Intro. Algorithms & Models of Computation

CS/ECE 374A, Fall 2024

Turing Machines

Lecture 8

Tuesday, September 17, 2024

LaTeXed: October 8, 2024 20:53

8.1

In the search for thinking machines

"Most General" computer?

1. DFAs are simple model of computation.
2. Accept only the regular languages.
3. Is there a kind of computer that can accept any language, or compute any function?
4. Recall counting argument. Set of all languages:
   $\{L \mid L \subseteq \{0,1\}^*\}$ is ~~countably infinite~~ / uncountably infinite
5. Set of all programs:
   $\{P \mid P$ is a finite length computer program$\}$:
   is countably infinite / ~~uncountably infinite~~.
6. **Conclusion:** There are languages for which there are no programs.

"Most General" computer?

1. DFAs are simple model of computation.
2. Accept only the regular languages.
3. Is there a kind of computer that can accept any language, or compute any function?
4. Recall counting argument. Set of all languages:
   $\{L \mid L \subseteq \{0,1\}^*\}$ is ~~countably infinite~~ / uncountably infinite
5. Set of all programs:
   $\{P \mid P \text{ is a finite length computer program}\}$:
   is countably infinite / ~~uncountably infinite~~.
6. **Conclusion:** There are languages for which there are no programs.

"Most General" computer?

1. DFAs are simple model of computation.
2. Accept only the regular languages.
3. Is there a kind of computer that can accept any language, or compute any function?
4. Recall counting argument. Set of all languages:
   $\{L \mid L \subseteq \{0,1\}^*\}$ is ~~countably infinite~~ / uncountably infinite
5. Set of all programs:
   $\{P \mid P \text{ is a finite length computer program}\}$:
   is countably infinite / ~~uncountably infinite~~.
6. **Conclusion:** There are languages for which there are no programs.

What can be computed?

Most General Computer:

1. If not all functions are computable, which are?
2. Is there a “most general” model of computer?
3. What languages can they recognize?

History: Formalizing mathematics

1. 19th century: Ooops. Math is a mess. Oy.
   Fix calculus, invented set theory (Cantor), etc.
2. David Hilbert (1862–1943)
   2.1 1900: The list of 23 problems.
   2.2 Early 1900s – crisis in math foundations
   attempts to formalize resulted in paradoxes, etc.
   2.3 1920: Hilbert's Program: "mechanize" mathematics.
   2.4 Finite axioms, inference rules turn crank, determine truth needed: axioms consistent & complete
   2.5 Hilbert: "No one shall expel us from the paradise that Cantor has created.".
3. Kurt Gödel (1906–1978)
   German logician, at age 25 (1931) proved: "There are true statements that can't be proved or disproved". (i.e., "no" to Hilbert)
   Shook the foundations of mathematics/philosophy/science/everything.

History: Formalizing mathematics

1. 19th century: Ooops. Math is a mess. Oy.
   Fix calculus, invented set theory (Cantor), etc.
2. David Hilbert (1862–1943)
   2.1 1900: The list of 23 problems.
   2.2 Early 1900s – crisis in math foundations
       attempts to formalize resulted in paradoxes, etc.
   2.3 1920: Hilbert's Program: "mechanize" mathematics.
   2.4 Finite axioms, inference rules turn crank, determine truth needed: axioms consistent & complete
   2.5 Hilbert: "No one shall expel us from the paradise that Cantor has created.".
3. Kurt Gödel (1906–1978)
   German logician, at age 25 (1931) proved: "There are true statements that can't be proved or disproved". (i.e., "no" to Hilbert)
   Shook the foundations of mathematics/philosophy/science/everything.

