COMP3630 / COMP6363

week 5: **Introduction to Turing Machines**

This Lecture Covers Chapter 8 of HMU: Introduction to Turing Machines

slides created by: Dirk Pattinson, based on material by
Peter Hoefner and Rob van Glabbeck; with improvements by Pascal Bercher

convenor & lecturer: Pascal Bercher

The Australian National University

Semester 1, 2025

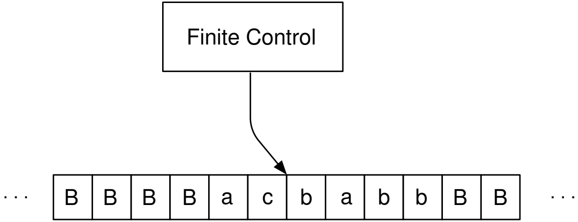
Content of this Chapter

- Turing Machine
- Extensions of Turing Machines
- Restrictions of Turing Machines
- Extensions of PDAs – and Relationship to TMs

Additional Reading: Chapter 8 of HMU.

Introduction to TMs

Turing Machine: Informal Definition



- A tape extending infinitely in both sides
- A reading head that can edit tape, move right or left
- A finite control
- A string is accepted if finite control ever reaches a final/accepting state

Heads-up: There are many variations of TMs (e.g., in COMP1600, the head could also stay stationary), and we will go through a few of them.

Turing Machine: Formal Definition

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is comprised of:

Turing Machine: Formal Definition

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is comprised of:

- Q: finite set of states

Turing Machine: Formal Definition

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is comprised of:

- Q: finite set of states
- Σ: finite set of input symbols

Turing Machine: Formal Definition

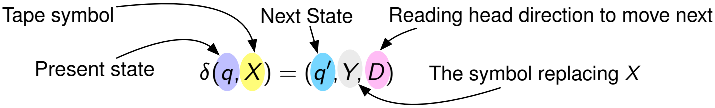
A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is comprised of:

- Q: finite set of states
- Σ: finite set of input symbols
- Γ: finite set of tape symbols such that $\Sigma \subseteq \Gamma$

Turing Machine: Formal Definition

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is comprised of:

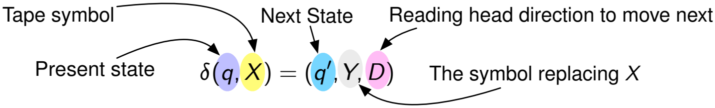
- Q: finite set of states
- Σ: finite set of input symbols
- Γ: finite set of tape symbols such that $\Sigma \subseteq \Gamma$
- δ: (deterministic) transition function. δ is a **partial function** over $Q \times \Gamma$, where the first component is viewed as the present state, and the second is viewed as the tape symbol read. If $\delta(q, X)$ is defined, then



Turing Machine: Formal Definition

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is comprised of:

- Q: finite set of states
- Σ: finite set of input symbols
- Γ: finite set of tape symbols such that $\Sigma \subseteq \Gamma$
- δ: (deterministic) transition function. δ is a **partial function** over $Q \times \Gamma$, where the first component is viewed as the present state, and the second is viewed as the tape symbol read. If $\delta(q, X)$ is defined, then

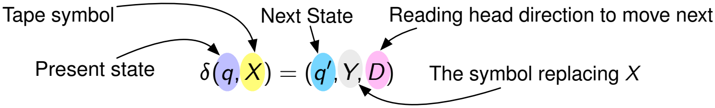


- $B \in \Gamma \setminus \Sigma$ is the blank symbol. All but a finite number of tape symbols are Bs.

Turing Machine: Formal Definition

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is comprised of:

- Q: finite set of states
- Σ: finite set of input symbols
- Γ: finite set of tape symbols such that $\Sigma \subseteq \Gamma$
- δ: (deterministic) transition function. δ is a **partial function** over $Q \times \Gamma$, where the first component is viewed as the present state, and the second is viewed as the tape symbol read. If $\delta(q, X)$ is defined, then

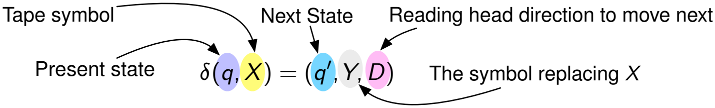


- $B \in \Gamma \setminus \Sigma$ is the blank symbol. All but a finite number of tape symbols are Bs.
- q_0: the initial state of the TM.

Turing Machine: Formal Definition

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is comprised of:

- Q: finite set of states
- Σ: finite set of input symbols
- Γ: finite set of tape symbols such that $\Sigma \subseteq \Gamma$
- δ: (deterministic) transition function. δ is a **partial function** over $Q \times \Gamma$, where the first component is viewed as the present state, and the second is viewed as the tape symbol read. If $\delta(q, X)$ is defined, then

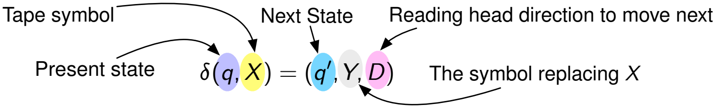


- $B \in \Gamma \setminus \Sigma$ is the blank symbol. All but a finite number of tape symbols are Bs.
- q_0: the initial state of the TM.
- F: the set of final/accepting states of the TM.

Turing Machine: Formal Definition

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is comprised of:

- Q: finite set of states
- Σ: finite set of input symbols
- Γ: finite set of tape symbols such that $\Sigma \subseteq \Gamma$
- δ: (deterministic) transition function. δ is a **partial function** over $Q \times \Gamma$, where the first component is viewed as the present state, and the second is viewed as the tape symbol read. If $\delta(q, X)$ is defined, then



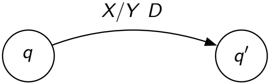
- $B \in \Gamma \setminus \Sigma$ is the blank symbol. All but a finite number of tape symbols are Bs.
- q_0: the initial state of the TM.
- F: the set of final/accepting states of the TM.

- Head **always** moves to the left or right. Being stationary is not an option. It can also be defined with such an option, see tutorial.

Describing TMs

- Turing machines can be defined by describing δ using a transition table.
- They can also be defined using transition diagrams (with labels appropriately altered)

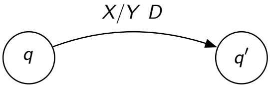
If $\delta(q, X) = (q', Y, D)$



Describing TMs

- Turing machines can be defined by describing δ using a transition table.
- They can also be defined using transition diagrams (with labels appropriately altered)

If $\delta(q, X) = (q', Y, D)$

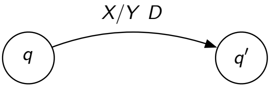


A TM that accepts any binary string that does not contain 111

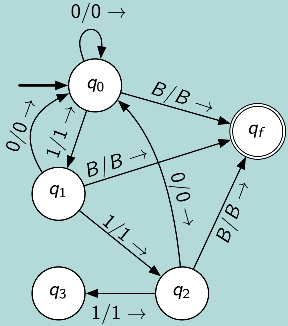
Describing TMs

- Turing machines can be defined by describing δ using a transition table.
- They can also be defined using transition diagrams (with labels appropriately altered)

If $\delta(q, X) = (q', Y, D)$



A TM that accepts any binary string that does not contain 111

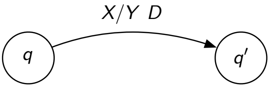


This encodes a DFA (almost).
Can you see why?

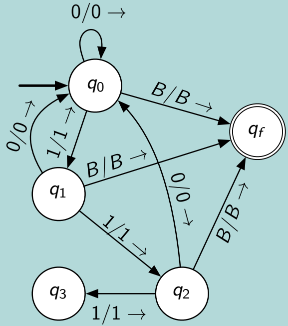
Describing TMs

- Turing machines can be defined by describing δ using a transition table.
- They can also be defined using transition diagrams (with labels appropriately altered)

If $\delta(q, X) = (q', Y, D)$



A TM that accepts any binary string that does not contain 111



This encodes a DFA (almost). Can you see why?

