Computational Models – Lecture 7

- Turing Machine – More Definition and Examples
- Notion of an Algorithm
- Hilbert’s Tenth Problem
- Decidability of DFAs and PDAs Questions
Non-Deterministic Turing Machines
(reminder)

Transition function:

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$
Non-Deterministic Turing Machines

(reminder)

Transition function:

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Computation is a tree.
Non-Deterministic Turing Machines (reminder)

Transition function:

\[ \delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}) \]

- Computation is a tree.
- Accepts if there is (\(\exists\)) an accepting branch.
Equivalence

**Theorem:** A language is enumerable if and only if there is some non-deterministic Turing machine that accepts it.
Equivalence

**Theorem:** A language is enumerable if and only if there is some non-deterministic Turing machine that accepts it.

One direction is trivial.
Equivalence

**Theorem:** A language is enumerable if and only if there is some non-deterministic Turing machine that accepts it.

One direction is trivial.

To prove the other direction, we will show how to convert a non-deterministic TM, $N$, into an equivalent deterministic TM, $D$. 

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
Equivalence

**Theorem:** A language is enumerable if and only if there is some non-deterministic Turing machine that accepts it.

One direction is trivial.

To prove the other direction, we will show how to convert a non-deterministic TM, $N$, into an equivalent deterministic TM, $D$.

$D$ will simulate $N$. 
Simulating Non-Determinism

Basic idea:

- $D$ tries all possible branches
Simulating Non-Determinism

Basic idea:

- $D$ tries all possible branches
- If $D$ finds any accepting branch, it accepts.
Simulating Non-Determinism

Basic idea:

- $D$ tries all possible branches
- If $D$ finds any accepting branch, it accepts.
- If all branches reject, $D$ rejects.
Simulating Non-Determinism

Basic idea:
- $D$ tries all possible branches
- If $D$ finds any accepting branch, it accepts.
- If all branches reject, $D$ rejects.
- If all branches reject or loop, $D$ loops.
Simulating Non-Determinism

Basic idea:

- $D$ tries all possible branches.
- If $D$ finds any **accepting** branch, it **accepts**.
- If all branches **reject**, $D$ **rejects**.
- If all branches **reject or loop**, $D$ **loops**.
- Of course, $D$ does not “**know**” this (loop) is the case. It just “follows” $N$. 

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
Simulating Non-Determinism (2)

\( D \) has three tapes
Simulating Non-Determinism (2)

\[ D \] has three tapes

- the input tape is never altered
Simulating Non-Determinism (2)

\(D\) has three tapes

- the input tape is never altered
- the simulation tape is a copy of \(N\)’s tape
Simulating Non-Determinism (2)

\(D\) has three tapes

- the input tape is never altered
- the simulation tape is a copy of \(N\)’s tape
- the address tape keeps track of \(D\)’s location in \(N\)’s computation tree.
Deciders

**Definition:** A non-deterministic TM is a decider if on all inputs, all branches halt (in either state $q_a$ or $q_r$).
Deciders

**Definition:** A non-deterministic TM is a decider if on all inputs, all branches halt (in either state $q_a$ or $q_r$).

**Definition (reminder):** A language is decidable if some deterministic Turing machine decides it.
Deciders

**Definition:** A non-deterministic TM is a **decider** if on all inputs, all branches halt (in either state $q_a$ or $q_r$).

**Definition (reminder):** A language is **decidable** if some deterministic Turing machine decides it.

**Theorem:** A language is **decidable** if and only if there is a non-deterministic Turing machine that decides it.
Enumerators

A language is **enumerable** if it is accepted by some Turing machine.

But why **enumerable**?

**Definition:** An enumerator is a TM with a printer.

- TM sends strings to printer
Enumerators

A language is **enumerable** if it is accepted by some Turing machine. But why *enumerable*?

**Definition:** An enumerator is a TM with a printer.

- TM sends strings to printer
- may create infinite list of strings
Enumerators

A language is **enumerable** if it is accepted by some Turing machine. But why **enumerable**?

**Definition:** An enumerator is a TM with a printer.

- TM sends strings to printer
- may create infinite list of strings
- TM **enumerates** a language – all strings produced.
Theorem

**Theorem:** A language is **accepted** by some Turing machine if and only if some enumerator **enumerates** it.
Theorem

**Theorem:** A language is accepted by some Turing machine if and only if some enumerator enumerates it.

Will show

- If $E$ enumerates language $A$, then some TM $M$ accepts $A$.
- If $M$ accepts $A$, then some enumerator $E$ enumerates it.
Theorem

**Claim:** If $E$ enumerates language $A$, then some TM $M$ accepts $A$.

On input $w$, TM $M$
Theorem

Claim: If $E$ enumerates language $A$, then some TM $M$ accepts $A$.