History: Formalizing mathematics

1. 19th century: Ooops. Math is a mess. Oy.
   Fix calculus, invented set theory (Cantor), etc.
2. David Hilbert (1862–1943)
   2.1 1900: The list of 23 problems.
   2.2 Early 1900s – crisis in math foundations
       attempts to formalize resulted in paradoxes, etc.
   2.3 1920: Hilbert's Program: "mechanize" mathematics.
   2.4 Finite axioms, inference rules turn crank, determine truth needed: axioms consistent & complete
   2.5 Hilbert: "No one shall expel us from the paradise that Cantor has created.".
3. Kurt Gödel (1906–1978)
   German logician, at age 25 (1931) proved: "There are true statements that can't be proved or disproved". (i.e., "no" to Hilbert)
   Shook the foundations of mathematics/philosophy/science/everything.

More history: Turing...

Alan Turing (1912–1954):

1. British mathematician
2. cryptoanalysis during WW II (enigma project)
3. Defined a computing model/program. In 1936 (age 23) provided foundations for investigating fundamental question of what is computable, what is not computable.
4. Gay, suicide.
5. Movies, UK apology.
6. Proved the halting theorem: Deciding if a computer program stops on a given input can not be decided by a program.

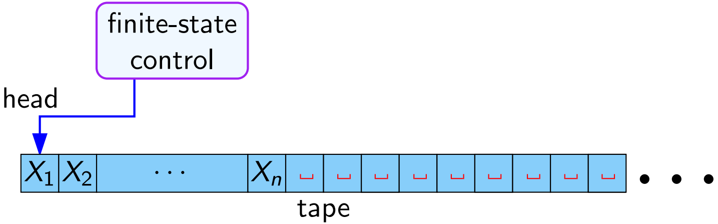
Turing original paper...

Is quite readable. Available here:

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

8.2
What is a Turing machine

Turing machine



1. Input written on (infinite) one sided tape.
2. Special blank characters.
3. Finite state control (similar to DFA).
4. Ever step: Read character under head, write character out, move the head right or left (or stay).

High level goals

1. Church-Turing thesis: TMs are the most general computing devices. So far no counter example.
2. Every TM can be represented as a string.
3. Existence of Universal Turing Machine which is the model/inspiration for stored program computing. UTM can simulate any TM
4. Implications for what can be computed and what cannot be computed

Turing machine: Formal definition

A **Turing machine** is a **7**-tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}})$$

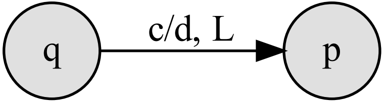
- Q: finite set of states.
- Σ: finite input alphabet.
- Γ: finite tape alphabet.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}$: Transition function.
- $q_0 \in Q$ is the initial state.
- $q_{\mathrm{acc}} \in Q$ is the **accepting**/**final** state.
- $q_{\mathrm{rej}} \in Q$ is the **rejecting** state.
- $\sqcup$ or ␣: Special blank symbol on the tape.

Turing machine: Transition function

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}$$

As such, the transition

$$\delta(q, c) = (p, d, \mathtt{L})$$



1. q: current state.
2. c: character under tape head.
3. p: new state.
4. d: character to write under tape head
5. $\mathtt{L}$: Move tape head left.

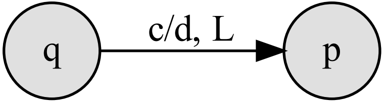
Missing transitions lead to hell state.
"Blue screen of death."
"Machine crashes."

Turing machine: Transition function

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}$$

As such, the transition

$$\delta(q, c) = (p, d, \mathtt{L})$$



1. q: current state.
2. c: character under tape head.
3. p: new state.
4. d: character to write under tape head
5. $\mathtt{L}$: Move tape head left.

Missing transitions lead to hell state.
"Blue screen of death."
"Machine crashes."