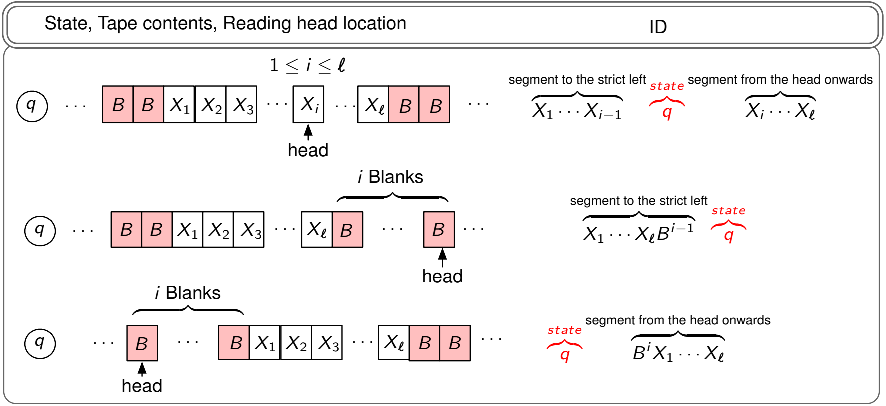
Because we never manipulate the tape and terminate once the String is read. The only difference is that not all edges are defined, but this can be fixed with a trap state.

Instantaneous Descriptions of TMs

- An instantaneous description (or configuration) of a TM is a complete description of the system that enables one to determine the trajectory of the TM as it operates.

Instantaneous Descriptions of TMs

- An instantaneous description (or configuration) of a TM is a complete description of the system that enables one to determine the trajectory of the TM as it operates.
- The instantaneous description or configuration or ID of a TM contains 3 parts:
  (a) The (finite, non-trivial) portion of tape to the left of the reading head;
  (b) the state that the TM is presently in; and
  (c) the (finite, non-trivial) portion of the tape to the right of the reading head.



'Moves' of a TM

- Just as in the case of a PDA, we use $\underset{M}{\vdash}$ to indicate a single move of a TM M, and $\overset{*}{\underset{M}{\vdash}}$ to indicate zero or a finite number of moves of a TM.

Present ID	Transition	**Next** ID
$X_1 \cdots X_{i-1} q X_i \cdots X_\ell$	$\delta(q, X_i) = (q', Y, R)$	$X_1 \cdots X_{i-1} Y q' X_{i+1} \cdots X_\ell$
$(1 < i < \ell)$	$\delta(q, X_i) = (q', Y, L)$	$X_1 \cdots X_{i-2} q' X_{i-1} Y X_{i+1} \cdots X_\ell$

'Moves' of a TM

- Just as in the case of a PDA, we use $\underset{M}{\vdash}$ to indicate a single move of a TM M, and $\overset{*}{\underset{M}{\vdash}}$ to indicate zero or a finite number of moves of a TM.

Present ID	Transition	**Next** ID
$X_1 \cdots X_{i-1} q X_i \cdots X_\ell$ $(1 < i < \ell)$	$\delta(q, X_i) = (q', Y, R)$	$X_1 \cdots X_{i-1} Y q' X_{i+1} \cdots X_\ell$
	$\delta(q, X_i) = (q', Y, L)$	$X_1 \cdots X_{i-2} q' X_{i-1} Y X_{i+1} \cdots X_\ell$
$X_1 \cdots X_\ell B^{i-1} q$	$\delta(q, B) = (q', Y, R)$	$X_1 \cdots X_\ell B^{i-1} Y q'$
	$\delta(q, B) = (q', Y, L)$	$\begin{cases} X_1 \cdots X_{\ell-1} q' X_\ell Y & i = 1 \\ X_1 \cdots X_\ell B^{i-2} q' B Y & i > 1 \end{cases}$

'Moves' of a TM

- Just as in the case of a PDA, we use $\underset{M}{\vdash}$ to indicate a single move of a TM M, and $\overset{*}{\underset{M}{\vdash}}$ to indicate zero or a finite number of moves of a TM.

Present ID	Transition	**Next** ID
$X_1 \cdots X_{i-1} q X_i \cdots X_\ell$ $(1 < i < \ell)$	$\delta(q, X_i) = (q', Y, R)$	$X_1 \cdots X_{i-1} Y q' X_{i+1} \cdots X_\ell$
	$\delta(q, X_i) = (q', Y, L)$	$X_1 \cdots X_{i-2} q' X_{i-1} Y X_{i+1} \cdots X_\ell$
$X_1 \cdots X_\ell B^{i-1} q$	$\delta(q, B) = (q', Y, R)$	$X_1 \cdots X_\ell B^{i-1} Y q'$
	$\delta(q, B) = (q', Y, L)$	$\begin{cases} X_1 \cdots X_{\ell-1} q' X_\ell Y & i = 1 \\ X_1 \cdots X_\ell B^{i-2} q' B Y & i > 1 \end{cases}$
$q B^i X_1 \ldots X_\ell$	$\delta(q, B) = (q', Y, R)$	$\begin{cases} Y q' X_2 \cdots X_\ell & i = 0 \\ Y q' B^{i-1} X_1 \cdots X_\ell & i > 0 \end{cases}$
	$\delta(q, B) = (q', Y, L)$	$\begin{cases} q' B Y X_2 \cdots X_\ell & i = 0 \\ q' B Y B^{i-1} X_1 \cdots X_\ell & i > 0 \end{cases}$

Language accepted by a TM

- A string w is in the language **accepted** by a TM M iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.

Language accepted by a TM

- A string w is in the language **accepted** by a TM M iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For accepted words, note that the TM may not necessarily halt (i.e., allow for no further transitions) in such a final state; it may even move to a non-final state afterwards. If a final state is ever visited, w is accepted in the language of M.

Language accepted by a TM

- A string w is in the language **accepted** by a TM M iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For accepted words, note that the TM may not necessarily halt (i.e., allow for no further transitions) in such a final state; it may even move to a non-final state afterwards. If a final state is ever visited, w is accepted in the language of M.
  - It is always possible to redesign a TM to halt for all accepted words without changing its language; just remove all transitions out of final states.

Language accepted by a TM

- A string w is in the language **accepted** by a TM M iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For accepted words, note that the TM may not necessarily halt (i.e., allow for no further transitions) in such a final state; it may even move to a non-final state afterwards. If a final state is ever visited, w is accepted in the language of M.
  - It is always possible to redesign a TM to halt for all accepted words without changing its language; just remove all transitions out of final states.
- Otherwise, w is **rejected** by M.

Language accepted by a TM

- A string w is in the language **accepted** by a TM M iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For accepted words, note that the TM may not necessarily halt (i.e., allow for no further transitions) in such a final state; it may even move to a non-final state afterwards. If a final state is ever visited, w is accepted in the language of M.
  - It is always possible to redesign a TM to halt for all accepted words without changing its language; just remove all transitions out of final states.
- Otherwise, w is **rejected** by M.
  - Careful: Rejection might be defined differently in different textbooks/courses. (In fact, our textbook doesn't even define it! It just "uses" it.)

Language accepted by a TM

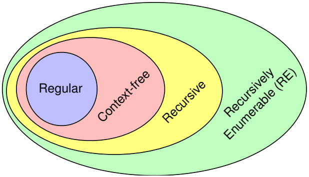
- A string w is in the language **accepted** by a TM M iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For accepted words, note that the TM may not necessarily halt (i.e., allow for no further transitions) in such a final state; it may even move to a non-final state afterwards. If a final state is ever visited, w is accepted in the language of M.
  - It is always possible to redesign a TM to halt for all accepted words without changing its language; just remove all transitions out of final states.
- Otherwise, w is **rejected** by M.
  - Careful: Rejection might be defined differently in different textbooks/courses. (In fact, our textbook doesn't even define it! It just "uses" it.)
  - We will learn that we can't always notice whether a word is rejected, because the TM might be stuck in a loop (and we don't know that). In general, we cannot redesign TMs to halt on all rejected words.

Language accepted by a TM

- A string w is in the language **accepted** by a TM M iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For accepted words, note that the TM may not necessarily halt (i.e., allow for no further transitions) in such a final state; it may even move to a non-final state afterwards. If a final state is ever visited, w is accepted in the language of M.
  - It is always possible to redesign a TM to halt for all accepted words without changing its language; just remove all transitions out of final states.
- Otherwise, w is **rejected** by M.
  - Careful: Rejection might be defined differently in different textbooks/courses. (In fact, our textbook doesn't even define it! It just "uses" it.)
  - We will learn that we can't always notice whether a word is rejected, because the TM might be stuck in a loop (and we don't know that). In general, we cannot redesign TMs to halt on all rejected words.
- A language L is **recursively enumerable** if it is accepted by some TM.

