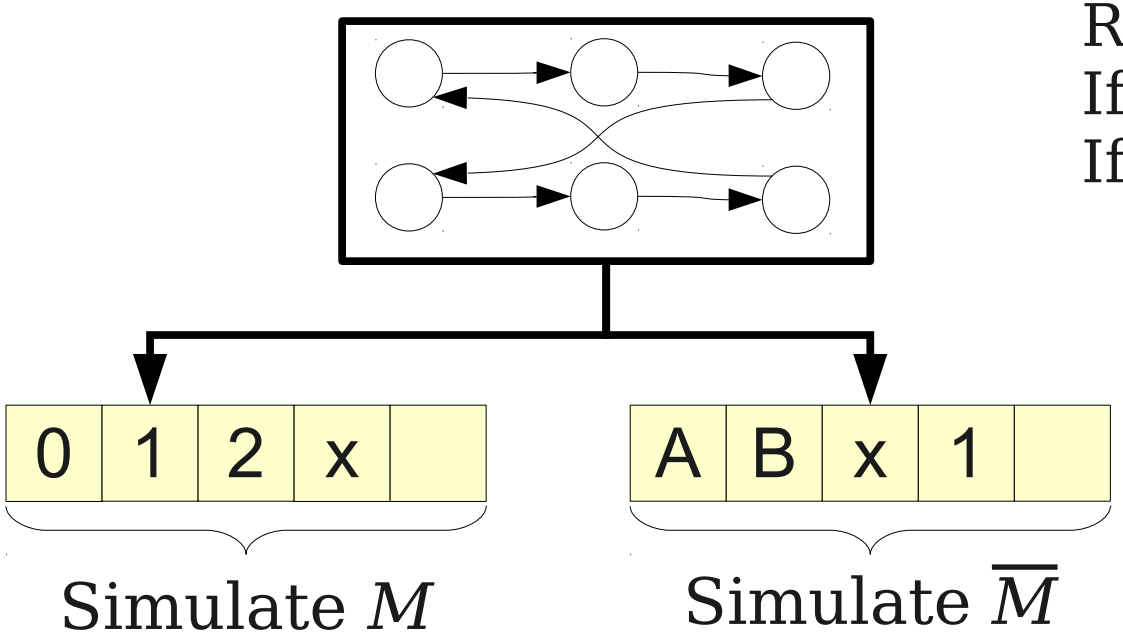


What happens if $w \in L$?

An Important Result

- Suppose that $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$.
- 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 w, accept.
If $\overline{M}$ accepts w, reject."

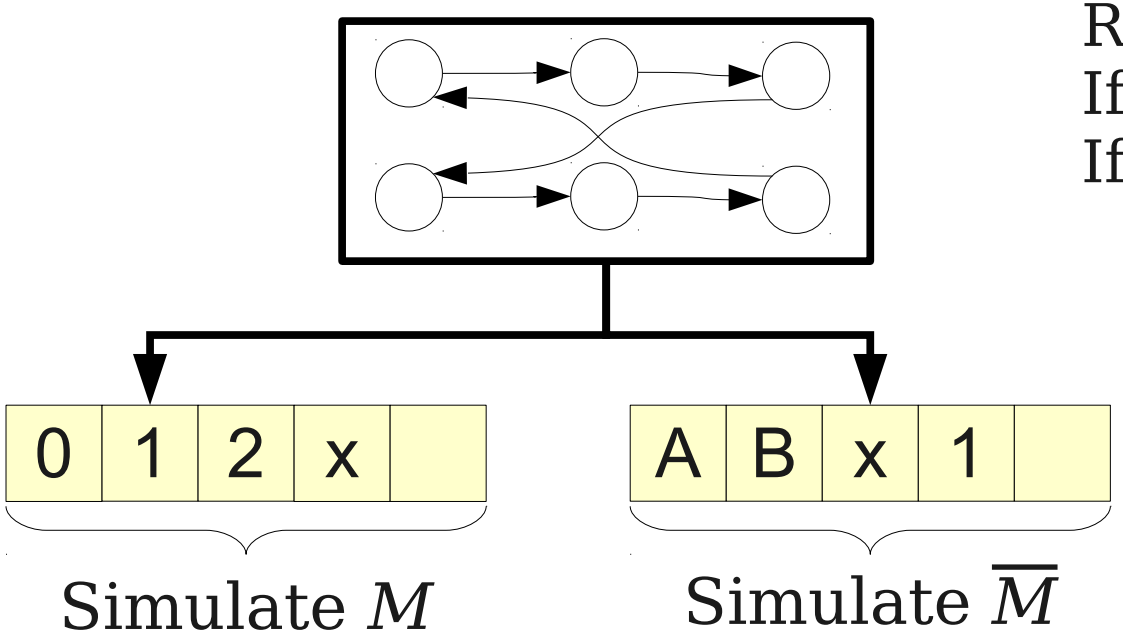


What happens if $w \in L$?

An Important Result

- Suppose that $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$.
- 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 w, accept.
If $\overline{M}$ accepts w, reject."

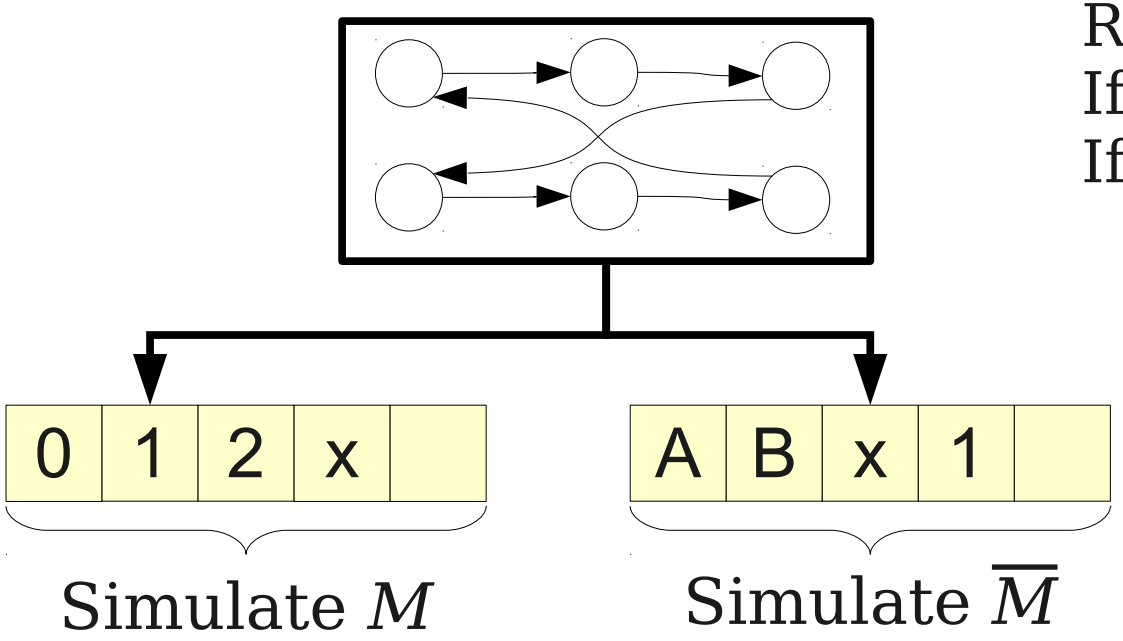


What happens if $w \in L$?

An Important Result

- Suppose that $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$.
- 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 w, accept.
If $\overline{M}$ accepts w, reject."

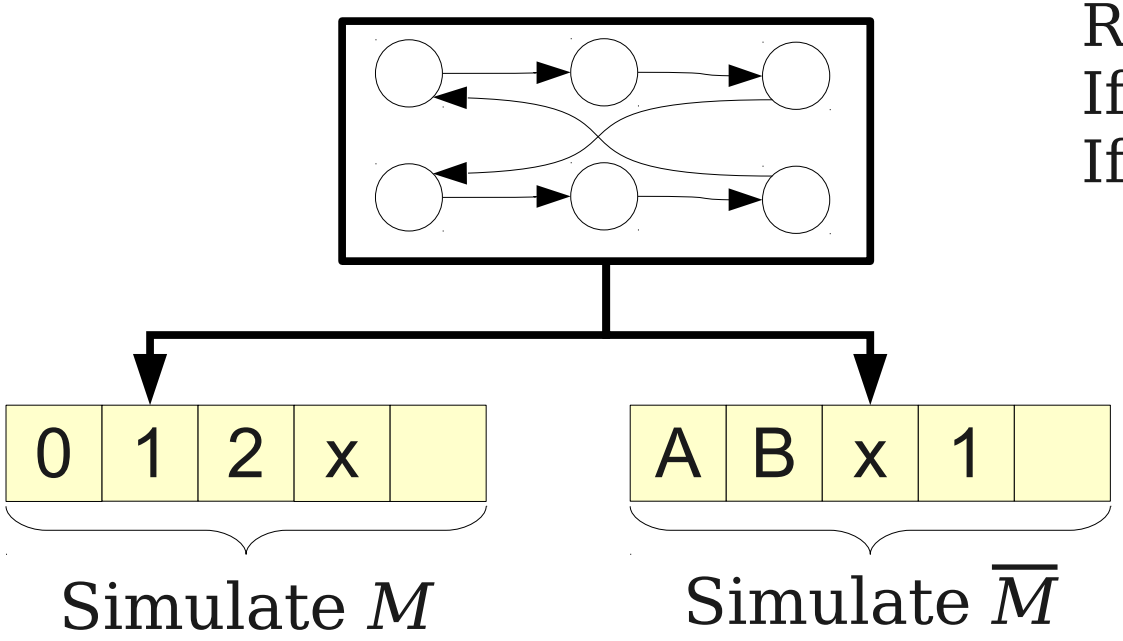


What happens if $w \notin L$?

An Important Result

- Suppose that $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$.
- 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 w, accept.
If $\overline{M}$ accepts w, reject."

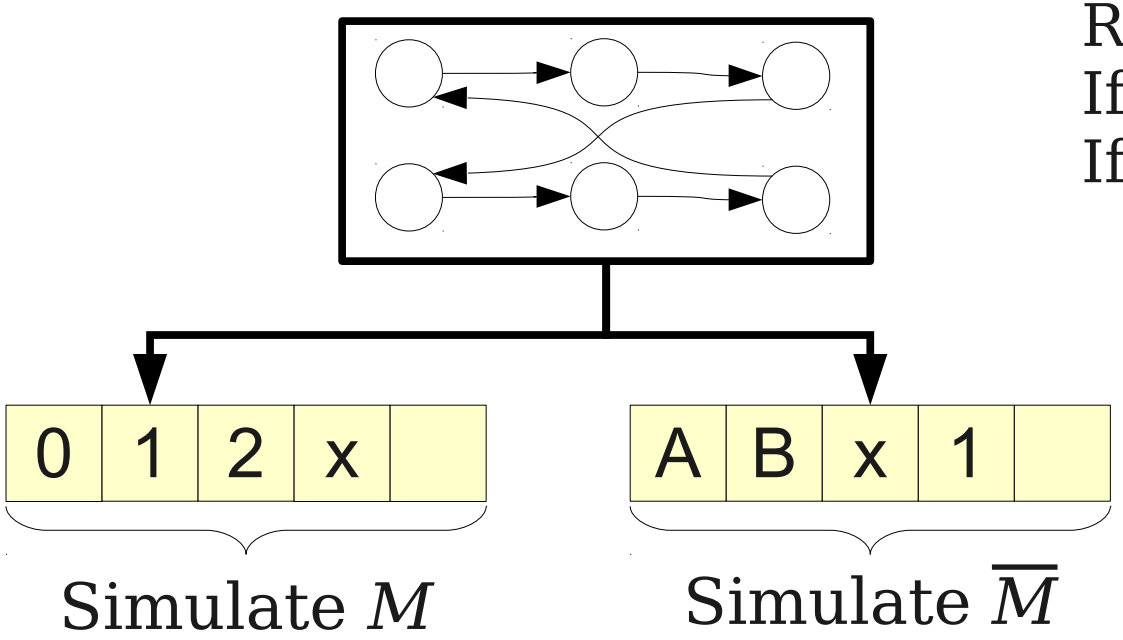


What happens if $w \in \overline{L}$?

An Important Result

- Suppose that $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$.
- 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 w, accept.
If $\overline{M}$ accepts w, reject."

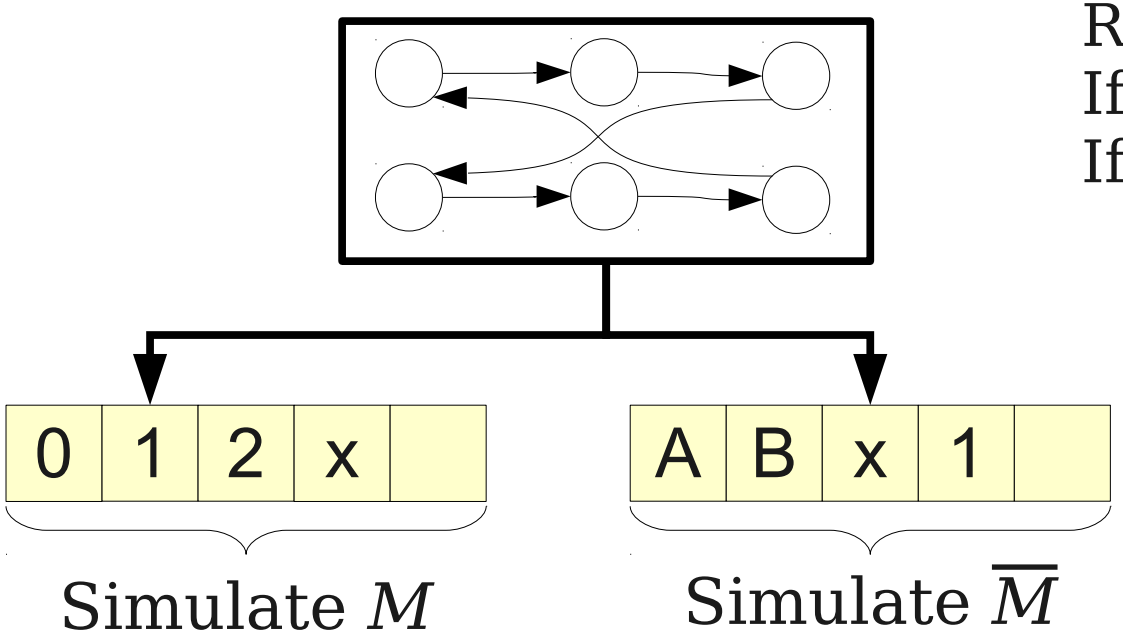


What happens if $w \in \overline{L}$?

An Important Result

- Suppose that $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$.
- 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 w, accept.
If $\overline{M}$ accepts w, reject."

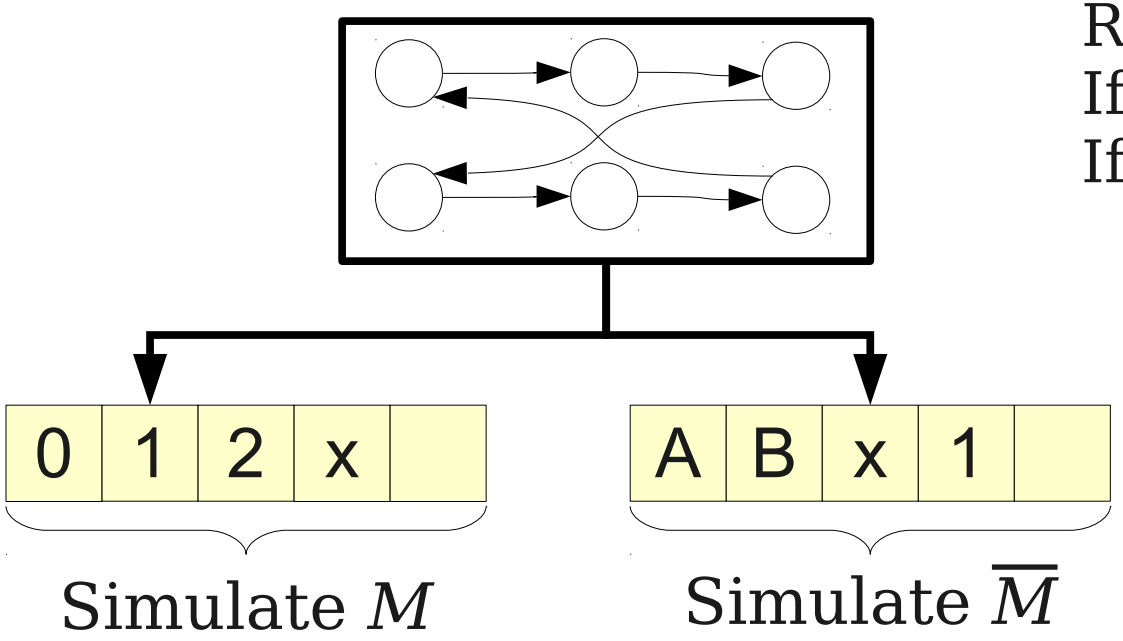


What happens if $w \in \overline{L}$?

An Important Result

- Suppose that $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$.
- 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 w, accept.
If $\overline{M}$ accepts w, reject."



M' is a decider!

Theorem: If $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$, then $L \in \mathbf{R}$.
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 $\mathscr{L}(M') = L$ and that M' is a decider. To see that M' is a decider, consider any string $w \in \Sigma^*$. 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 $\mathscr{L}(M') = L$, 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 $\mathscr{L}(M') = L$.

Since $\mathscr{L}(M') = L$ and M' is a decider, we have $L \in \mathbf{R}$, as required. ■

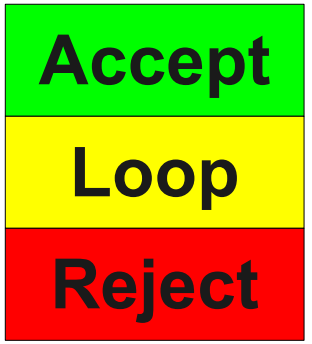
Nondeterministic Turing Machines

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.

Nondeterminisic TMs

- An NTM **accepts** a string *w* if it enters an accept state *on some path*.
- An NTM **rejects** a string *w* if it enters a reject state *on every path*.
- An NTM **loops** on a string *w* if neither of these happen (it doesn't accept on any path and doesn't reject on every path).

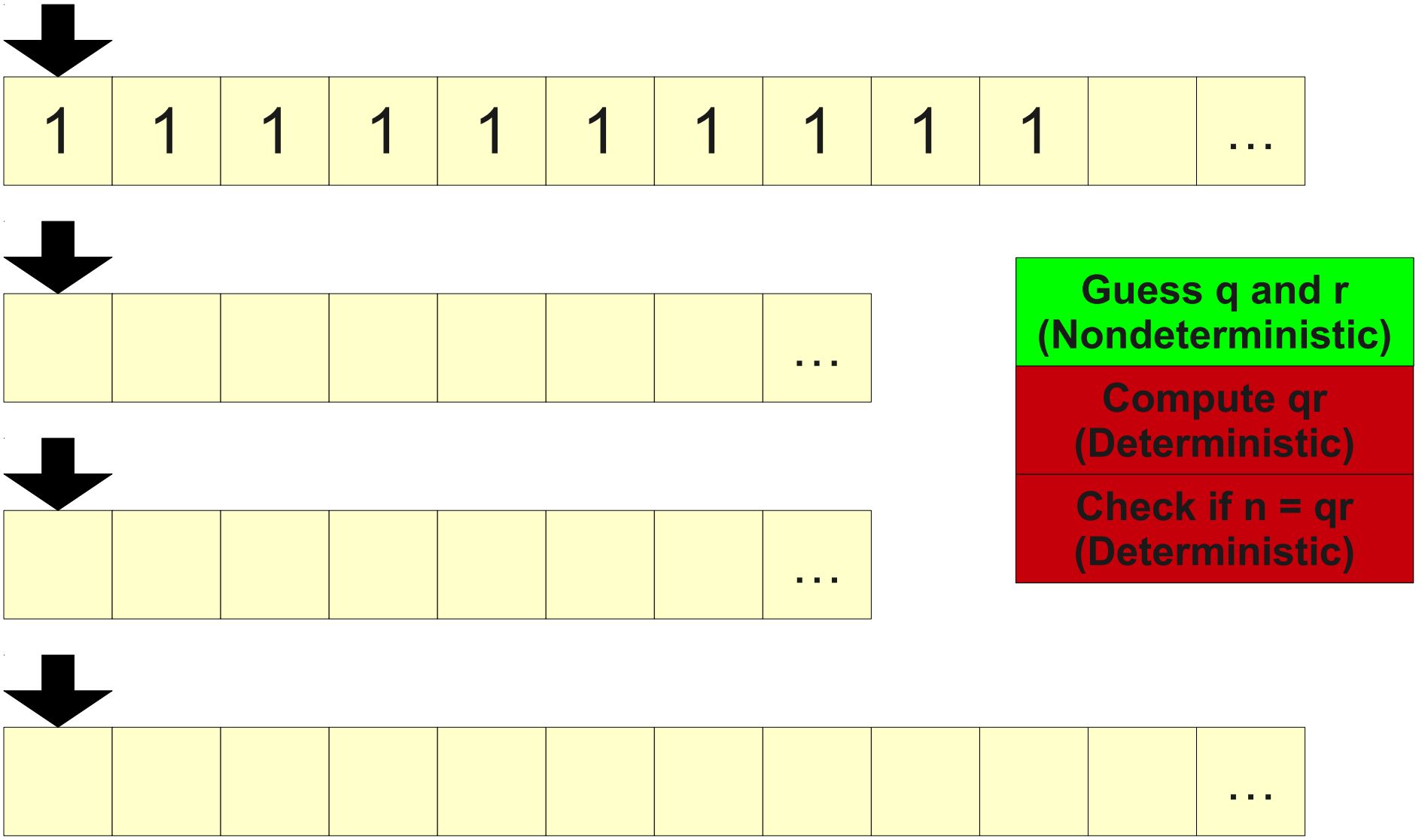


Nondeterministic Algorithms

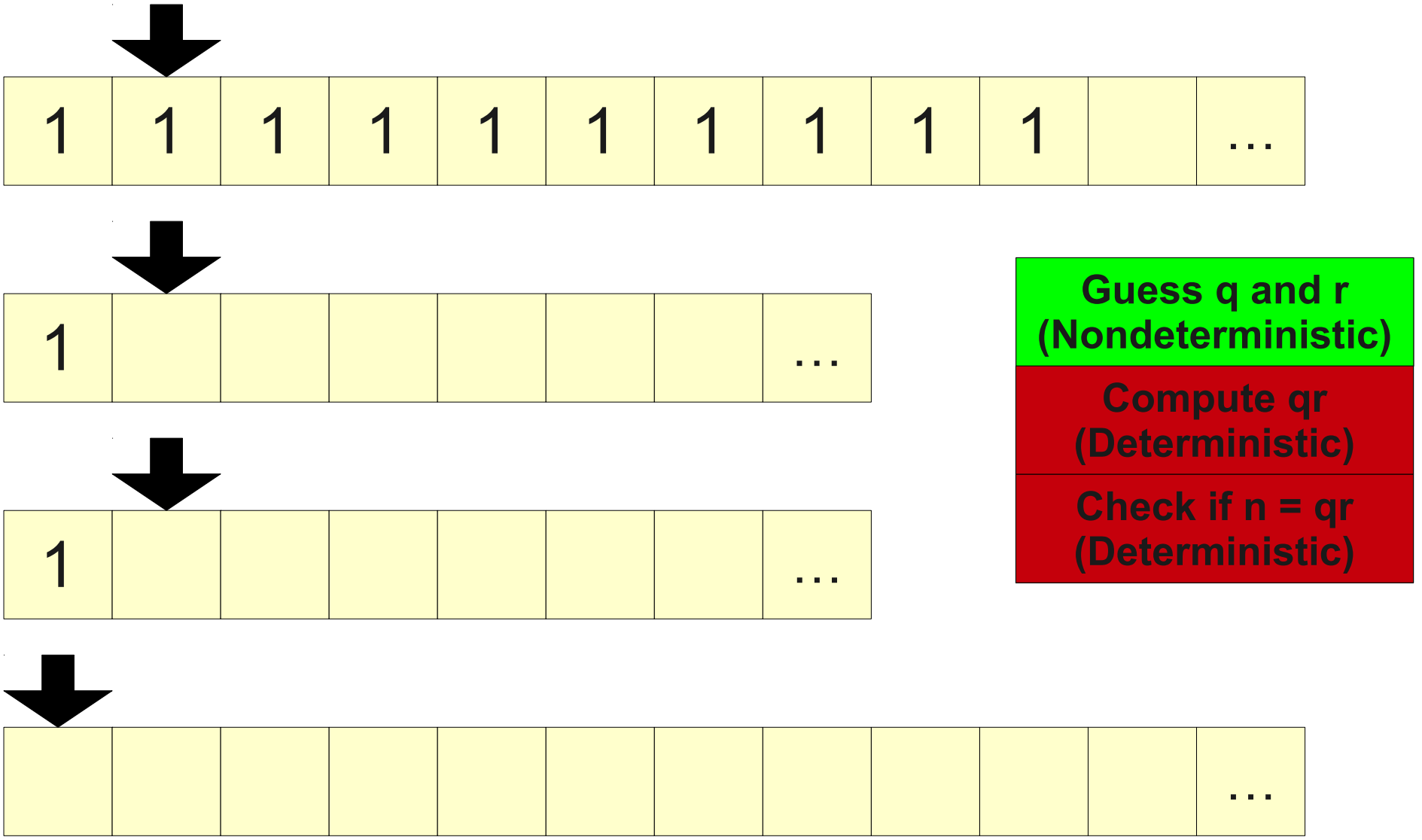
- A natural number greater than 1 is **composite** if it is not prime.
- Let $\Sigma = \{ \texttt{1} \}$ and consider the language

  $COMPOSITE = \{ \texttt{1}^n \mid n \in \Sigma^* \text{ is composite} \}$
- We can build a **multitape, nondeterministic TM** for *COMPOSITE* as follows:
- M = “On input $\texttt{1}^n$:
  - **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”