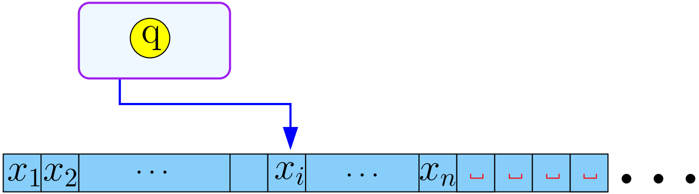
8.3
Snapshots, computation as sequence of strings

Snapshot = ID: Instantaneous Description

1. Contains all necessary information to capture “state of the computation”.
2. Includes
   2.1 state q of M
   2.2 location of read/write head
   2.3 contents of tape from left edge to rightmost non-blank (or to head, whichever is rightmost).

Snapshot = ID: Instantaneous Description

As a string



ID: $x_1 x_2 \ldots x_{i-1} q x_i x_{i+1} \ldots x_n$

$x_1, \ldots, x_n \in \Gamma$, $q \in Q$.

A step in computation as rewriting strings

$x_1 x_2 \ldots x_{i-1} q x_i x_{i+1} \ldots x_n$
If transition is $\delta(q, X_i) = (p, Y, \mathtt{L})$, new ID is:

$$\begin{array}{rcl} \text{current ID :} & & x_1 x_2 \ldots x_{i-2} x_{i-1} q x_i x_{i+1} \ldots x_n \\ \delta(q, X_i) = (p, y, \mathtt{L}) & \Longrightarrow & x_1 x_2 \ldots x_{i-2} p x_{i-1} y x_{i+1} \ldots x_n \end{array}$$

If no transition defined, or illegal transition, then no next ID (crash).
Shockingly: Computation is just a string rewriting system.

A step in computation as rewriting strings

$x_1 x_2 \ldots x_{i-1} q x_i x_{i+1} \ldots x_n$
If transition is $\delta(q, X_i) = (p, Y, \mathtt{L})$, new ID is:

$$\begin{array}{rcc} \text{current ID :} & & x_1 x_2 \ldots x_{i-2} x_{i-1} q x_i x_{i+1} \ldots x_n \\ \delta(q, X_i) = (p, y, \mathtt{L}) & \Longrightarrow & x_1 x_2 \ldots x_{i-2} p x_{i-1} y x_{i+1} \ldots x_n \end{array}$$

If no transition defined, or illegal transition, then no next ID (crash).
Shockingly: Computation is just a string rewriting system.

A step in computation as rewriting strings

1. Initial ID: $q_0 w$:
2. Accepting ID: $\alpha q_{\mathrm{acc}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
3. Rejecting ID: $\alpha q_{\mathrm{rej}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
4. $\mathcal{I} \rightsquigarrow \mathcal{J}$:Denotes that if we start execution of TM with configuration/ID encoded by $\mathcal{I}$, leads TM (after maybe several steps) to ID $\mathcal{J}$
5. M accepts w: If for some $\alpha, \alpha' \in \Gamma^*$, we have

$$q_0 w \rightsquigarrow \alpha q_{\mathrm{acc}} \alpha'.$$

   Acceptance happens as soon as TM enters accept state.
6. Language of TM M: $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

A step in computation as rewriting strings

1. Initial ID: $q_0 w$:
2. Accepting ID: $\alpha q_{\mathrm{acc}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
3. Rejecting ID: $\alpha q_{\mathrm{rej}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
4. $\mathcal{I} \rightsquigarrow \mathcal{J}$:Denotes that if we start execution of TM with configuration/ID encoded by $\mathcal{I}$, leads TM (after maybe several steps) to ID $\mathcal{J}$
5. M **accepts** w: If for some $\alpha, \alpha' \in \Gamma^*$, we have

$$q_0 w \rightsquigarrow \alpha q_{\mathrm{acc}} \alpha'.$$

   Acceptance happens as soon as TM enters accept state.
6. Language of TM M: $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