Language accepted by a TM

- A string w is in the language **accepted** by a TM M iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For accepted words, note that the TM may not necessarily halt (i.e., allow for no further transitions) in such a final state; it may even move to a non-final state afterwards. If a final state is ever visited, w is accepted in the language of M.
  - It is always possible to redesign a TM to halt for all accepted words without changing its language; just remove all transitions out of final states.
- Otherwise, w is **rejected** by M.
  - Careful: Rejection might be defined differently in different textbooks/courses. (In fact, our textbook doesn't even define it! It just "uses" it.)
  - We will learn that we can't always notice whether a word is rejected, because the TM might be stuck in a loop (and we don't know that). In general, we cannot redesign TMs to halt on all rejected words.
- A language L is **recursively enumerable** if it is accepted by some TM.
- A language L is **recursive** if it is accepted by a TM that **always** halts on all inputs.



(Context-sensitive languages sit between the CFLs and recursive languages.)

On Acceptance, Rejection, Halting, and Decidability

Usually, there is no need to define outgoing transitions for final states, since as soon as we enter a final state, the input word is accepted (so why proceed?)

On Acceptance, Rejection, Halting, and Decidability

Usually, there is no need to define outgoing transitions for final states, since as soon as we enter a final state, the input word is accepted (so why proceed?)

- Thus, when you pick/design a TM to accept a given language L, you can consider "reasonable" TMs that always halt on accepted words.

On Acceptance, Rejection, Halting, and Decidability

Usually, there is no need to define outgoing transitions for final states, since as soon as we enter a final state, the input word is accepted (so why proceed?)

- Thus, when you pick/design a TM to accept a given language L, you can consider "reasonable" TMs that always halt on accepted words.
- However, if you have to judge properties of a given TM (e.g., whether it always halts, or accepts a particular word etc.), then you have to deal with *any* TM – reasonable or not... (*Motivation:* You might want to judge properties of somebody's "program")

On Acceptance, Rejection, Halting, and Decidability

Usually, there is no need to define outgoing transitions for final states, since as soon as we enter a final state, the input word is accepted (so why proceed?)

- Thus, when you pick/design a TM to accept a given language L, you can consider "reasonable" TMs that always halt on accepted words.
- However, if you have to judge properties of a given TM (e.g., whether it always halts, or accepts a particular word etc.), then you have to deal with *any* TM – reasonable or not... (*Motivation:* You might want to judge properties of somebody's "program")

Let L be given and M a TM, such that $L(M) = L$.

On Acceptance, Rejection, Halting, and Decidability

Usually, there is no need to define outgoing transitions for final states, since as soon as we enter a final state, the input word is accepted (so why proceed?)

- Thus, when you pick/design a TM to accept a given language L, you can consider "reasonable" TMs that always halt on accepted words.
- However, if you have to judge properties of a given TM (e.g., whether it always halts, or accepts a particular word etc.), then you have to deal with *any* TM – reasonable or not... (*Motivation:* You might want to judge properties of somebody's "program")

Let L be given and M a TM, such that $L(M) = L$.

- If M halts on all inputs in Σ^*, we call it a total TM, say that it decides L, and call L a decidable (or recursive) language.

On Acceptance, Rejection, Halting, and Decidability

Usually, there is no need to define outgoing transitions for final states, since as soon as we enter a final state, the input word is accepted (so why proceed?)

- Thus, when you pick/design a TM to accept a given language L, you can consider "reasonable" TMs that always halt on accepted words.
- However, if you have to judge properties of a given TM (e.g., whether it always halts, or accepts a particular word etc.), then you have to deal with *any* TM – reasonable or not... (*Motivation:* You might want to judge properties of somebody's "program")

Let L be given and M a TM, such that $L(M) = L$.

- If M halts on all inputs in Σ^*, we call it a total TM, say that it decides L, and call L a decidable (or recursive) language.
- If M halts on all inputs taken from L (but does not necessarily halt for all inputs in $\Sigma^* \setminus L$), then L is semi-decidable (or recursively enumerable).
- In both cases, M 'accepts' L. (That's literally what $L(M) = L$ means!)

On Acceptance, Rejection, Halting, and Decidability

Usually, there is no need to define outgoing transitions for final states, since as soon as we enter a final state, the input word is accepted (so why proceed?)

- Thus, when you pick/design a TM to accept a given language L, you can consider "reasonable" TMs that always halt on accepted words.
- However, if you have to judge properties of a given TM (e.g., whether it always halts, or accepts a particular word etc.), then you have to deal with *any* TM – reasonable or not... (*Motivation:* You might want to judge properties of somebody's "program")

Let L be given and M a TM, such that $L(M) = L$.

- If M halts on all inputs in Σ^*, we call it a total TM, say that it decides L, and call L a decidable (or recursive) language.
- If M halts on all inputs taken from L (but does not necessarily halt for all inputs in $\Sigma^* \setminus L$), then L is semi-decidable (or recursively enumerable).
- In both cases, M 'accepts' L. (That's literally what $L(M) = L$ means!)
- Note: Just because $L \in RE$ does not mean $L \notin R$ since $R \subseteq RE$. Also, $R \subsetneq RE$.

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$,

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).
- "Rejecting w", i.e. $w \notin L(M)$,

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).
- "Rejecting w", i.e. $w \notin L(M)$, also does not imply halting. Either w was rejected because the TM halted after a finite number of steps having never reached a final state, or the TM loops through non-final states forever (and never reached a final state before).

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).
- "Rejecting w", i.e. $w \notin L(M)$, also does not imply halting. Either w was rejected because the TM halted after a finite number of steps having never reached a final state, or the TM loops through non-final states forever (and never reached a final state before).
- "Not accepting w"

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).
- "Rejecting w", i.e. $w \notin L(M)$, also does not imply halting. Either w was rejected because the TM halted after a finite number of steps having never reached a final state, or the TM loops through non-final states forever (and never reached a final state before).
- "Not accepting w" is the same as "rejecting w."

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).
- "Rejecting w", i.e. $w \notin L(M)$, also does not imply halting. Either w was rejected because the TM halted after a finite number of steps having never reached a final state, or the TM loops through non-final states forever (and never reached a final state before).
- "Not accepting w" is the same as "rejecting w."
- "Not rejecting w"

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).
- "Rejecting w", i.e. $w \notin L(M)$, also does not imply halting. Either w was rejected because the TM halted after a finite number of steps having never reached a final state, or the TM loops through non-final states forever (and never reached a final state before).
- "Not accepting w" is the same as "rejecting w."
- "Not rejecting w" is the same as "accepting w."

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).
- "Rejecting w", i.e. $w \notin L(M)$, also does not imply halting. Either w was rejected because the TM halted after a finite number of steps having never reached a final state, or the TM loops through non-final states forever (and never reached a final state before).
- "Not accepting w" is the same as "rejecting w."
- "Not rejecting w" is the same as "accepting w."
- "Halting on w"

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).
- "Rejecting w", i.e. $w \notin L(M)$, also does not imply halting. Either w was rejected because the TM halted after a finite number of steps having never reached a final state, or the TM loops through non-final states forever (and never reached a final state before).
- "Not accepting w" is the same as "rejecting w."
- "Not rejecting w" is the same as "accepting w."
- "Halting on w" neither implies $w \in L(M)$ nor $w \notin L(M)$, i.e., it is not enough information to determine whether w was accepted or rejected. This is because we do not know whether we have previously traversed a final state.

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).
- "Rejecting w", i.e. $w \notin L(M)$, also does not imply halting. Either w was rejected because the TM halted after a finite number of steps having never reached a final state, or the TM loops through non-final states forever (and never reached a final state before).
- "Not accepting w" is the same as "rejecting w."
- "Not rejecting w" is the same as "accepting w."
- "Halting on w" neither implies $w \in L(M)$ nor $w \notin L(M)$, i.e., it is not enough information to determine whether w was accepted or rejected. This is because we do not know whether we have previously traversed a final state.
- "Not halting on w"

On Acceptance, Rejection, Halting, and Decidability

- "Accepting w", i.e. $w \in L(M)$, does not imply halting, since the machine may keep going after reaching a final state. If we design a TM, there's no point in defining transitions out of any final state (so, it would halt).
- "Rejecting w", i.e. $w \notin L(M)$, also does not imply halting. Either w was rejected because the TM halted after a finite number of steps having never reached a final state, or the TM loops through non-final states forever (and never reached a final state before).
- "Not accepting w" is the same as "rejecting w."
- "Not rejecting w" is the same as "accepting w."
- "Halting on w" neither implies $w \in L(M)$ nor $w \notin L(M)$, i.e., it is not enough information to determine whether w was accepted or rejected. This is because we do not know whether we have previously traversed a final state.
- "Not halting on w" also neither implies $w \in L(M)$ nor $w \notin L(M)$. Those are determined by whether the TM ever traverses a final state. However, if the TM is "reasonable", then $w \notin L(M)$ (i.e., the word is rejected).