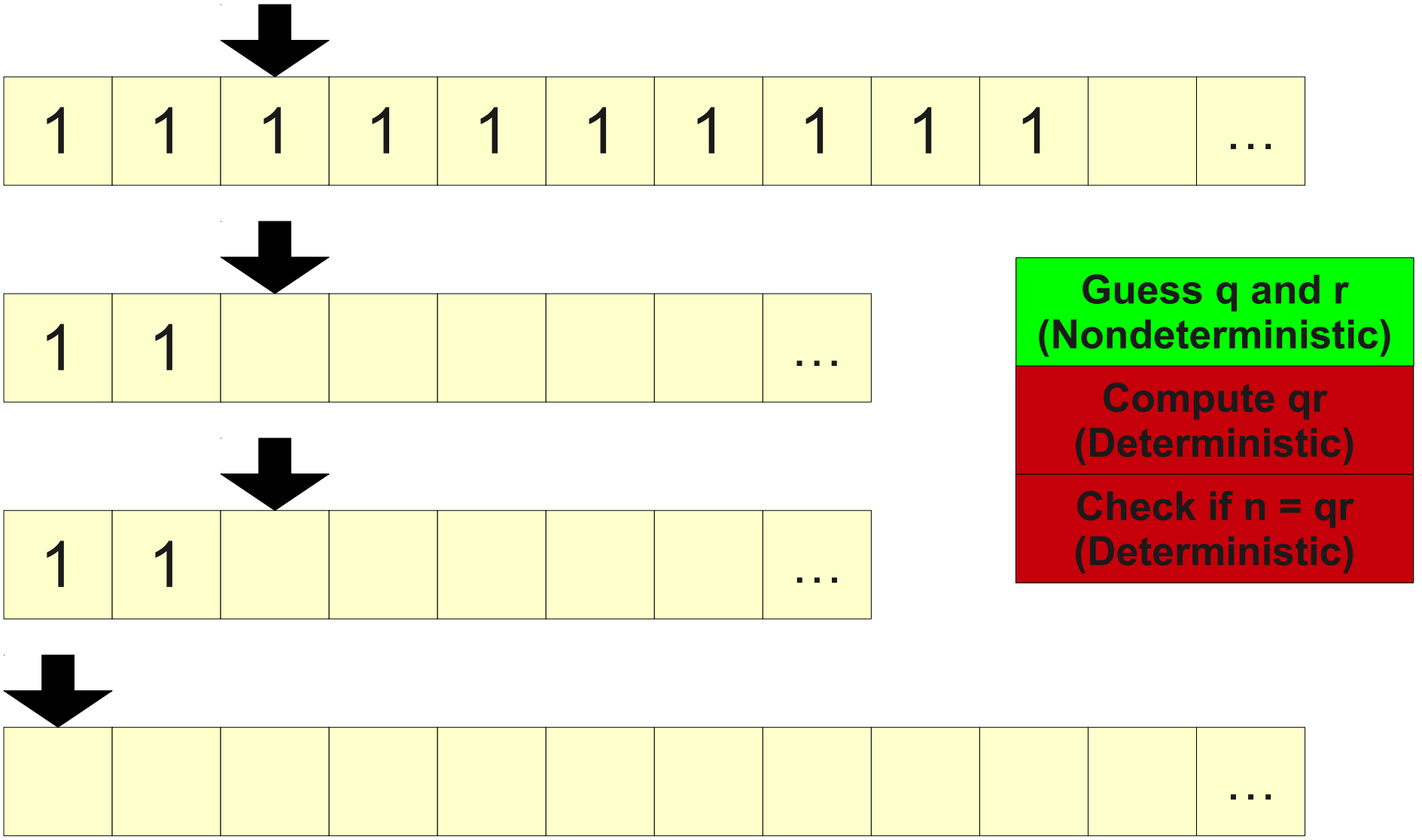
Nondeterministic Algorithms



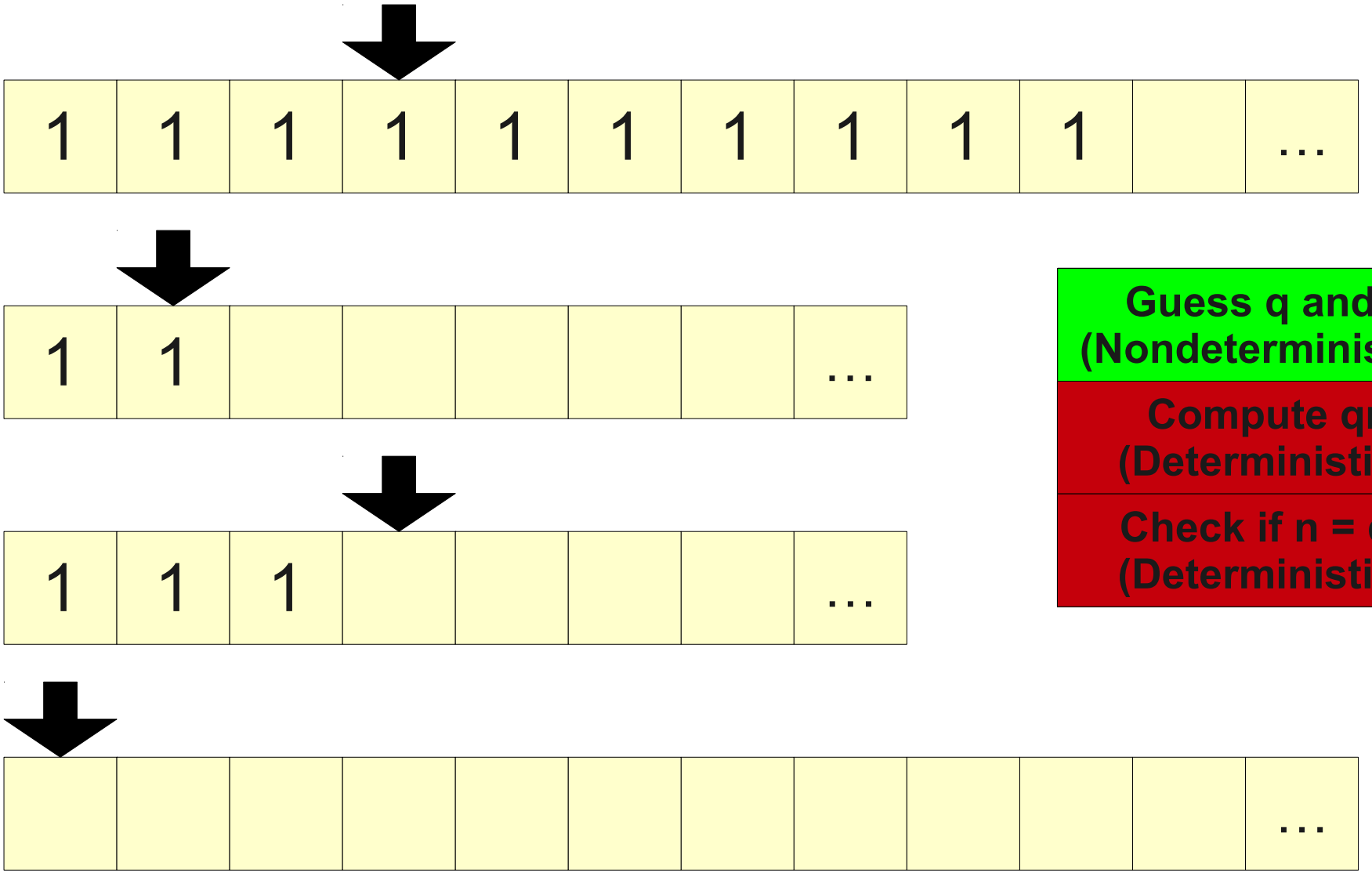
Nondeterministic Algorithms



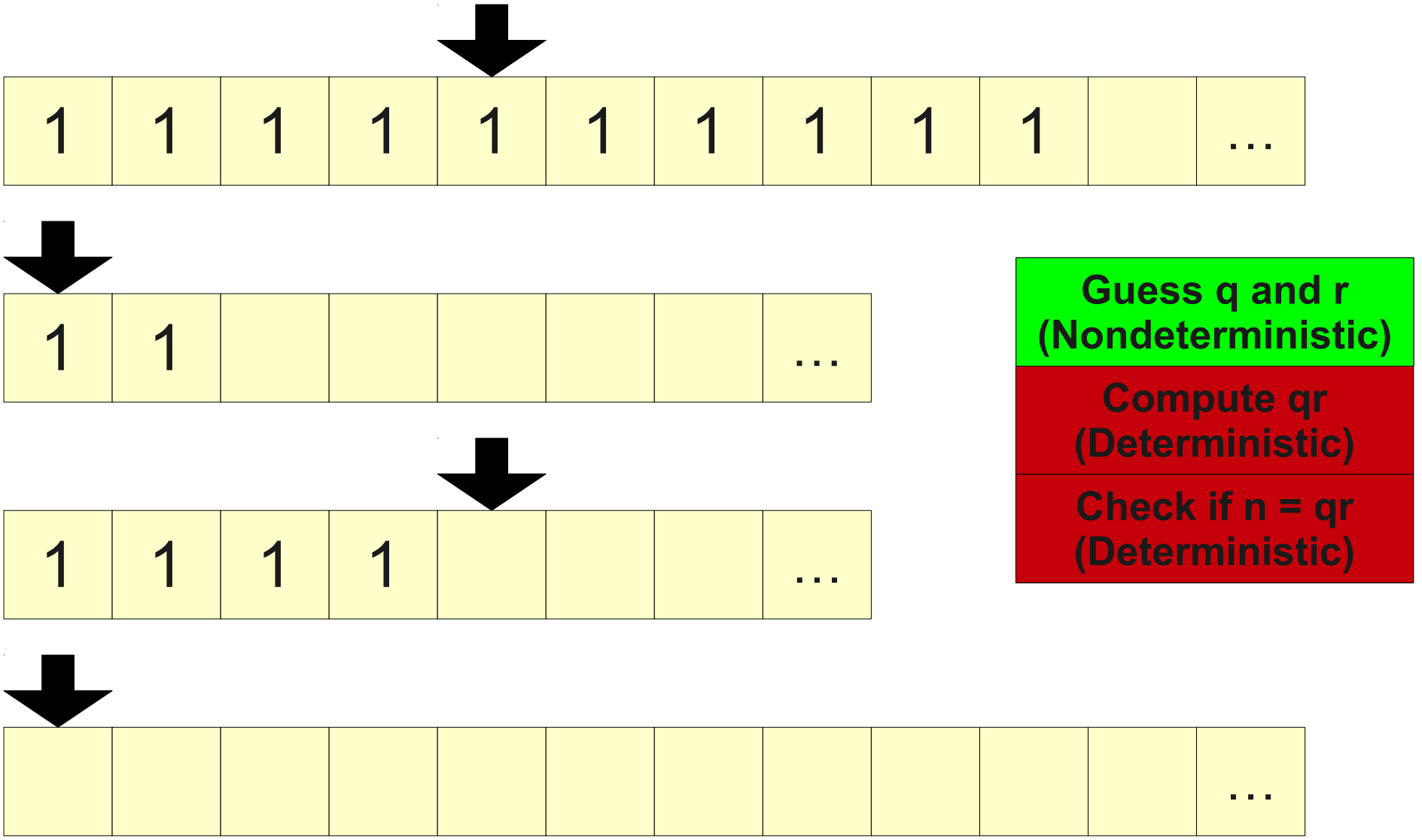
Nondeterministic Algorithms



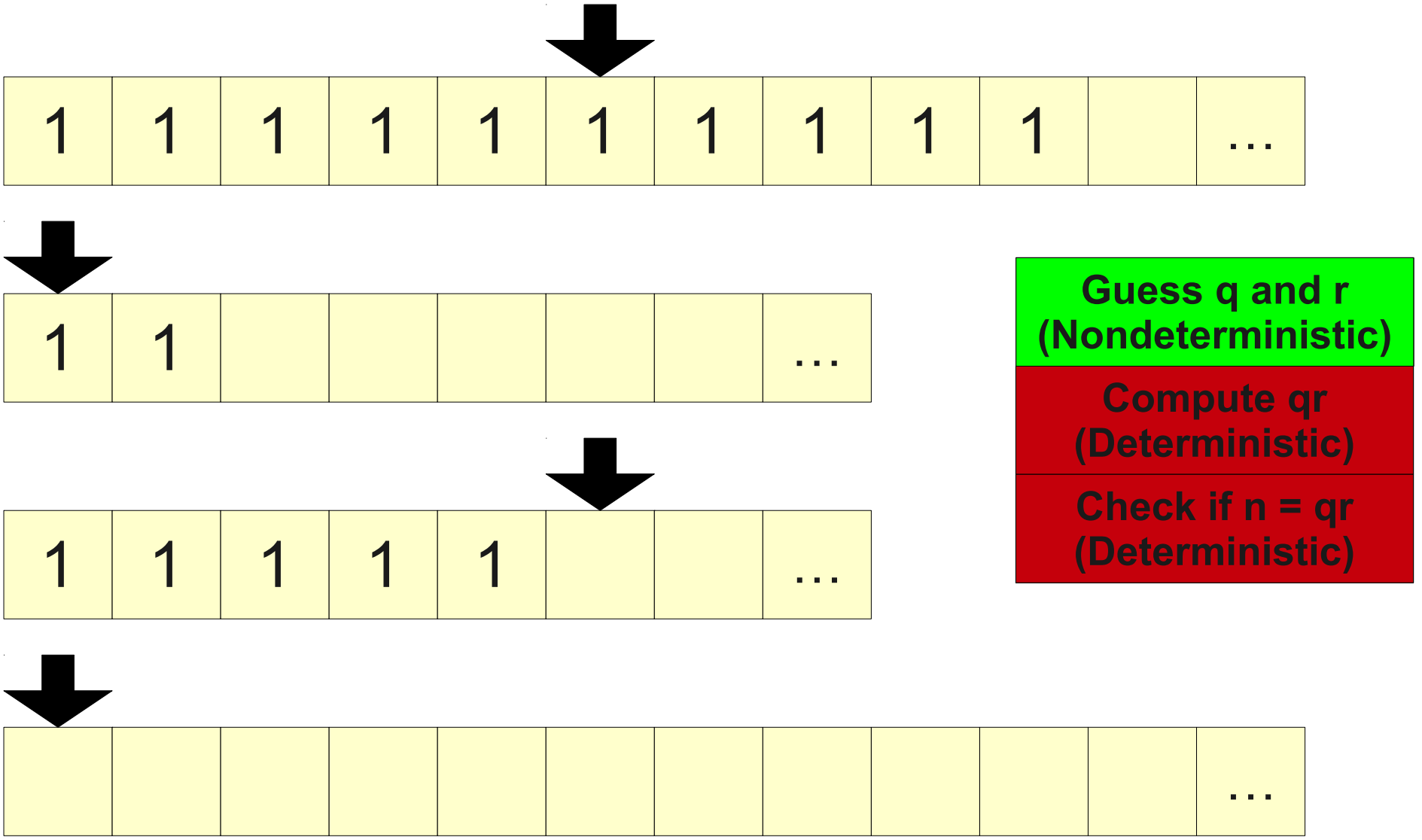
Nondeterministic Algorithms



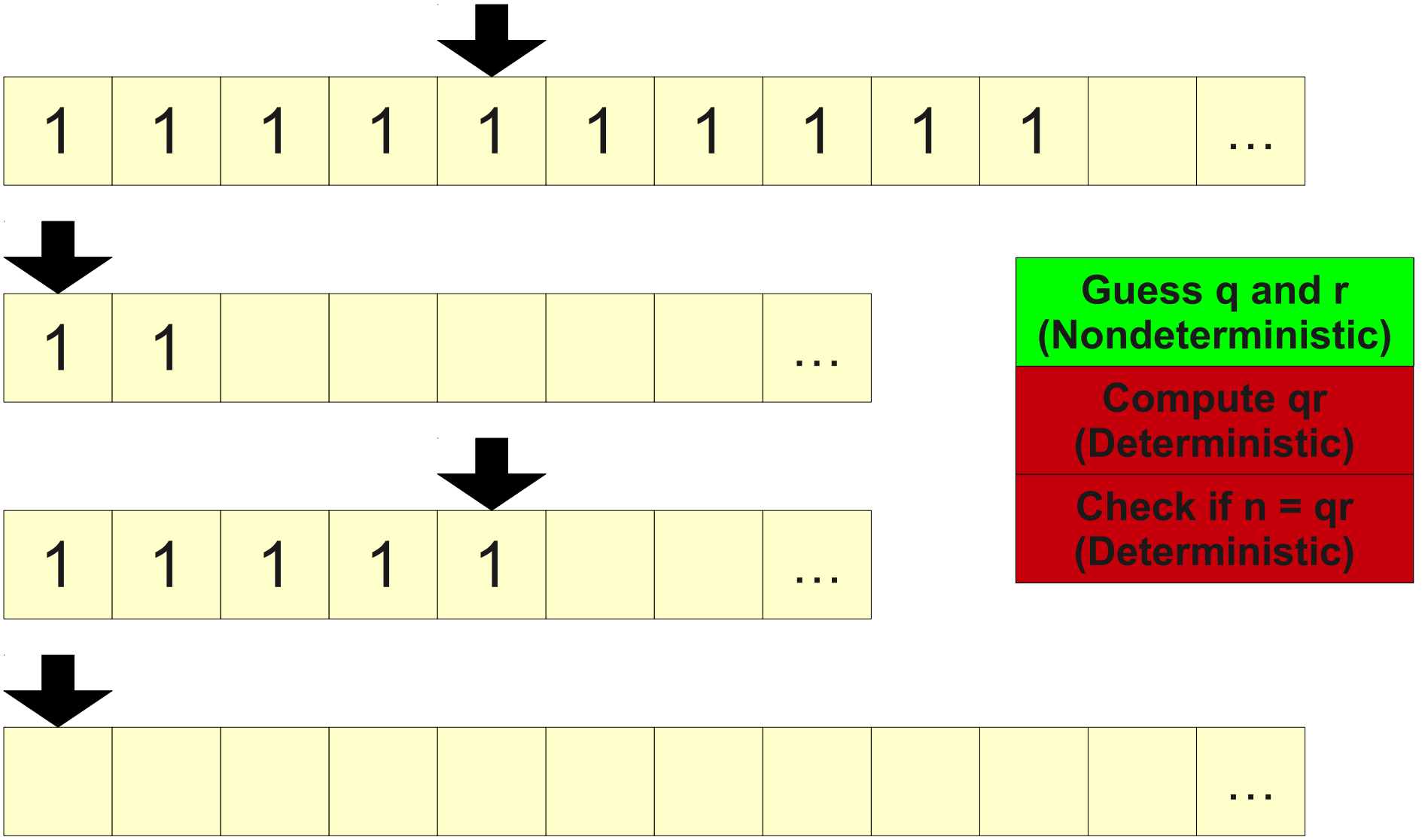
Nondeterministic Algorithms



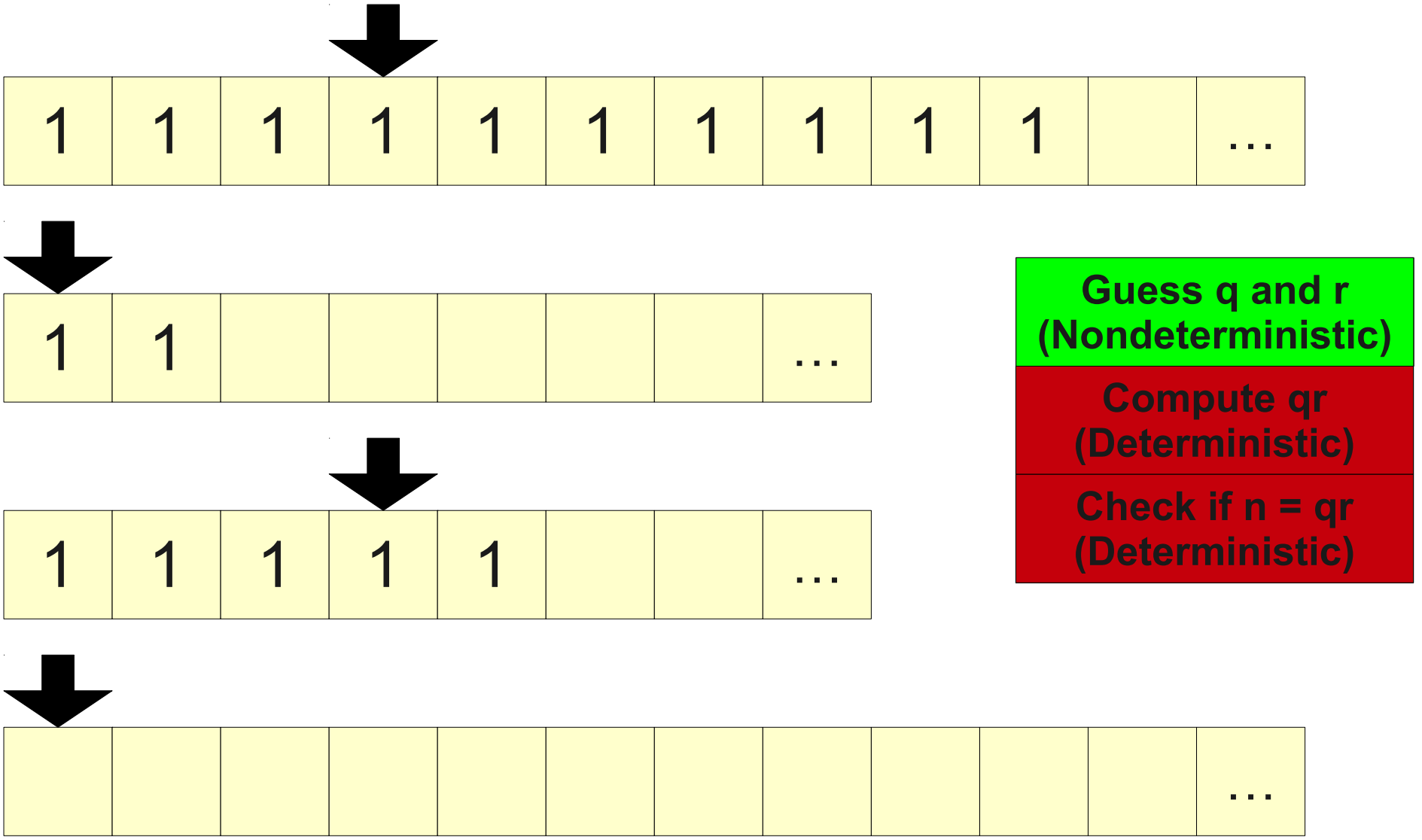
Nondeterministic Algorithms



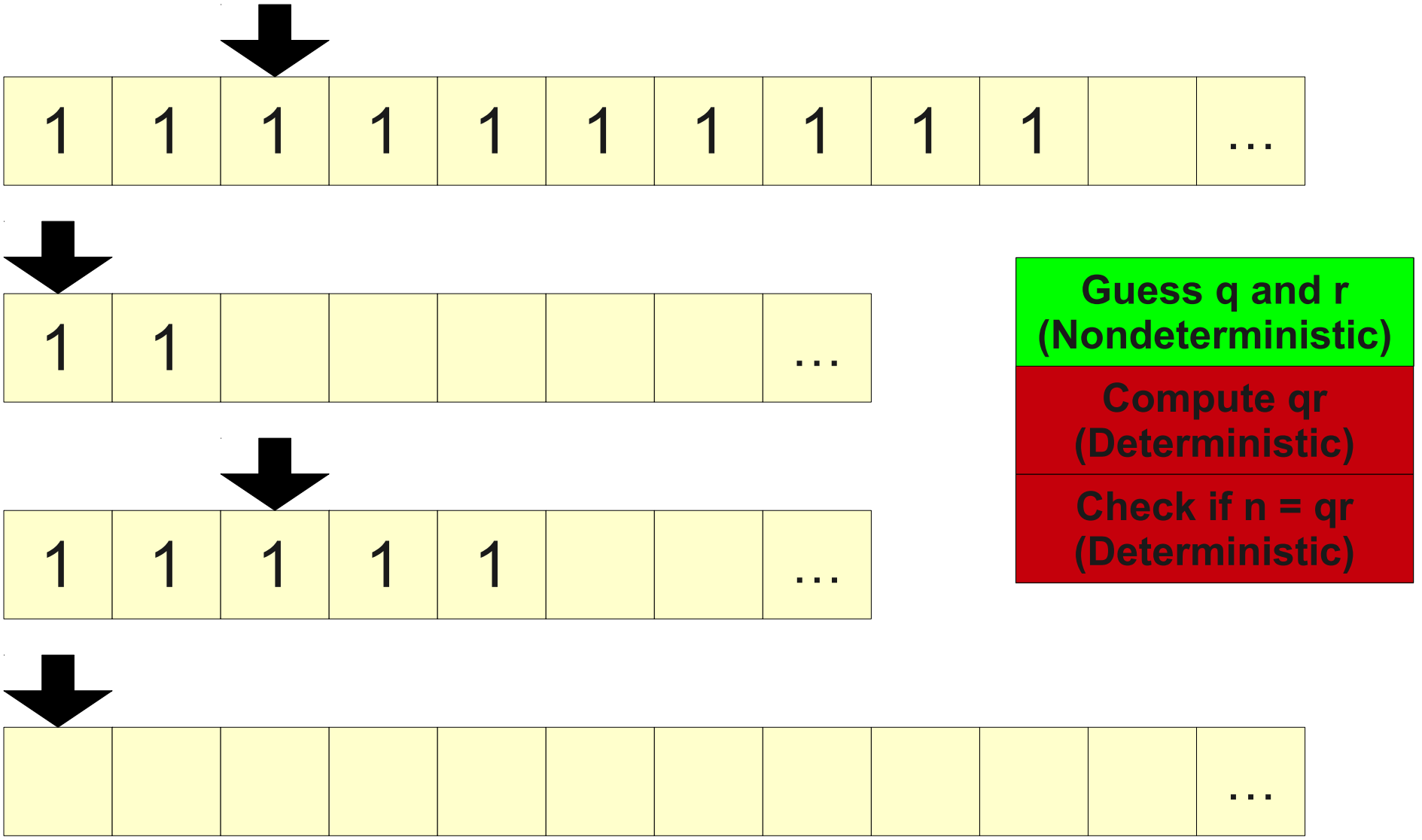
Nondeterministic Algorithms



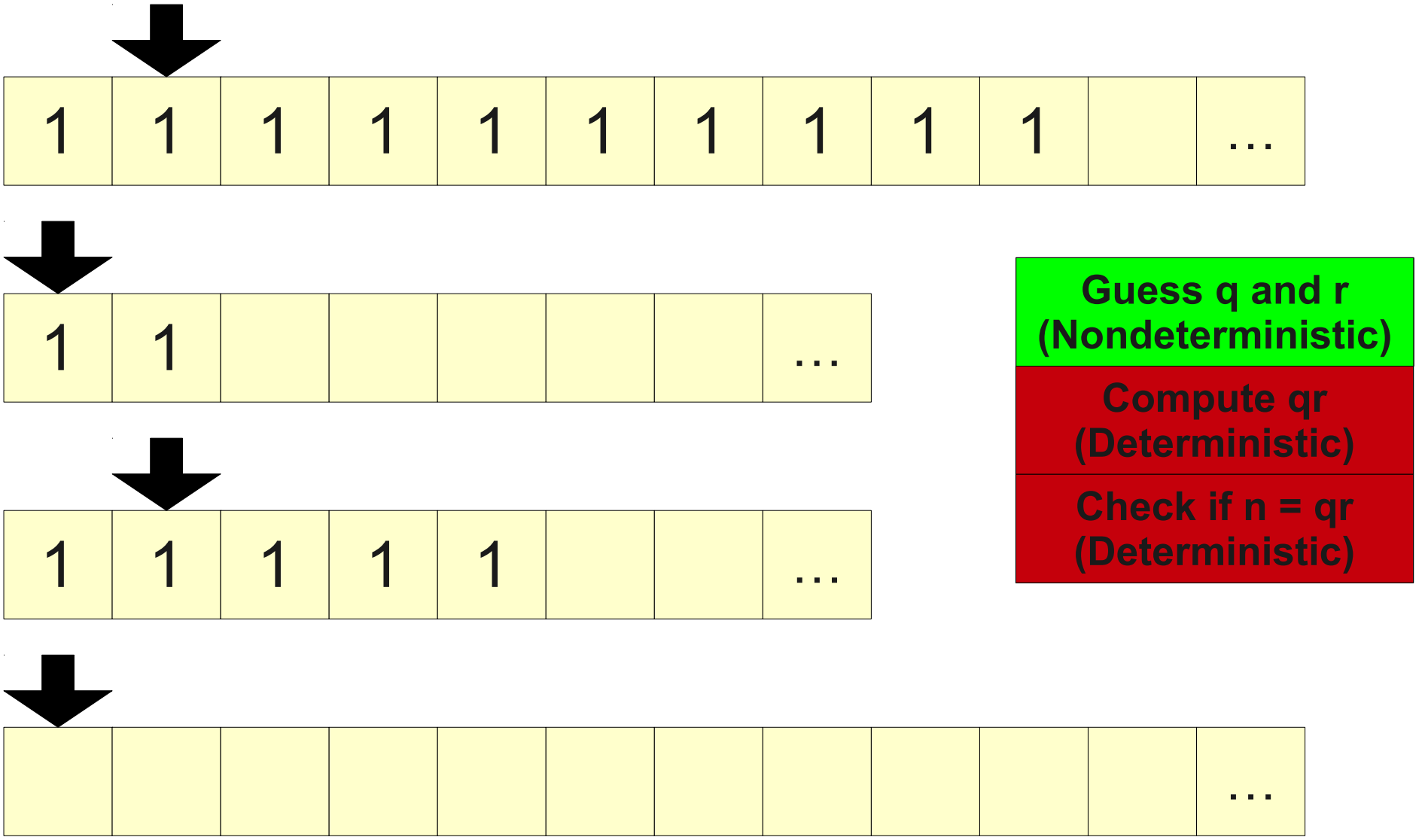
Nondeterministic Algorithms



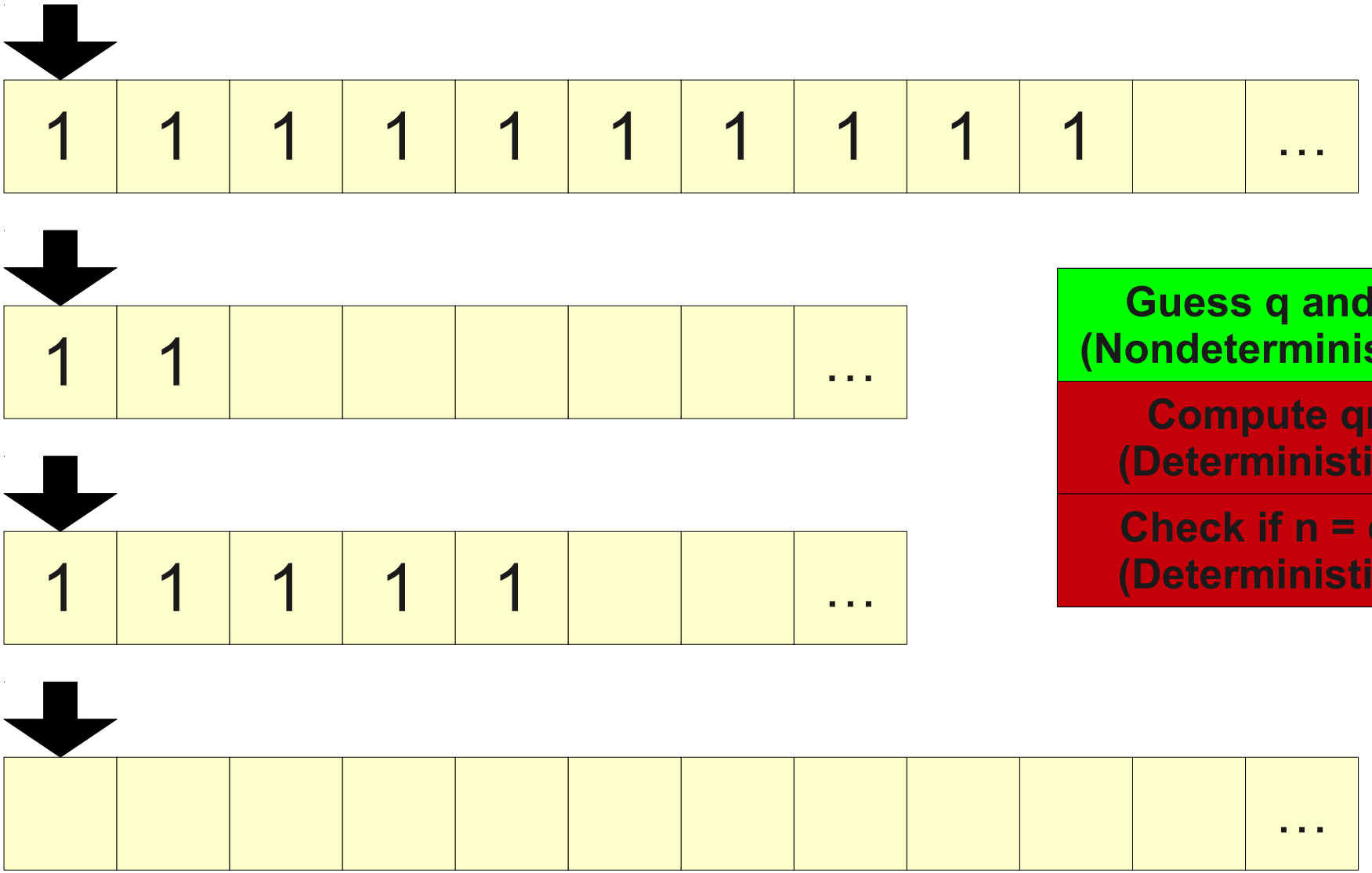
Nondeterministic Algorithms



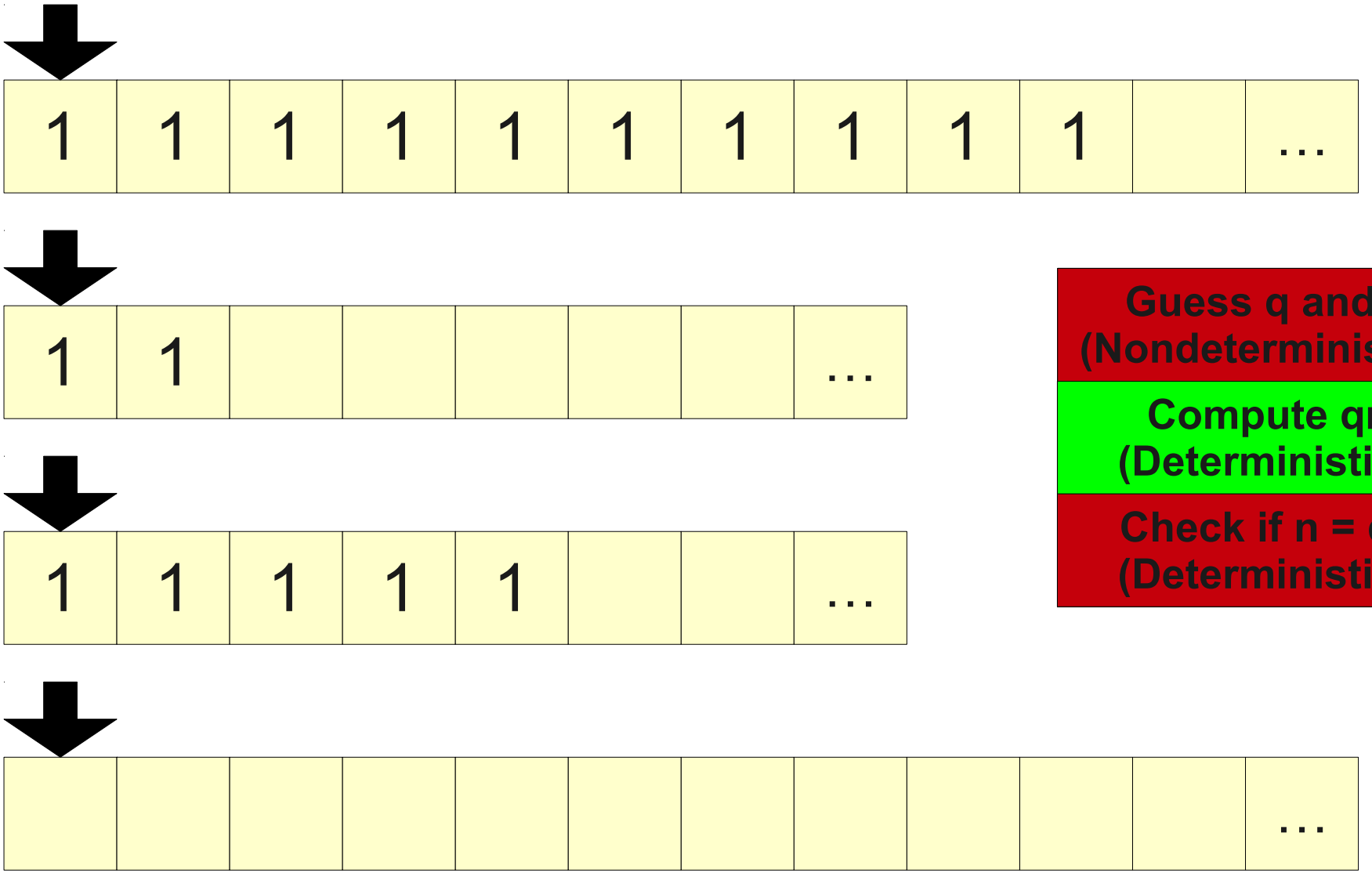
Nondeterministic Algorithms



Nondeterministic Algorithms



Nondeterministic Algorithms

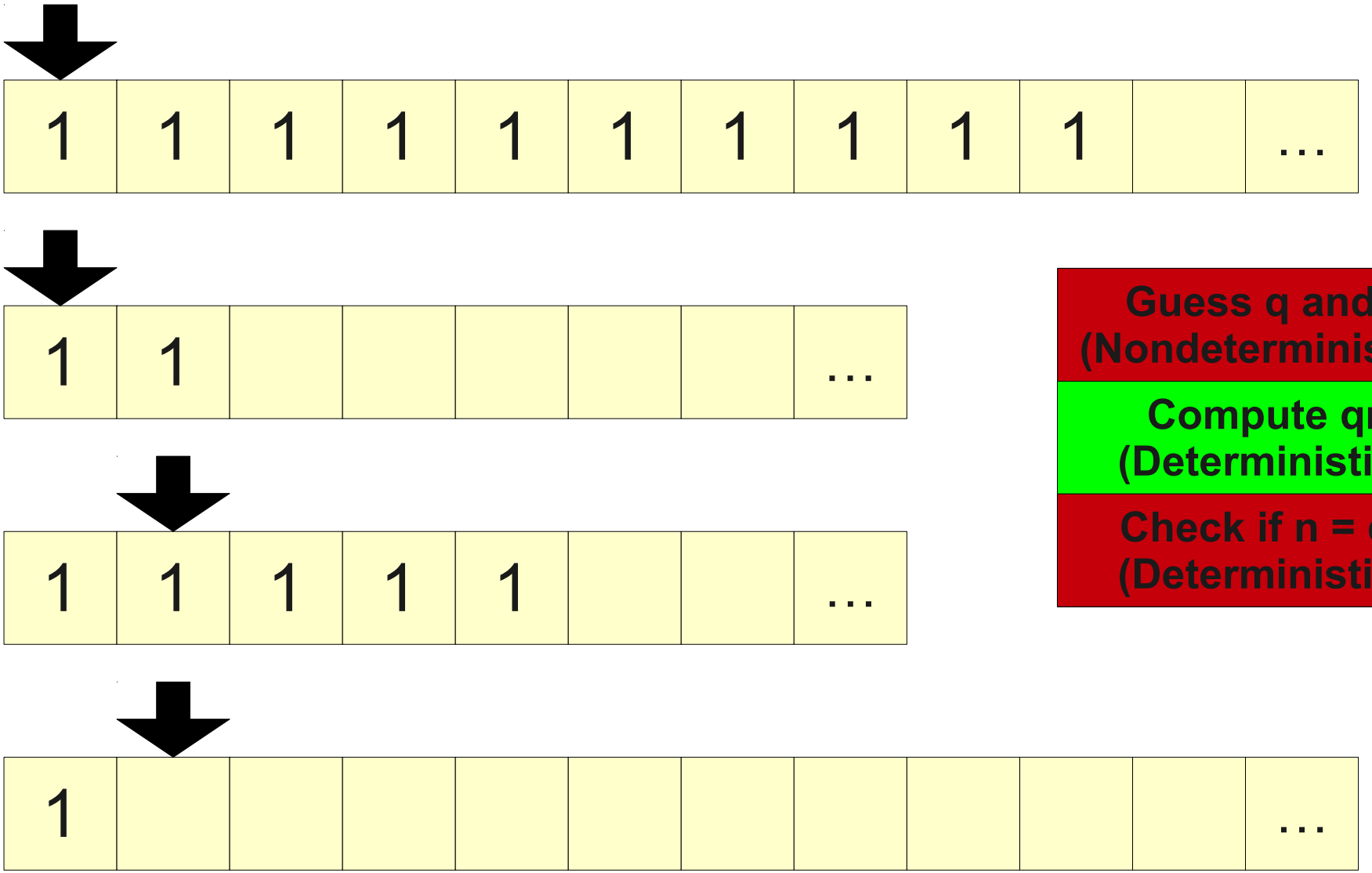


Guess q and r (Nondeterministic)

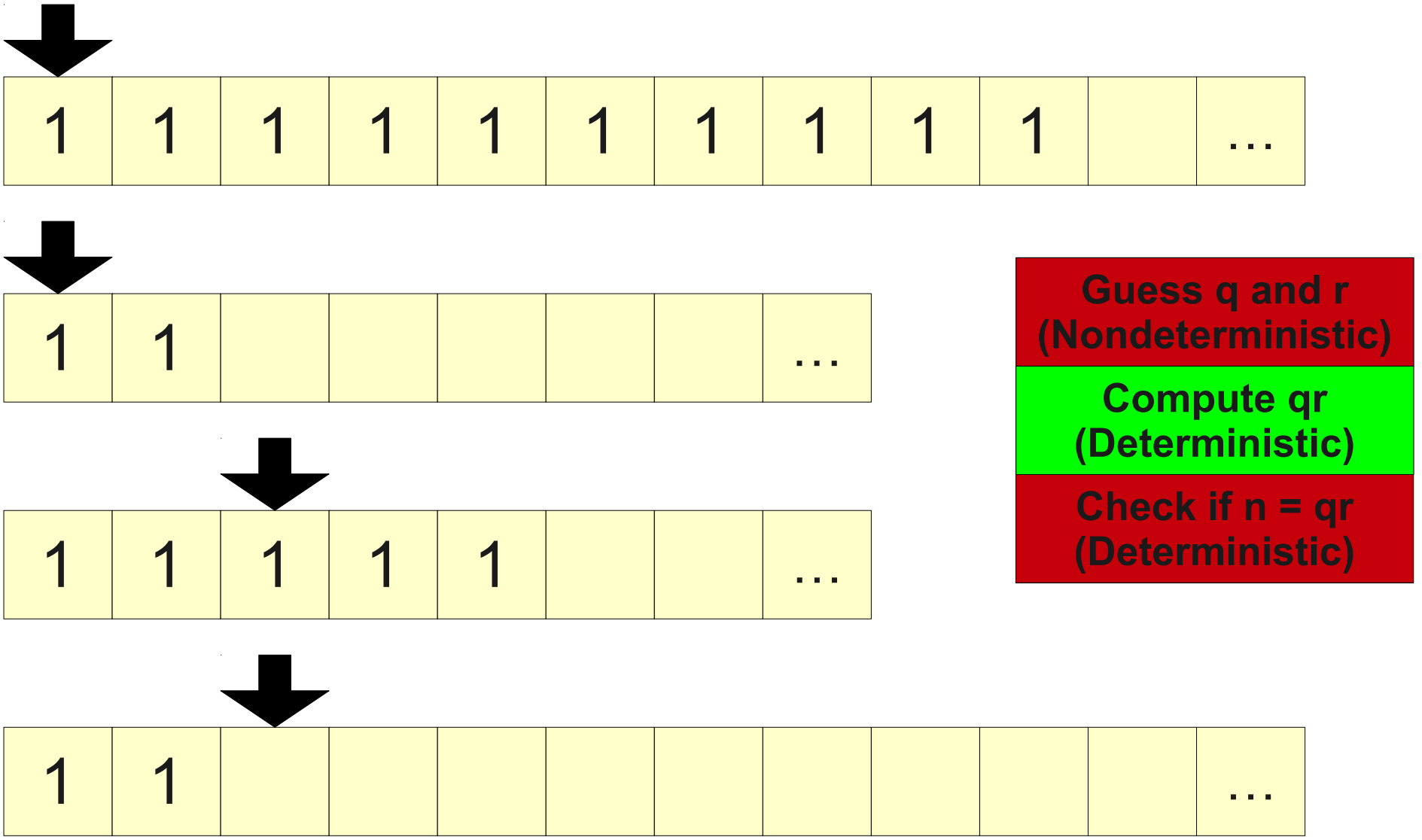
Compute qr (Deterministic)

Check if n = qr (Deterministic)