A step in computation as rewriting strings

1. Initial ID: $q_0 w$:
2. Accepting ID: $\alpha q_{\mathrm{acc}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
3. Rejecting ID: $\alpha q_{\mathrm{rej}} \alpha'$, for some $\alpha, \alpha' \in \Gamma^*$.
4. $\mathcal{I} \rightsquigarrow \mathcal{J}$:Denotes that if we start execution of TM with configuration/ID encoded by $\mathcal{I}$, leads TM (after maybe several steps) to ID $\mathcal{J}$
5. M **accepts** w: If for some $\alpha, \alpha' \in \Gamma^*$, we have

   $$q_0 w \rightsquigarrow \alpha q_{\mathrm{acc}} \alpha'.$$

   Acceptance happens as soon as TM enters accept state.
6. Language of TM M: $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

Non-accepting computation

M **does not accept** w if:

1. M enters q_{rej} (i.e., M **rejects** w)
2. M crashes (moves to left of tape, no transition available, etc).
3. M runs forever.

If the TM keeps running, should we wait, or is it rejection?

Non-accepting computation

M **does not accept** w if:

1. M enters q_{rej} (i.e., M **rejects** w)
2. M crashes (moves to left of tape, no transition available, etc).
3. M runs forever.

If the TM keeps running, should we wait, or is it rejection?

Everything is a number

8.4
Languages defined by a Turing machine

Recursive vs. Recursively Enumerable

1. **Recursively enumerable** (aka **RE**) languages

$$L = \{L(M) \mid M \text{ some Turing machine}\}\,.$$

2. **Recursive** / **decidable** languages

$$L = \{L(M) \mid M \text{ some Turing machine that halts on all inputs}\}\,.$$

3. Fundamental questions:
   3.1 What languages are RE?
   3.2 Which are recursive?
   3.3 What is the difference?
   3.4 What makes a language decidable?
   3.5 How much wood would a TM chuck, if a TM could chuck wood?

Recursive vs. Recursively Enumerable

1. **Recursively enumerable** (aka **RE**) languages (bad)

$$L = \{L(M) \mid M \text{ some Turing machine}\}.$$

2. **Recursive** / **decidable** languages (good)

$$L = \{L(M) \mid M \text{ some Turing machine that halts on all inputs}\}.$$

3. Fundamental questions:
   3.1 What languages are RE?
   3.2 Which are recursive?
   3.3 What is the difference?
   3.4 What makes a language decidable?
   3.5 How much wood would a TM chuck, if a TM could chuck wood?

Recursive vs. Recursively Enumerable

1. **Recursively enumerable** (aka **RE**) languages (bad)

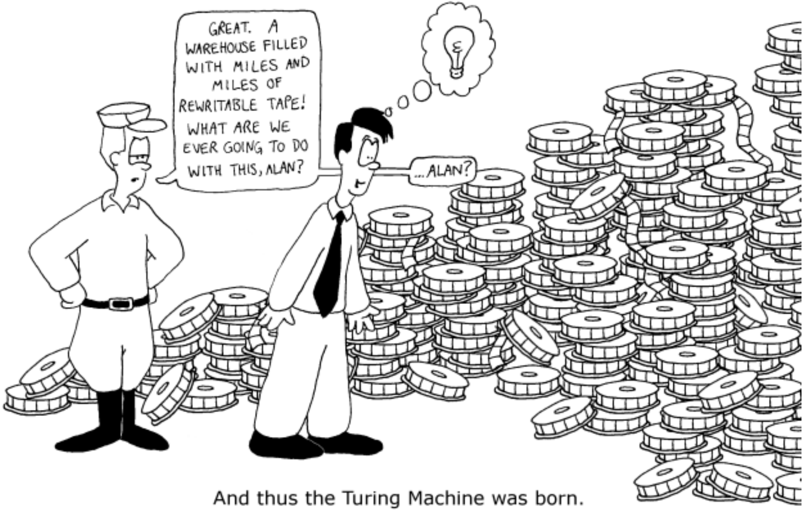
$$L = \{L(M) \mid M \text{ some Turing machine}\}.$$

2. **Recursive** / **decidable** languages (good)

$$L = \{L(M) \mid M \text{ some Turing machine that halts on all inputs}\}.$$

3. Fundamental questions:
   3.1 What languages are RE?
   3.2 Which are recursive?
   3.3 What is the difference?
   3.4 What makes a language decidable?
   3.5 How much wood would a TM chuck, if a TM could chuck wood?