On Acceptance, Rejection, Halting, and Decidability

	Accepted $w \in L(M)$	**Rejected** $w \notin L(M)$
Halted on *w*		
Looped forever on *w*		

On Acceptance, Rejection, Halting, and Decidability

	Accepted $w \in L(M)$	**Rejected** $w \notin L(M)$
Halted on w	reached a final state, possibly kept going, then eventually reached a point where no further transition was possible	
Looped forever on w		

On Acceptance, Rejection, Halting, and Decidability

	Accepted $w \in L(M)$	**Rejected** $w \notin L(M)$
Halted on w	reached a final state, possibly kept going, then eventually reached a point where no further transition was possible	eventually reached a point where no further transition was possible and never traversed a final state
Looped forever on w		

On Acceptance, Rejection, Halting, and Decidability

	Accepted $w \in L(M)$	**Rejected** $w \notin L(M)$
Halted on w	reached a final state, possibly kept going, then eventually reached a point where no further transition was possible	eventually reached a point where no further transition was possible and never traversed a final state
Looped forever on w	reached a final state and keeps making transitions forever	

On Acceptance, Rejection, Halting, and Decidability

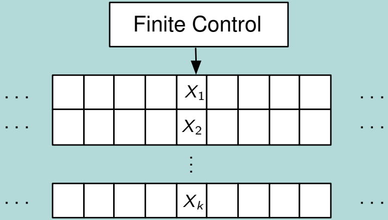
	Accepted $w \in L(M)$	**Rejected** $w \notin L(M)$
Halted on w	reached a final state, possibly kept going, then eventually reached a point where no further transition was possible	eventually reached a point where no further transition was possible and never traversed a final state
Looped forever on w	reached a final state and keeps making transitions forever	keeps making transitions forever, traversing non-final states only

Extensions of TMs

Multiple-Track TMs

Multiple-track TM

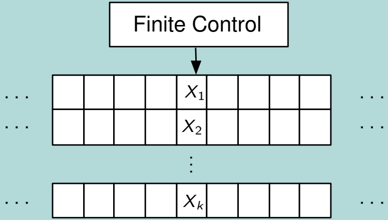
- We do not provide a formal definition (but assume you could provide one).
- There are k tracks, each having symbols written on them. They are essentially tapes, but we call them that way since they are not independent.



Multiple-Track TMs

Multiple-track TM

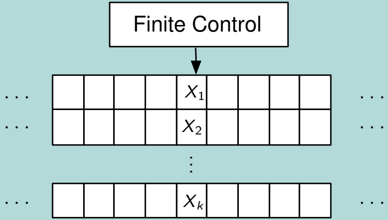
- We do not provide a formal definition (but assume you could provide one).
- There are k tracks, each having symbols written on them. They are essentially tapes, but we call them that way since they are not independent.
- The machine can only read symbols from each tape corresponding to **one** location, i.e., all symbols in a column at any one time.
- Likewise, all tapes move simultaneously in the same direction.



Multiple-Track TMs

Multiple-track TM

- We do not provide a formal definition (but assume you could provide one).
- There are k tracks, each having symbols written on them. They are essentially tapes, but we call them that way since they are not independent.
- The machine can only read symbols from each tape corresponding to **one** location, i.e., all symbols in a column at any one time.
- Likewise, all tapes move simultaneously in the same direction.

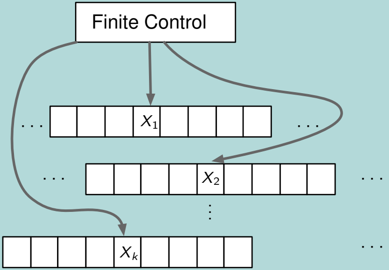


- A k-track TM with tape alphabet Γ has the same language-acceptance power as a TM with tape alphabet Γ^k. (E.g., each cell contains the "symbol" $(X_1, \ldots, X_k)$)

Multi-tape TMs

Multiple-tape TM

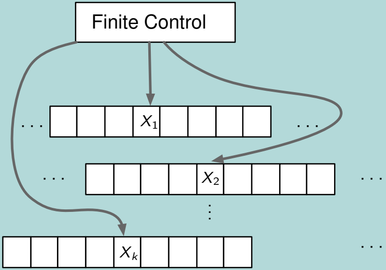
- We (again) don't provide a formal definition (but assume you could provide one).
- There are k (independent) tapes, each having symbols written on them.



Multi-tape TMs

Multiple-tape TM

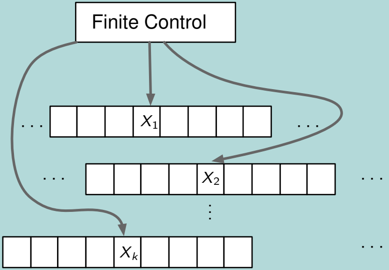
- We (again) don't provide a formal definition (but assume you could provide one).
- There are k (independent) tapes, each having symbols written on them.
- The machine can read each tape independently, i.e., the symbols read from each tape need not correspond to the same location.



Multi-tape TMs

Multiple-tape TM

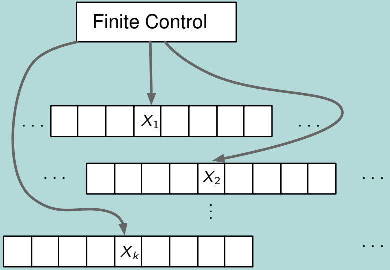
- We (again) don't provide a formal definition (but assume you could provide one).
- There are k (independent) tapes, each having symbols written on them.
- The machine can read each tape independently, i.e., the symbols read from each tape need not correspond to the same location.
- After all tapes are read, all tape transitions must happen (now we can also stay with a head – the entire TM is for convenience anyway!).



Multi-tape TMs

Multiple-tape TM

- We (again) don't provide a formal definition (but assume you could provide one).
- There are k (independent) tapes, each having symbols written on them.
- The machine can read each tape independently, i.e., the symbols read from each tape need not correspond to the same location.
- After all tapes are read, all tape transitions must happen (now we can also stay with a head – the entire TM is for convenience anyway!).



- The rest stays the same (e.g., one set of states, acceptance, etc.).

Multi-tape TMs

Theorem 8.2.1

Every language that is accepted by a multi-tape TM is also recursively enumerable (i.e., accepted by some 'standard' TM).

Proof of Theorem 8.2.1

Multi-tape TMs

Theorem 8.2.1

Every language that is accepted by a multi-tape TM is also recursively enumerable (i.e., accepted by some 'standard' TM).

Proof of Theorem 8.2.1

- Let L be accepted by a k-tape TM M. We'll devise a $2k$-track TM M' that accepts L.

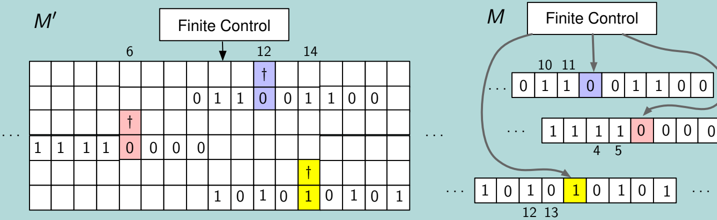
Multi-tape TMs

Theorem 8.2.1

Every language that is accepted by a multi-tape TM is also recursively enumerable (i.e., accepted by some 'standard' TM).

Proof of Theorem 8.2.1

- Let L be accepted by a k-tape TM M. We'll devise a $2k$-track TM M' that accepts L.
- Every even tape of M' has the same alphabet as that of the k-tape TM.
  The $2i^{\text{th}}$ track of M' contains exactly the same contents as the i^{th} tape of M.



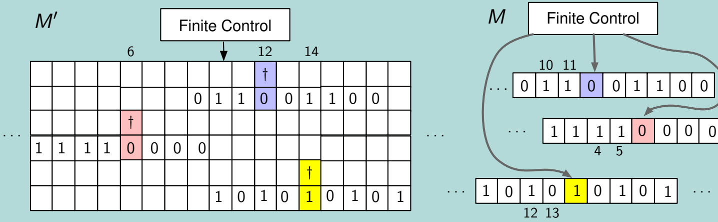
Multi-tape TMs

Theorem 8.2.1

Every language that is accepted by a multi-tape TM is also recursively enumerable (i.e., accepted by some 'standard' TM).