On input $w$, TM $M$

- Runs $E$. Every time $E$ outputs a string $v$, $M$ compares it to $w$. 
Theorem

Claim: If $E$ enumerates language $A$, then some TM $M$ accepts $A$.

On input $w$, TM $M$

- Runs $E$. Every time $E$ outputs a string $v$, $M$ compares it to $w$.
- If $v = w$, $M$ accept.
Theorem

**Claim:** If \( E \) enumerates language \( A \), then some TM \( M \) accepts \( A \).

On input \( w \), TM \( M \)

- Runs \( E \). Every time \( E \) outputs a string \( v \), \( M \) compares it to \( w \).
- If \( v = w \), \( M \) accept.
- If \( v \neq w \), \( M \) continues running \( E \).
Theorem

Claim: If $M$ accepts $A$, then some enumerator $E$ enumerates it.

Let $s_1, s_2, s_3, \ldots$ is a list of all strings in $\Sigma^*$ (e.g. strings in lexicographic order).

The enumerator, $E$

repeat the following for $i = 1, 2, 3, \ldots$
Theorem

**Claim:** If $M$ accepts $A$, then some enumerator $E$ enumerates it.

Let $s_1, s_2, s_3, \ldots$ is a list of all strings in $\Sigma^*$ (e.g. strings in lexicographic order). The enumerator, $E$

- repeat the following for $i = 1, 2, 3, \ldots$
- run $M$ for $i$ steps on each input $s_1, s_2, \ldots, s_i$. 
Theorem

Claim: If $M$ accepts $A$, then some enumerator $E$ enumerates it.

Let $s_1, s_2, s_3, \ldots$ is a list of all strings in $\Sigma^*$ (e.g. strings in lexicographic order).

The enumerator, $E$
- repeat the following for $i = 1, 2, 3, \ldots$
- run $M$ for $i$ steps on each input $s_1, s_2, \ldots, s_i$.
- if any computation accepts, print out the corresponding $s$. 

♣ Note that with this procedure, each output is duplicated infinitely often. Think how can this duplication be avoided?
Theorem

Claim: If $M$ accepts $A$, then some enumerator $E$ enumerates it.

Let $s_1, s_2, s_3, \ldots$ is a list of all strings in $\Sigma^*$ (e.g. strings in lexicographic order).

The enumerator, $E$

- repeat the following for $i = 1, 2, 3, \ldots$
- run $M$ for $i$ steps on each input $s_1, s_2, \ldots, s_i$.
- if any computation accepts, print out the corresponding $s$.

Note that with this procedure, each output is duplicated infinitely often.
Theorem

**Claim:** If $M$ accepts $A$, then some enumerator $E$ enumerates it.

Let $s_1, s_2, s_3, \ldots$ is a list of all strings in $\Sigma^*$ (e.g. strings in lexicographic order).

The enumerator, $E$

- repeat the following for $i = 1, 2, 3, \ldots$
- run $M$ for $i$ steps on each input $s_1, s_2, \ldots, s_i$.
- if any computation accepts, print out the corresponding $s$.

Note that with this procedure, each output is duplicated infinitely often.

Think how can this duplication be avoided?
Decidability vs. Enumerability

Decidability is a stronger notion than enumerability.
Decidability vs. Enumerability

- Decidability is a stronger notion than enumerability.

- If a language $L$ is decidable then clearly it is enumerable (the other direction does not hold, as we’ll show in a couple of lectures).
Decidability vs. Enumerability

- Decidability is a stronger notion than enumerability.
- If a language $L$ is decidable then clearly it is enumerable (the other direction does not hold, as we’ll show in a couple of lectures).
- It is also clear that if $L$ is decidable then so is $\overline{L}$, and thus $\overline{L}$ is also enumerable.
Decidability vs. Enumerability

- Decidability is a **stronger notion** than enumerability.

- If a language $L$ is decidable then clearly it is enumerable (the other direction does **not** hold, as we’ll show in a couple of lectures).

- It is also clear that if $L$ is decidable then so is $\overline{L}$, and thus $\overline{L}$ is also enumerable.

- Let $\mathcal{RE}$ denote the class of enumerable languages, and let $\text{co}\mathcal{RE}$ denote the class of languages whose complement is enumerable.
Decidability vs. Enumerability

- Decidability is a **stronger notion** than enumerability.

- If a language $L$ is **decidable** then clearly it is **enumerable** (the other direction does not hold, as we’ll show in a couple of lectures).