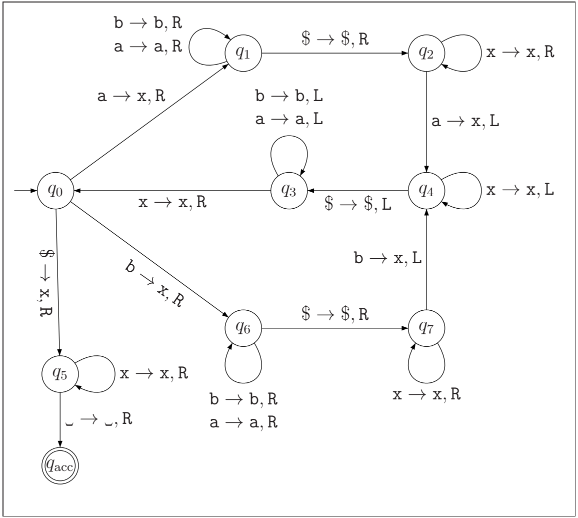
How was the Turing Machine invented...



8.5
Some examples of Turing machines

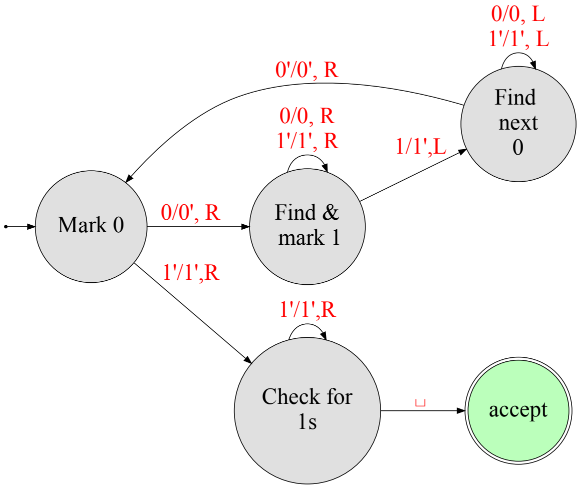
8.5.1
Turing machine for $w\$w$

Example: Turing machine for **w$w**



8.5.2
Turing machine for $\mathbf{0}^n\mathbf{1}^n$

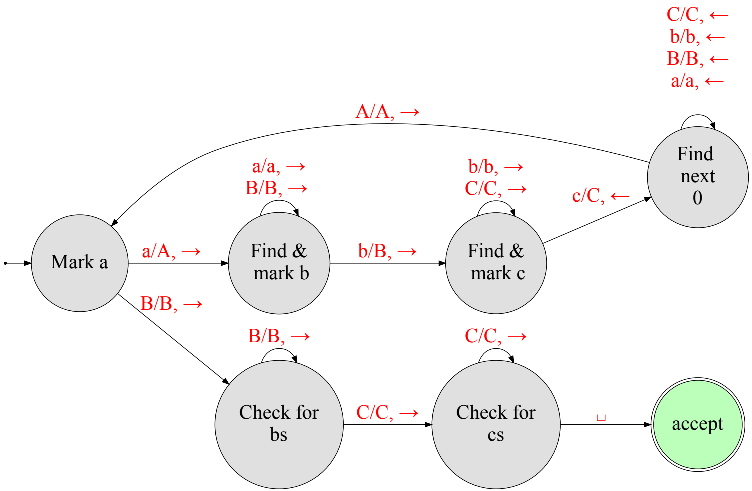
Example: Turing machine for $\mathbf{0^n 1^n}$



8.5.3
Turing machine for $a^n b^n c^n$

Example: Turing machine for $\mathbf{a^n b^n c^n}$

A language that is not context free...