Proof of Theorem 8.2.1

- Let L be accepted by a k-tape TM M. We'll devise a $2k$-track TM M' that accepts L.
- Every even tape of M' has the same alphabet as that of the k-tape TM.
  The $2i^{\text{th}}$ track of M' contains exactly the same contents as the i^{th} tape of M.
- Every odd track has an alphabet $\{B, \dagger\}$, and contains a single $\dagger$.
  The $2i-1^{\text{th}}$ track of M' contains $\dagger$ at the location where the i^{th} head of M is located.

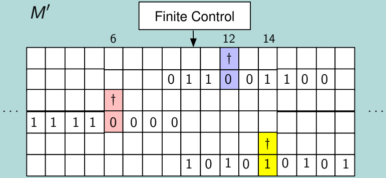


Multi-tape TMs

Proof of Theorem 8.2.1 (1 of 3)

What is the main problem we need to solve?

- In the Multi-tape TM M, heads move independently, whereas in the Multi-track TM M' they do not. So the heads can diverge:



(But M' has just a single head position!)

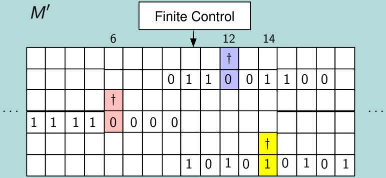
So, how to solve it?

Multi-tape TMs

Proof of Theorem 8.2.1 (1 of 3)

What is the main problem we need to solve?

- In the Multi-tape TM M, heads move independently, whereas in the Multi-track TM M' they do not. So the heads can diverge:



(But M' has just a single head position!)

So, how to solve it?

- Make sure that in each transition of M, we visit all heads of M'.
- "Store" all head positions in a state with k (number of tapes) entries.

Multi-tape TMs

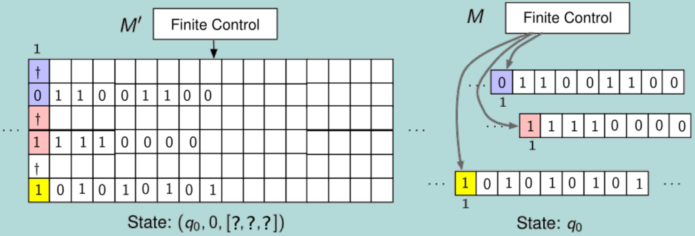
Proof of Theorem 8.2.1 (2 of 3)

- The state of M' has 3 components: (a) the state of M; (b) the number of †s to its head's strict left; and (c) a k-length tuple from $(\Gamma \cup \{?\})^k$.

Multi-tape TMs

Proof of Theorem 8.2.1 (2 of 3)

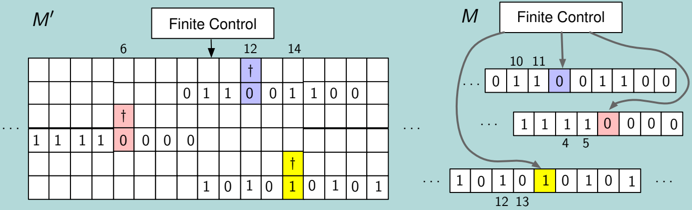
- The state of M' has 3 components: (a) the state of M; (b) the number of †s to its head's strict left; and (c) a k-length tuple from $(\Gamma \cup \{?\})^k$.
- At the beginning of the sweep, the head of M' is at the location of the leftmost † and the state of M' is $(q, 0, [?, \cdots, ?])$. The head moves to the right uncovering †s and the corresponding track symbols (are stored in the third component of the state).



Multi-tape TMs

Proof of Theorem 8.2.1 (2 of 3)

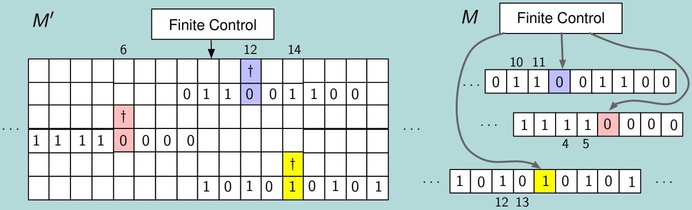
- The state of M' has 3 components: (a) the state of M; (b) the number of †s to its head's strict left; and (c) a k-length tuple from $(\Gamma \cup \{?\})^k$.
- At the beginning of the sweep, the head of M' is at the location of the leftmost † and the state of M' is $(q, 0, [?, \cdots, ?])$. The head moves to the right uncovering †s and the corresponding track symbols (are stored in the third component of the state).
- Each move of M takes multiple moves of M', and is a sweep of the tape from the location of the leftmost † to that of the rightmost † and back performing the changes to tracks that M would do to its corresponding tapes.
- The right sweep ends when the second component is k.



Multi-tape TMs

Proof of Theorem 8.2.1 (3 of 3)

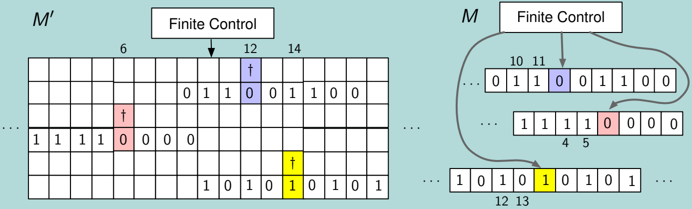
- At this stage (once the i in $(q, i, [\gamma_1, \cdots, \gamma_k])$ is k and all γ_j are set), M' knows the head symbols M will have read, and knows what actions to take.



Multi-tape TMs

Proof of Theorem 8.2.1 (3 of 3)

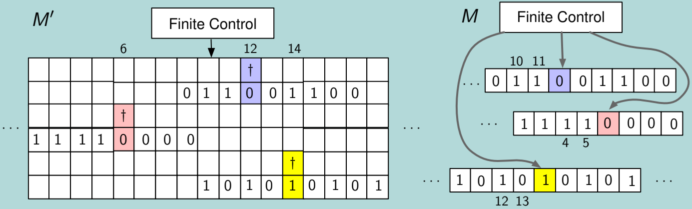
- At this stage (once the i in $(q, i, [\gamma_1, \cdots, \gamma_k])$ is k and all γ_j are set), M' knows the head symbols M will have read, and knows what actions to take.
- It then sweeps left making appropriate changes to the tracks (just like M does to its tape) each time a † is encountered. M' also moves the †s accordingly.



Multi-tape TMs

Proof of Theorem 8.2.1 (3 of 3)

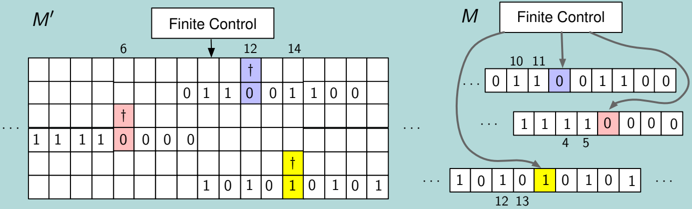
- At this stage (once the i in $(q, i, [\gamma_1, \cdots, \gamma_k])$ is k and all γ_j are set), M' knows the head symbols M will have read, and knows what actions to take.
- It then sweeps left making appropriate changes to the tracks (just like M does to its tape) each time a † is encountered. M' also moves the †s accordingly.
- The left sweep ends when the second component is zero. At this time, M' would have completed moving the †s and the track contents; they'll now match those of M.



Multi-tape TMs

Proof of Theorem 8.2.1 (3 of 3)

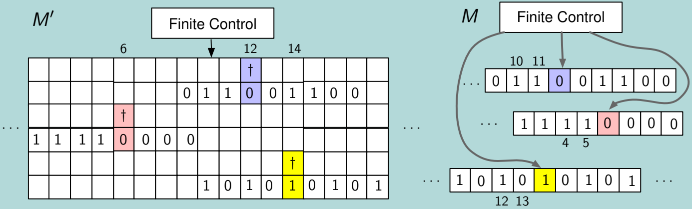
- At this stage (once the i in $(q, i, [\gamma_1, \cdots, \gamma_k])$ is k and all γ_j are set), M' knows the head symbols M will have read, and knows what actions to take.
- It then sweeps left making appropriate changes to the tracks (just like M does to its tape) each time a † is encountered. M' also moves the †s accordingly.
- The left sweep ends when the second component is zero. At this time, M' would have completed moving the †s and the track contents; they'll now match those of M.
- M' then moves the state to $(q', 0, [?, \cdots, ?])$ and starts the next sweep if q' is not a final state.