- It is also clear that if $L$ is **decidable** then so is $\overline{L}$, and thus $\overline{L}$ is also **enumerable**.

- Let $\mathcal{RE}$ denote the class of enumerable languages, and let $\mathcal{coRE}$ denote the class of languages whose complement is enumerable.

- Let $\mathcal{R}$ denote the class of decidable languages. Then what we just saw is $\mathcal{R} \subseteq \mathcal{RE} \cap \mathcal{coRE}$. 
Decidability vs. Enumerability (2)

**Theorem:** \( R = \text{RE} \cap \text{coRE} \).

**Proof:** We should prove the \( \supseteq \) direction. Namely if \( L \in \text{RE} \cap \text{coRE} \), then \( L \in R \).

In other words, if both \( L \) and its complement are enumerable, then \( L \) is decidable.
Decidability vs. Enumerability (2)

**Theorem:** $\mathcal{R} = \mathcal{RE} \cap \text{coRE}$. 

**Proof:** We should prove the $\supseteq$ direction. Namely if $L \in \mathcal{RE} \cap \text{coRE}$, then $L \in \mathcal{R}$.

In other words, if both $L$ and its complement are enumerable, then $L$ is decidable.
Decidability vs. Enumerability (2)

**Theorem:** \( \mathcal{R} = \mathcal{RE} \cap \text{co}\mathcal{RE} \).

**Proof:** We should prove the \( \supseteq \) direction. Namely if \( L \in \mathcal{RE} \cap \text{co}\mathcal{RE} \), then \( L \in \mathcal{R} \).

In other words, if both \( L \) and its complement are enumerable, then \( L \) is decidable.

Let \( M_1 \) be a TM that accepts \( L \).

Let \( M_2 \) be a TM that accepts \( \overline{L} \).

We describe a TM, \( M \), that decides \( L \).
Decidability vs. Enumerability (2)

**Theorem:** \( R = \mathcal{RE} \cap \text{co}\mathcal{RE} \).

**Proof:** We should prove the \( \supseteq \) direction. Namely if \( L \in \mathcal{RE} \cap \text{co}\mathcal{RE} \), then \( L \in R \).

In other words, if both \( L \) and its complement are enumerable, then \( L \) is decidable.

Let \( M_1 \) be a TM that accepts \( L \).

Let \( M_2 \) be a TM that accepts \( \overline{L} \).

We describe a TM, \( M \), that decides \( L \).

On input \( x \), \( M \) runs \( M_1 \) and \( M_2 \) in parallel.

If \( M_1 \) accepts, \( M \) accepts.

If \( M_2 \) accepts, \( M \) rejects.
Decidability vs. Enumerability (2)

**Theorem:** $\mathcal{R} = \mathcal{RE} \cap \text{coRE}.$

**Proof:** We should prove the $\supseteq$ direction. Namely if $L \in \mathcal{RE} \cap \text{coRE}$, then $L \in \mathcal{R}$.

In other words, if both $L$ and its complement are enumerable, then $L$ is decidable.

Let $M_1$ be a TM that accepts $L$.

Let $M_2$ be a TM that accepts $\overline{L}$.

We describe a TM, $M$, that decides $L$.

On input $x$, $M$ runs $M_1$ and $M_2$ in parallel.

If $M_1$ accepts, $M$ accepts.

If $M_2$ accepts, $M$ rejects.

Should now show that indeed $M$ decides $L$.

Not too hard... ♣
Some Perspective

- Many models have been proposed for general-purpose computation.
Some Perspective

- Many models have been proposed for general-purpose computation.
- Remarkably, all “reasonable” models are equivalent to Turing machines.
Some Perspective

- Many models have been proposed for general-purpose computation.
- Remarkably, all “reasonable” models are equivalent to Turing machines.
- All “reasonable” programming languages (e.g. Java, Pascal, C, Scheme, Mathematica, Maple, Cobol, . . . ) are equivalent.
Some Perspective

- Many models have been proposed for general-purpose computation.
- Remarkably, all “reasonable” models are equivalent to Turing machines.
- All “reasonable” programming languages (e.g. Java, Pascal, C, Scheme, Mathematica, Maple, Cobol, . . . ) are equivalent.
- The notion of an algorithm is model-independent!
Some Perspective

- Many models have been proposed for general-purpose computation.
- Remarkably, all “reasonable” models are equivalent to Turing machines.
- All “reasonable” programming languages (e.g. Java, Pascal, C, Scheme, Mathematica, Maple, Cobol, . . .) are equivalent.
- The notion of an algorithm is model-independent!
- We don’t really care about Turing machines per se, we care about understanding computation.
What is an Algorithm?

Informally
What is an Algorithm?