8.6
Why Turing Machine is a “real” computer?

Why Turing Machine is a “real” computer?

TM can compute anything that a real computer can, if very very very tediously.

1. Add/multiply two numbers in binary representation.
2. Move input tape one position to the right.
3. Simulate a TM with two tapes.
4. Simulate a TM with many tapes.
5. Stack.
6. Subroutines.
7. Compile say any C program into a TM.
8. Conclusion: TM can do what a regular program can do.
9. Turing brilliant observation: A TM can simulate/modify another TM.
10. Modern equivalent: An interpreter can run a program that might be the interpreter itself (you don't say).

Why Turing Machine is a "real" computer?

TM can compute anything that a real computer can, if very very very tediously.

1. Add/multiply two numbers in binary representation.
2. Move input tape one position to the right.
3. Simulate a TM with two tapes.
4. Simulate a TM with many tapes.
5. Stack.
6. Subroutines.
7. Compile say any C program into a TM.
8. Conclusion: TM can do what a regular program can do.
9. Turing brilliant observation: A TM can simulate/modify another TM.
10. Modern equivalent: An interpreter can run a program that might be the interpreter itself (you don't say).

Why Turing Machine is a “real” computer?

TM can compute anything that a real computer can, if very very very tediously.

1. Add/multiply two numbers in binary representation.
2. Move input tape one position to the right.
3. Simulate a TM with two tapes.
4. Simulate a TM with many tapes.
5. Stack.
6. Subroutines.
7. Compile say any C program into a TM.
8. Conclusion: TM can do what a regular program can do.
9. Turing brilliant observation: A TM can simulate/modify another TM.
10. Modern equivalent: An interpreter can run a program that might be the interpreter itself (you don't say).

Why Turing Machine is a "real" computer?

TM can compute anything that a real computer can, if very very very tediously.

1. Add/multiply two numbers in binary representation.
2. Move input tape one position to the right.
3. Simulate a TM with two tapes.
4. Simulate a TM with many tapes.
5. Stack.
6. Subroutines.
7. Compile say any C program into a TM.
8. Conclusion: TM can do what a regular program can do.
9. Turing brilliant observation: A TM can simulate/modify another TM.
10. Modern equivalent: An interpreter can run a program that might be the interpreter itself (you don't say).

Why Turing Machine is a “real” computer?

TM can compute anything that a real computer can, if very very very tediously.

1. Add/multiply two numbers in binary representation.
2. Move input tape one position to the right.
3. Simulate a TM with two tapes.
4. Simulate a TM with many tapes.
5. Stack.
6. Subroutines.
7. Compile say any C program into a TM.
8. Conclusion: TM can do what a regular program can do.
9. Turing brilliant observation: A TM can simulate/modify another TM.
10. Modern equivalent: An interpreter can run a program that might be the interpreter itself (you don’t say).

So what Turing Machines are good for?

1. Simplest mathematical way to describe a computer/program.
2. A good sandbox to argue about what programs can and can not do.
3. A terrible counter-intuitive model, completely unlike real world programs.
4. TM = PROGRAM.

So what Turing Machines are good for?

1. Simplest mathematical way to describe a computer/program.
2. A good sandbox to argue about what programs can and can not do.
3. A terrible counter-intuitive model, completely unlike real world programs.
4. TM = PROGRAM.

So what Turing Machines are good for?

1. Simplest mathematical way to describe a computer/program.
2. A good sandbox to argue about what programs can and can not do.
3. A terrible counter-intuitive model, completely unlike real world programs.
4. TM = PROGRAM.

Universal Turing Machine

Turing Machine that simulates another Turing Machine

UTM: A Turing machine that can simulate another Turing machine.

1. Programs can self replicate.
2. Program can modify themselves (a big no no nowadays).
3. Program can rewrite a program.
4. Turing had created a Pandora box...
   ...which we will open in the next lecture.