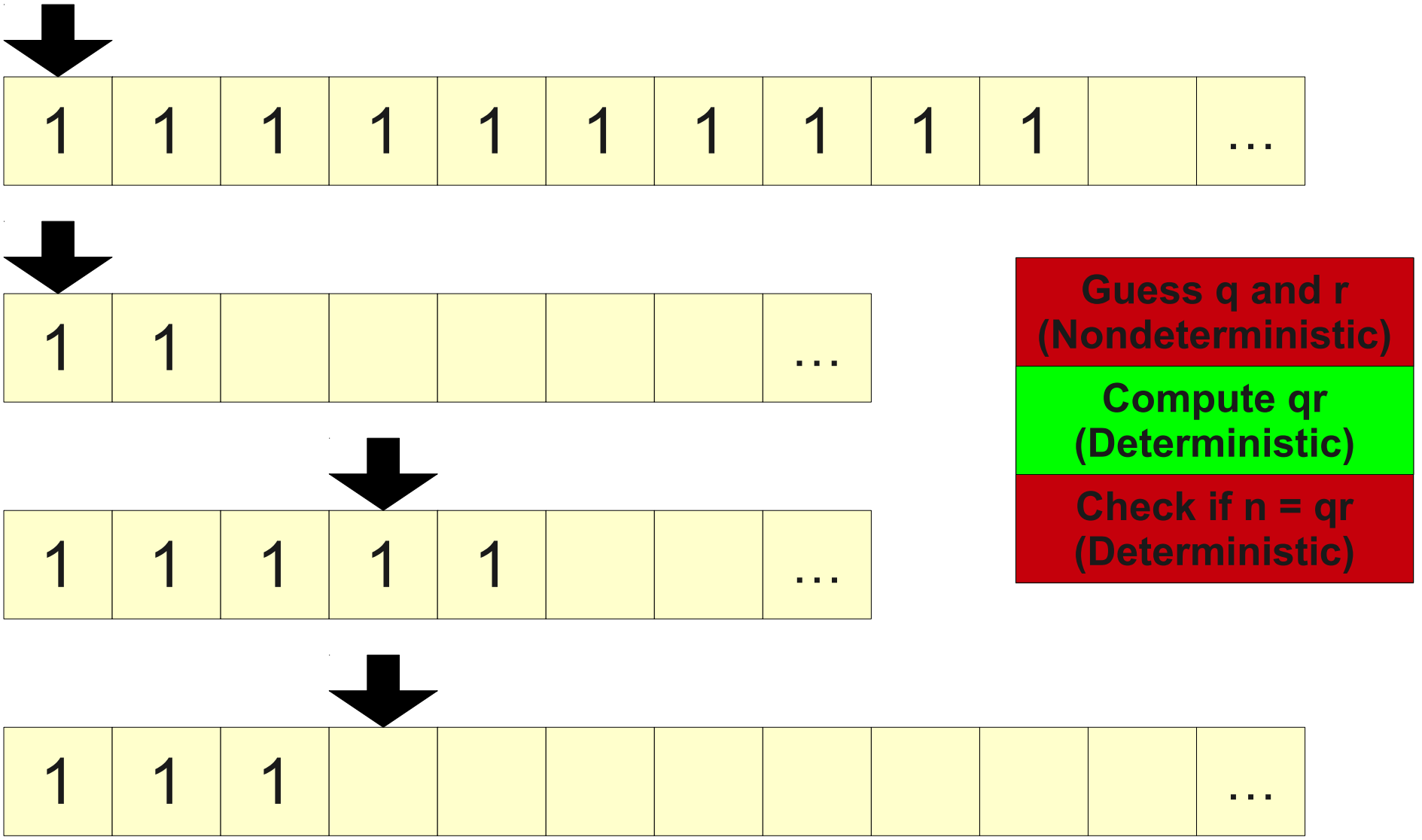
Nondeterministic Algorithms



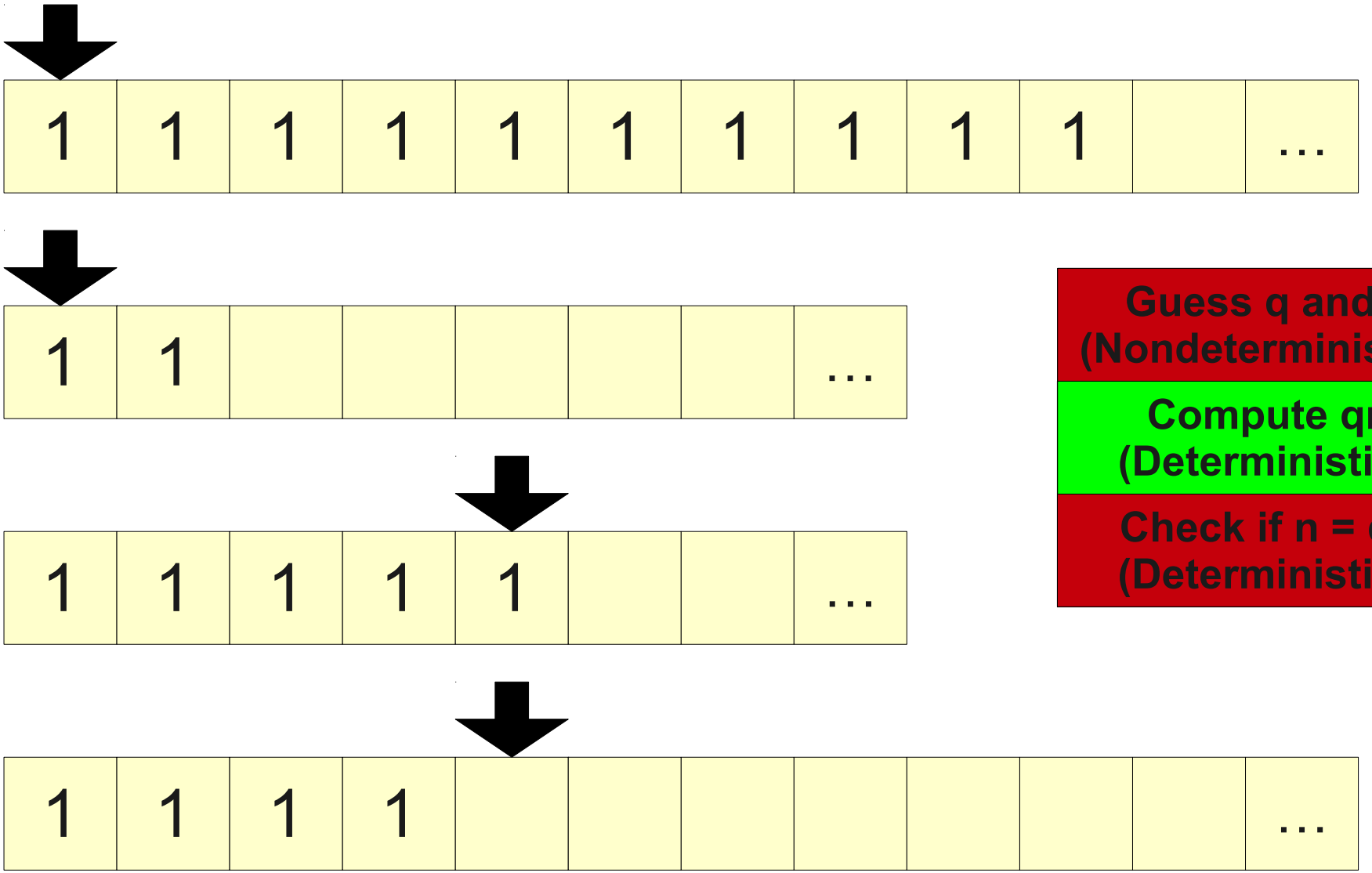
Nondeterministic Algorithms



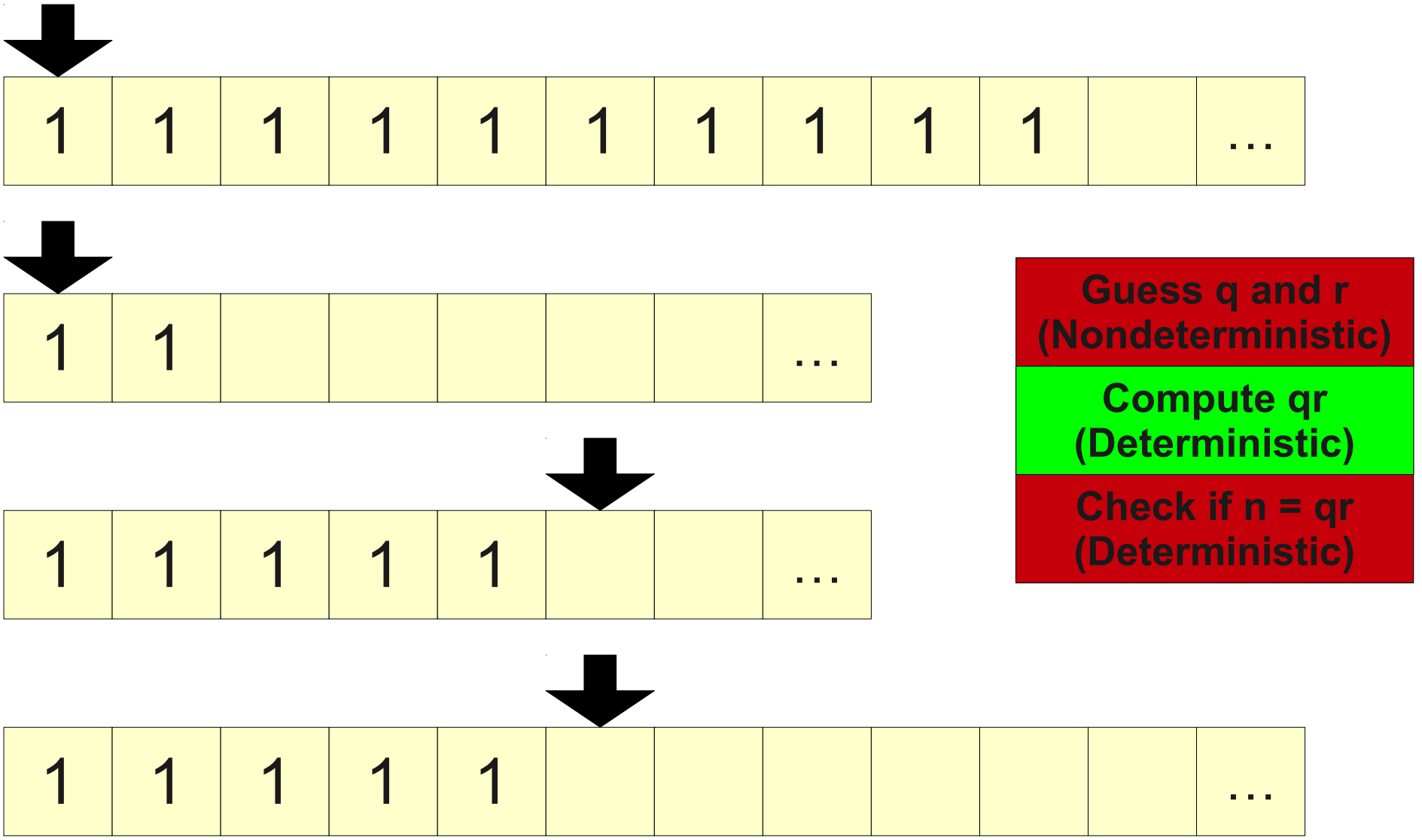
Nondeterministic Algorithms



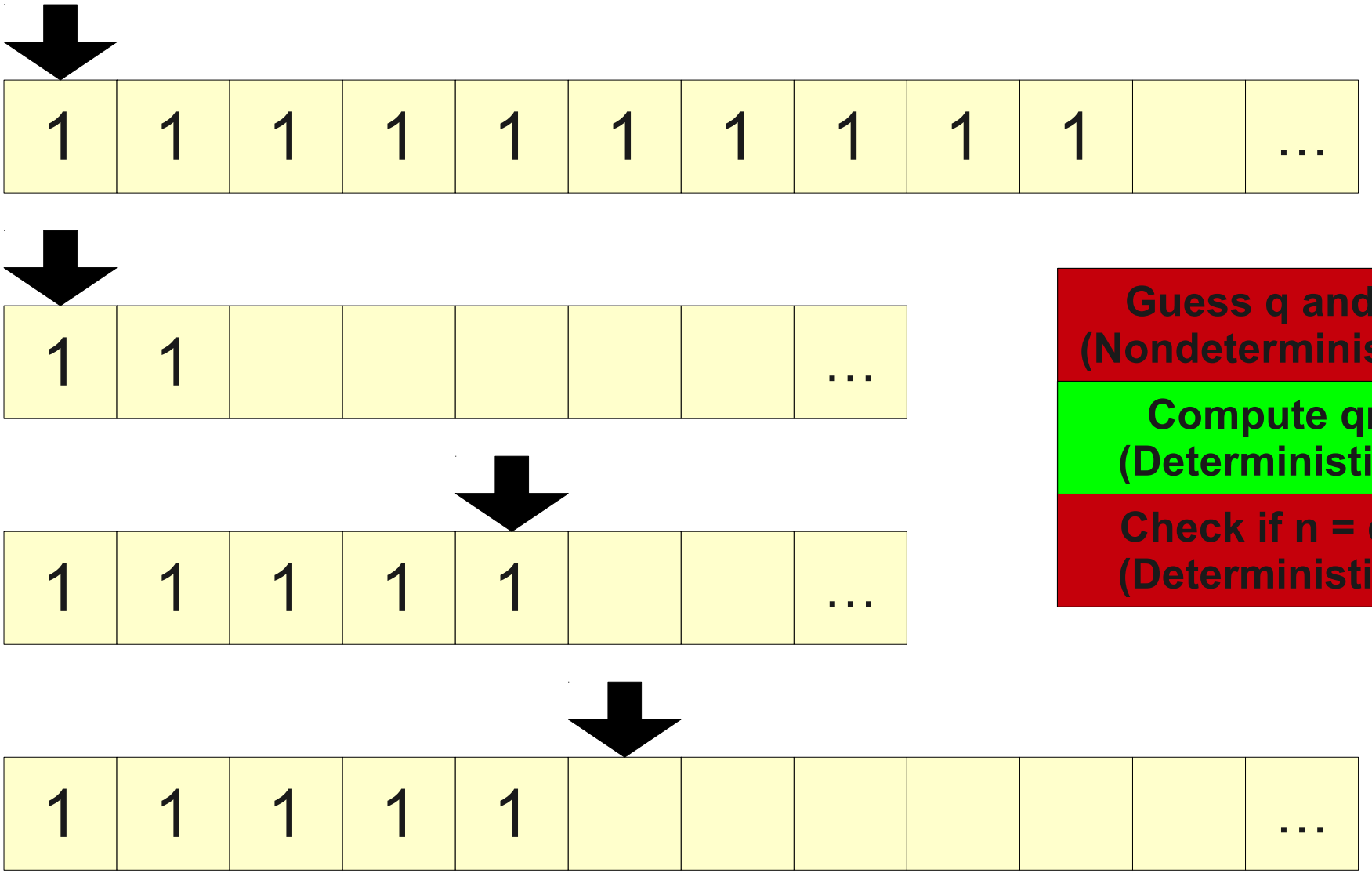
Nondeterministic Algorithms



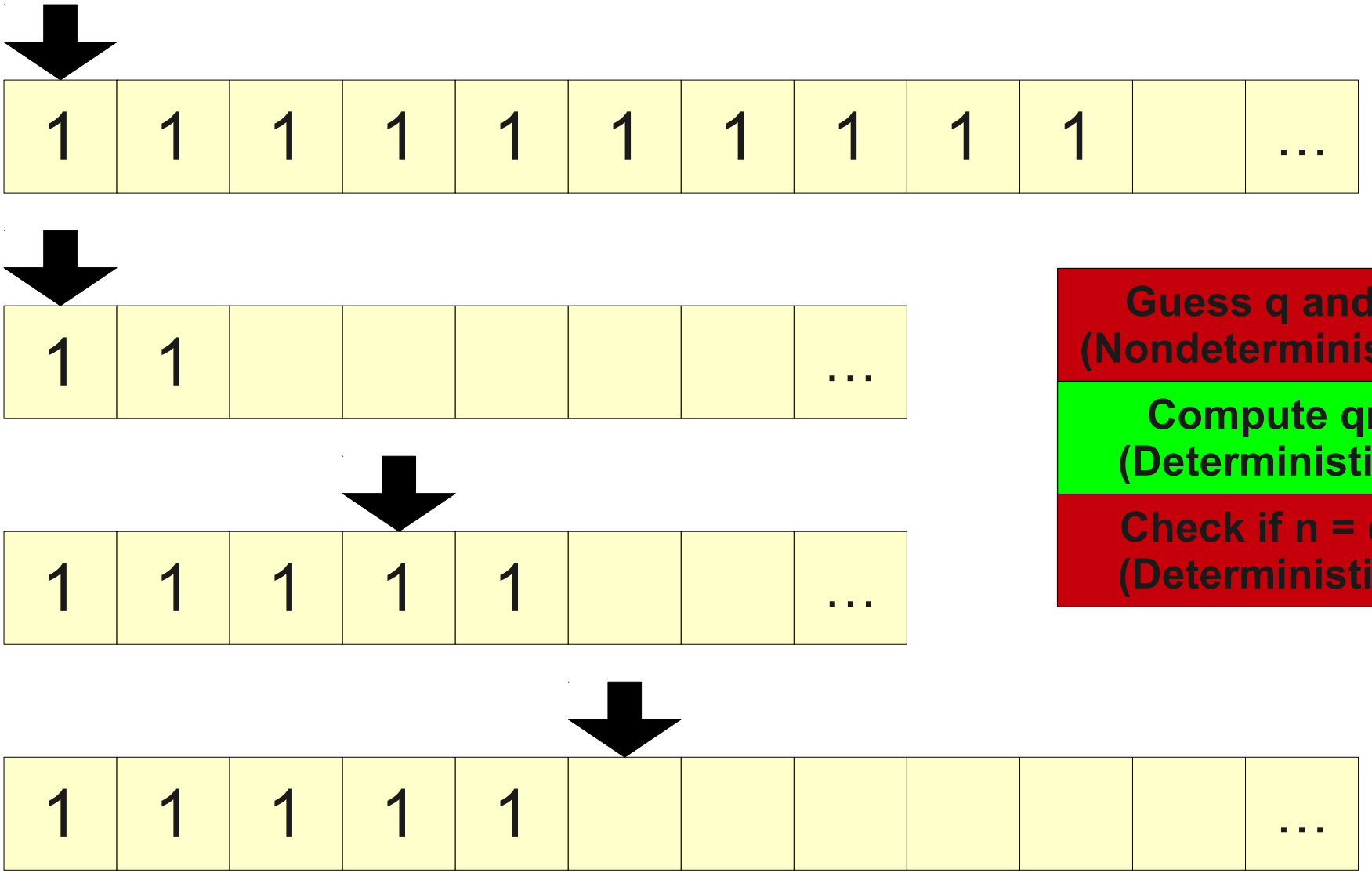
Nondeterministic Algorithms



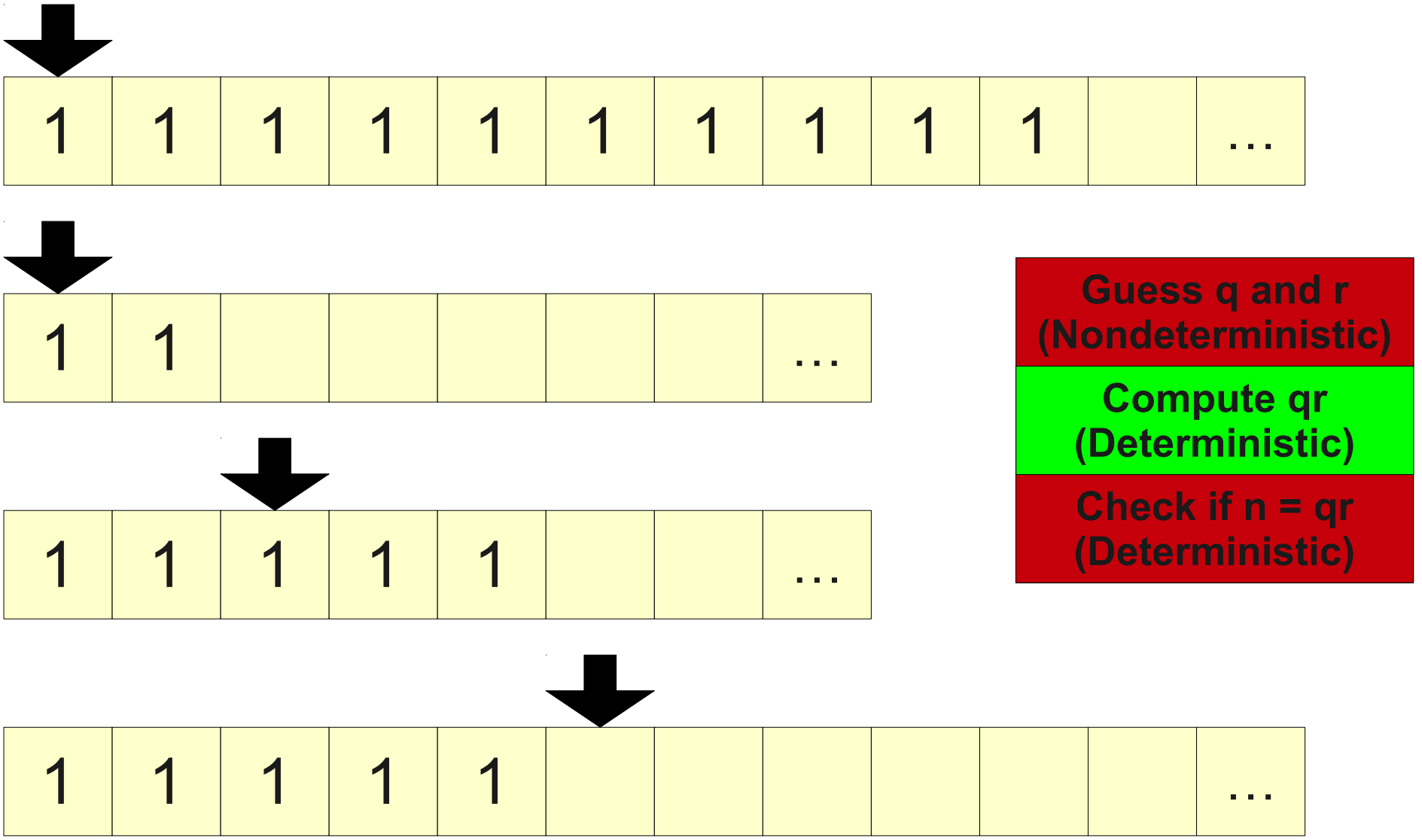
Nondeterministic Algorithms



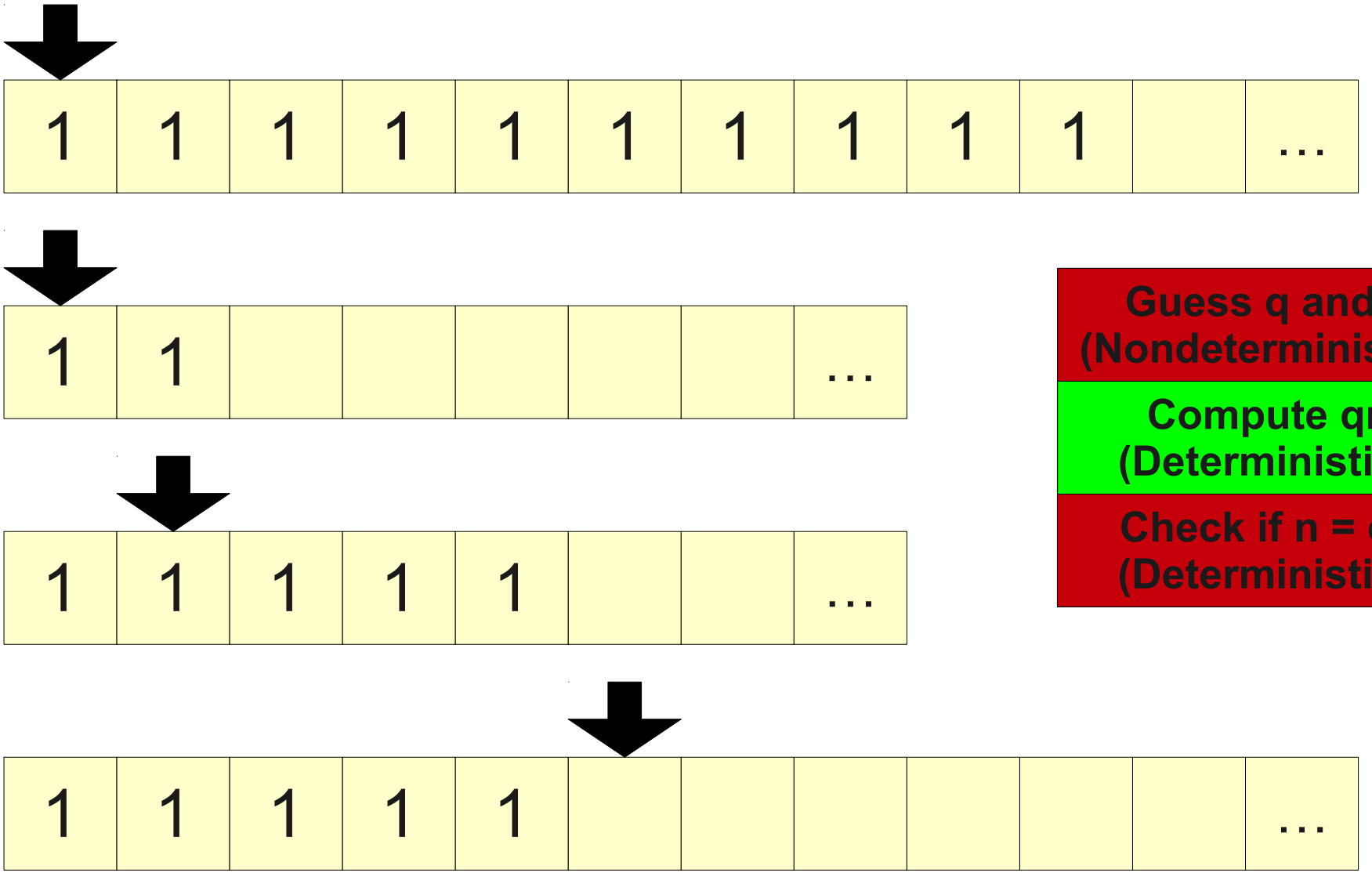
Nondeterministic Algorithms



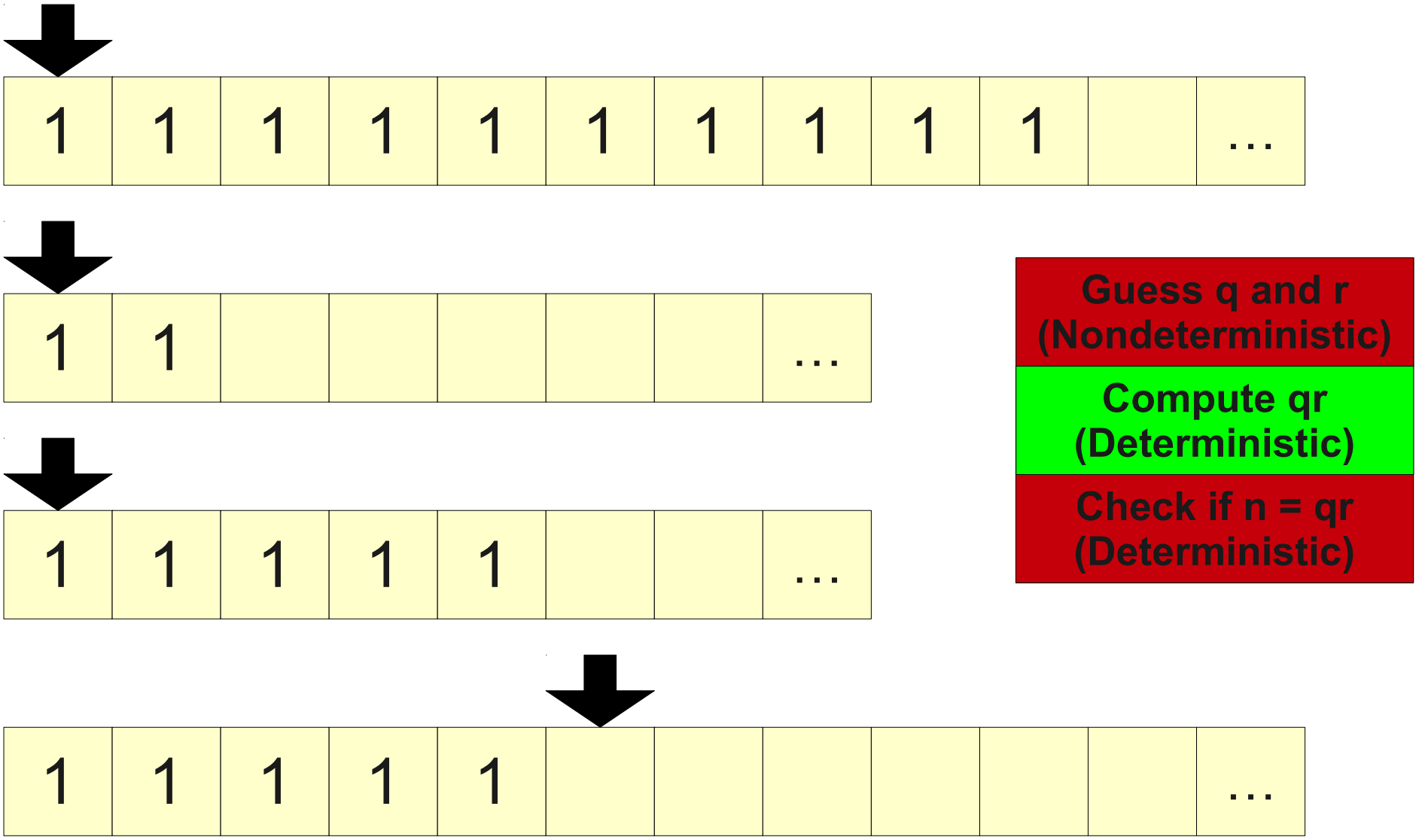
Nondeterministic Algorithms



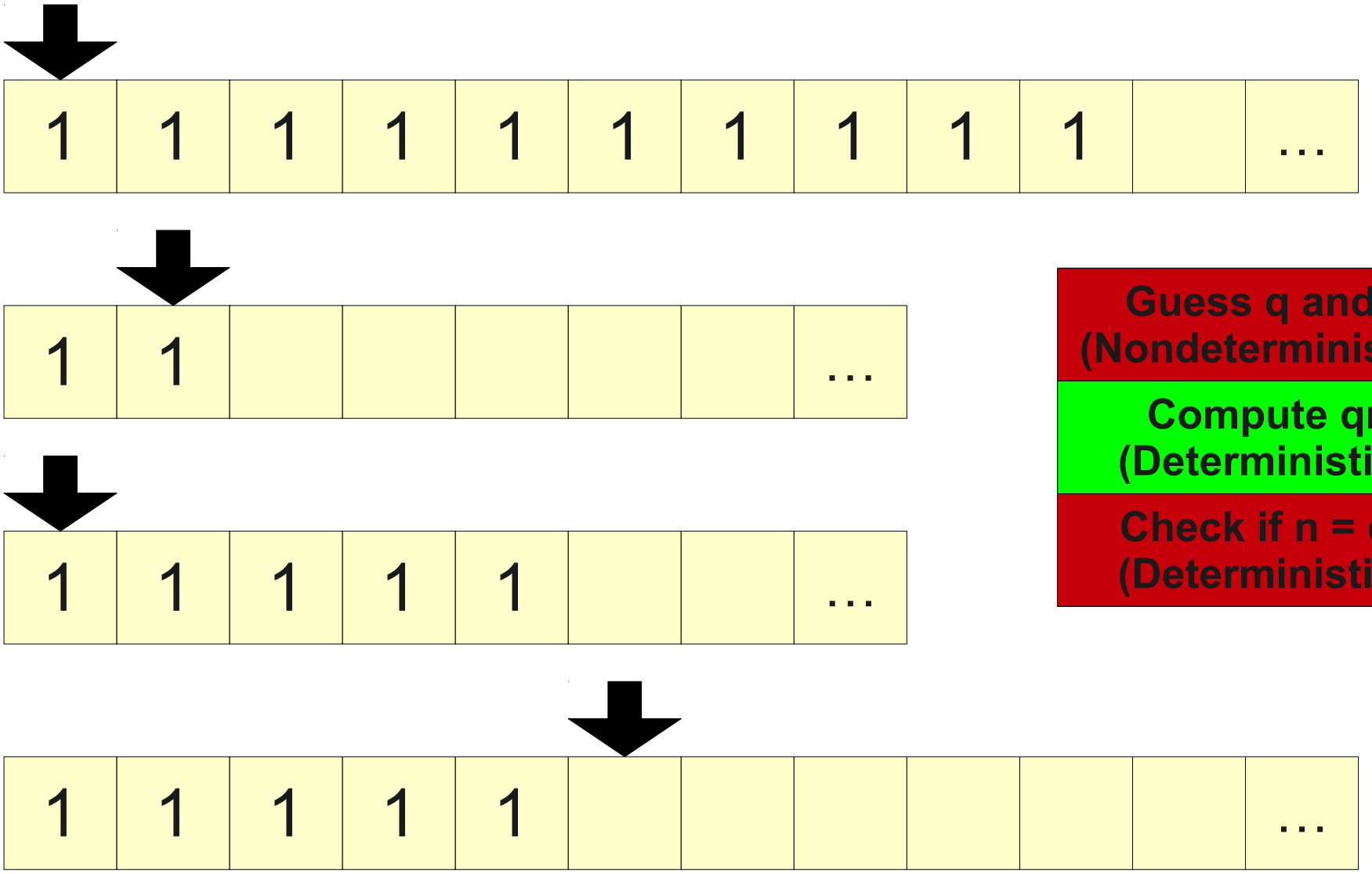
Nondeterministic Algorithms



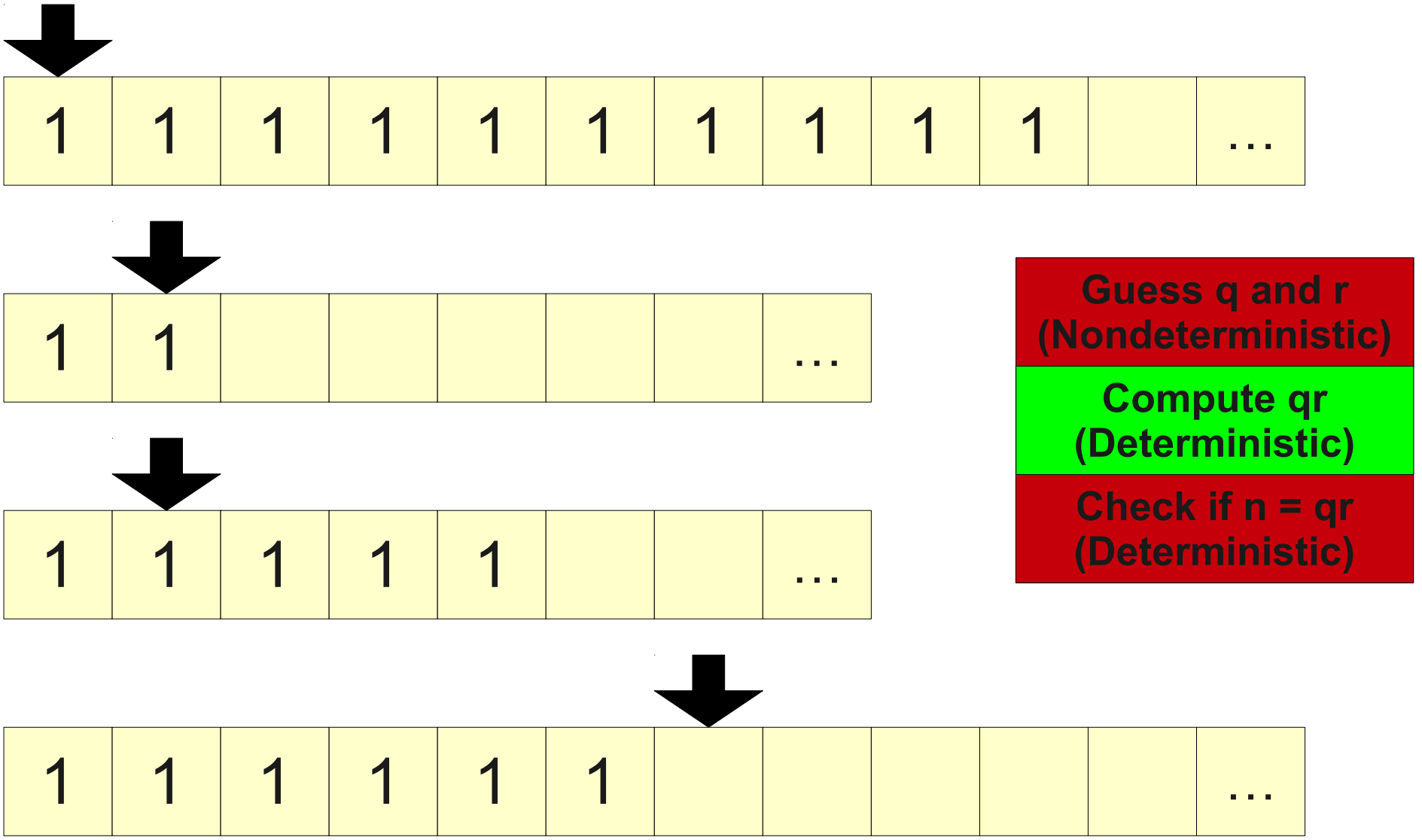
Nondeterministic Algorithms



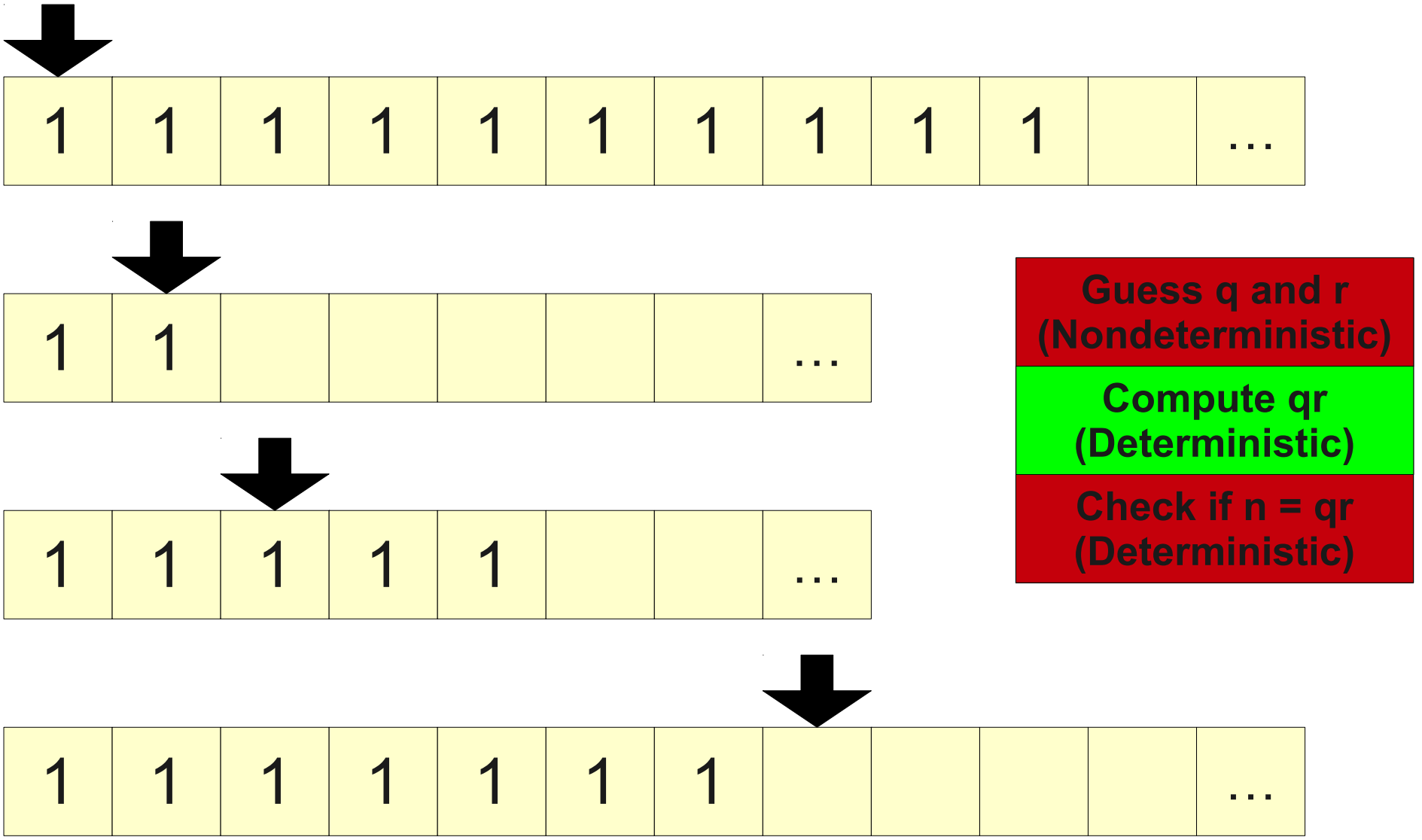
Nondeterministic Algorithms



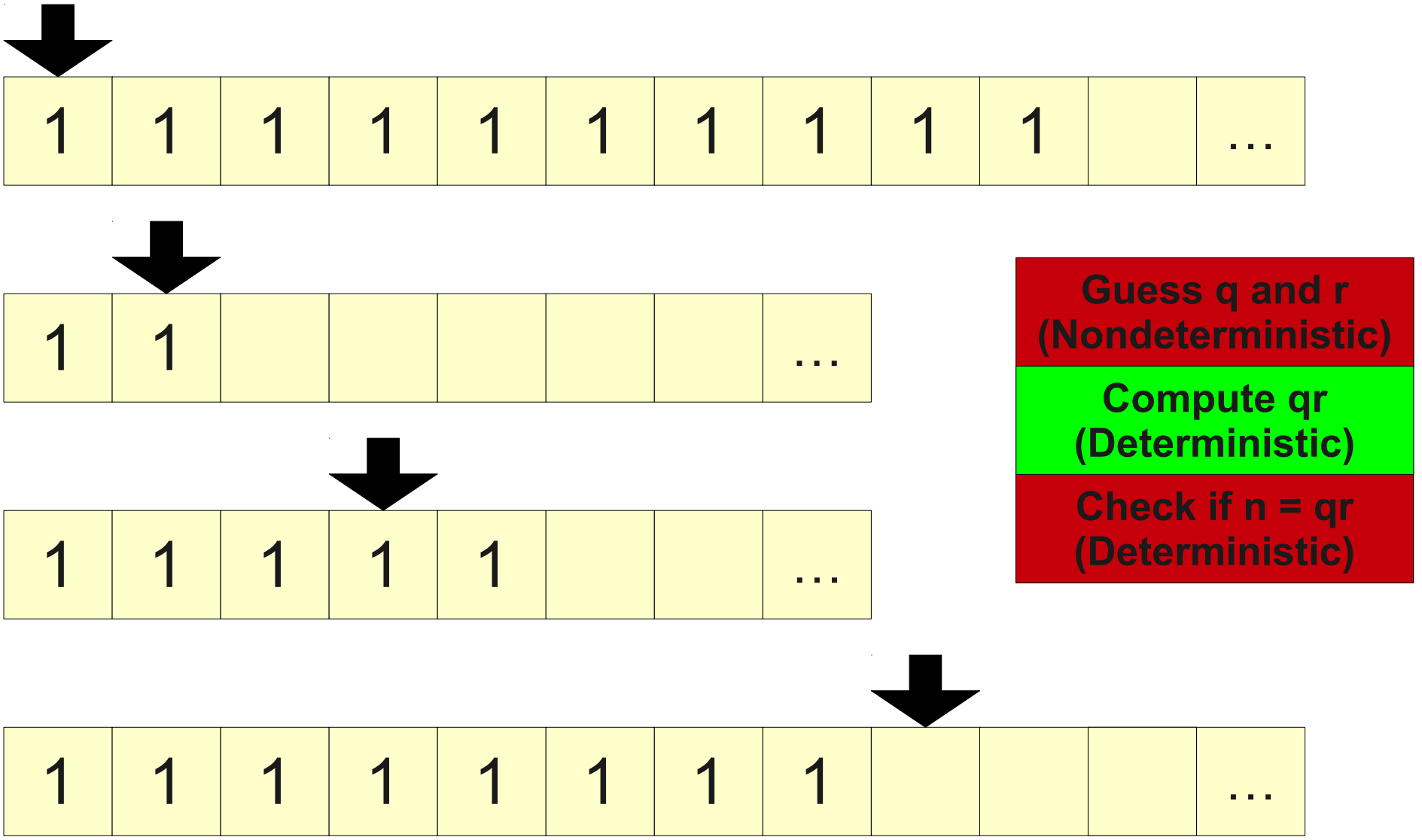
Nondeterministic Algorithms



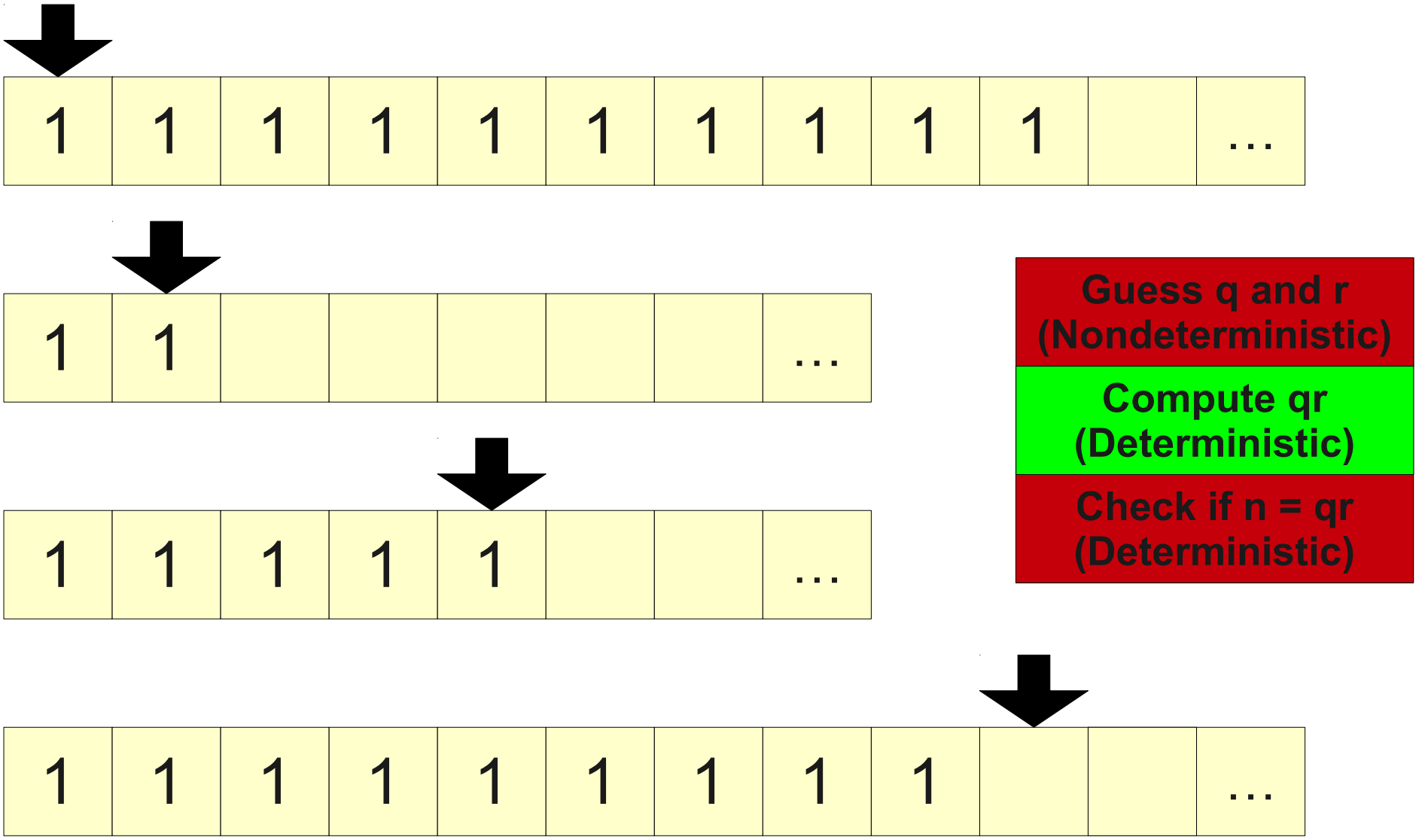
Nondeterministic Algorithms



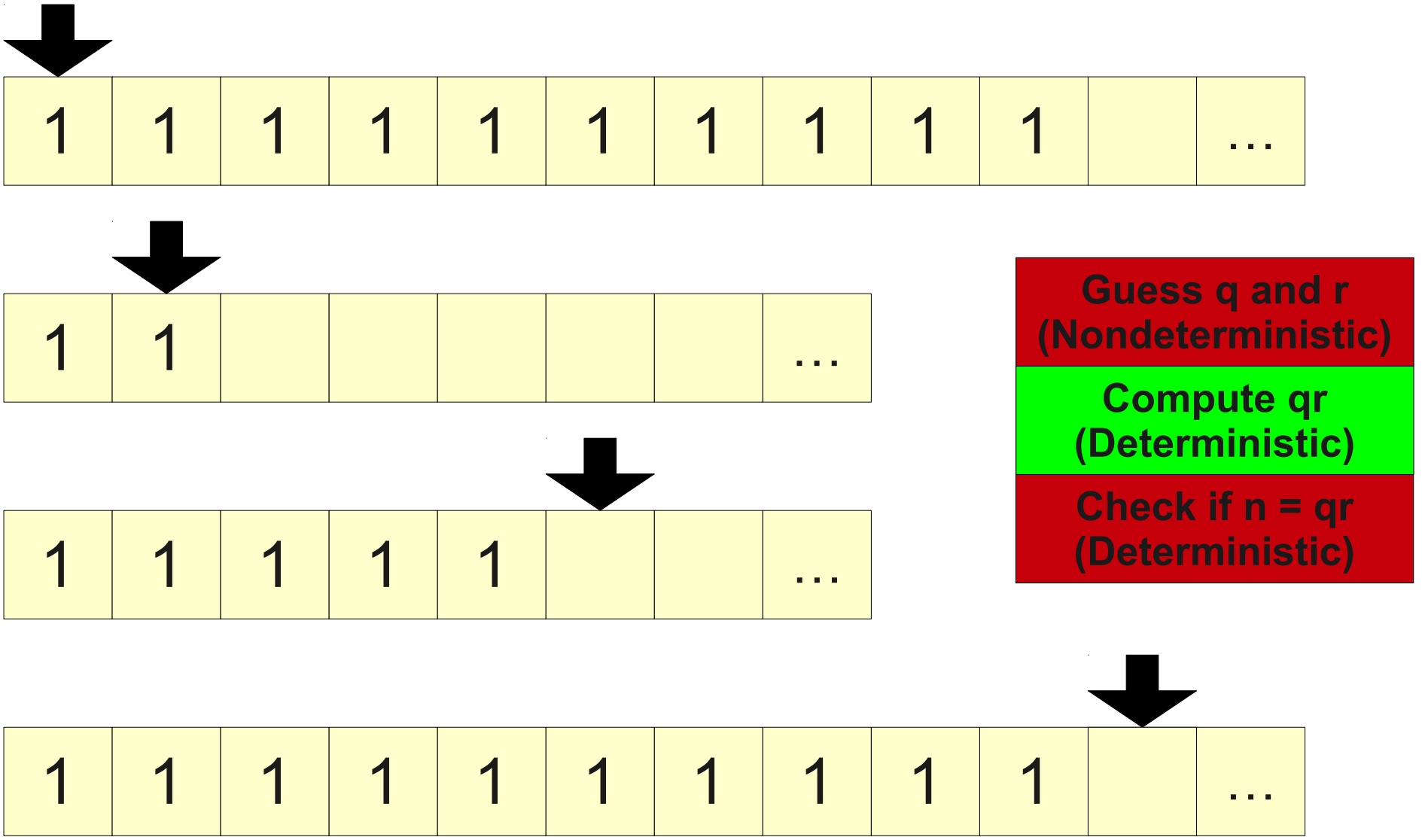
Nondeterministic Algorithms



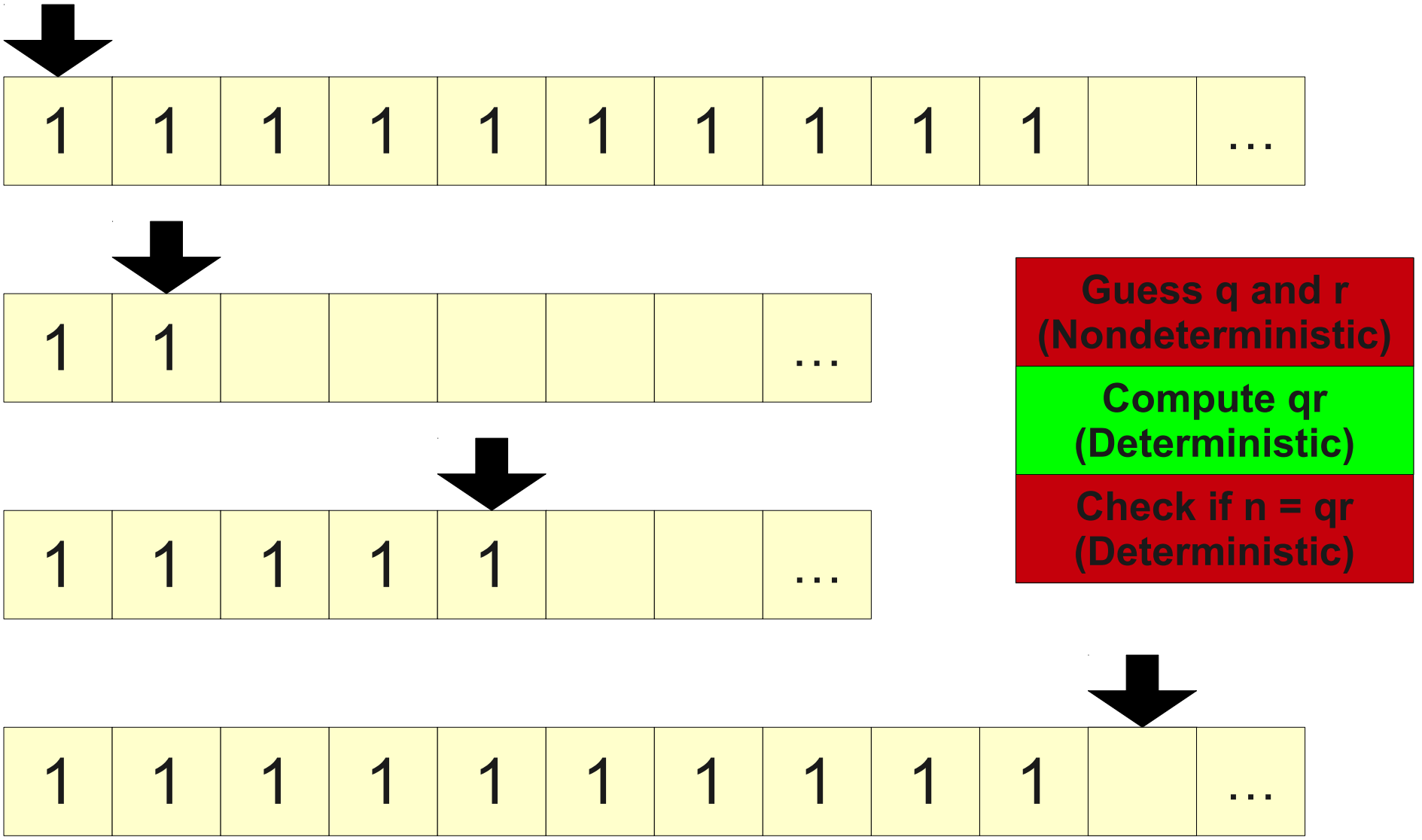