Multi-tape TMs

Proof of Theorem 8.2.1 (3 of 3)

- At this stage (once the i in $(q, i, [\gamma_1, \cdots, \gamma_k])$ is k and all γ_j are set), M' knows the head symbols M will have read, and knows what actions to take.
- It then sweeps left making appropriate changes to the tracks (just like M does to its tape) each time a † is encountered. M' also moves the †s accordingly.
- The left sweep ends when the second component is zero. At this time, M' would have completed moving the †s and the track contents; they'll now match those of M.
- M' then moves the state to $(q', 0, [?, \cdots, ?])$ and starts the next sweep if q' is not a final state.
- Note that M' mimics M and hence the languages accepted are identical.



Multi-tape TMs

- The running time of a TM M with input w is the number of moves M makes before it halts. (If it does not, the running time is ∞).

Multi-tape TMs

- The running time of a TM M with input w is the number of moves M makes before it halts. (If it does not, the running time is ∞).
- The time complexity $T_M : \{0, 1, \ldots\} \rightarrow \{0, 1, \ldots\} \cup \{\infty\}$ of a TM M is defined as:
  - $T_M(n) :=$ maximum running time of M for an input w of length n symbols.

Multi-tape TMs

- The running time of a TM M with input w is the number of moves M makes before it halts. (If it does not, the running time is ∞).
- The time complexity $T_M : \{0, 1, \ldots\} \to \{0, 1, \ldots\} \cup \{\infty\}$ of a TM M is defined as:
  - $T_M(n) :=$ maximum running time of M for an input w of length n symbols.

Theorem 8.2.2

The time taken for M' in Theorem 8.2.1 to process n moves of k-tape M is $O(n^2)$.

Multi-tape TMs

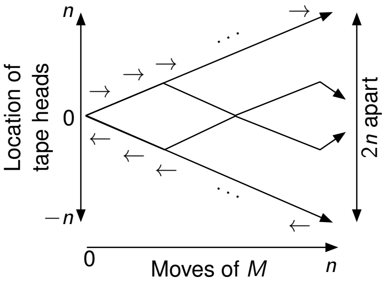
- The running time of a TM M with input w is the number of moves M makes before it halts. (If it does not, the running time is ∞).
- The time complexity $T_M : \{0, 1, \ldots\} \to \{0, 1, \ldots\} \cup \{\infty\}$ of a TM M is defined as:
  - $T_M(n) :=$ maximum running time of M for an input w of length n symbols.

Theorem 8.2.2

The time taken for M' in Theorem 8.2.1 to process n moves of k-tape M is $O(n^2)$.

Outline of Proof of Theorem 8.2.2

- In the ith move of M, any two heads of M can be at most $2i$ locations apart.



Multi-tape TMs

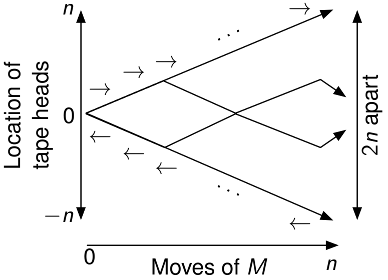
- The running time of a TM M with input w is the number of moves M makes before it halts. (If it does not, the running time is ∞).
- The time complexity $T_M : \{0, 1, \ldots\} \to \{0, 1, \ldots\} \cup \{\infty\}$ of a TM M is defined as:
  - $T_M(n) :=$ maximum running time of M for an input w of length n symbols.

Theorem 8.2.2

The time taken for M' in Theorem 8.2.1 to process n moves of k-tape M is $O(n^2)$.

Outline of Proof of Theorem 8.2.2

- In the ith move of M, any two heads of M can be at most $2i$ locations apart.
- Each sweep then requires $4i$ moves of M'.



Multi-tape TMs

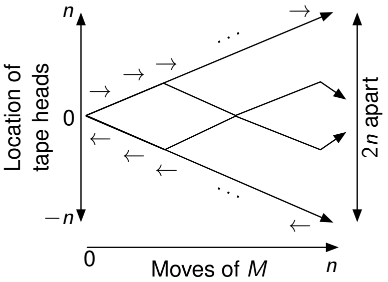
- The running time of a TM M with input w is the number of moves M makes before it halts. (If it does not, the running time is ∞).
- The time complexity $T_M : \{0, 1, \ldots\} \rightarrow \{0, 1, \ldots\} \cup \{\infty\}$ of a TM M is defined as:
  - $T_M(n) :=$ maximum running time of M for an input w of length n symbols.

Theorem 8.2.2

The time taken for M' in Theorem 8.2.1 to process n moves of k-tape M is $O(n^2)$.

Outline of Proof of Theorem 8.2.2

- In the ith move of M, any two heads of M can be at most $2i$ locations apart.
- Each sweep then requires $4i$ moves of M'.
- Each track update requires $\Theta(k)$ time.



Multi-tape TMs

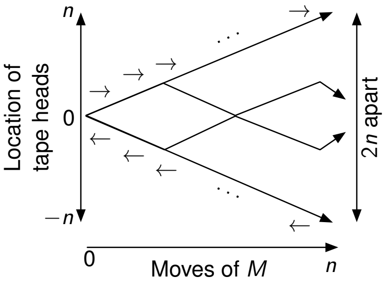
- The running time of a TM M with input w is the number of moves M makes before it halts. (If it does not, the running time is ∞).
- The time complexity $T_M : \{0, 1, \dots\} \to \{0, 1, \dots\} \cup \{\infty\}$ of a TM M is defined as:
  - $T_M(n) :=$ maximum running time of M for an input w of length n symbols.

Theorem 8.2.2

The time taken for M' in Theorem 8.2.1 to process n moves of k-tape M is $O(n^2)$.

Outline of Proof of Theorem 8.2.2

- In the ith move of M, any two heads of M can be at most $2i$ locations apart.
- Each sweep then requires $4i$ moves of M'.
- Each track update requires $\Theta(k)$ time.
- So, n moves in M need $O(n^2)$ moves in M'.



Non-deterministic TMs

Non-deterministic TM: $\delta(q, X)$ is a set of triples representing possible moves.

Non-deterministic TMs

Non-deterministic TM: $\delta(q, X)$ is a set of triples representing possible moves.

Theorem 8.2.3

For every non-deterministic TM M, there is a TM N such that $L(M) = L(N)$.

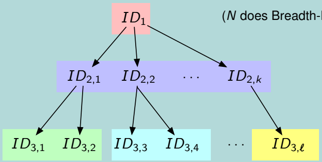
Non-deterministic TMs

Non-deterministic TM: $\delta(q, X)$ is a set of triples representing possible moves.

Theorem 8.2.3

For every non-deterministic TM M, there is a TM N such that $L(M) = L(N)$.

Outline of Proof of Theorem 8.2.3



(N does Breadth-First exploration of IDs of M)

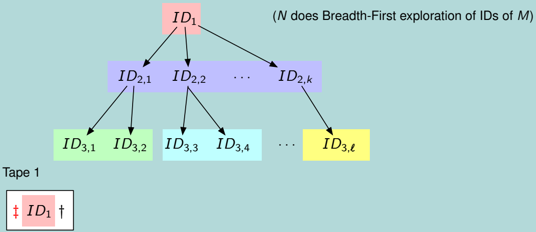
Non-deterministic TMs

Non-deterministic TM: $\delta(q, X)$ is a set of triples representing possible moves.

Theorem 8.2.3

For every non-deterministic TM M, there is a TM N such that $L(M) = L(N)$.

Outline of Proof of Theorem 8.2.3



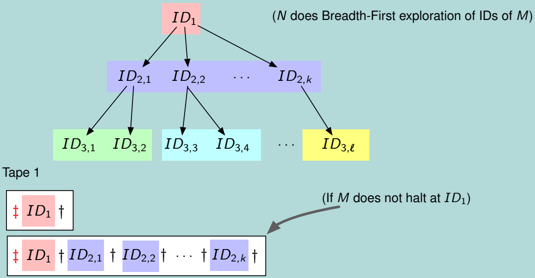
Non-deterministic TMs

Non-deterministic TM: $\delta(q, X)$ is a set of triples representing possible moves.

Theorem 8.2.3

For every non-deterministic TM M, there is a TM N such that $L(M) = L(N)$.