- Informally
  - a recipe

Historically, the notion has a long history in Mathematics (starting with Euclid’s gcd algorithm), but not precisely defined until the 20th century. Informal notions rarely questioned, still, they were insufficient.
What is an Algorithm?

Informally

- a recipe
- a procedure
What is an Algorithm?

- Informally
- a recipe
- a procedure
- a computer program
What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it

Historically, notion has long history in Mathematics (starting with Euclid's gcd algorithm), but not precisely defined until 20th century. Informal notions rarely questioned, still, they were insufficient.
What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it

- Historically,
What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it

- Historically,
  - notion has long history in Mathematics (starting with Euclid’s gcd algorithm), but
What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it

- Historically,
  - notion has long history in Mathematics (starting with Euclid’s gcd algorithm), but
  - not precisely defined until 20th century
What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it

- Historically,
  - notion has long history in Mathematics (starting with Euclid’s gcd algorithm), but
  - not precisely defined until 20th century
  - informal notions rarely questioned,
What is an Algorithm?

- Informally
  - a recipe
  - a procedure
  - a computer program
  - who cares? I know it when I see it 🤔

- Historically,
  - notion has long history in Mathematics (starting with Euclid’s gcd algorithm), but
  - not precisely defined until 20th century
  - informal notions rarely questioned,
  - still, they were insufficient
Dilbert’s Problems
Dilbert’s Problems

Too much beer last night? These are D. Hilbert’s problems, not Dilbert’s problems, we are supposed to talk about...
Hilbert’s 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented 23 central mathematical problems
Hilbert’s 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented **23 central mathematical problems**
- challenge for the next (20th) century
Hilbert’s 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented **23 central mathematical problems**
- challenge for the next (20th) century
- the **10th problem** directly concerned algorithms
Hilbert’s 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented **23 central mathematical problems**
- challenge for the next (20th) century
- the 10th problem directly concerned algorithms

- In **November 2003**, significant progress made on the 6th problem.
Hilbert’s 10th Problem

In 1900, David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris, France.

- Presented 23 central mathematical problems
- Challenge for the next (20th) century
- The 10th problem directly concerned algorithms

In November 2003, significant progress made on the 6th problem.

But it is the 10th problem we care about. Will start with some background.
Polynomials

A term is a product of variables and a constant coefficient, e.g. $6x^3yz^2$. 
Polynomials

- A term is a product of variables and a constant coefficient, e.g. $6x^3yz^2$.
- A polynomial is a sum of terms, e.g. $6x^3yz^2 + 3xy^2 - x^3 - 10$. 
Polynomials

- A term is a product of variables and a constant coefficient, e.g. $6x^3yz^2$.
- A polynomial is a sum of terms, e.g. $6x^3yz^2 + 3xy^2 - x^3 - 10$.
- A root of a polynomial is an assignment of values to variables so that the polynomial equals zero.
Polynomials

- A term is a product of variables and a constant coefficient, e.g. $6x^3yz^2$.
- A polynomial is a sum of terms, e.g. $6x^3yz^2 + 3xy^2 - x^3 - 10$.
- A root of a polynomial is an assignment of values to variables so that the polynomial equals zero.
- For example, $x = 5$, $y = 3$, and $z = 0$ is a root of the polynomial above.
Polynomials

- A **term** is a product of **variables** and a constant **coefficient**, *e.g.* $6x^3yz^2$.

- A **polynomial** is a sum of terms, *e.g.*
  $$6x^3yz^2 + 3xy^2 - x^3 - 10.$$

- A **root** of a polynomial is an assignment of values to variables so that the polynomial equals **zero**.

- For example, $x = 5$, $y = 3$, and $z = 0$ is a root of the polynomial above.

- Here, we are interested in **integral** roots, namely an assignment of **integers** to all variables.
Polynomials

- A **term** is a product of **variables** and a constant coefficient, *e.g.* $6x^3yz^2$.

- A **polynomial** is a sum of terms, *e.g.*
  \[6x^3yz^2 + 3xy^2 - x^3 - 10\].

- A **root** of a polynomial is an assignment of values to variables so that the polynomial equals **zero**.

  For example, $x = 5$, $y = 3$, and $z = 0$ is a root of the polynomial above.

- Here, we are interested in **integral** roots, namely an assignment of **integers** to all variables.

- Some polynomials have integral roots, some don’t (*e.g.* $x^2 - 2$).
Hilbert’s Tenth Problem

The Problem: Devise an algorithm that tests whether a polynomial has an integral root.

Actually, what he said (translated from German) was “to devise a process according to which it can be determined by a finite number of operations”.