Nondeterministic Algorithms



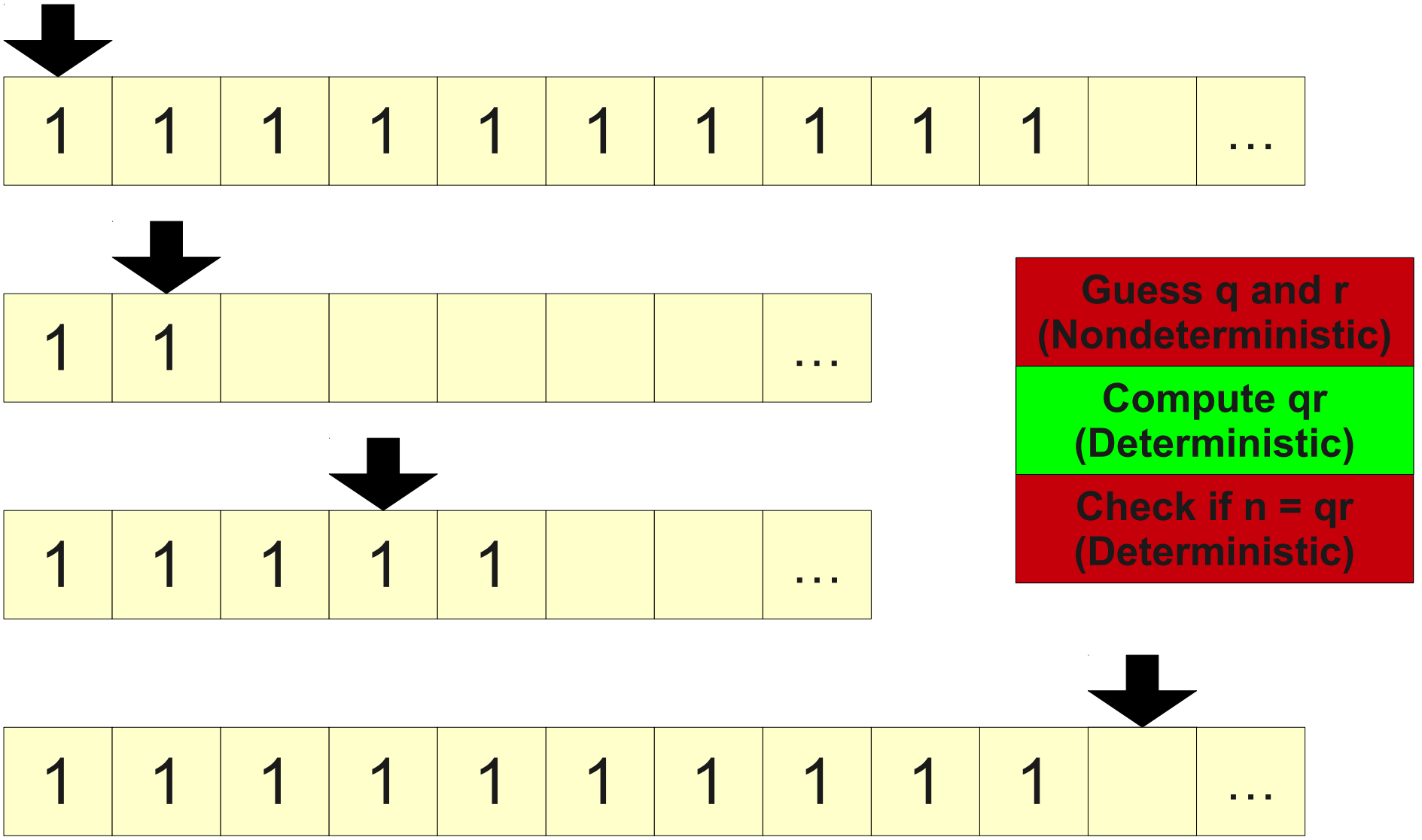
Nondeterministic Algorithms



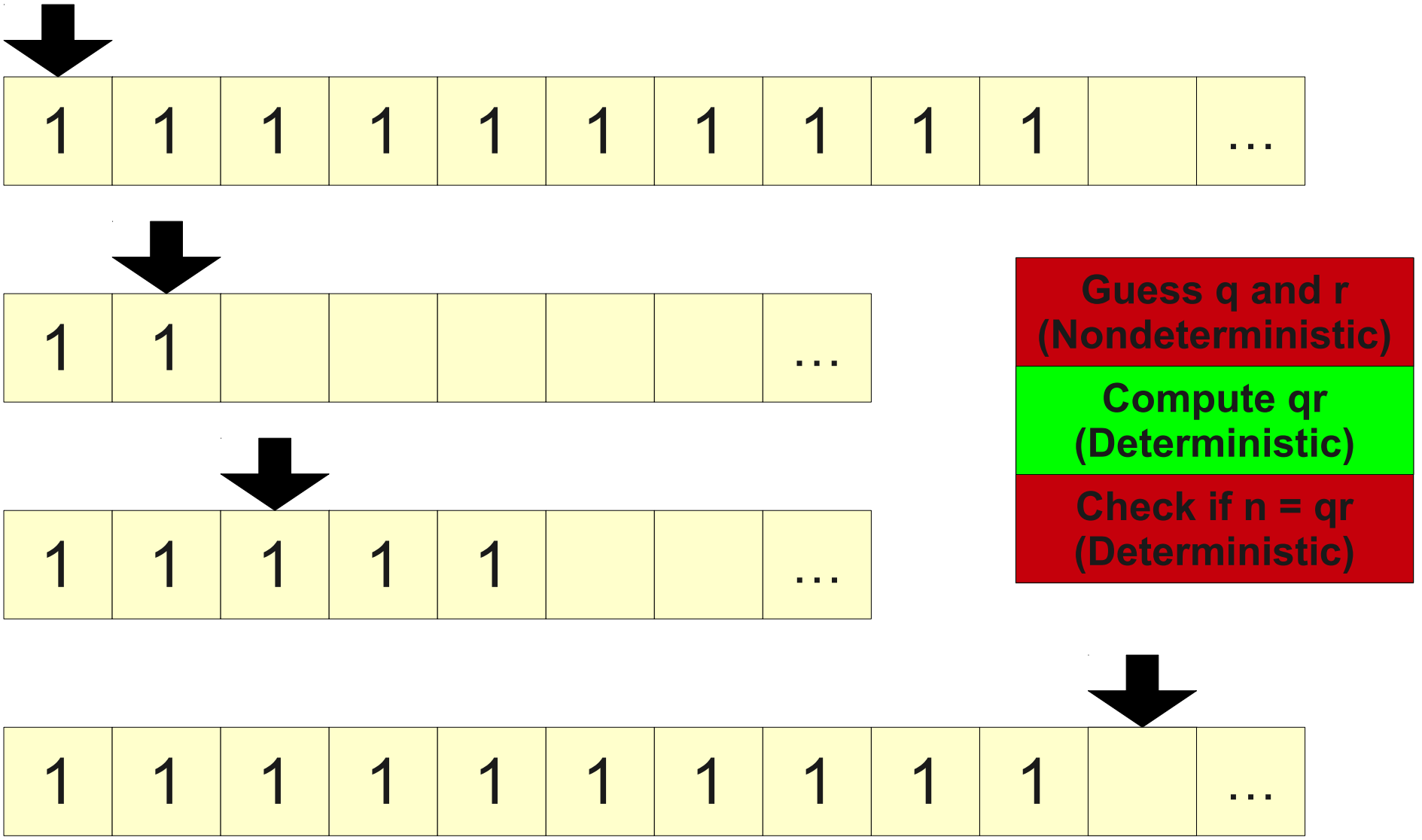
Nondeterministic Algorithms



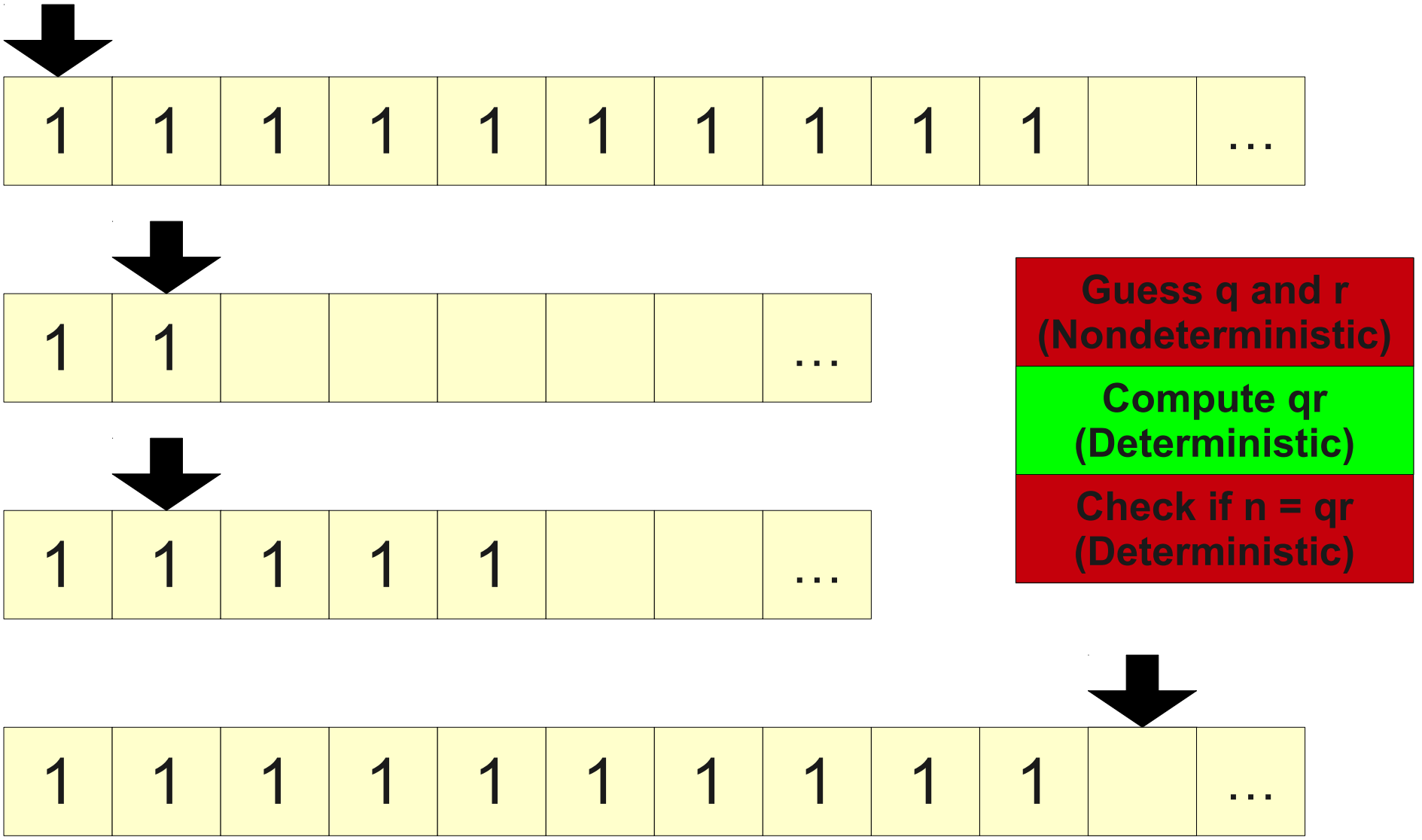
Nondeterministic Algorithms



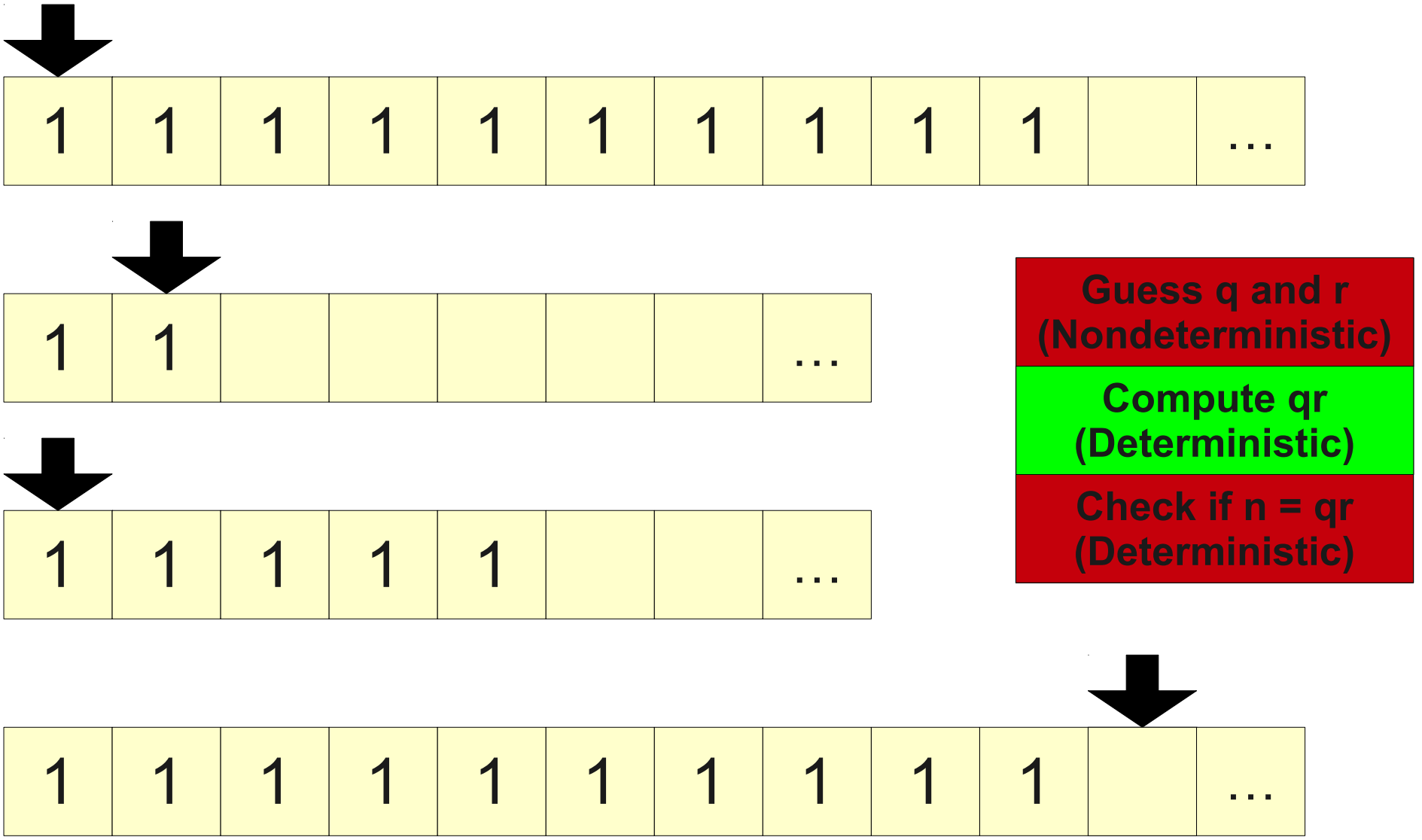
Nondeterministic Algorithms



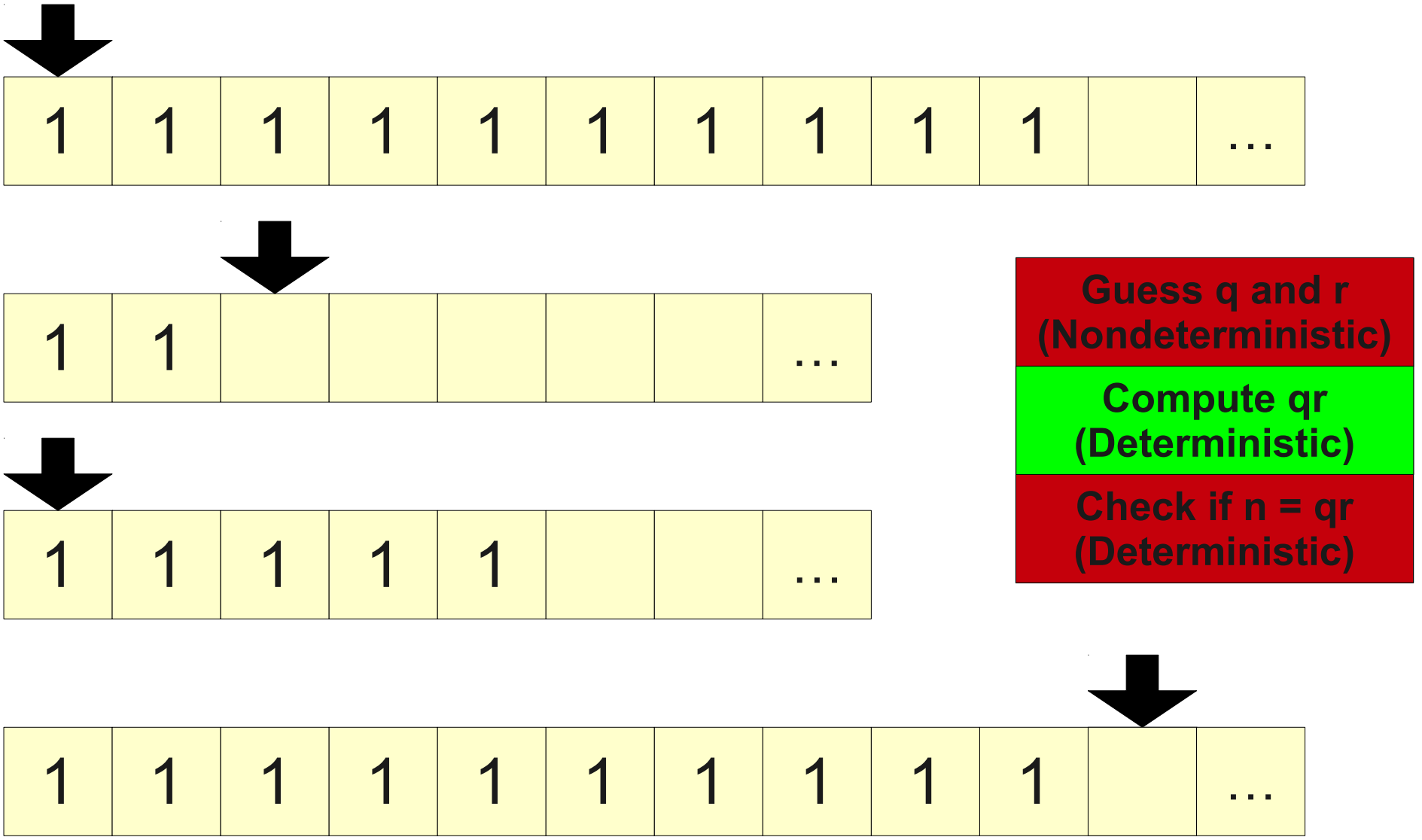
Nondeterministic Algorithms



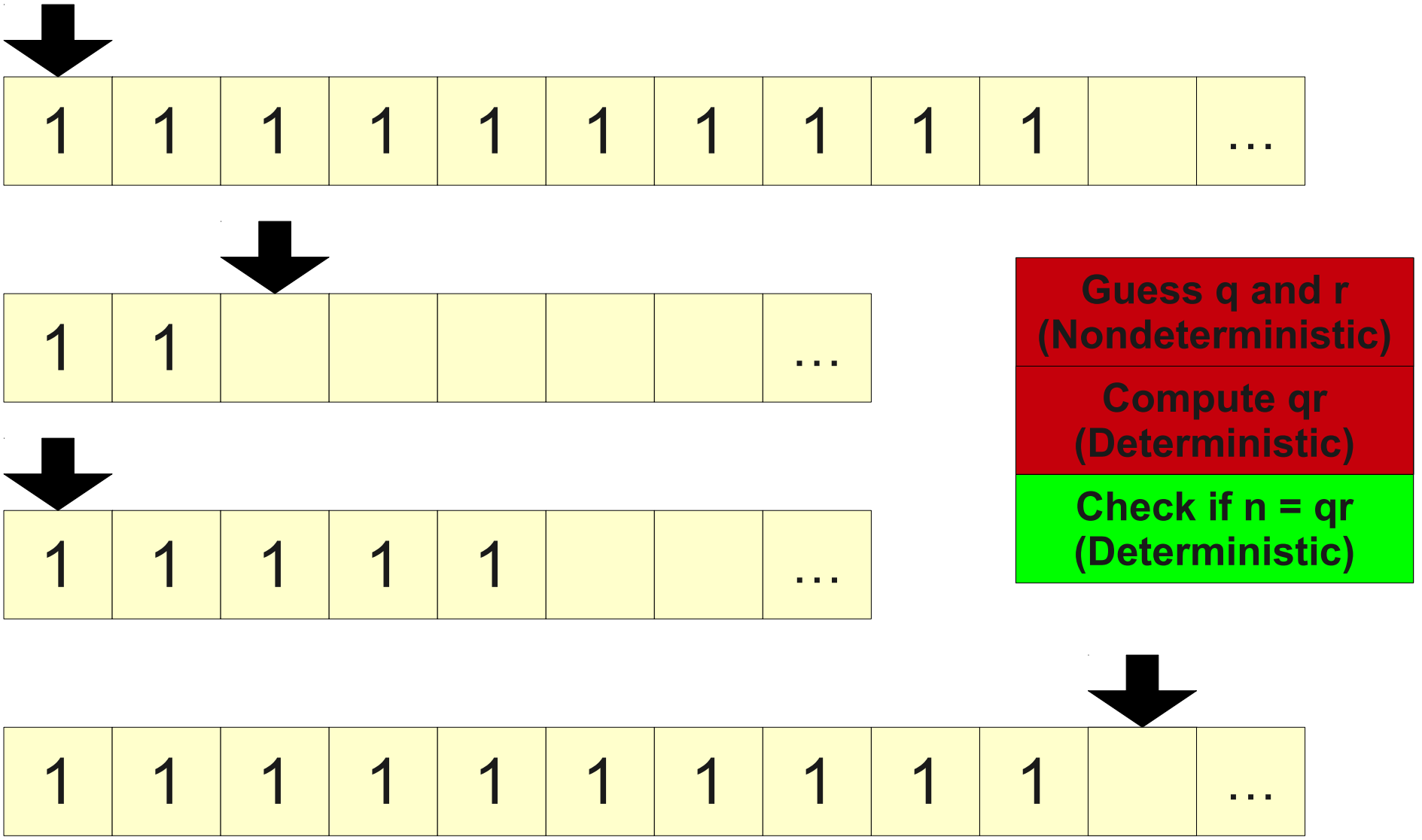
Nondeterministic Algorithms



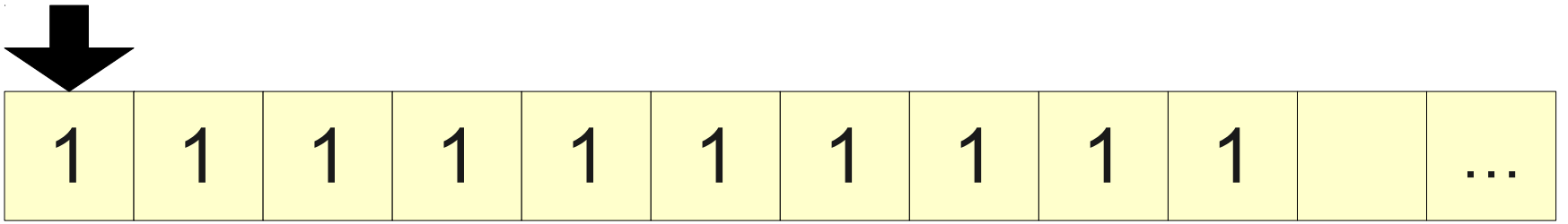
Nondeterministic Algorithms

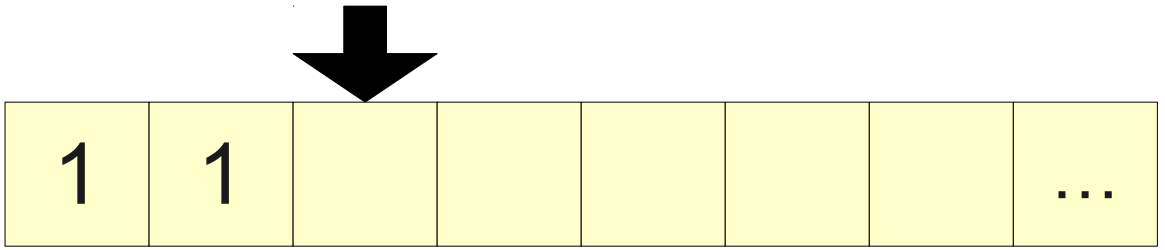


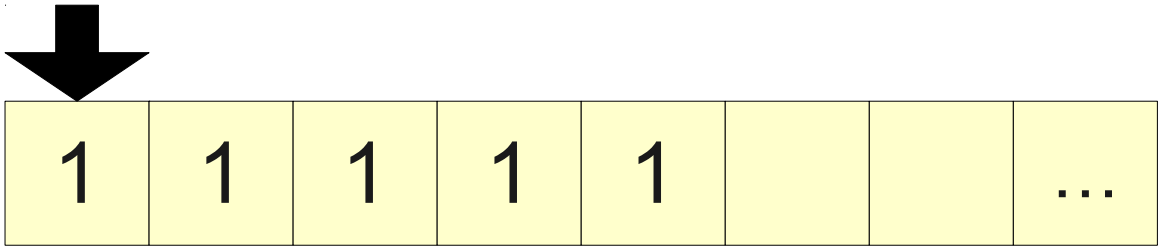
Nondeterministic Algorithms

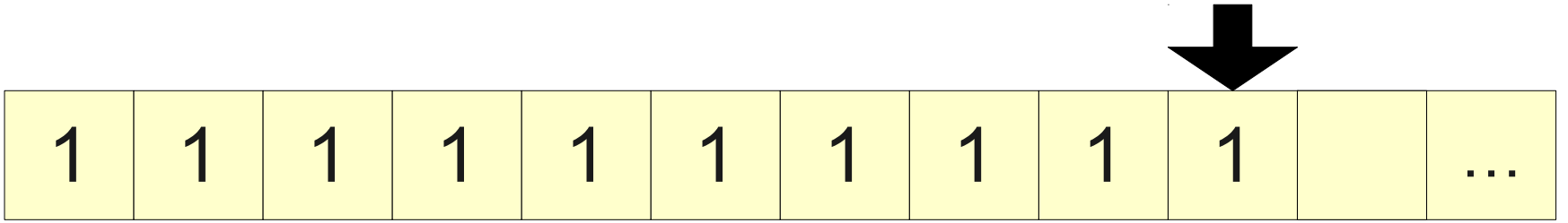


Nondeterministic Algorithms







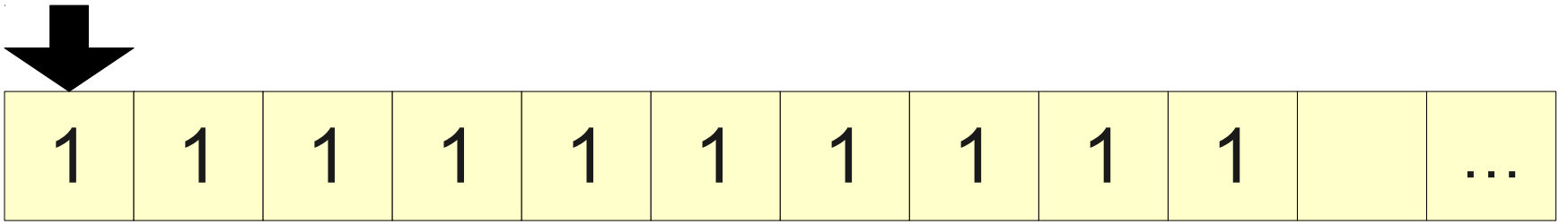


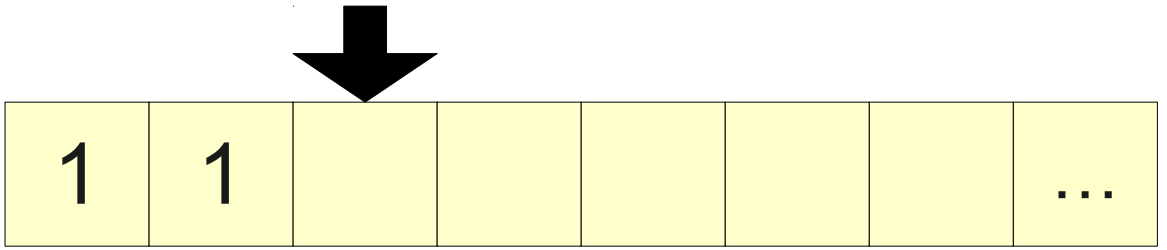
Guess q and r (Nondeterministic)

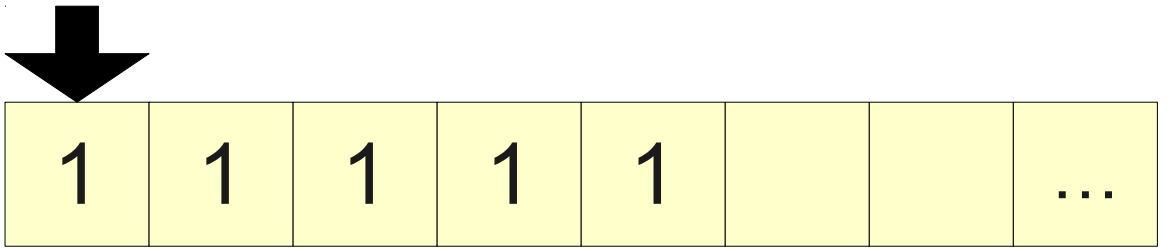
Compute qr (Deterministic)

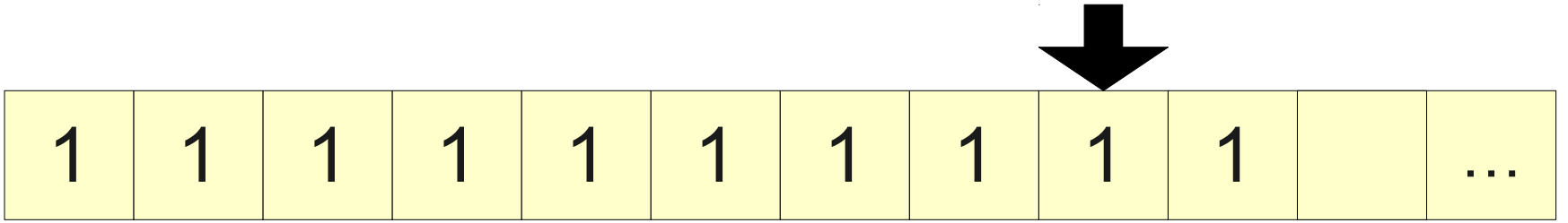
Check if n = qr (Deterministic)

Nondeterministic Algorithms







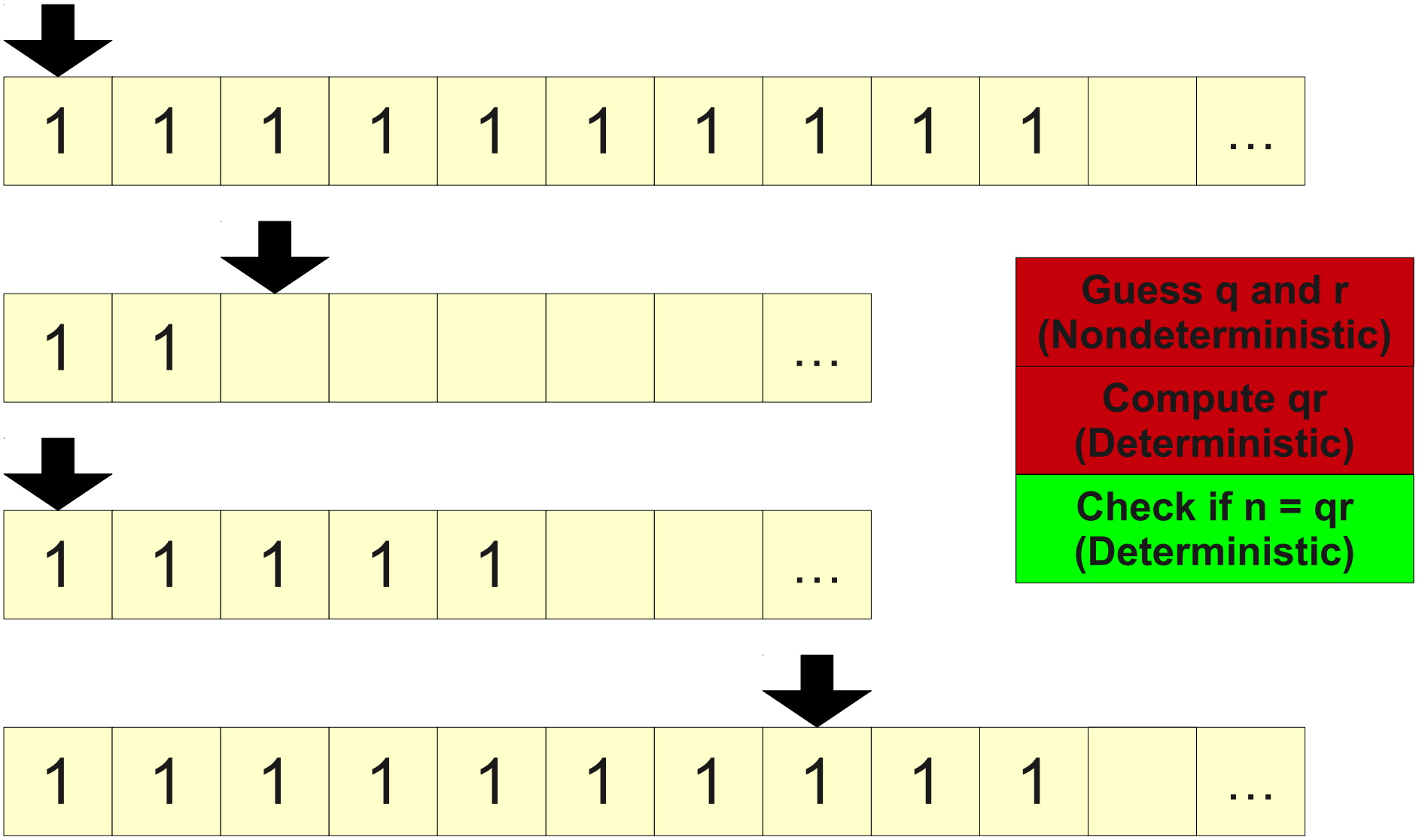