Outline of Proof of Theorem 8.2.3



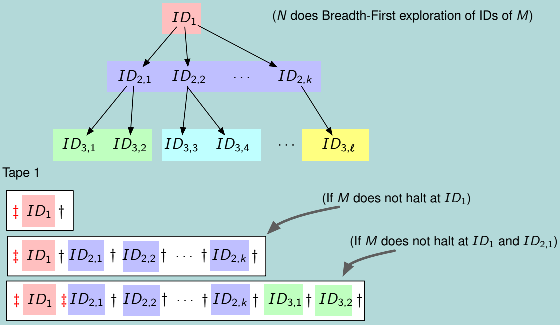
Non-deterministic TMs

Non-deterministic TM: $\delta(q, X)$ is a set of triples representing possible moves.

Theorem 8.2.3

For every non-deterministic TM M, there is a TM N such that $L(M) = L(N)$.

Outline of Proof of Theorem 8.2.3



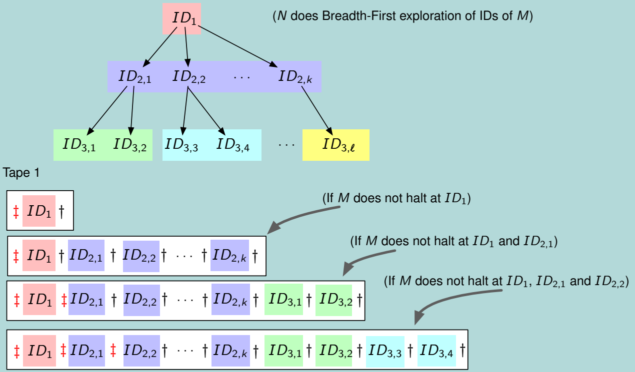
Non-deterministic TMs

Non-deterministic TM: $\delta(q, X)$ is a set of triples representing possible moves.

Theorem 8.2.3

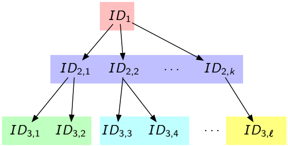
For every non-deterministic TM M, there is a TM N such that $L(M) = L(N)$.

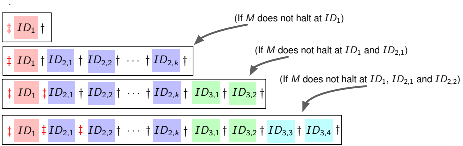
Outline of Proof of Theorem 8.2.3



Outline of Proof of Theorem 8.2.3

- We can devise a 2-tape TM N that simulates M.
- N first replaces the content of the first tape by ‡ followed by the ID that M is initially in, which is then followed by a special symbol †, which serves as ID separator. (N uses the second tape as scratch tape to enable this operation).
- If the ID corresponds to a final state, N accepts (as would M).
- If not, N then identifies all possible choices for the next IDs for M and enters each one of them followed by † at the right end of its first tape. (Again, N uses the second tape as scratch tape to enable this operation.)
- N then searches for † to the right of ‡, changes the † to a ‡ (to signify that it is processing the succeeding ID), and processes that ID in the similar way.
- N halts at an ID iff M would at that ID. (To have halting equivalence.)





Restrictions of TMs

TM Semi-infinite Tape

A TM with a semi-infinite tape is a TM that only has blanks on one of its sides, but not on the other.

Phrased (slightly) more formally:
A TM with a semi-infinite tape is a TM that can never move to left of the left-most input symbol.

We don't provide a formal definition, but a way of simulating this is by providing a special symbol, placed on the left of the input, and defining the transitions to always go to the right when this is read.

TM Semi-infinite Tape

Theorem 8.3.1

Every recursively enumerable language is also accepted by a TM with semi-infinite tape.

TM Semi-infinite Tape

Theorem 8.3.1

Every recursively enumerable language is also accepted by a TM with semi-infinite tape.

Outline of Proof of Theorem 8.3.1

TM Semi-infinite Tape

Theorem 8.3.1

Every recursively enumerable language is also accepted by a TM with semi-infinite tape.

Outline of Proof of Theorem 8.3.1

- Given a TM M that accepts a language L, construct a two-track TM M' as follows.

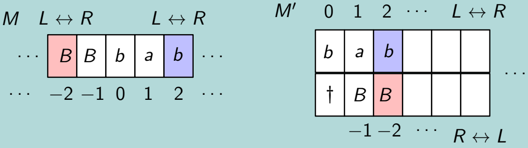
TM Semi-infinite Tape

Theorem 8.3.1

Every recursively enumerable language is also accepted by a TM with semi-infinite tape.

Outline of Proof of Theorem 8.3.1

- Given a TM M that accepts a language L, construct a two-track TM M' as follows.
- The first/second tracks of M' are the right/left parts of the tape of M.



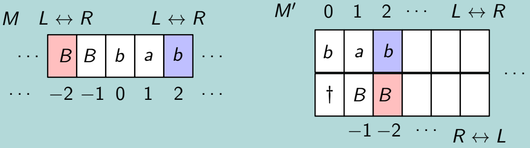
TM Semi-infinite Tape

Theorem 8.3.1

Every recursively enumerable language is also accepted by a TM with semi-infinite tape.

Outline of Proof of Theorem 8.3.1

- Given a TM M that accepts a language L, construct a two-track TM M' as follows.
- The first/second tracks of M' are the right/left parts of the tape of M.
- First, write a special symbol, say † at the leftmost part of the second track; this indicates to M' that a left move is not to be attempted at this location.



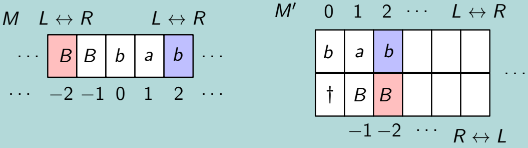
TM Semi-infinite Tape

Theorem 8.3.1

Every recursively enumerable language is also accepted by a TM with semi-infinite tape.

Outline of Proof of Theorem 8.3.1

- Given a TM M that accepts a language L, construct a two-track TM M' as follows.
- The first/second tracks of M' are the right/left parts of the tape of M.
- First, write a special symbol, say † at the leftmost part of the second track; this indicates to M' that a left move is not to be attempted at this location.
- At any time, M' keeps track of whether M is to the right or left of its start location.



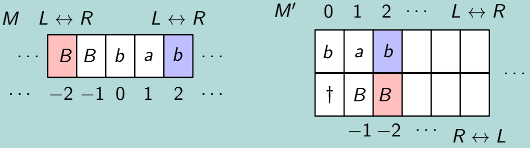
TM Semi-infinite Tape

Theorem 8.3.1

Every recursively enumerable language is also accepted by a TM with semi-infinite tape.

Outline of Proof of Theorem 8.3.1

- Given a TM M that accepts a language L, construct a two-track TM M' as follows.
- The first/second tracks of M' are the right/left parts of the tape of M.
- First, write a special symbol, say † at the leftmost part of the second track; this indicates to M' that a left move is not to be attempted at this location.
- At any time, M' keeps track of whether M is to the right or left of its start location.
  - If M is to the strict right of its start location, M' mimics M on the first track.
  - If M is to the strict left of its start location, M' mimics M on second track, but with the head directions reversed. M' detects the start by the † symbol.



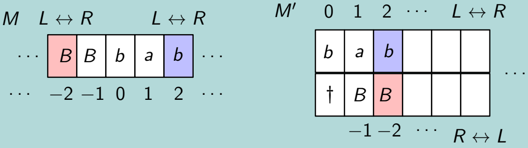
TM Semi-infinite Tape

Theorem 8.3.1

Every recursively enumerable language is also accepted by a TM with semi-infinite tape.

Outline of Proof of Theorem 8.3.1

- Given a TM M that accepts a language L, construct a two-track TM M' as follows.
- The first/second tracks of M' are the right/left parts of the tape of M.
- First, write a special symbol, say † at the leftmost part of the second track; this indicates to M' that a left move is not to be attempted at this location.
- At any time, M' keeps track of whether M is to the right or left of its start location.
  - If M is to the strict right of its start location, M' mimics M on the first track.
  - If M is to the strict left of its start location, M' mimics M on second track, but with the head directions reversed. M' detects the start by the † symbol.
- It can be formally shown that M' accepts a string iff M accepts it.



Extensions of PDAs

Multi-stack Machines

A multistack machine is a PDA with several independent stacks (i.e., one can be popping a symbol, while another is pushing a symbol).

Multi-stack Machines

A multistack machine is a PDA with several independent stacks (i.e., one can be popping a symbol, while another is pushing a symbol).

Theorem 8.4.1

Every recursively enumerable language is accepted by a two-stack PDA

Multi-stack Machines

A multistack machine is a PDA with several independent stacks (i.e., one can be popping a symbol, while another is pushing a symbol).