Hilbert’s Tenth Problem

The Problem: Devise an algorithm that tests whether a polynomial has an integral root.

Actually, what he said (translated from German) was “to devise a process according to which it can be determined by a finite number of operations”.

Note that

- Hilbert explicitly asks that algorithm be “devised”
- apparently Hilbert assumes that such an algorithm must exist, and someone “only” need find it.
Hilbert’s Tenth Problem

We now know no algorithm exists for this task.
Hilbert’s Tenth Problem

- We now know no algorithm exists for this task.
- Mathematicians of 1900 could not have proved this, because they didn’t have a formal notion of an algorithm.
Hilbert’s Tenth Problem

- We now know no algorithm exists for this task.
- Mathematicians of 1900 could not have proved this, because they didn’t have a formal notion of an algorithm.
- Intuitive notions work fine for constructing algorithms (we know one when we see it).
Hilbert’s Tenth Problem

- We now know no algorithm exists for this task.
- Mathematicians of 1900 could not have proved this, because they didn’t have a formal notion of an algorithm.
- Intuitive notions work fine for constructing algorithms (we know one when we see it).
- Formal notions are required to show that no algorithm exists.
Church-Turing Thesis

Formal notions appeared in 1936:
- \( \lambda \)-calculus of Alonzo Church
Church-Turing Thesis

Formal notions appeared in 1936:

- $\lambda$-calculus of Alonzo Church
- Turing machines of Alan Turing
Church-Turing Thesis

Formal notions appeared in 1936:
- $\lambda$-calculus of Alonzo Church
- Turing machines of Alan Turing
- Recursive functions of Stephen Kleene
Church-Turing Thesis

Formal notions appeared in 1936:

- $\lambda$-calculus of Alonzo Church
- Turing machines of Alan Turing
- Recursive functions of Stephen Kleene
- Correspondence systems of Emil Post
Church-Turing Thesis

Formal notions appeared in 1936:
- $\lambda$-calculus of Alonzo Church
- Turing machines of Alan Turing
- Recursive functions of Stephen Kleene
- Correspondence systems of Emil Post

These definitions look very different and have very different characteristics, yet they are provably equivalent.
Church-Turing Thesis

Formal notions appeared in 1936:

- $\lambda$-calculus of Alonzo Church
- Turing machines of Alan Turing
- Recursive functions of Stephen Kleene
- Correspondence systems of Emil Post

These definitions look very different and have very different characteristics, yet they are provably equivalent.

Church-Turing Thesis:

“The intuitive notion of algorithms equals Turing machine algorithms”.
Hilbert’s Tenth Problem

In 1970, 23 years old Yuri Matijasevič, building on work of Martin Davis, Hilary Putnam, and Julia Robinson, proved that no algorithm exists for testing whether a polynomial has integral roots (a survey of the proof)
Reformulating Hilbert’s Tenth Problem

Consider the language:

\[ D = \{ p \mid p \text{ is a polynomial with an integral root} \} \]
Reformulating Hilbert’s Tenth Problem

Consider the language:

\[ D = \{ p \mid p \text{ is a polynomial with an integral root} \} \]

Hilbert’s tenth problem asks whether this language is decidable.
Reformulating Hilbert’s Tenth Problem

Consider the language:

\[ D = \{ p \mid p \text{ is a polynomial with an integral root} \} \]

Hilbert’s tenth problem asks whether this language is **decidable**.

We now know it is **not decidable**, but it is **enumerable**!
Univariate Polynomials

Consider the \textit{simpler} language:

$$D_1 = \{ p \mid p \text{ is a polynomial over } x \text{ with an integral root} \}$$
Univariate Polynomials

Consider the simpler language:

$$D_1 = \{ p \mid p \text{ is a polynomial over } x \text{ with an integral root} \}$$

Here is a Turing machine that accepts $D_1$. 
Univariate Polynomials

Consider the simpler language:

\[ D_1 = \{ p \mid p \text{ is a polynomial over } x \text{ with an integral root} \} \]

Here is a Turing machine that accepts \( D_1 \). On input \( p \),

- evaluate \( p \) with \( x \) set successively to \( 0, 1, -1, 2, -2, \ldots \).
- if \( p \) evaluates to zero, accept.
Univariate Polynomials (2)

\[ D_1 = \{ p \mid p \text{ is a polynomial over } x \text{ with an integral root} \} \]

Note that
Univariate Polynomials (2)

\[ D_1 = \{ p \mid p \text{ is a polynomial over } x \text{ with an integral root} \} \]

Note that

- If \( p \) has an integral root, the machine accepts.
Univariate Polynomials (2)

\[ D_1 = \{ p \mid p \text{ is a polynomial over } x \text{ with an integral root} \} \]