Guess q and r (Nondeterministic)

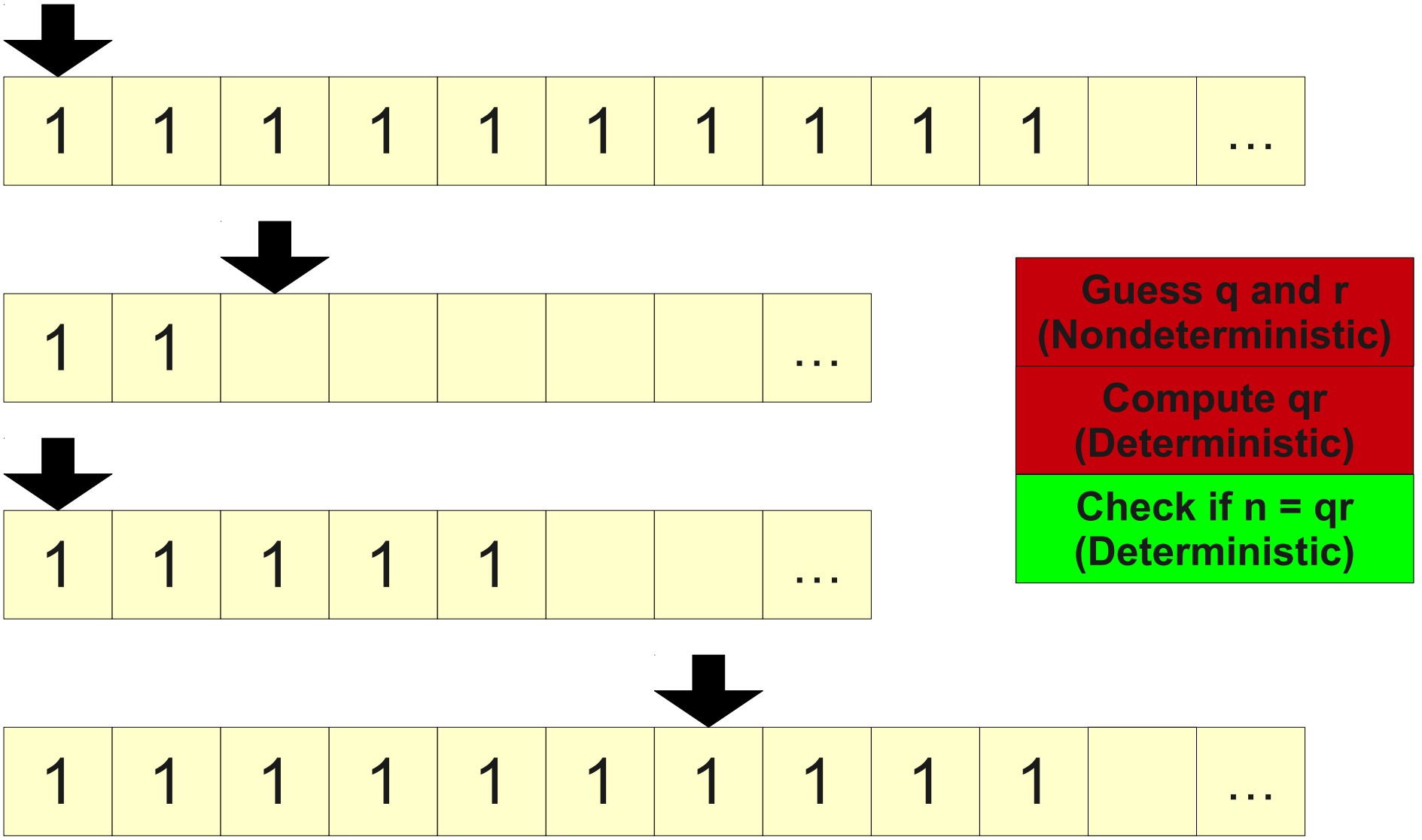
Compute qr (Deterministic)

Check if n = qr (Deterministic)

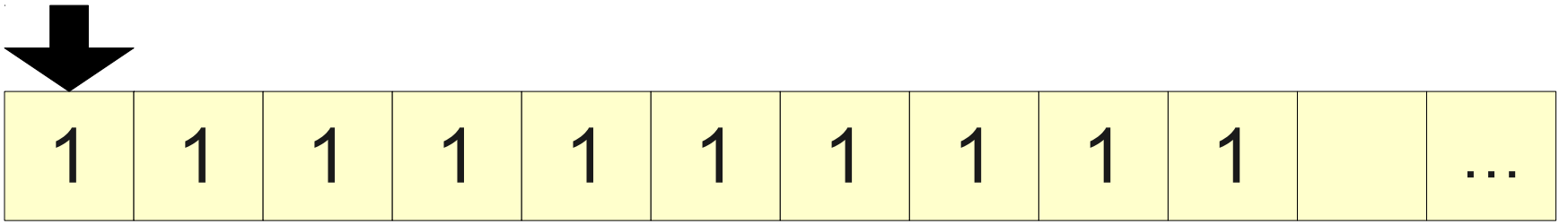
Nondeterministic Algorithms

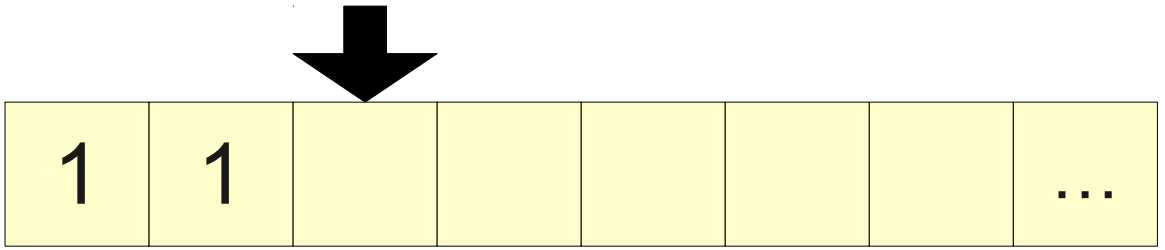


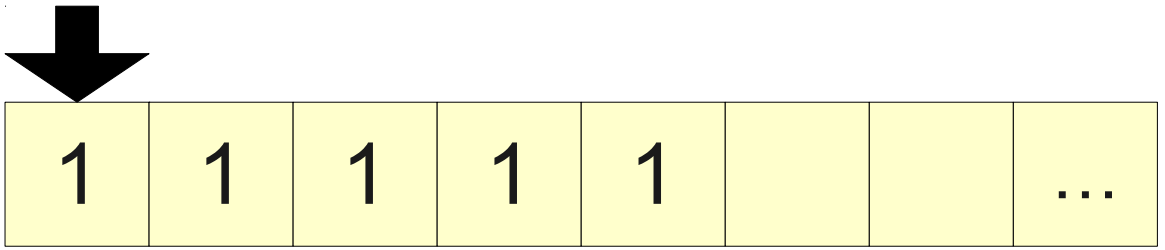
Nondeterministic Algorithms

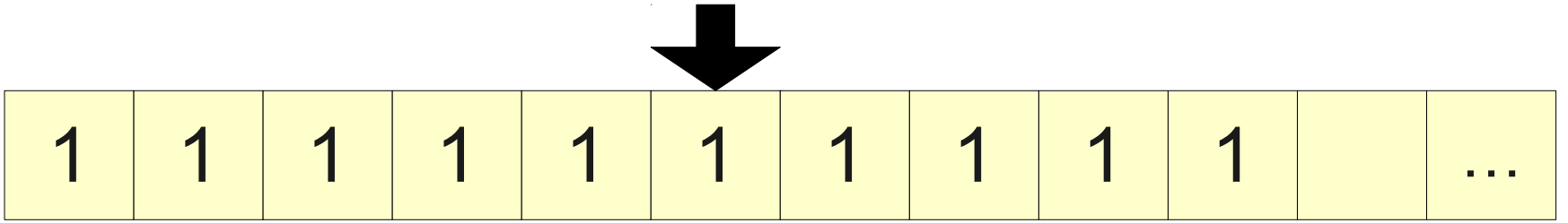


Nondeterministic Algorithms







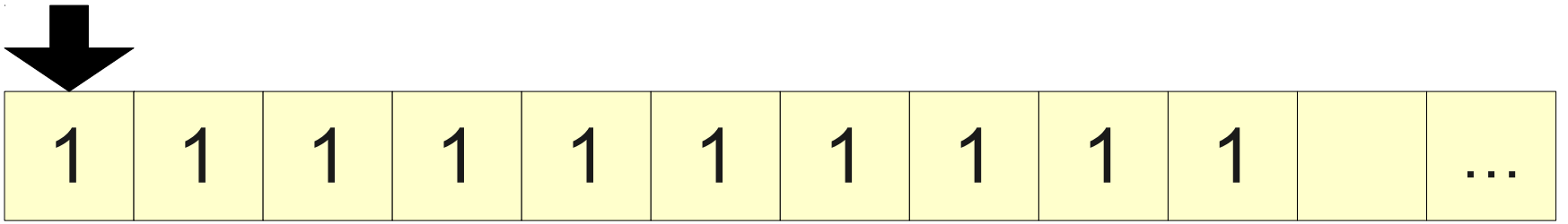


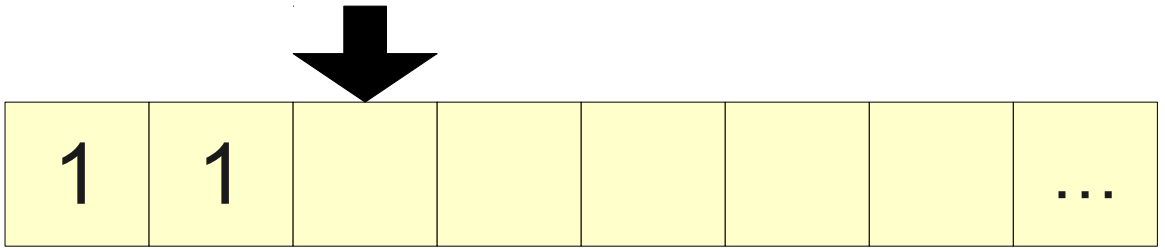
Guess q and r (Nondeterministic)

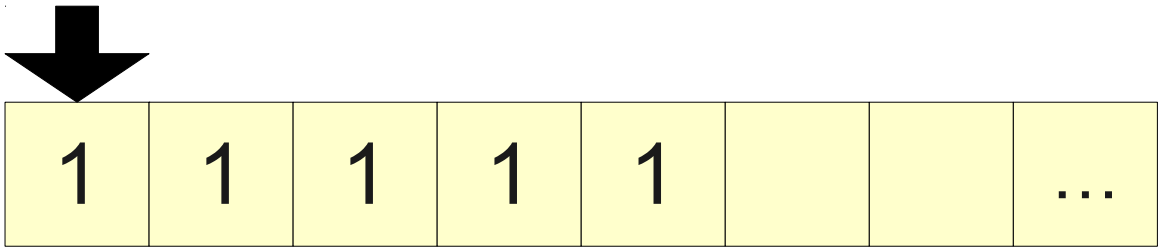
Compute qr (Deterministic)

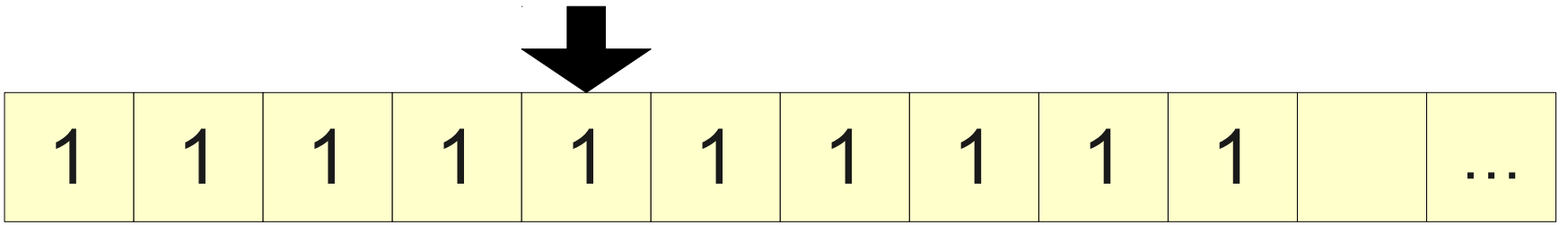
Check if n = qr (Deterministic)

Nondeterministic Algorithms







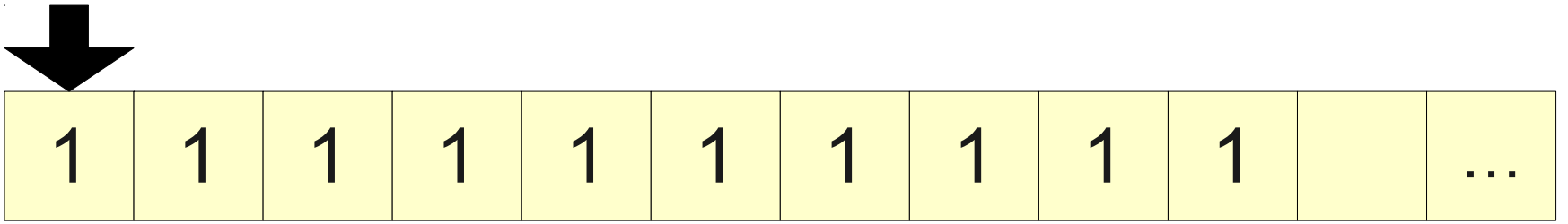


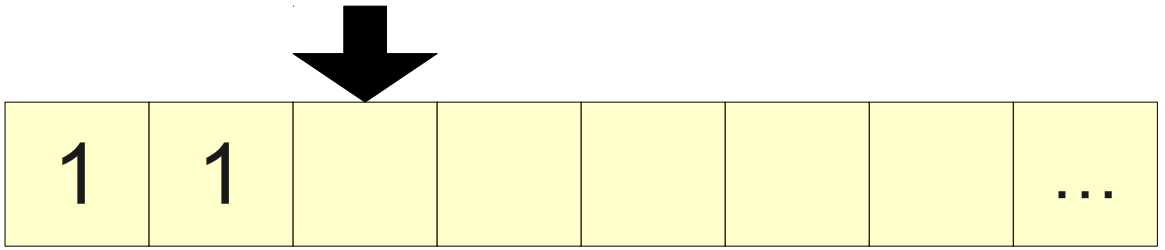
Guess q and r (Nondeterministic)

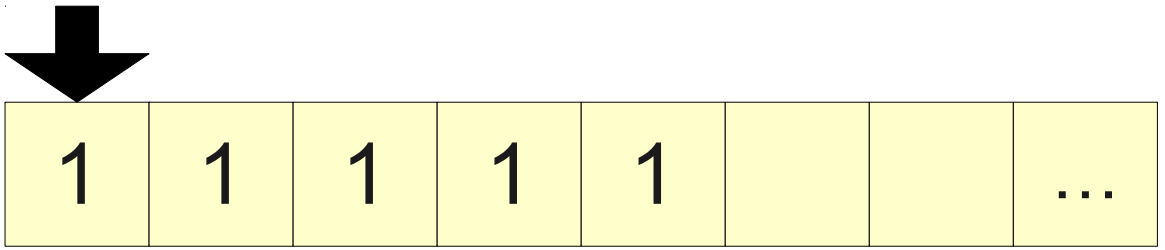
Compute qr (Deterministic)

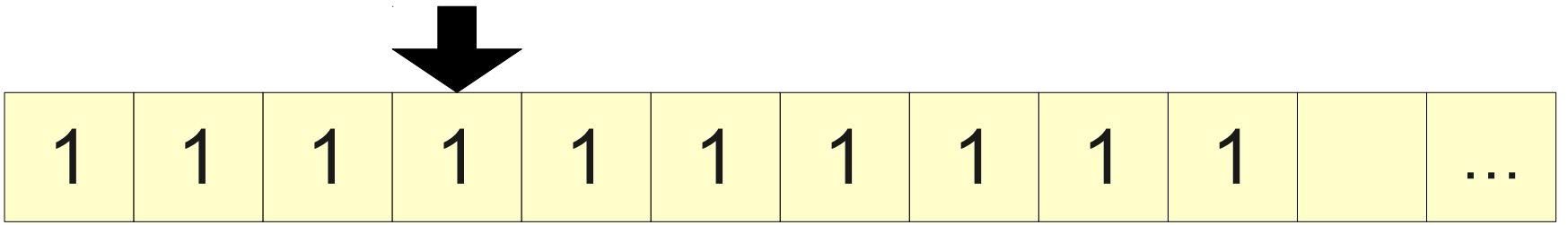
Check if n = qr (Deterministic)

Nondeterministic Algorithms







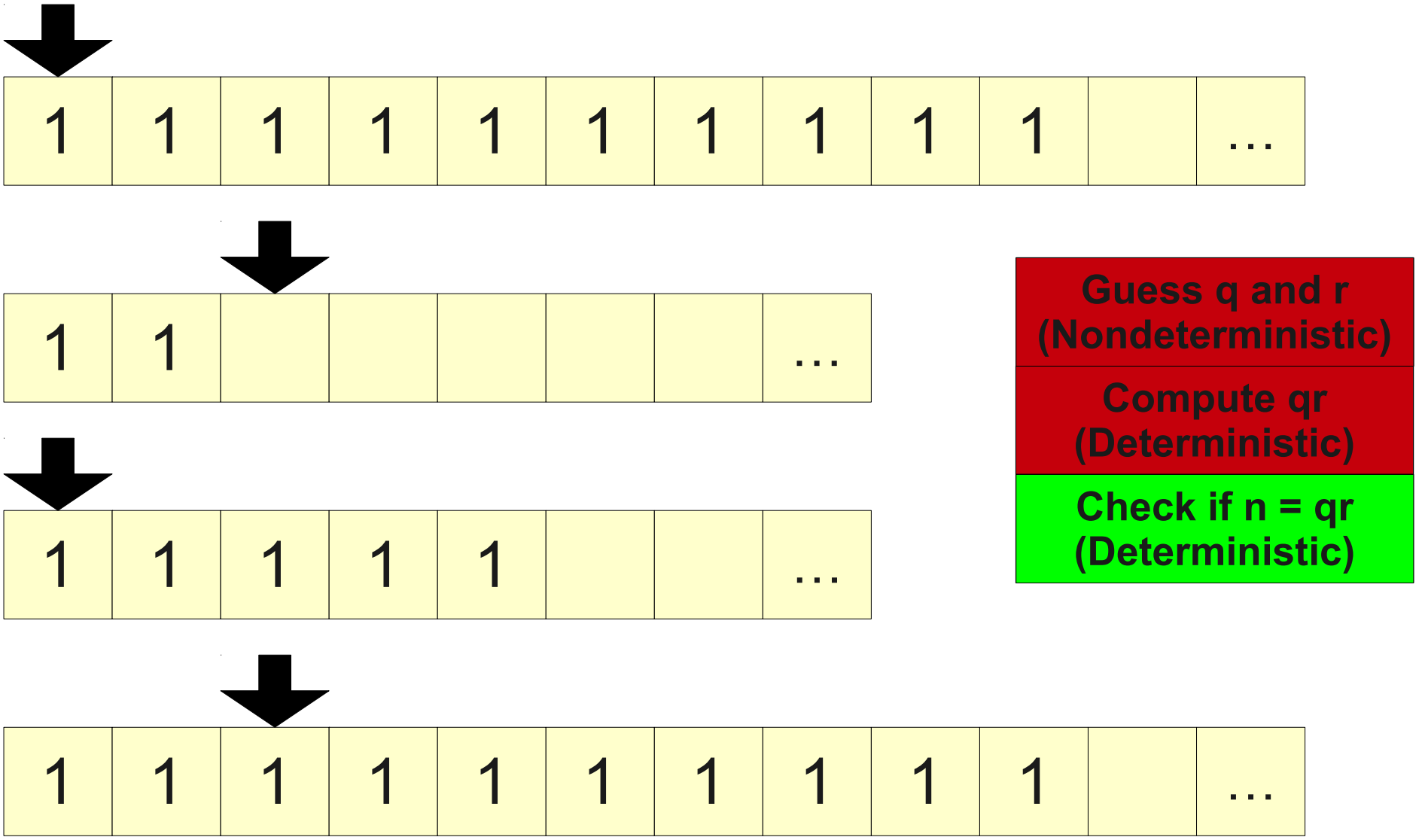


Guess q and r (Nondeterministic)