Theorem 8.4.1

Every recursively enumerable language is accepted by a two-stack PDA

Outline of Proof of Theorem 8.4.1

- Let each stack again contain a bottom-most start symbol.
- Let ID $= x_{-3}x_{-2}x_{-1}qx_0x_1x_2$, i.e., $w = x_{-3}x_{-2}x_{-1}x_0x_1x_2$, and head read reads x_0
  - Let stack-1 contain $x_0x_1x_2$ (with x_0 at the top), representing the head position and the symbols to its right.
  - Let stack-2 contain $x_{-1}x_{-2}x_{-3}$ (with x_{-1} at the top), representing the symbols to the left of the head in reversed order.

Multi-stack Machines

A multistack machine is a PDA with several independent stacks (i.e., one can be popping a symbol, while another is pushing a symbol).

Theorem 8.4.1

Every recursively enumerable language is accepted by a two-stack PDA

Outline of Proof of Theorem 8.4.1

- Let each stack again contain a bottom-most start symbol.
- Let ID $= x_{-3}x_{-2}x_{-1}qx_0x_1x_2$, i.e., $w = x_{-3}x_{-2}x_{-1}x_0x_1x_2$, and head read reads x_0
  - Let stack-1 contain $x_0x_1x_2$ (with x_0 at the top), representing the head position and the symbols to its right.
  - Let stack-2 contain $x_{-1}x_{-2}x_{-3}$ (with x_{-1} at the top), representing the symbols to the left of the head in reversed order.
- What if we move the head to the right? Then, ID' $= x_{-3}x_{-2}x_{-1}x_0q'x_1x_2$.
  We can easily do this with our stacks:
  - How should the stack now look like?
  - stack-1: x_1x_2 and stack-2: $x_0x_{-1}x_{-2}x_{-3}$.
  - But that's just a simple pop and push!
- Moving to the left, and changing the symbol that's written can be simulated as well.

Multi-stack Machines

Outline of Proof of Theorem 8.4.1, cont'd

- Remaining problem: How to fill the stacks initially?
- Recall: stack-1 contains what's right of the head and stack-2 what's left (but reversed).

Multi-stack Machines

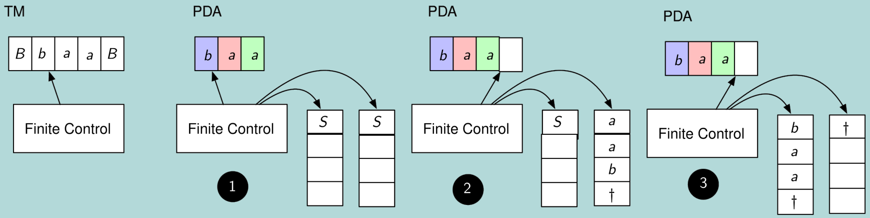
Outline of Proof of Theorem 8.4.1, cont'd

- Remaining problem: How to fill the stacks initially?
- Recall: stack-1 contains what's right of the head and stack-2 what's left (but reversed).
- Initial configuration is $q_0 w$, so stack-1 should be w and stack-2 "empty".

Multi-stack Machines

Outline of Proof of Theorem 8.4.1, cont'd

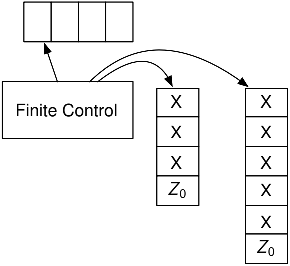
- Remaining problem: How to fill the stacks initially?
- Recall: stack-1 contains what's right of the head and stack-2 what's left (but reversed).
- Initial configuration is $q_0 w$, so stack-1 should be w and stack-2 "empty".
- We achieve this by the following procedure:



- I.e., run to the right filling stack-2, then run back putting it on stack-1.

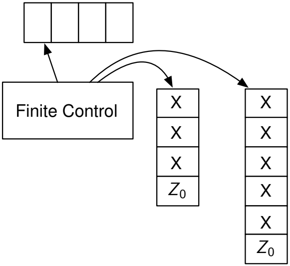
Counter Machines

- A counter machine is a multi-stack machine whose stack alphabet contains two symbols: Z_0 (stack end marker) and X
- Z_0 is initially in the stack.
- Z_0 may be replaced by $X^i Z_0$ for some $i \geq 0$
- X may be replaced by X^i for some $i \geq 0$.



Counter Machines

- A counter machine is a multi-stack machine whose stack alphabet contains two symbols: Z_0 (stack end marker) and X
- Z_0 is initially in the stack.
- Z_0 may be replaced by $X^i Z_0$ for some $i \geq 0$
- X may be replaced by X^i for some $i \geq 0$.
- A counter machine effectively stores a non-negative number.



Simulating a 2-Stack PDA with a 3-Counter Machine

Theorem 8.4.2

Every recursively enumerable language is accepted by a three-counter machine.

Key Challenge

- Challenges:
  - A 2-stack PDA uses arbitrary symbols on its stacks.
  - A counter machine can only store and manipulate numbers.
  - We must encode a stack's contents into a single number so that counter operations can simulate stack operations.
  - We must implement mathematical operations in a stack to encode push and pop operations on the "single numbers".
- How stacks work?
  - counter 1 and 2 encode stacks 1 and 3
  - counter 3 is used for additional computations

Encoding a Stack into a Counter

Encoding Method

- Assign each symbol in the stack alphabet a unique number:
  - $A = 0, B = 1, C = 2, \ldots, D = 3$, etc.
- Represent a stack as a single number using positional encoding:

$$X = Y_1 + rY_2 + r^2 Y_3 + \cdots + r^{k-1} Y_k$$

  where:
  - Y_1 is the top symbol,
  - $Y_2, Y_3, \ldots$ are symbols below it,
  - $r = |\Gamma|$, the size of the stack alphabet.

Example

- Suppose the stack contains (top to bottom): B, C, A, A, D.
- Let $r = 4$ (since the alphabet has 4 symbols).
- Encode as: $X = 1 + 4(2) + 4^2(0) + 4^3(0) + 4^4(3) = 777$.
- The counter now stores $X = 777$.

Simulating Stack Operations

Popping the Top Symbol

- Extract the top symbol using $X \mod r$:

$$777 \mod 4 = 1 \quad \Rightarrow \quad \text{Top symbol was } B$$

- Remove it by dividing by r: $X' = \lfloor X/4 \rfloor = 194$.
- The counter now stores $X' = 194$, encoding stack $C, A, A, D = 2 + 4(0) + 4^2(0) + 4^3(3) = 194$.

Pushing a Symbol

- Suppose we push symbol A onto the stack C, A, A, D.
- The current stack encoding is: $X = 194$.
- Compute the new encoding:

$$X' = 4 \cdot X + 0 = 4 \cdot 194 + 0 = 776.$$

- The counter now stores $X' = 776$, encoding stack A, C, A, A, D:

$$776 = 0 + 4(2) + 4^2(0) + 4^3(0) + 4^4(3).$$

Counter Machine Implementation

- Computing $X \mod r$ (extracting top symbol from stack 1 or 2):
  - Subtract r repeatedly from the respective stack counter (1 or 2) until value is less than r.
  - This remaining value is the top symbol.
- Computing $X' = \lfloor X/r \rfloor$ (removing top symbol from stack 1 or 2):
  - Move remainder to counter 3.
  - Subtract remainder from stack counter.
  - Divide stack counter by r by decrementing it while incrementing counter 3 in steps of r.
- Computing $X' = rX + Z$ (pushing a symbol onto a stack):
  - Copy X to counter 3.
  - Multiply counter 3 by r by adding it to itself r times.
  - Add Z to counter 3.
  - Copy the result back to the respective stack counter (1 or 2).
- Ensuring correct stack operations:
  - Stack 1 and stack 2 each use a separate counter.
  - When operating on stack 1, stack 2 stays unchanged, and vice versa.
  - Counter 3 is used only as temporary storage.

Simulating a 3-Counter Machine with 2 Counters

Theorem 8.4.3

Every recursively enumerable language is accepted by a two-counter machine.

Key Idea

- Encode three counters using prime factorization:

$$X = 2^i 3^j 5^k, \tag{1}$$

  where i, j, k are the values of the three counters.
- Updates involve:
  - Multiplying by 2, 3, or 5 to increment.
  - Dividing by 2, 3, or 5 to decrement.
  - Checking divisibility to test for zero.
- Since a second counter can store temporary results, a 2-counter machine can simulate a 3-counter machine.