Note that

- If \( p \) has an integral root, the machine accepts.
- If not, \( M_1 \) loops.
Univariate Polynomials (2)

\[ D_1 = \{ p \mid p \text{ is a polynomial over } x \text{ with an integral root} \} \]

Note that

- If \( p \) has an integral root, the machine accepts.
- If not, \( M_1 \) loops.
- \( M_1 \) is an acceptor, but not a decider.
Univariate Polynomials (3)

\[ f := x \rightarrow x^3 - 300x^2 + 10000x + 1000000; \]
\[ g := x \rightarrow 200x^2 - 2000x - 1000000; \]

\[ \text{plot}([f(x), g(x)], x=-100..300, \text{color}=[red, blue], \text{thickness}=3); \]
Univariate Polynomials (4)

In fact, $D_1$ is decidable.

Can show that all real roots of $p[x]$ lie inside interval

$$
\left( -|kc_{max}/c_1|, |kc_{max}/c_1| \right),
$$

where $k$ is number of terms, $c_{max}$ is max coefficient, and $c_1$ is high-order coefficient.
Univariate Polynomials (4)

In fact, $D_1$ is decidable.

Can show that all real roots of $p[x]$ lie inside interval

$$\left( -\frac{kc_{\text{max}}}{c_1}, \frac{kc_{\text{max}}}{c_1} \right),$$

where $k$ is number of terms, $c_{\text{max}}$ is max coefficient, and $c_1$ is high-order coefficient.

By Matijasevič theorem, such effective bounds on range of real roots cannot be computed for multivariable polynomials.
Wild Models

What about “unreasonable” models of computation? Consider MUintel’s \( \aleph \)-AXP\(^{\copyright} \) processor (to be released XMAS 2003).

Like a Turing machine, except
Wild Models

What about “unreasonable” models of computation? Consider MUn tel’s $\aleph$-AXP© processor (to be released XMAS 2003).

- Like a Turing machine, except
- Takes first step in 1 second.
Wild Models

What about “unreasonable” models of computation? Consider MUntel’s $\aleph$-AXP© processor (to be released XMAS 2003).

- Like a Turing machine, except
- Takes first step in 1 second.
- Takes second step in 1/2 second.
Wild Models

What about “unreasonable” models of computation? Consider MUntel’s $\aleph$-AXP© processor (to be released XMAS 2003).

- Like a Turing machine, except
- Takes first step in 1 second.
- Takes second step in 1/2 second.
- Takes $i$-th step in $2^{-i}$ seconds …
Wild Models

What about “unreasonable” models of computation? Consider MUntel’s $\aleph$-AXP© processor (to be released XMAS 2003).

- Like a Turing machine, except
- Takes first step in 1 second.
- Takes second step in $1/2$ second.
- Takes $i$-th step in $2^{-i}$ seconds . . .

After 2 seconds, the $\aleph$-AXP© decides any enumerable language!
Wild Models

What about “unreasonable” models of computation?
Consider MUntel’s $\aleph$-AXP© processor (to be released XMAS 2003).

- Like a Turing machine, except
- Takes first step in 1 second.
- Takes second step in $1/2$ second.
- Takes $i$-th step in $2^{-i}$ seconds …

After 2 seconds, the $\aleph$-AXP© decides any enumerable language!

**Question:** Does the $\aleph$-AXP© invalidate the Church-Turing Thesis?
Encoding

- Input to a Turing machine is a string of symbols.

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
Encoding

- Input to a Turing machine is a **string of symbols**.
- But we want algorithms that work on graphs, matrices, polynomials, Turing machines, etc.
Encoding

- Input to a Turing machine is a string of symbols.
- But we want algorithms that work on graphs, matrices, polynomials, Turing machines, etc.
- Need to choose an encoding for objects.
Encoding

- Input to a Turing machine is a string of symbols.
- But we want algorithms that work on graphs, matrices, polynomials, Turing machines, etc.
- Need to choose an encoding for objects.
- Can often be done in many reasonable ways.
Encoding

- Input to a Turing machine is a string of symbols.
- But we want algorithms that work on graphs, matrices, polynomials, Turing machines, etc.
- Need to choose an encoding for objects.
- Can often be done in many reasonable ways.
- Sometimes distinguish between $X$, the object, and $\langle X \rangle$, its encoding.
Encoding

Consider strings representing **undirected graphs**.

A graph is **connected** if every node can be reached from any other node by traveling along edges.