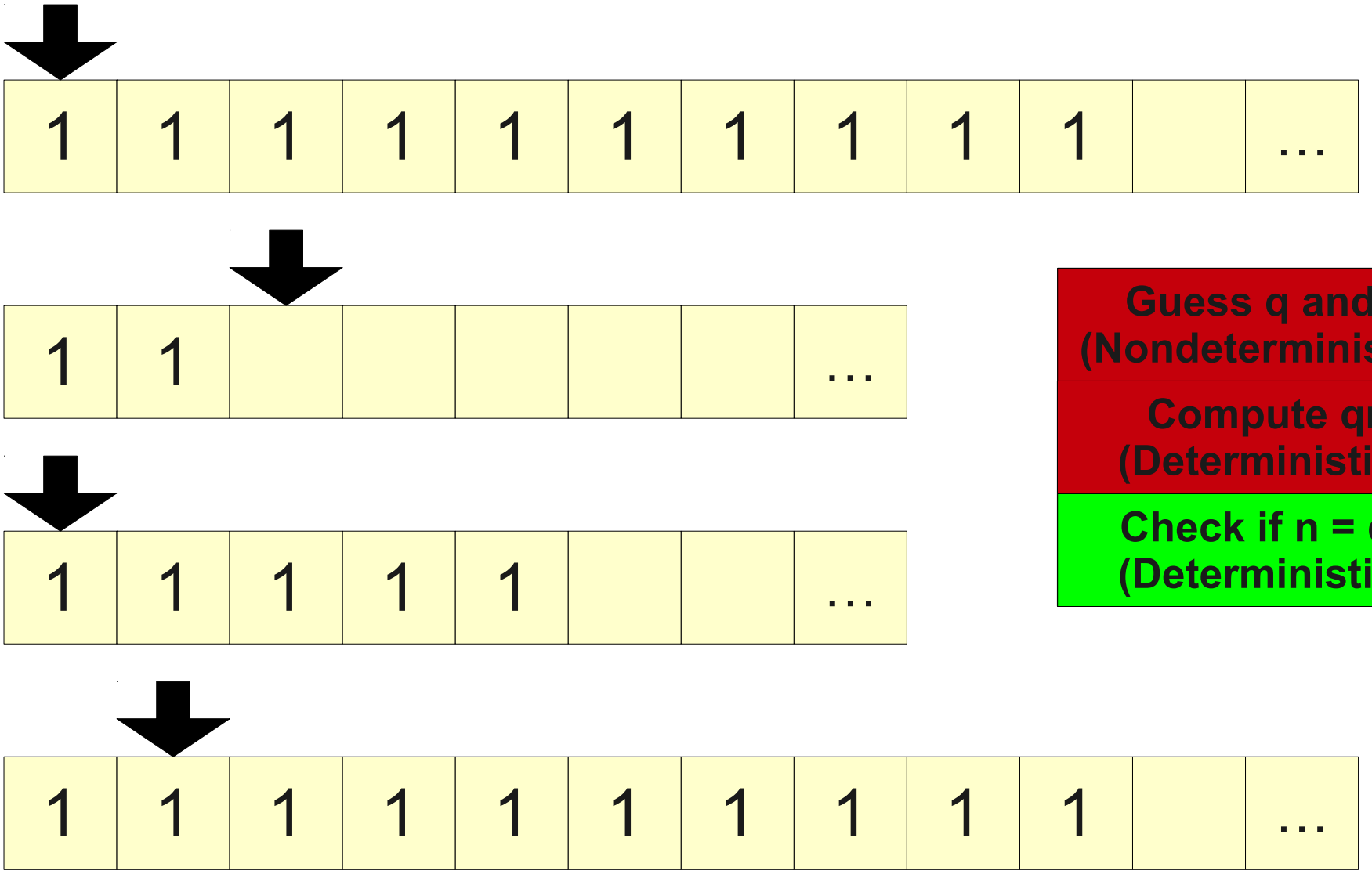
Compute qr (Deterministic)

Check if n = qr (Deterministic)