Define the language:

\[ A = \{ \langle G \rangle \mid G \text{ is a connected undirected graph} \} \]
High-Level Description

High-level description of a machine that decides

$$A = \{ \langle G \rangle \mid G \text{ is a connected undirected graph} \}$$

On input $\langle G \rangle$, encoding of graph $G$

- select first node of $G$ and mark it.
High-Level Description

High-level description of a machine that decides

\[ A = \{ \langle G \rangle \mid G \text{ is a connected undirected graph} \} \]

On input \( \langle G \rangle \), encoding of graph \( G \)

- select first node of \( G \) and mark it.
- repeat until no new nodes marked:
High-Level Description

High-level description of a machine that decides

\[ A = \{ \langle G \rangle \mid G \text{ is a connected undirected graph} \} \]

On input \( \langle G \rangle \), encoding of graph \( G \)

- select first node of \( G \) and mark it.
- repeat until no new nodes marked:
- For each node in \( G \), mark if attached by an edge to a node already marked.
High-Level Description

High-level description of a machine that decides

\[ A = \{ \langle G \rangle \mid G \text{ is a connected undirected graph} \} \]

On input \( \langle G \rangle \), encoding of graph \( G \)

1. select first node of \( G \) and mark it.
2. repeat until no new nodes marked:
   - For each node in \( G \), mark if attached by an edge to a node already marked.
3. scan nodes of \( G \) to determine whether they are all marked. If so, accept, otherwise reject.
Some Details

**Question:** How is $G$ encoded?

**Answer:** List of nodes, followed by list of edges.
More Details

On input $M$ checks that input is valid graph encoding
- two lists
More Details

On input $M$ checks that input is valid graph encoding
- two lists
- first is list of numbers
More Details

On input $M$ checks that input is valid graph encoding
- two lists
- first is list of numbers
- second is list of pairs
More Details

On input $M$ checks that input is valid graph encoding
- two lists
- first is list of numbers
- second is list of pairs
- first list contains no duplicates (element distinctness subroutine)
More Details

On input $M$ checks that input is valid graph encoding

- two lists
- first is list of numbers
- second is list of pairs
- first list contains no duplicates (element distinctness subroutine)
- every node in second list appears in first
More Details

On input $M$ checks that input is valid graph encoding

- two lists
- first is list of numbers
- second is list of pairs
- first list contains no duplicates (element distinctness subroutine)
- every node in second list appears in first

Now ready to start “step one”.
Detailed Algorithm

On input $\langle G \rangle$, encoding of graph $G$

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and “underlines” undotted node $n_1$.

3. $M$ scans the list of nodes. If all dotted, accept, otherwise reject.
Detailed Algorithm

On input \( \langle G \rangle \), encoding of graph \( G \)

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and “underlines” undotted node \( n_1 \).
   - \( M \) rescans and “underlines” dotted node \( n_2 \).
3. \( M \) scans the list of nodes. If all dotted, accept, otherwise reject.
Detailed Algorithm

On input $\langle G \rangle$, encoding of graph $G$

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and “underlines” undotted node $n_1$.
   - $M$ rescans and “underlines” dotted node $n_2$.
   - $M$ scans edges.
3. $M$ scans the list of nodes. If all dotted, accept, otherwise reject.
Detailed Algorithm

On input $\langle G \rangle$, encoding of graph $G$

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and “underlines” undotted node $n_1$.
   - $M$ rescans and “underlines” dotted node $n_2$.
   - $M$ scans edges.
   - $M$ tests each edge if it is $(n_1, n_2)$.
3. $M$ scans the list of nodes. If all dotted, accept, otherwise reject.
Detailed Algorithm

On input \( \langle G \rangle \), encoding of graph \( G \)

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and “underlines” undotted node \( n_1 \).
   - \( M \) rescans and “underlines” dotted node \( n_2 \).
   - \( M \) scans edges.
   - \( M \) tests each edge if it is \( (n_1, n_2) \).
   - If so, dot \( n_1 \), remove underlines, goto Step 2.

3. \( M \) scans the list of nodes. If all dotted, accept, otherwise reject.
Detailed Algorithm

On input $\langle G \rangle$, encoding of graph $G$

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and “underlines” undotted node $n_1$.
   - $M$ rescans and “underlines” dotted node $n_2$.
   - $M$ scans edges.
   - $M$ tests each edge if it is $(n_1, n_2)$.
   - If so, dot $n_1$, remove underlines, goto Step 2.
   - If not, check next edge. When no more edges, move underline to next dotted $n_2$.