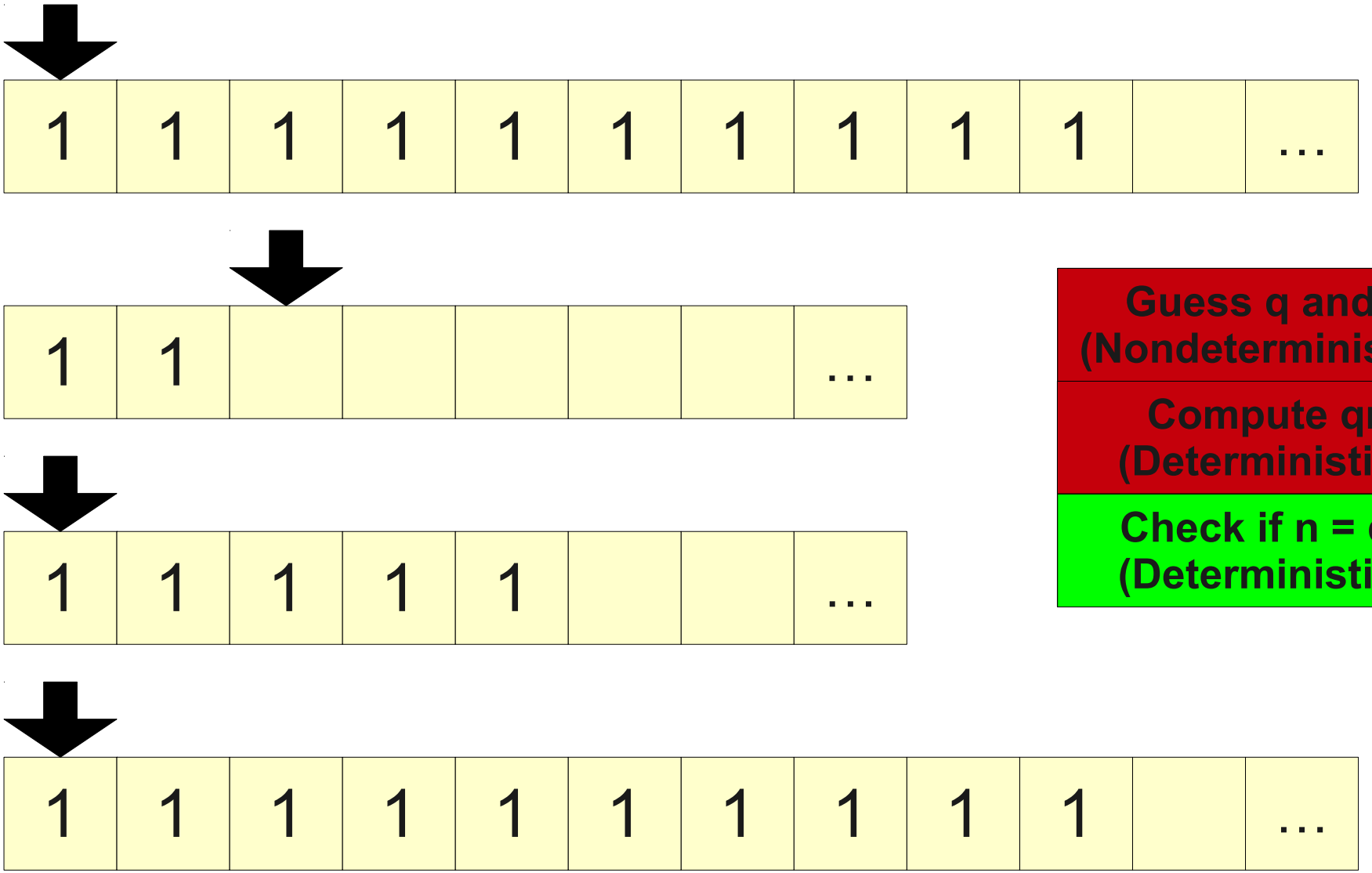
Nondeterministic Algorithms



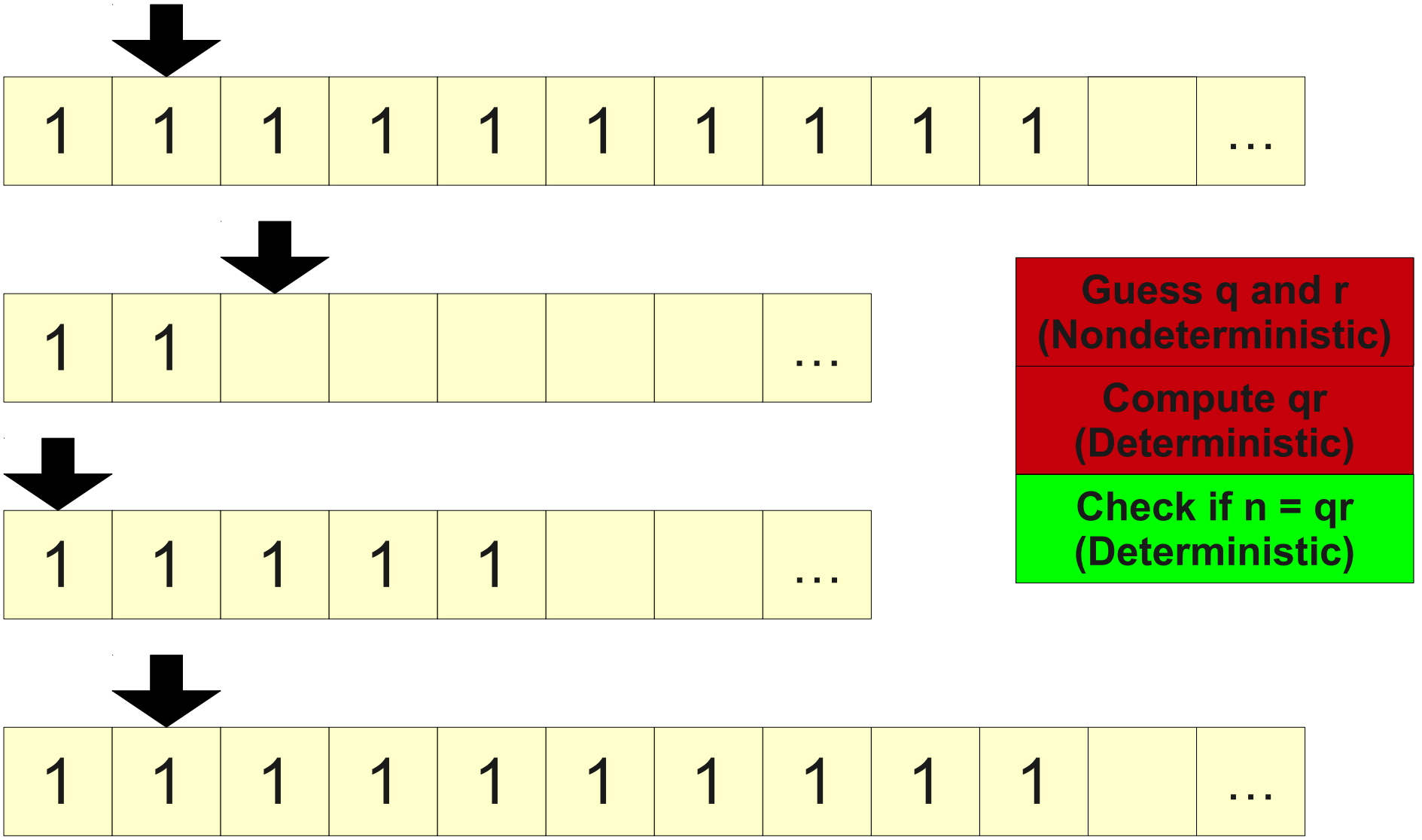
Nondeterministic Algorithms



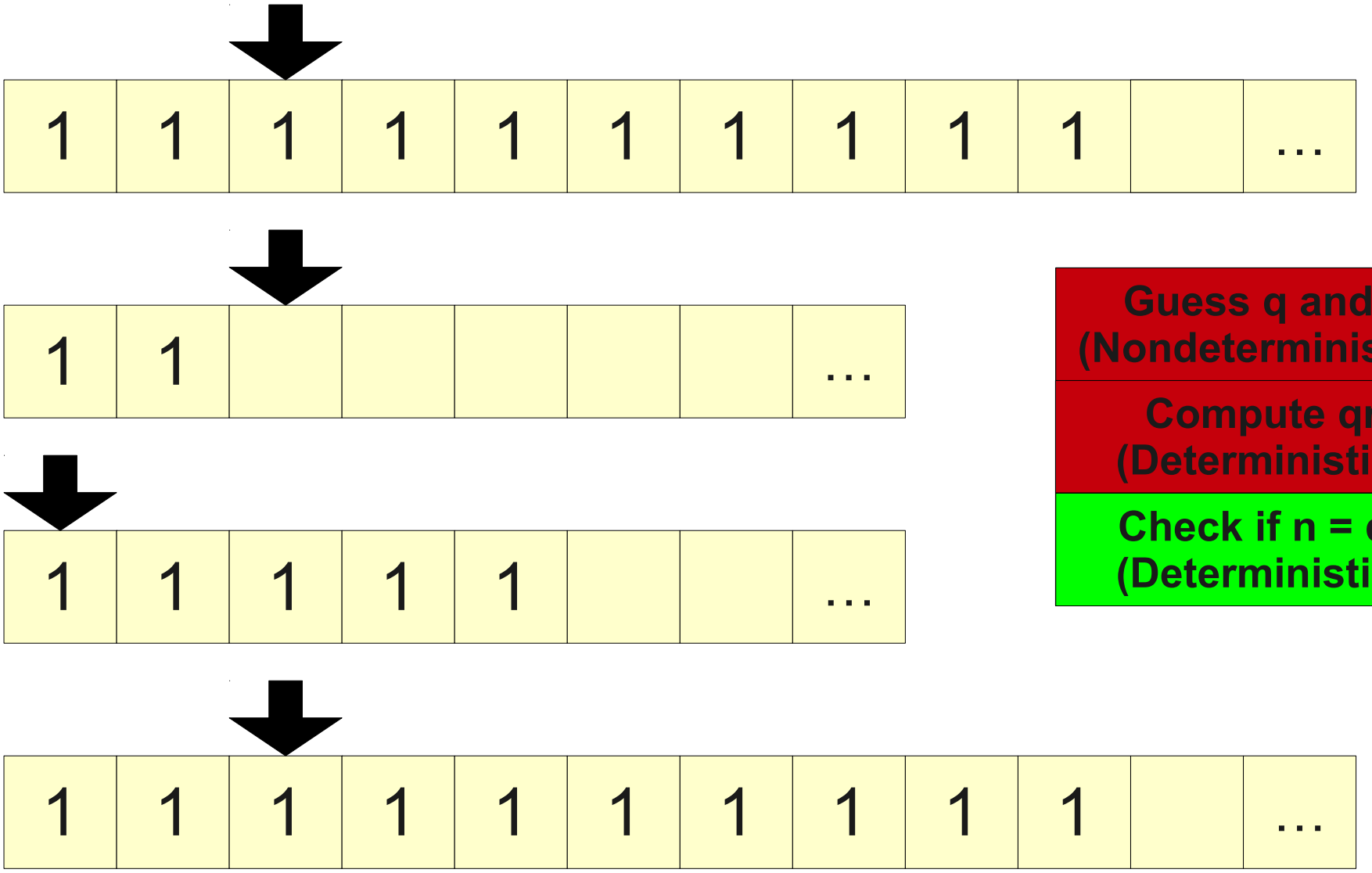
Nondeterministic Algorithms



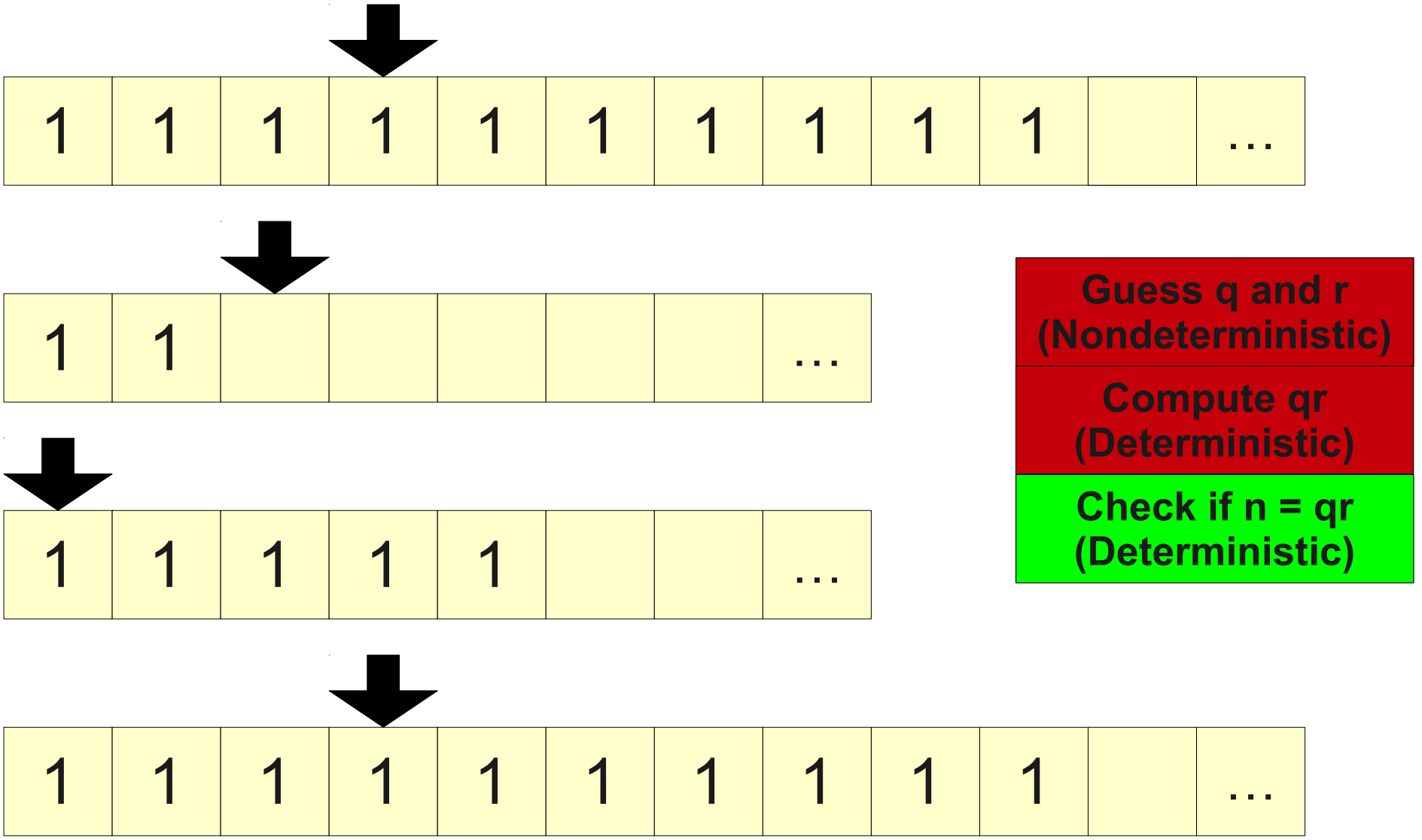
Nondeterministic Algorithms



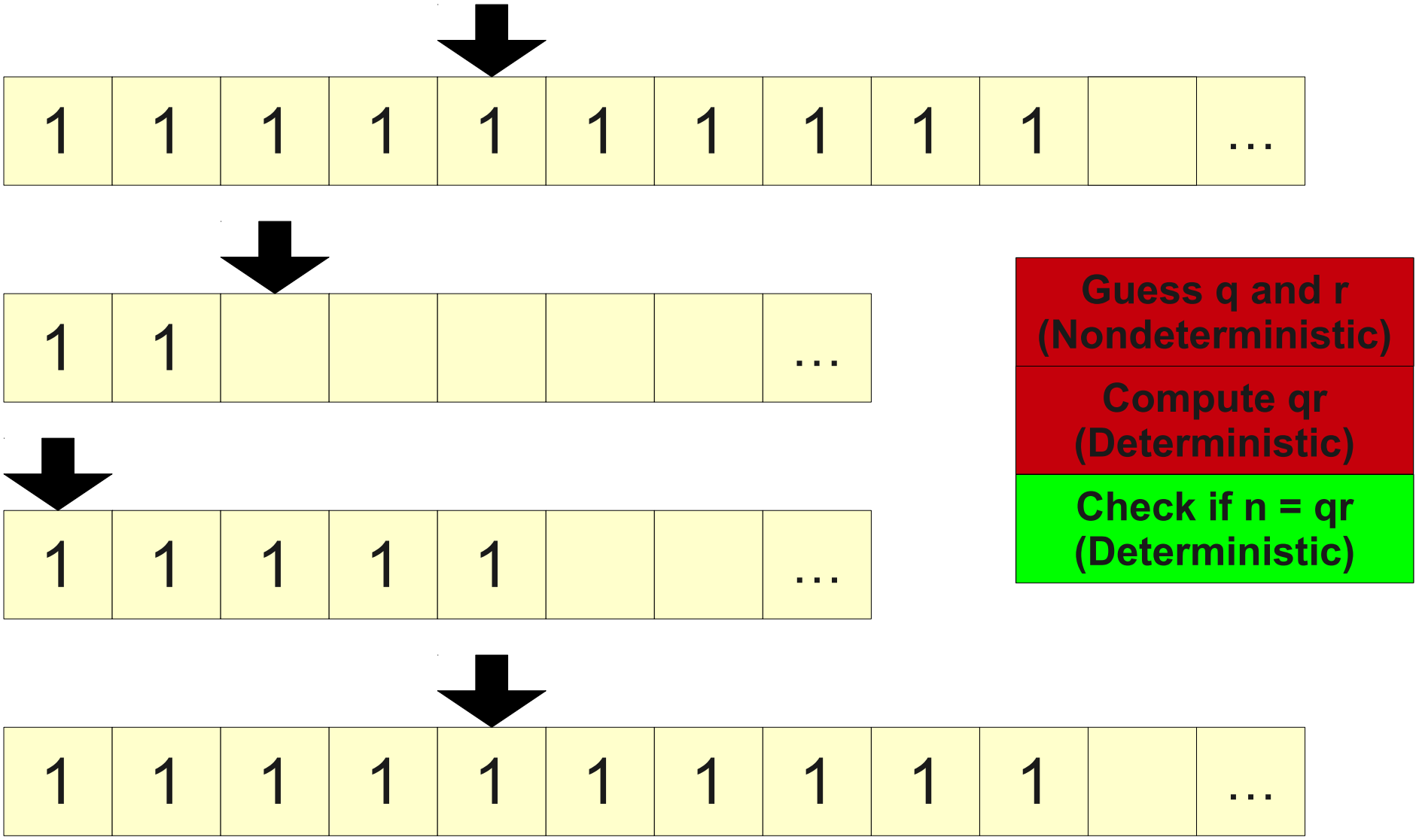
Nondeterministic Algorithms



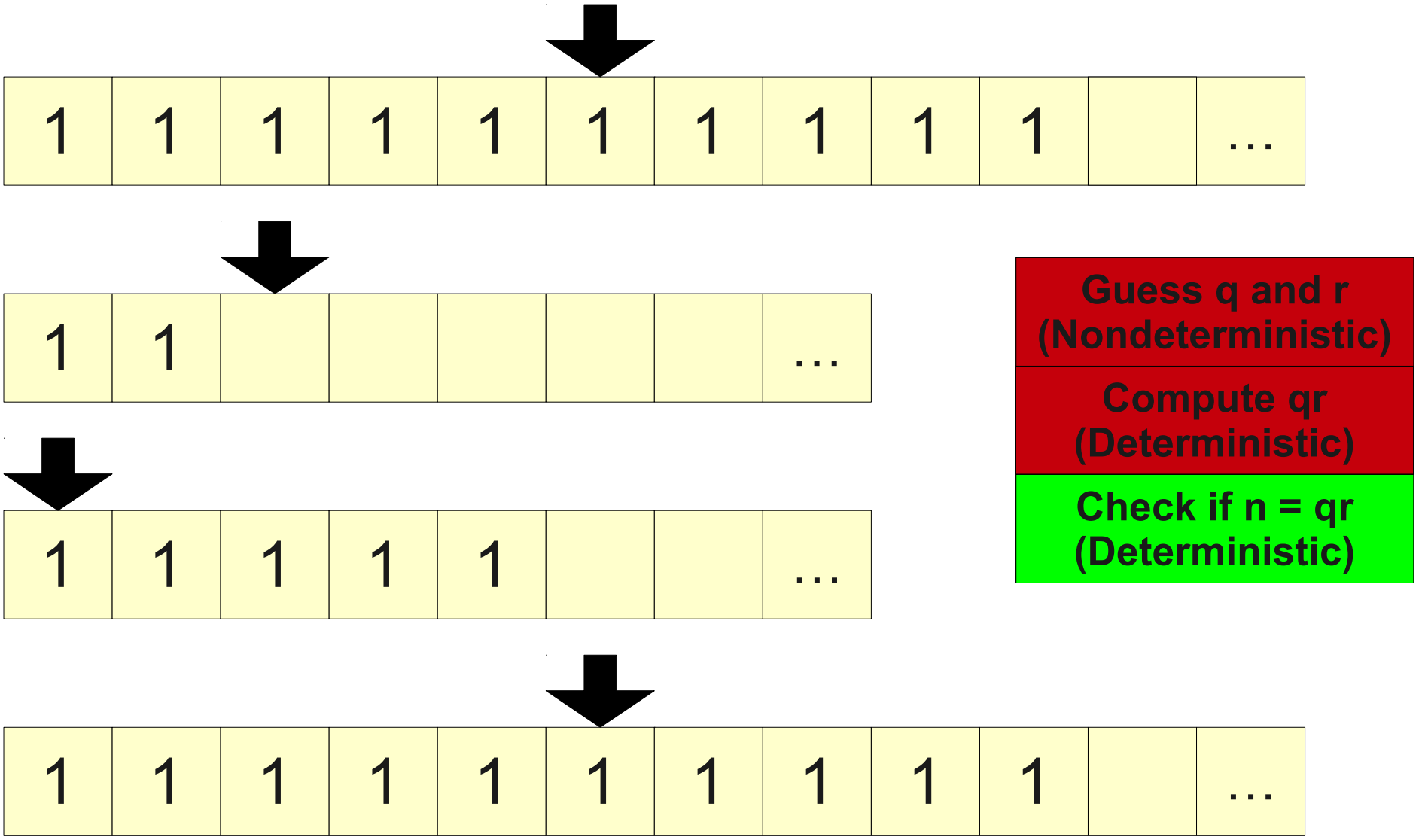
Nondeterministic Algorithms



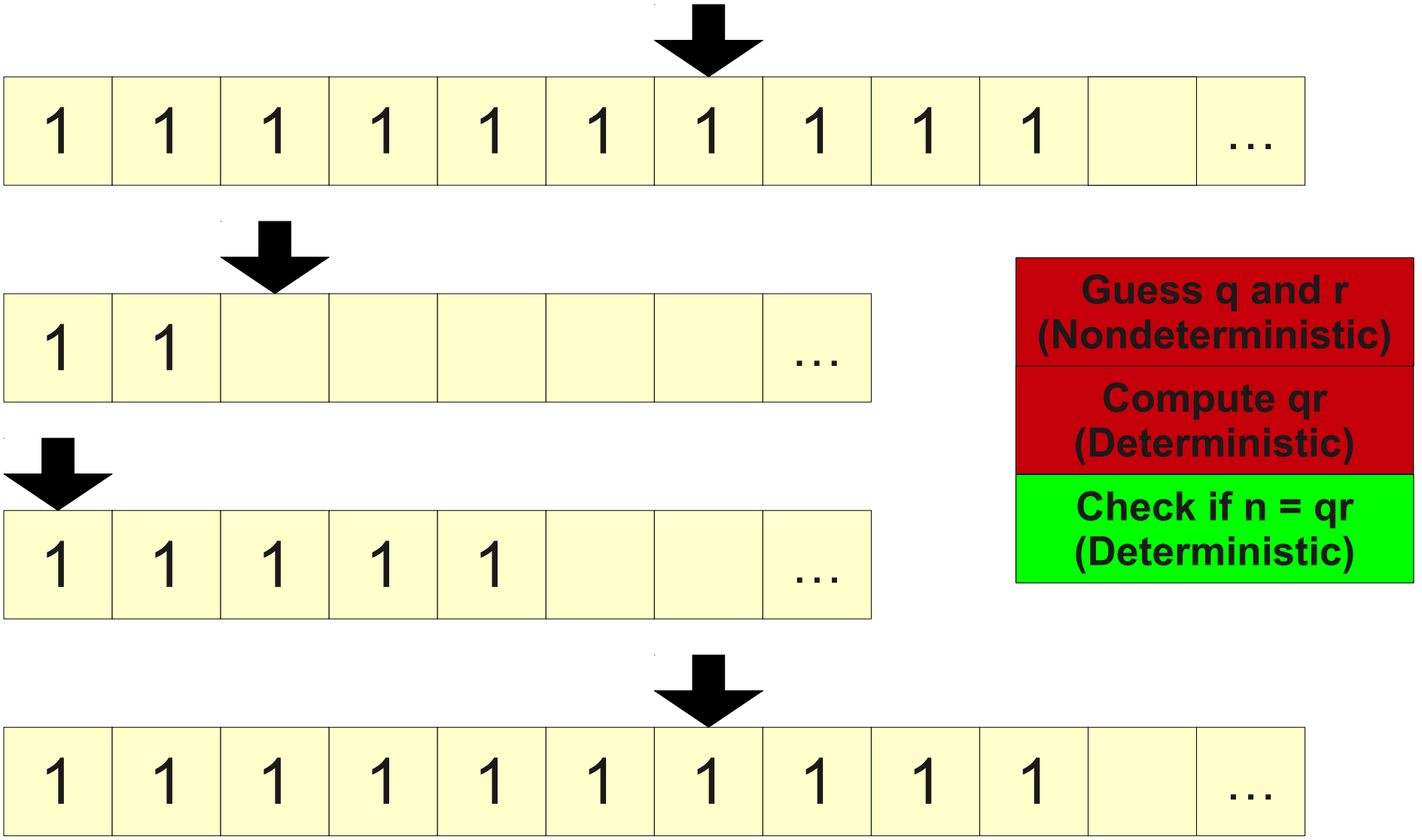
Nondeterministic Algorithms



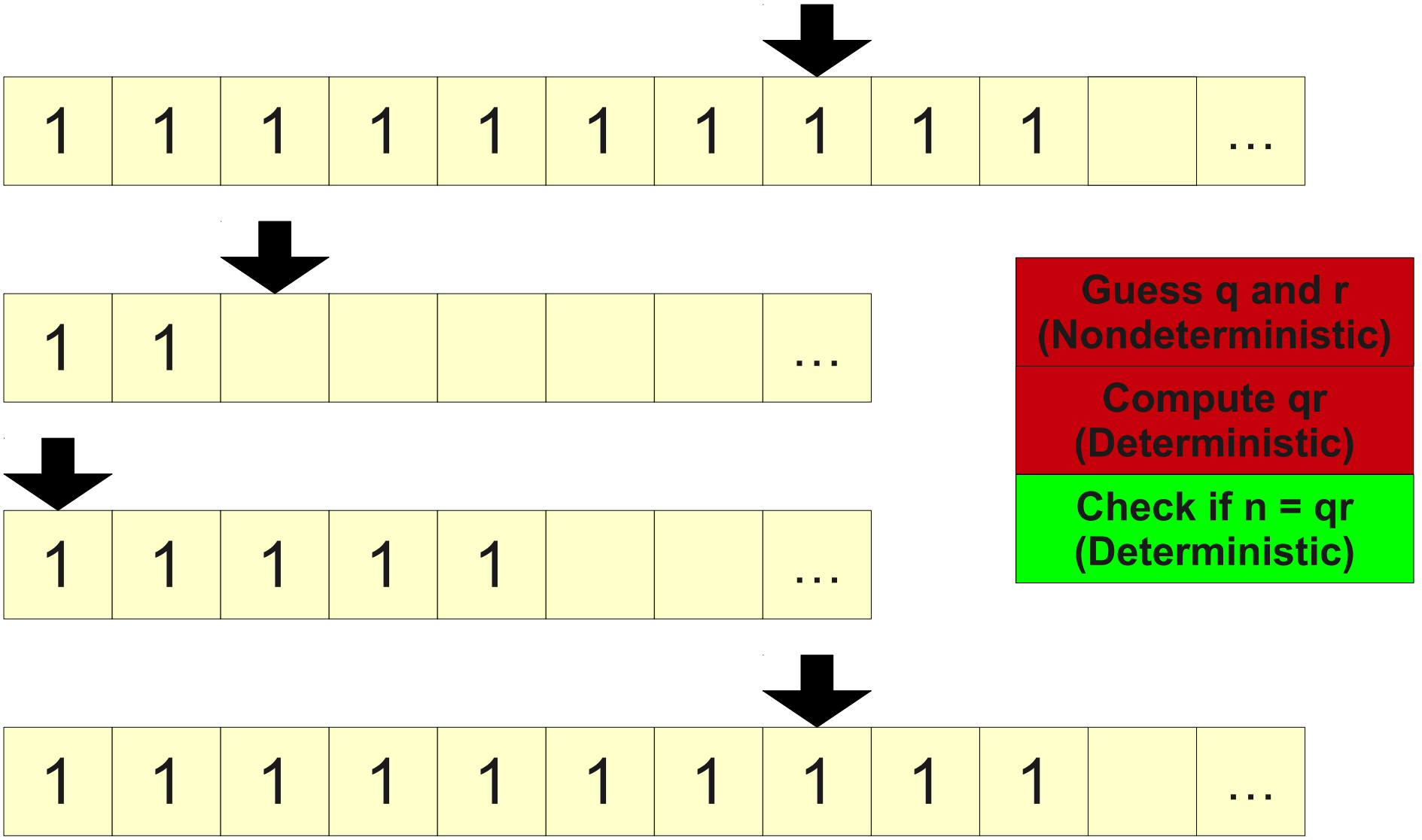
Nondeterministic Algorithms



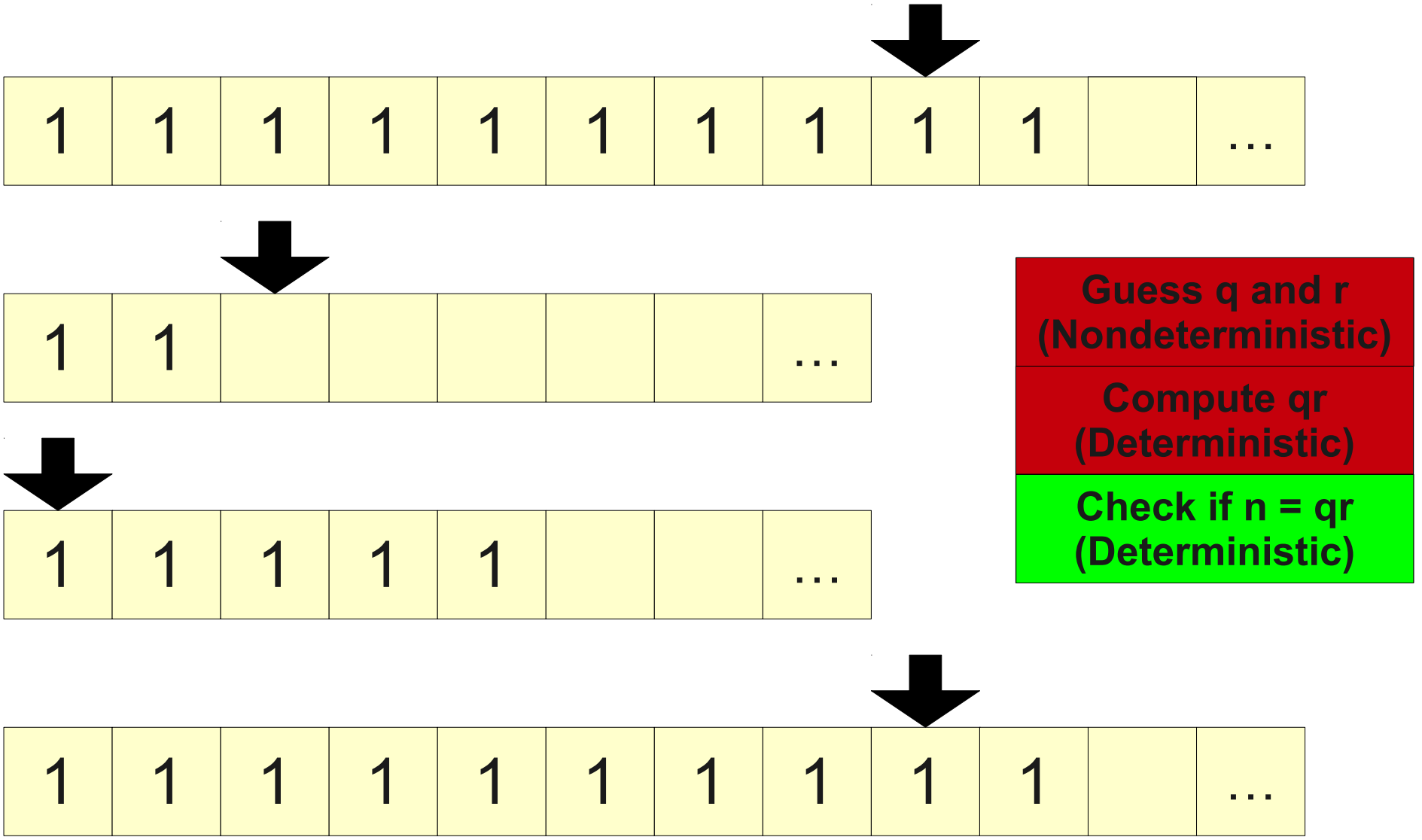
Nondeterministic Algorithms



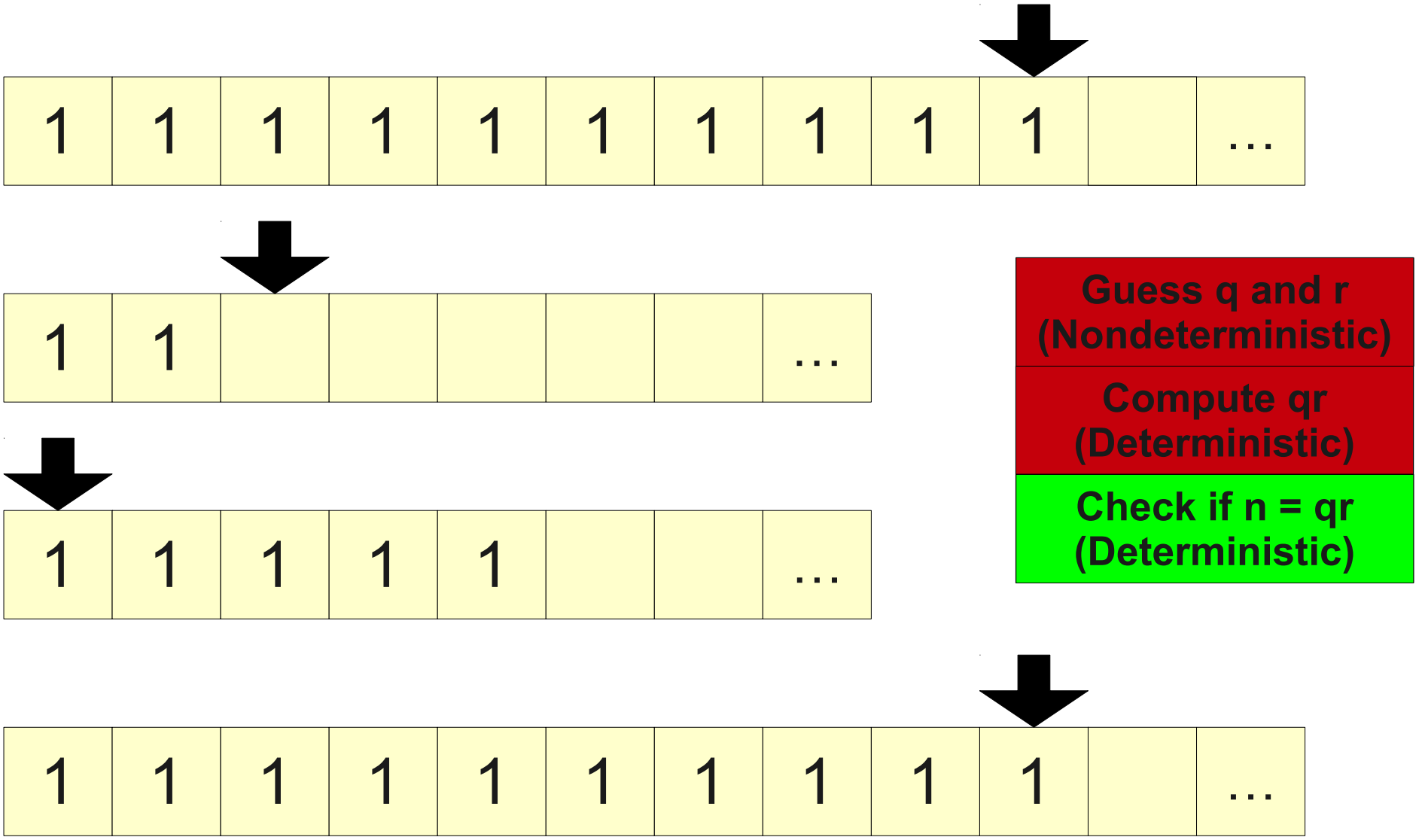
Nondeterministic Algorithms



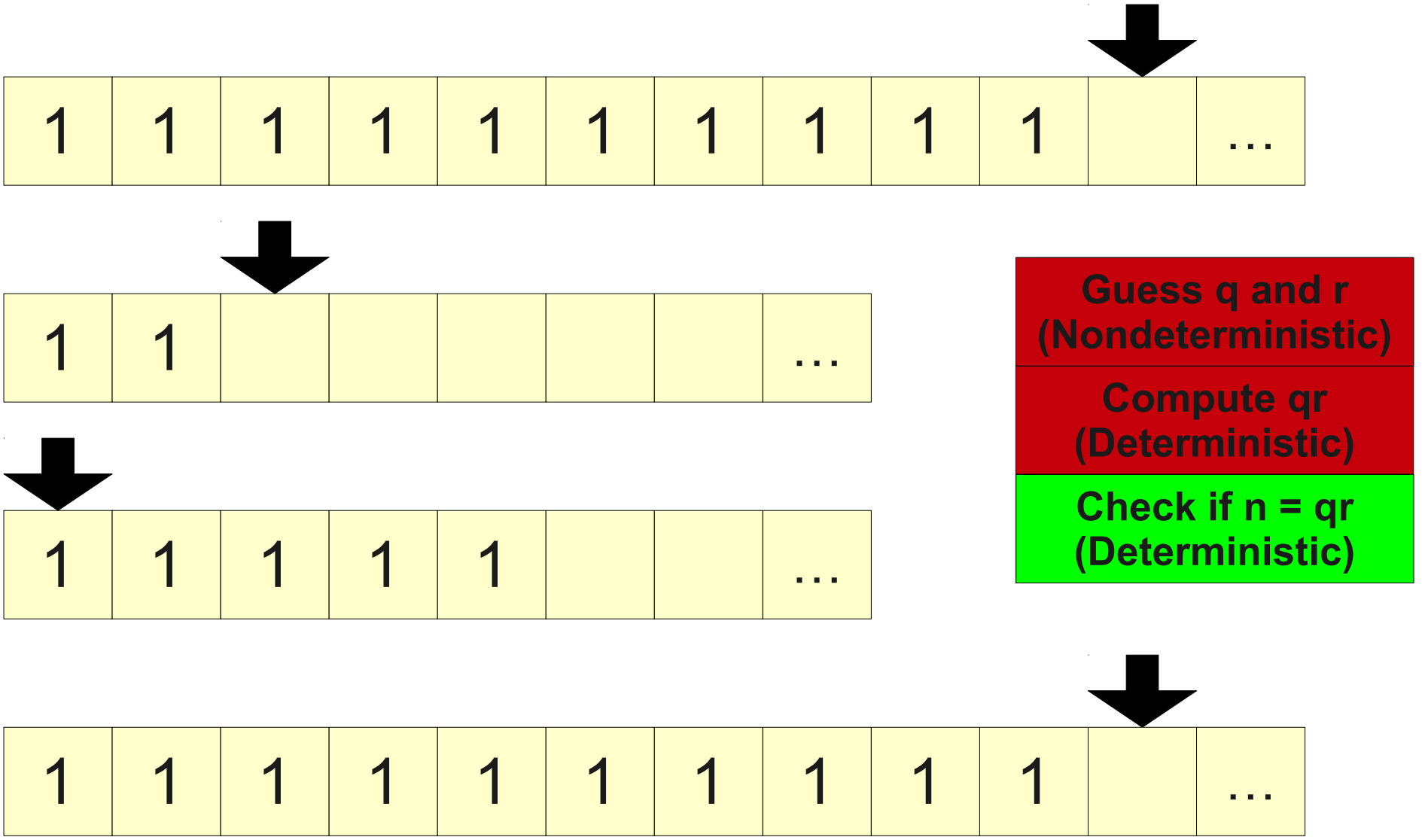
Nondeterministic Algorithms



Nondeterministic Algorithms



Nondeterministic Algorithms



Applications of NTMs

- Consider the following question:

  Given a TM M, does M accept any strings?

- Equivalently:

  Given a TM M, is $\mathscr{L}(M) \neq \varnothing$?

- As a language question:

  $$L_{NE} = \{\langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) \neq \varnothing\}$$

- Question: Is $L_{NE} \in \mathbf{RE}$?

Nondeterminism to the Rescue!

- Consider the following NTM:

  N = "On input $\langle M \rangle$, where M is a Turing machine:

  Nondeterministically guess a string $x \in \Sigma^*$.

  Deterministically run M on x.

  If M accepts x, accept.

  If M rejects x, reject."

- Is this a legal nondeterministic Turing machine?
- If so, how would we prove $\mathscr{L}(M) = L_{NE}$?
- Does this say whether $L_{NE} \in \mathbf{RE}$?

Nondeterminism to the Rescue!

Consider the following NTM:

N = "On input $\langle M \rangle$, where M is a Turing machine:

Nondeterministically guess a string $x \in \Sigma^*$.

Deterministically run M on x.

If M accepts x, accept.

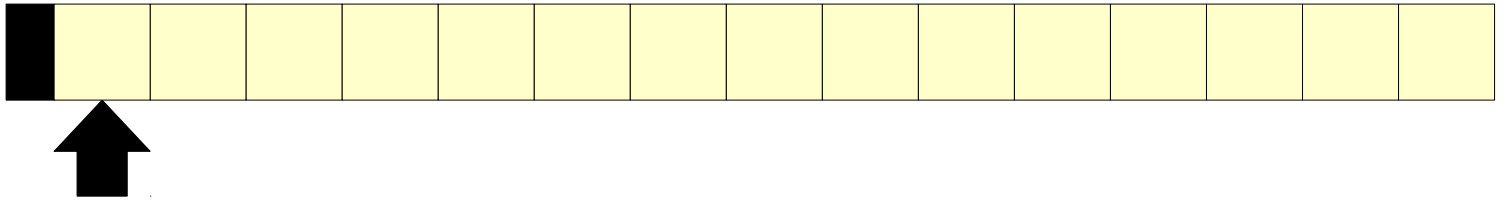
If M rejects x, reject."

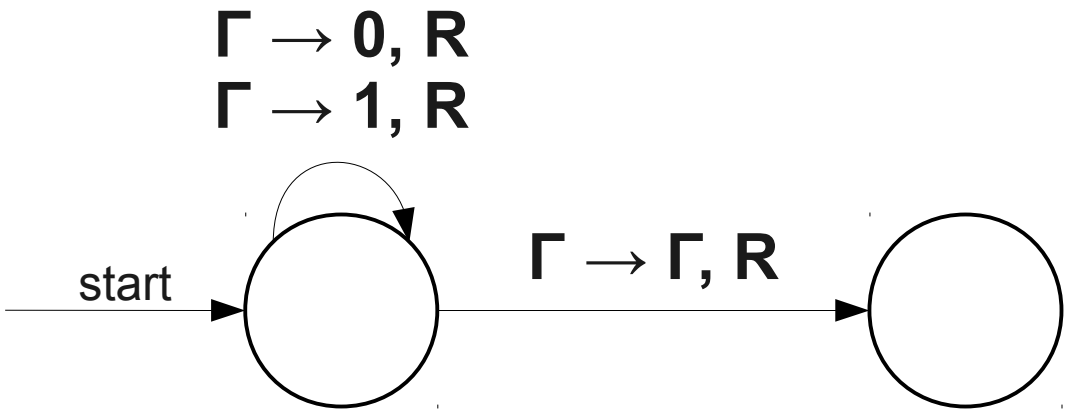
- Is this a legal nondeterministic Turing machine?

If so, how would we prove $\mathscr{L}(M) = L_{NE}$?

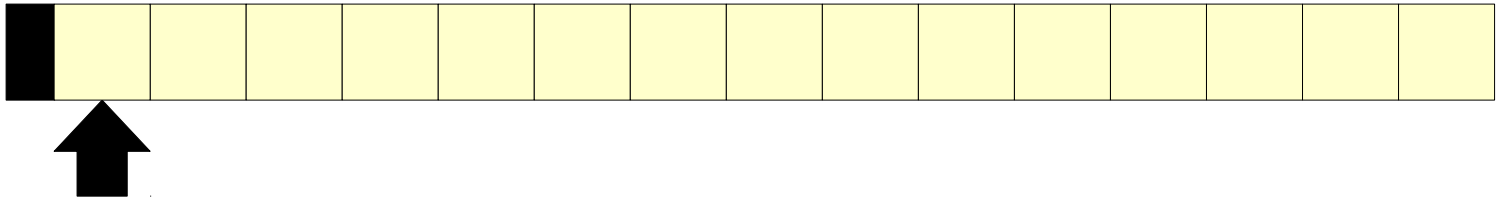
Does this say whether $L_{NE} \in \mathbf{RE}$?

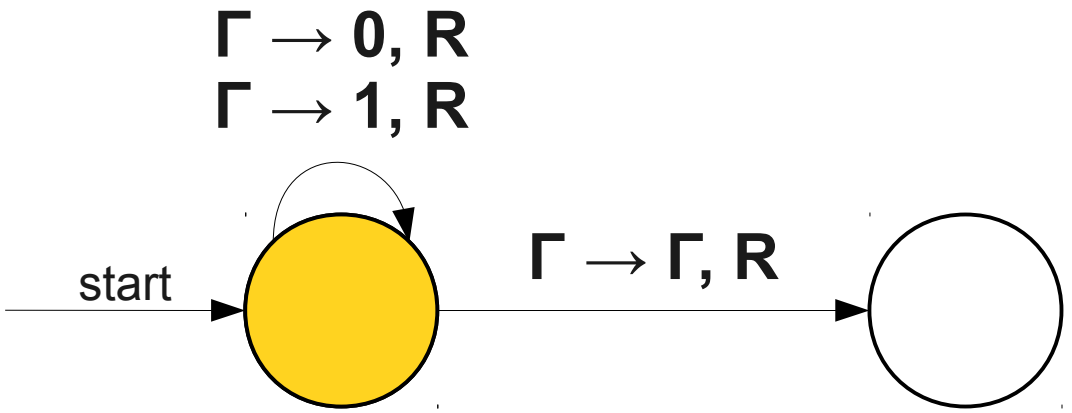
Guessing an Arbitrary String



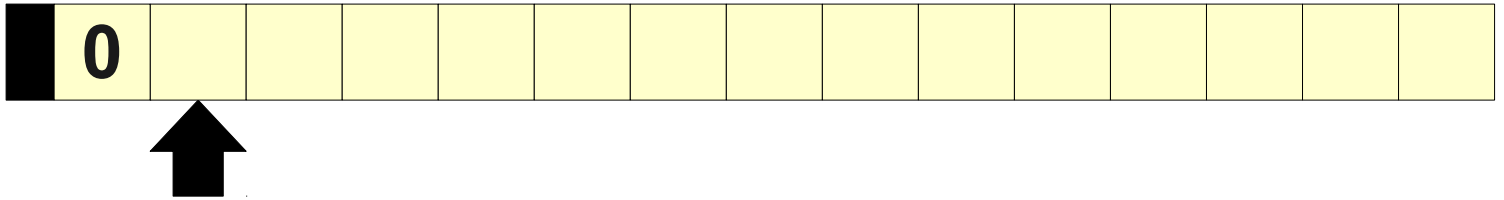


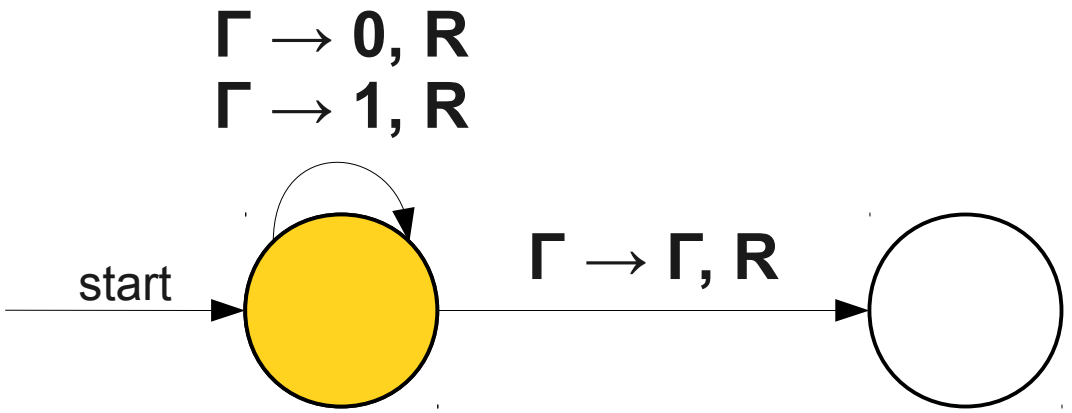
Guessing an Arbitrary String



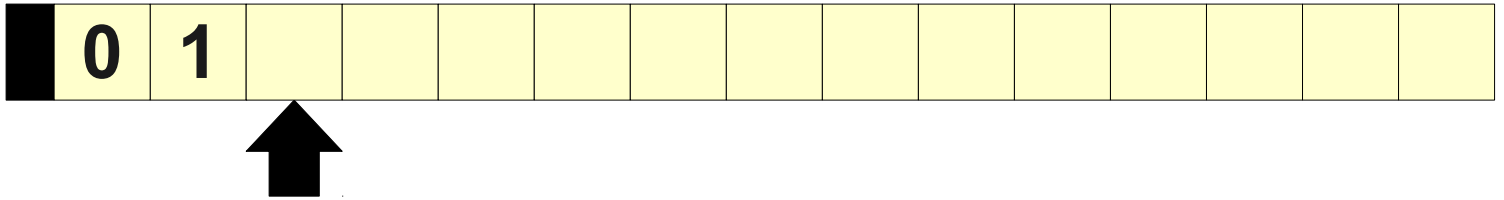


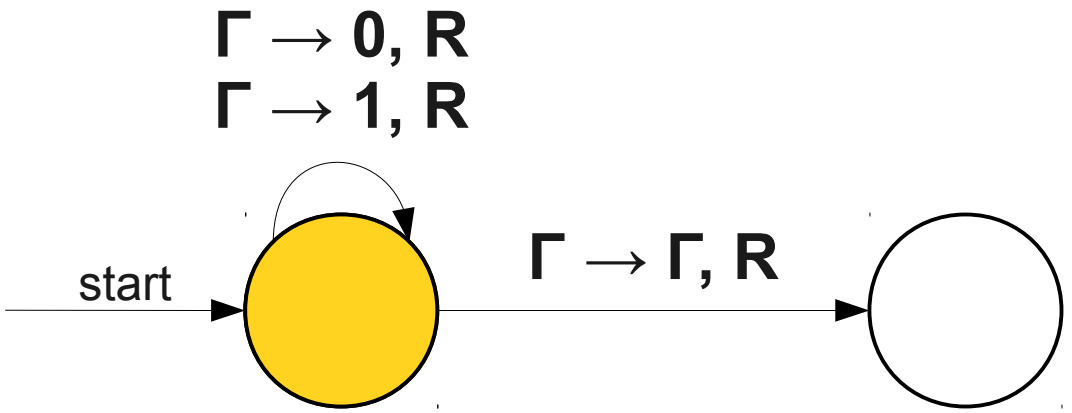
Guessing an Arbitrary String



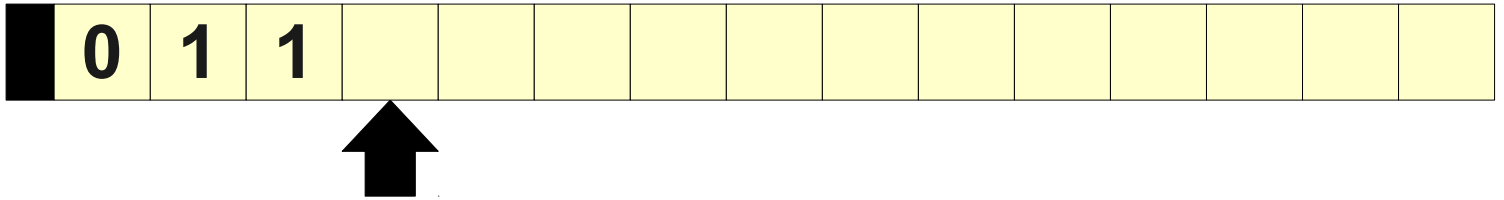


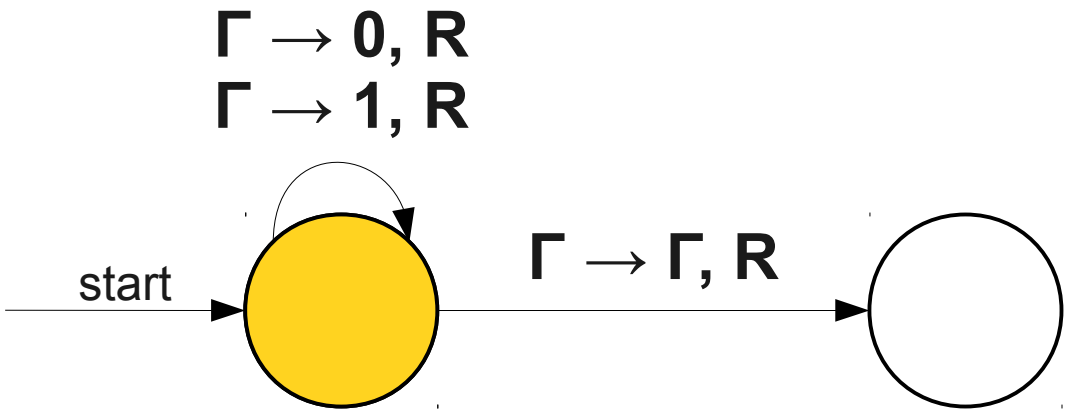
Guessing an Arbitrary String



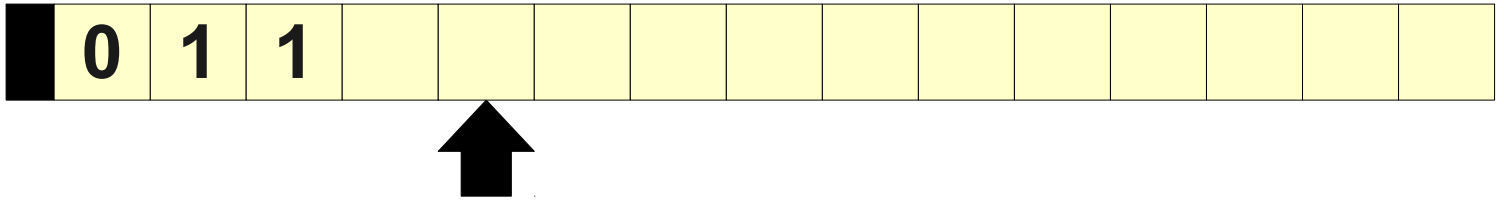


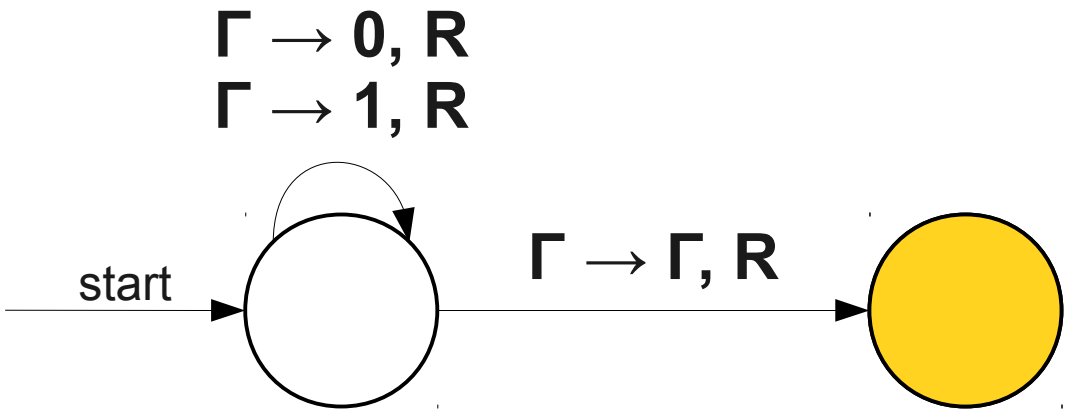
Guessing an Arbitrary String





Guessing an Arbitrary String





Nondeterminism to the Rescue!

- Consider the following NTM:

  N = "On input $\langle M \rangle$, where M is a Turing machine:

  Nondeterministically guess a string $x \in \Sigma^*$.

  Deterministically run M on x.

  If M accepts x, accept.

  If M rejects x, reject."

- Is this a legal nondeterministic Turing machine?
- If so, how would we prove $\mathscr{L}(M) = L_{NE}$?
- Does this say whether $L_{NE} \in \mathbf{RE}$?

Nondeterminism to the Rescue!

Consider the following NTM:

N = "On input $\langle M \rangle$, where M is a Turing machine:

Nondeterministically guess a string $x \in \Sigma^*$.

Deterministically run M on x.

If M accepts x, accept.

If M rejects x, reject."

Is this a legal nondeterministic Turing machine?

- If so, how would we prove $\mathscr{L}(M) = L_{NE}$?

Does this say whether $L_{NE} \in \mathbf{RE}$?

Proofs on NTMs

- Given a nondeterministic TM M and a language L, how do we prove that $\mathscr{L}(M) = L$?
- Prove the following:

For any $w \in \Sigma^*$, $w \in L$ iff there is a series of choices M can make such that M accepts w.

- Note the biconditional *and* the existential.

N = “On input $\langle M \rangle$, where M is a Turing machine:
Nondeterministically guess a string $x \in \Sigma^*$.
Deterministically run M on x.
If M accepts x, accept.
If M rejects x, reject.”

$$L_{NE} = \{ \langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) \neq \emptyset \}$$

Theorem: $\mathscr{L}(N) = L_{NE}$.

N = "On input $\langle M \rangle$, where M is a Turing machine:
Nondeterministically guess a string $x \in \Sigma^*$.
Deterministically run M on x.
If M accepts x, accept.
If M rejects x, reject."

$$L_{NE} = \{ \langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) \neq \emptyset \}$$

Theorem: $\mathscr{L}(N) = L_{NE}$.

Proof: We will prove that N accepts w iff $w \in L_{NE}$.

N = "On input $\langle M \rangle$, where M is a Turing machine:
Nondeterministically guess a string $x \in \Sigma^*$.
Deterministically run M on x.
If M accepts x, accept.
If M rejects x, reject."

$$L_{NE} = \{ \langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) \neq \emptyset \}$$

Theorem: $\mathscr{L}(N) = L_{NE}$.

Proof: We will prove that N accepts w iff $w \in L_{NE}$. To do this, note that by construction, N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such M accepts x.

N = "On input $\langle M \rangle$, where M is a Turing machine:
Nondeterministically guess a string $x \in \Sigma^*$.
Deterministically run M on x.
If M accepts x, accept.
If M rejects x, reject."

$$L_{NE} = \{ \langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) \neq \emptyset \}$$

Theorem: $\mathscr{L}(N) = L_{NE}$.

Proof: We will prove that N accepts w iff $w \in L_{NE}$. To do this, note that by construction, N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such M accepts x. Also note that M accepts x iff $x \in \mathscr{L}(M)$, so N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such that $x \in \mathscr{L}(M)$.

N = "On input $\langle M \rangle$, where M is a Turing machine:
Nondeterministically guess a string $x \in \Sigma^*$.
Deterministically run M on x.
If M accepts x, accept.
If M rejects x, reject."

$$L_{NE} = \{ \langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) \neq \emptyset \}$$

Theorem: $\mathscr{L}(N) = L_{NE}$.

Proof: We will prove that N accepts w iff $w \in L_{NE}$. To do this, note that by construction, N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such M accepts x. Also note that M accepts x iff $x \in \mathscr{L}(M)$, so N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such that $x \in \mathscr{L}(M)$. Finally, note that there is some choice of x such that $x \in \mathscr{L}(M)$ iff $\mathscr{L}(M) \neq \emptyset$.

N = "On input $\langle M \rangle$, where M is a Turing machine:
Nondeterministically guess a string $x \in \Sigma^*$.
Deterministically run M on x.
If M accepts x, accept.
If M rejects x, reject."

$$L_{NE} = \{ \langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) \neq \emptyset \}$$

Theorem: $\mathscr{L}(N) = L_{NE}$.

Proof: We will prove that N accepts w iff $w \in L_{NE}$. To do this, note that by construction, N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such M accepts x. Also note that M accepts x iff $x \in \mathscr{L}(M)$, so N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such that $x \in \mathscr{L}(M)$. Finally, note that there is some choice of x such that $x \in \mathscr{L}(M)$ iff $\mathscr{L}(M) \neq \emptyset$. This means that N accepts w iff $w = \langle M \rangle$ for some TM M and $\mathscr{L}(M) \neq \emptyset$.

N = "On input $\langle M \rangle$, where M is a Turing machine:
Nondeterministically guess a string $x \in \Sigma^*$.
Deterministically run M on x.
If M accepts x, accept.
If M rejects x, reject."

$$L_{NE} = \{ \langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) \neq \emptyset \}$$

Theorem: $\mathscr{L}(N) = L_{NE}$.

Proof: We will prove that N accepts w iff $w \in L_{NE}$. To do this, note that by construction, N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such M accepts x. Also note that M accepts x iff $x \in \mathscr{L}(M)$, so N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such that $x \in \mathscr{L}(M)$. Finally, note that there is some choice of x such that $x \in \mathscr{L}(M)$ iff $\mathscr{L}(M) \neq \emptyset$. This means that N accepts w iff $w = \langle M \rangle$ for some TM M and $\mathscr{L}(M) \neq \emptyset$. Thus by definition of L_{NE}, we have that N accepts w iff $w \in L_{NE}$.

N = "On input $\langle M \rangle$, where M is a Turing machine:
Nondeterministically guess a string $x \in \Sigma^*$.
Deterministically run M on x.
If M accepts x, accept.
If M rejects x, reject."

$$L_{NE} = \{ \langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) \neq \emptyset \}$$

Theorem: $\mathscr{L}(N) = L_{NE}$.

Proof: We will prove that N accepts w iff $w \in L_{NE}$. To do this, note that by construction, N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such M accepts x. Also note that M accepts x iff $x \in \mathscr{L}(M)$, so N accepts w iff $w = \langle M \rangle$ for some TM M and there is some choice of x such that $x \in \mathscr{L}(M)$. Finally, note that there is some choice of x such that $x \in \mathscr{L}(M)$ iff $\mathscr{L}(M) \neq \emptyset$. This means that N accepts w iff $w = \langle M \rangle$ for some TM M and $\mathscr{L}(M) \neq \emptyset$. Thus by definition of L_{NE}, we have that N accepts w iff $w \in L_{NE}$. ■

Nondeterminism to the Rescue!

- Consider the following NTM:

  N = "On input $\langle M \rangle$, where M is a Turing machine:

  Nondeterministically guess a string $x \in \Sigma^*$.

  Deterministically run M on x.

  If M accepts x, accept.

  If M rejects x, reject."

- Is this a legal nondeterministic Turing machine?
- If so, how would we prove $\mathscr{L}(M) = L_{NE}$?
- Does this say whether $L_{NE} \in \mathbf{RE}$?

Nondeterminism to the Rescue!

Consider the following NTM:

N = "On input $\langle M \rangle$, where M is a Turing machine:

Nondeterministically guess a string $x \in \Sigma^*$.

Deterministically run M on x.

If M accepts x, accept.

If M rejects x, reject."

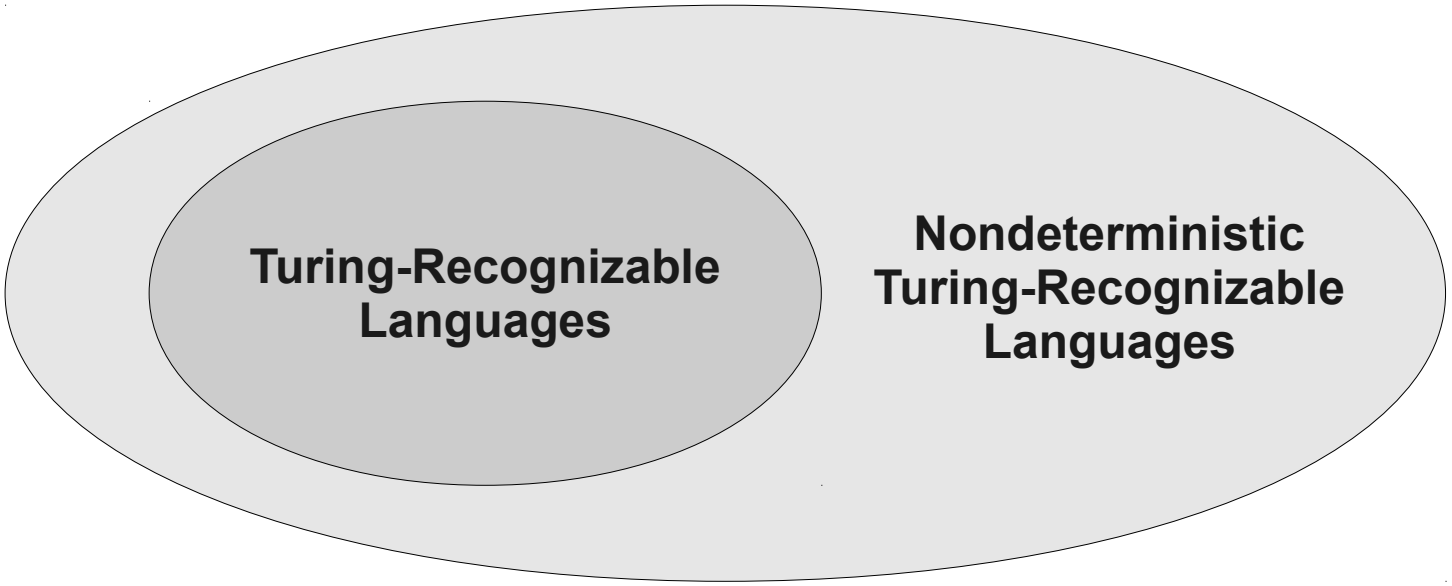
Is this a legal nondeterministic Turing machine?

If so, how would we prove $\mathscr{L}(M) = L_{NE}$?

- Does this say whether $L_{NE} \in \mathbf{RE}$?

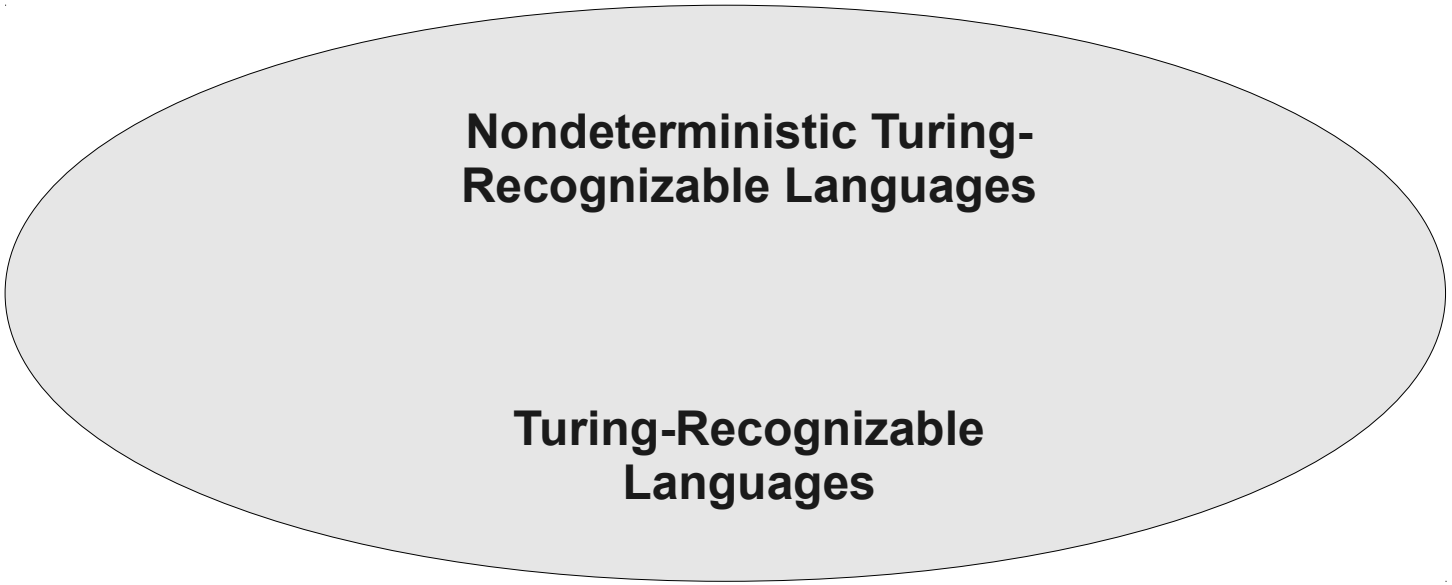
The Power of Nondeterminism

- 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?

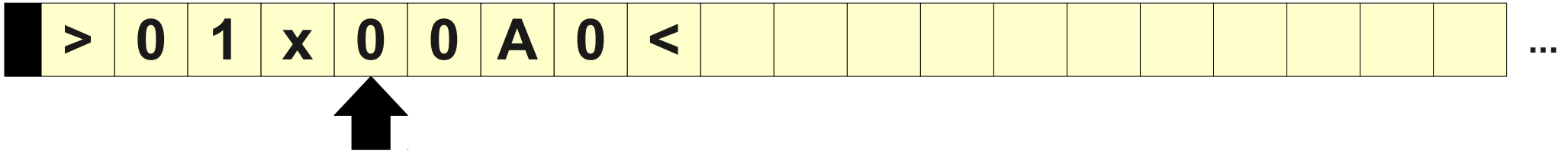


A Rather Remarkable Theorem

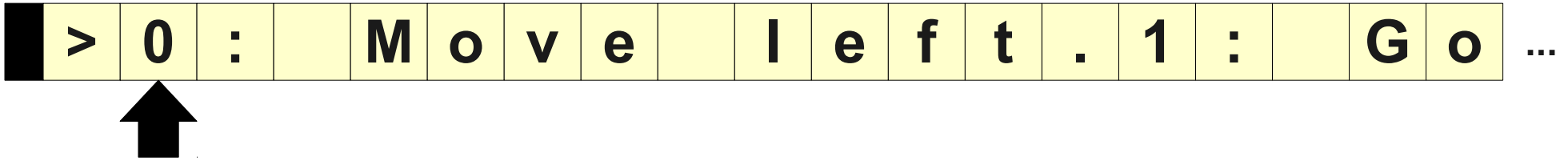
- **Theorem:** A language is recursively enumerable iff it is accepted by a nondeterministic Turing machine.
- How is this possible?

Sketch of the Universal **WB** Program

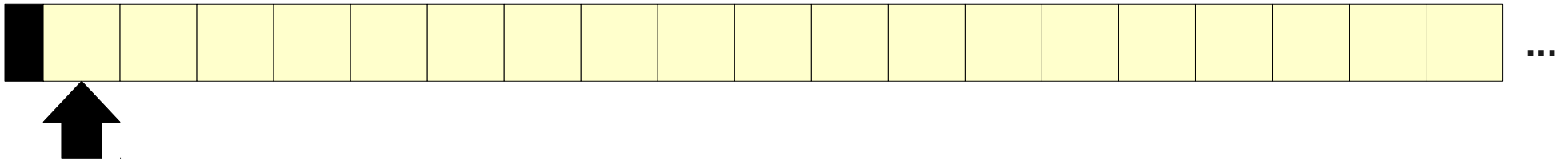
Simulated tape of the program being executed.



Program tape holding the program being executed.



Scratch tape for intermediate computation.

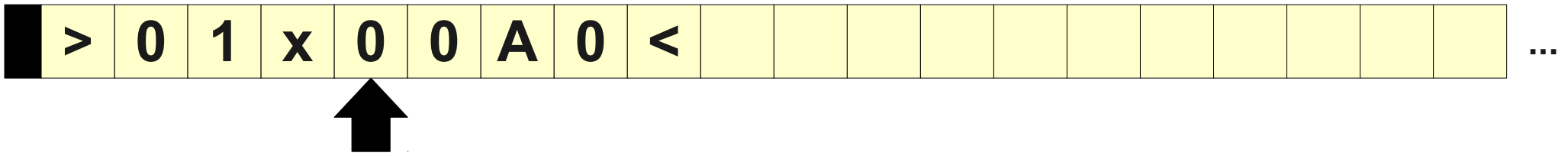


Variables for intermediate storage.

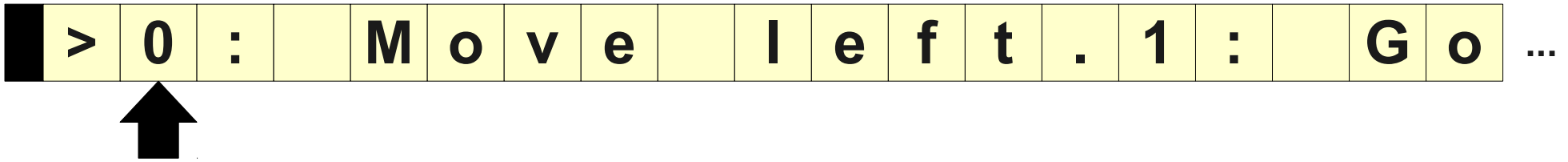
Instr **Letter**

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

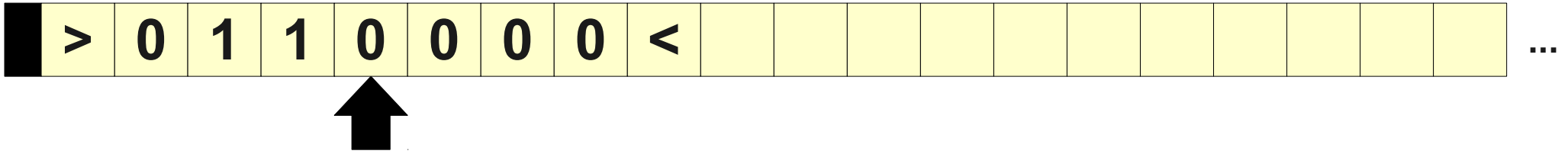
- An **instantaneous description** (or **ID**) of the execution of a program (TM, **WB** program, etc.) is a string encoding of a snapshot of that program at one instant in time.
- For Turing machines, it contains
  - The contents of the tape,
  - Where the tape head is, and
  - What state the machine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

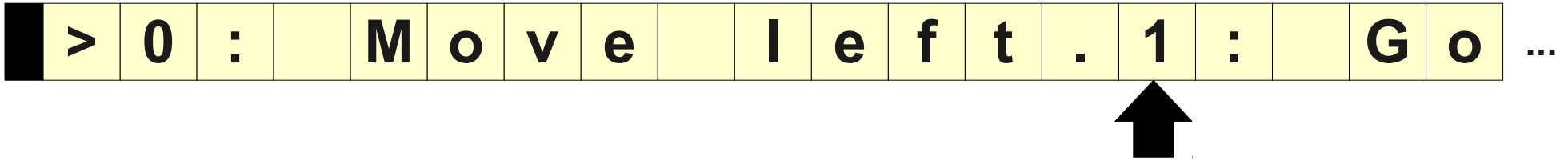
- There is a close connection between an ID and universal machines.
- The universal machine $U_{\mathbb{TM}}$ works by repeatedly reading the ID of the machine being simulated, then executing one step of that machine.

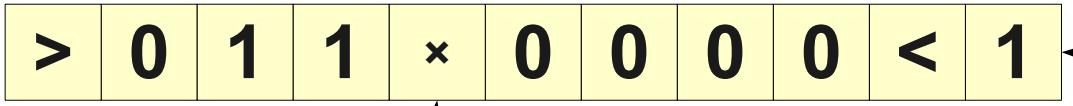
An ID for **WB** Programs

Simulated tape of the program being executed.



Program tape holding the program being executed.





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

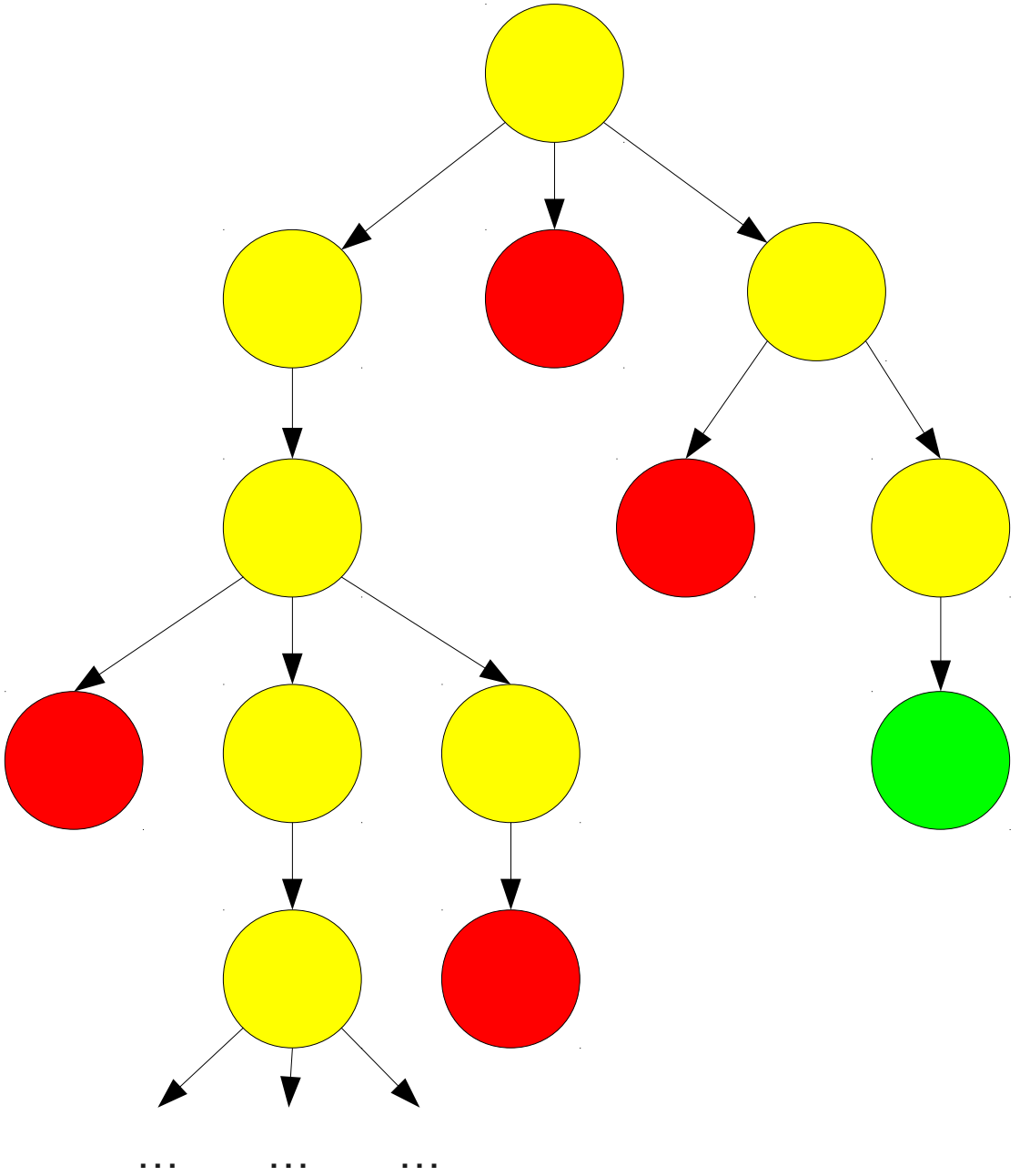
We write the line number at the end of the ID.

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.

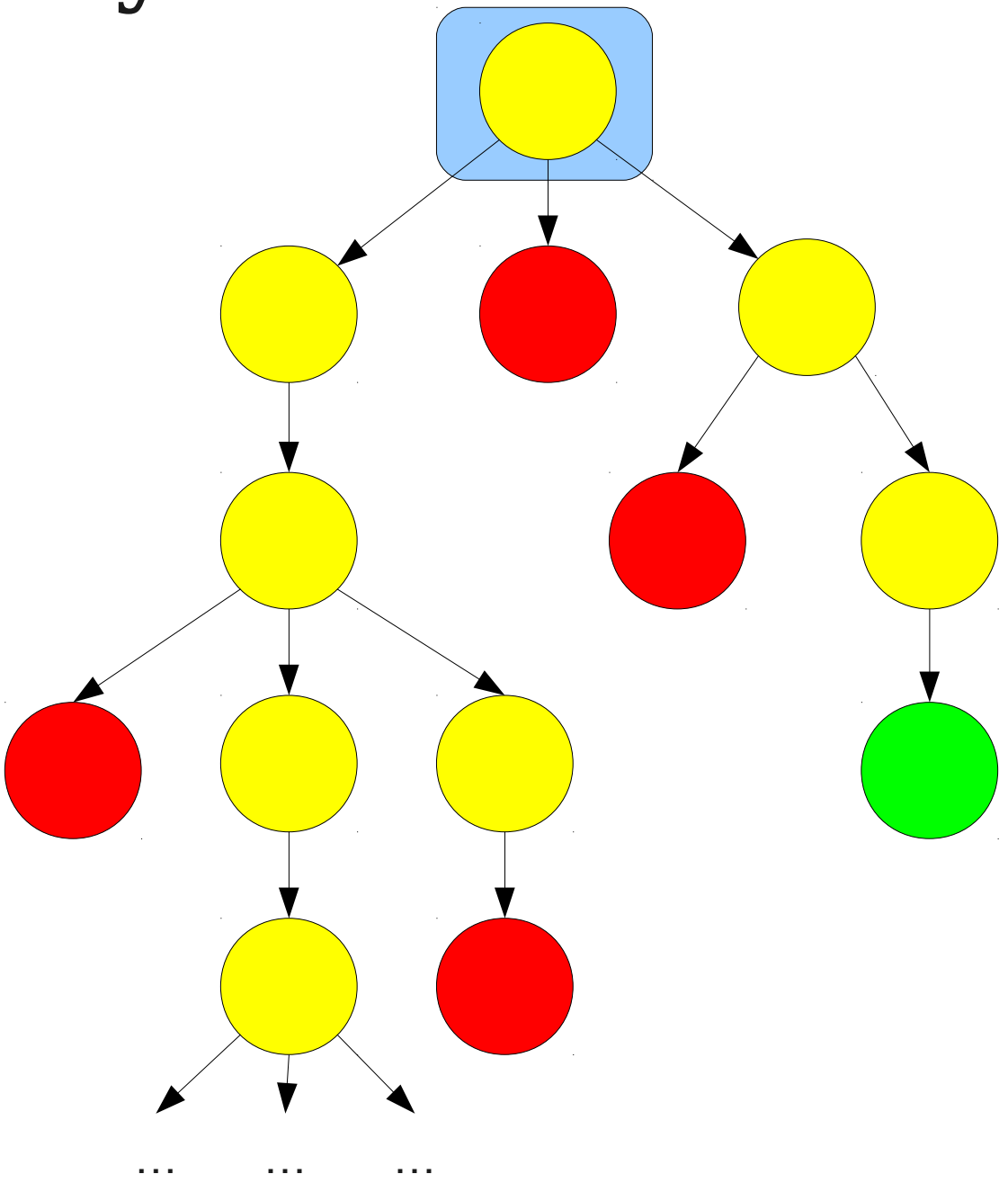
The Key Idea

- Store IDs of all of branches of computation and execute them in a breadth-first search.
- Uses the “tree computation” interpretation of nondeterminism.
- If there is an accepting computation, we will eventually find it.



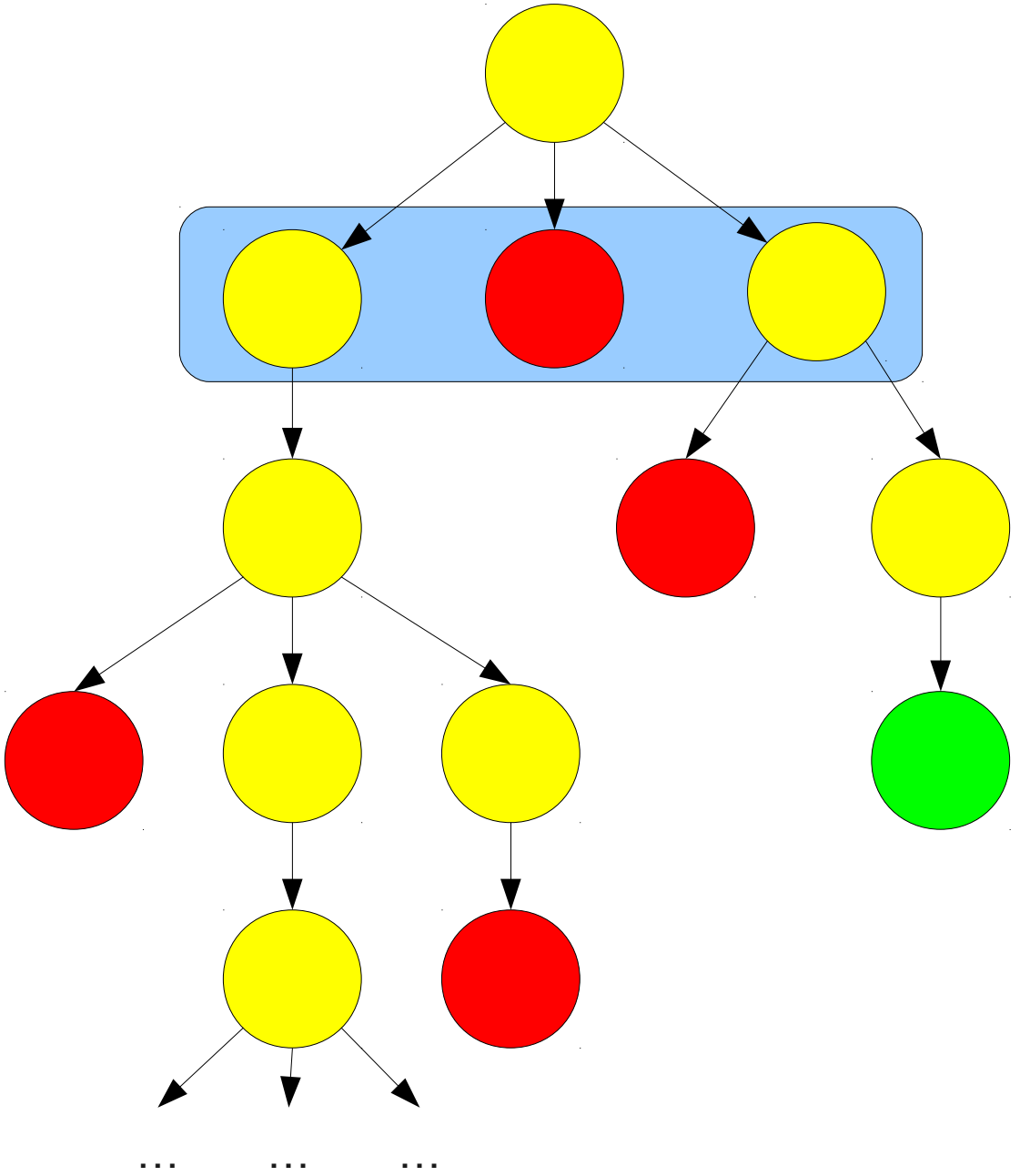
The Key Idea

- Store IDs of all of branches of computation and execute them in a breadth-first search.
- Uses the “tree computation” interpretation of nondeterminism.
- If there is an accepting computation, we will eventually find it.



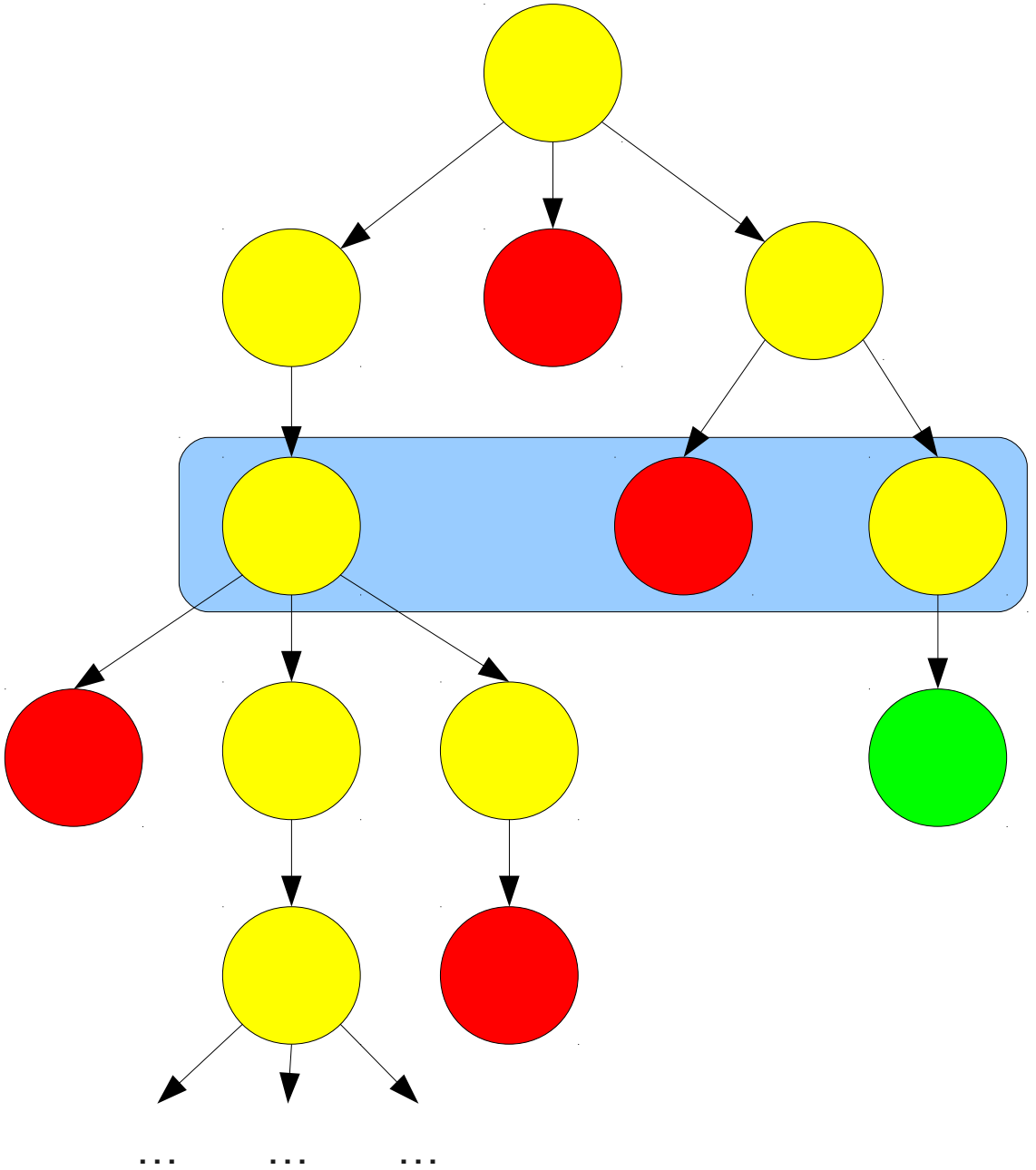
The Key Idea

- Store IDs of all of branches of computation and execute them in a breadth-first search.
- Uses the “tree computation” interpretation of nondeterminism.
- If there is an accepting computation, we will eventually find it.



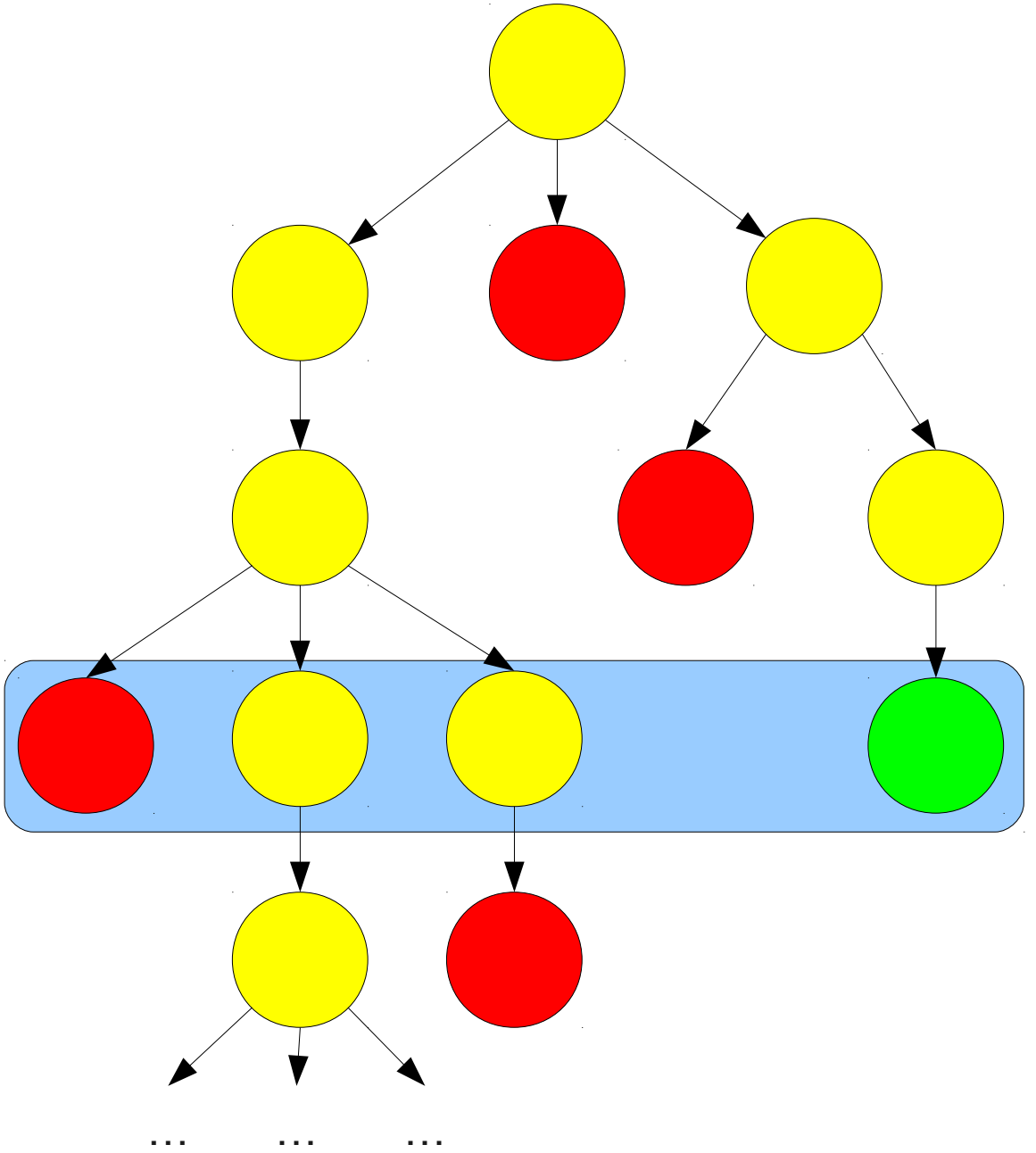
The Key Idea

- Store IDs of all of branches of computation and execute them in a breadth-first search.
- Uses the “tree computation” interpretation of nondeterminism.
- If there is an accepting computation, we will eventually find it.



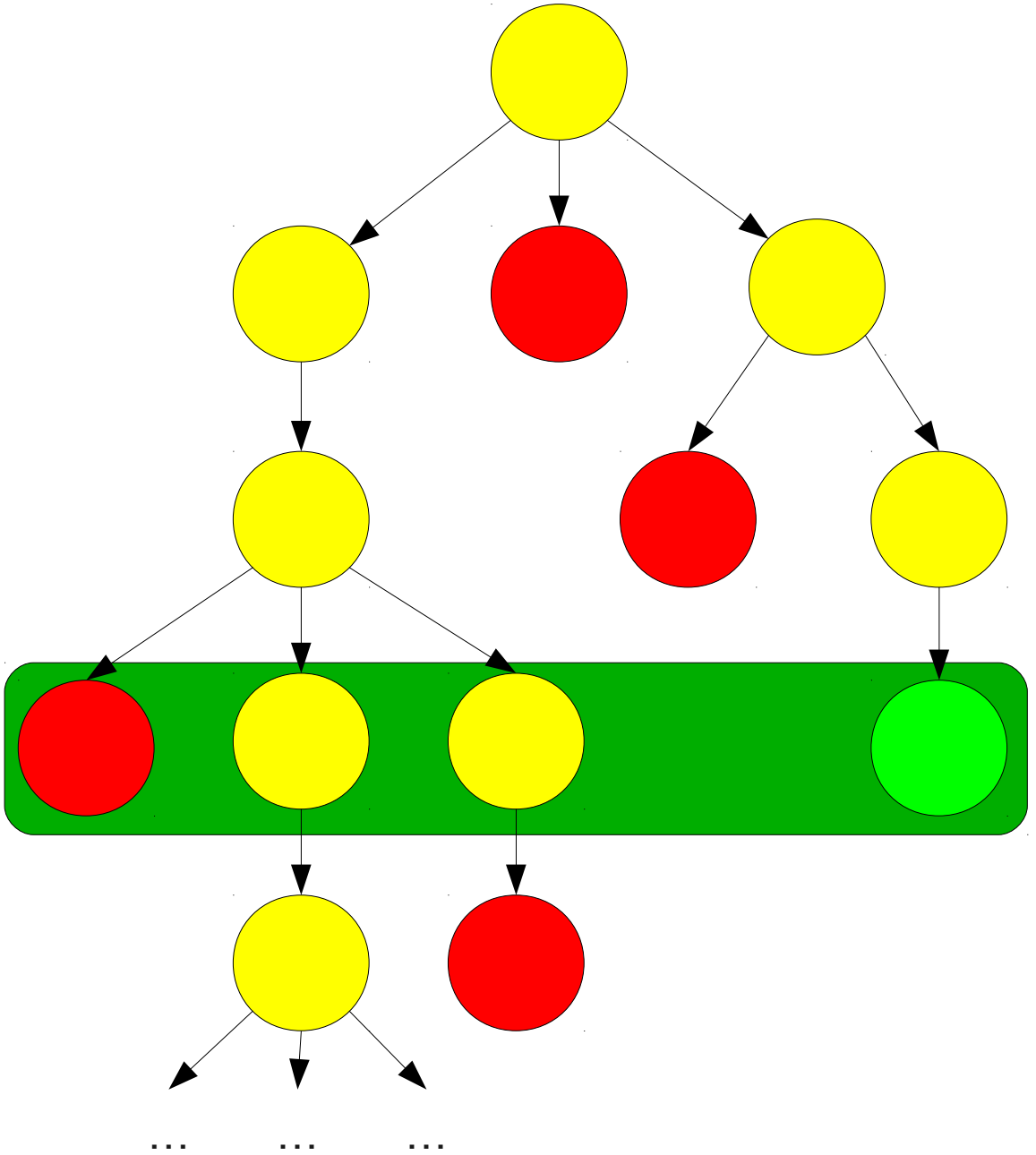
The Key Idea

- Store IDs of all of branches of computation and execute them in a breadth-first search.
- Uses the “tree computation” interpretation of nondeterminism.
- If there is an accepting computation, we will eventually find it.



The Key Idea

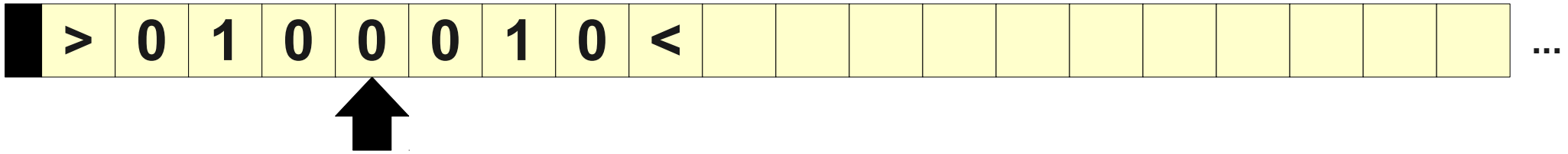
- Store IDs of all of branches of computation and execute them in a breadth-first search.
- Uses the “tree computation” interpretation of nondeterminism.
- If there is an accepting computation, we will eventually find it.



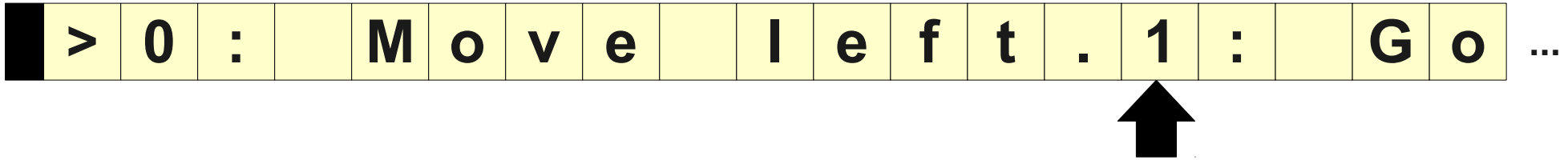
Simulating an NTM

- 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:
    - Move 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."

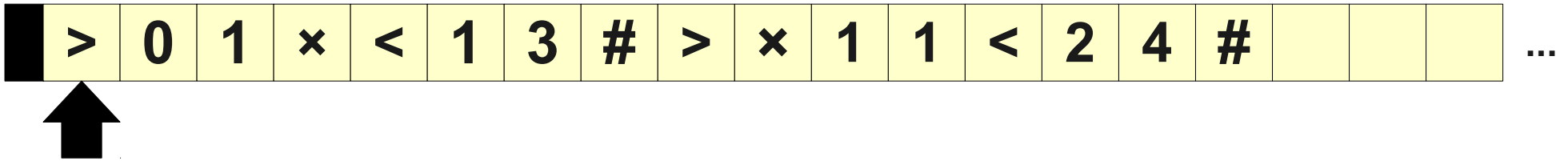
Simulated tape of the program being executed.



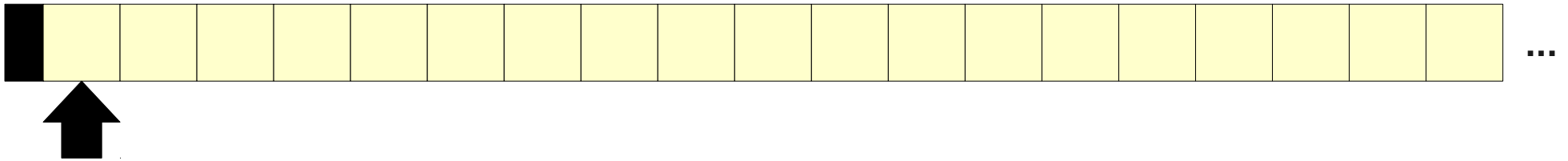
Program tape holding the program being executed.



Stored IDs



Scratch tape for intermediate computation.



Variables for intermediate storage.

Instr **Letter**

Why This Matters

- NTMs make it easier to solve a variety of problems.

Theorem: $L_{NE} \in \mathbf{RE}$.

- This is incredibly difficult to prove without nondeterminism.
- You will see more applications of nondeterminism in the problem set.

Next Time

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