3. $M$ scans the list of nodes. If all dotted, accept, otherwise reject.
Detailed Algorithm

On input \( \langle G \rangle \), encoding of graph \( G \)

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and “underlines” undotted node \( n_1 \).
   - \( M \) rescans and “underlines” dotted node \( n_2 \).
   - \( M \) scans edges.
   - \( M \) tests each edge if it is \((n_1, n_2)\).
   - If so, dot \( n_1 \), remove underlines, goto Step 2.
   - If not, check next edge. When no more edges, move underline to next dotted \( n_2 \).
   - when no more dotted vertexes, move underlines: new \( n_1 \) is next undotted node and new \( n_1 \) is first dotted node. Repeat Step 2. When no more und dotted nodes, go to Step 3.
3. \( M \) scans the list of nodes. If all dotted, accept, otherwise reject.
Detailed Algorithm

On input $\langle G \rangle$, encoding of graph $G$

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and “underlines” undotted node $n_1$.
   - $M$ rescans and “underlines” dotted node $n_2$.
   - $M$ scans edges.
   - $M$ tests each edge if it is $(n_1, n_2)$.
   - If so, dot $n_1$, remove underlines, goto Step 2.
   - If not, check next edge. When no more edges, move underline to next dotted $n_2$.
   - when no more dotted vertexes, move underlines: new $n_1$ is next undotted node and new $n_1$ is first dotted node. Repeat Step 2. When no more undotted nodes, go to Step 3.
3. $M$ scans the list of nodes. If all dotted, accept, otherwise reject.
Detailed Algorithm

On input \( \langle G \rangle \), encoding of graph \( G \)

1. mark first node with a dot on leftmost digit.
2. loop:
   - Scans list and “underlines” undotted node \( n_1 \).
   - \( M \) rescans and “underlines” dotted node \( n_2 \).
   - \( M \) scans edges.
   - \( M \) tests each edge if it is \( (n_1, n_2) \).
   - If so, dot \( n_1 \), remove underlines, goto Step 2.
   - If not, check next edge. When no more edges, move underline to next dotted \( n_2 \).
   - when no more dotted vertexes, move underlines: new \( n_1 \) is next undotted node and new \( n_1 \) is first dotted node. Repeat Step 2. When no more undotted nodes, go to Step 3.
3. \( M \) scans the list of nodes. If all dotted, accept, otherwise reject.
Decidability of Languages

**Question:** Why study decidability?

- Good for improving your imagination.
Decidability of Languages

**Question:** Why study decidability?

- Good for improving your imagination.
- Some of the most beautiful and important mathematics of the 20th century, and you can actually understand it!
Decidability of Languages

**Question:** Why study decidability?

- Good for improving your imagination.
- Some of the most beautiful and important mathematics of the 20th century, and you can actually understand it!
- Your boss (who never took this course...) orders you to solve Hilbert’s 10th, or else.
Examples of Decidable Languages

Finite automata problems can be reformulated as languages.

Does DFA, $B$, accept input string $w$?
Examples of Decidable Languages

Finite automata problems can be reformulated as languages.

Does DFA, $B$, accept input string $w$?

Consider the language:

$$A_{\text{DFA}} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts } w \}$$
Examples of Decidable Languages

Finite automata problems can be reformulated as languages.

Does DFA, $B$, accept input string $w$?

Consider the language:

$$A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts } w \}$$

The following are equivalent:

- $B$ accepts $w$
- $\langle B, w \rangle \in A_{DFA}$
Decidability of DFA Acceptance

**Theorem:** $A_{\text{DFA}}$ is a decidable language.
Decidability of DFA Acceptance

**Theorem:** $A_{\text{DFA}}$ is a decidable language.

**Proof:** On input $\langle B, w \rangle$, where $B$ is a DFA and $w$ a string:

1. Simulate $B$ on input $w$
2. if simulation ends in accepting state, accept, otherwise reject.
Decidability of DFA Acceptance

**Theorem:** $A_{\text{DFA}}$ is a decidable language.

**Proof:** On input $\langle B, w \rangle$, where $B$ is a DFA and $w$ a string:

1. Simulate $B$ on input $w$
2. if simulation ends in accepting state, accept, otherwise reject.

Remarks

- “where” clause means scan and check condition.
- $B$ represented by a list of $(Q, \Sigma, \delta, q_0, F)$.
- simulation straightforward.
Decidability of NFA Acceptance

Does an NFA accept a string?

$$A_{\text{NFA}} = \{ \langle B, w \rangle | B \text{ is an NFA that accepts } w \}$$
Decidability of NFA Acceptance

Does an NFA accept a string?

\[ A_{\text{NFA}} = \{ \langle B, w \rangle | B \text{ is an NFA that accepts } w \} \]

**Theorem:** \( A_{\text{NFA}} \) is a decidable language.
Decidability of NFA Acceptance

Does an NFA accept a string?

\[ A_{\text{NFA}} = \{ \langle B, w \rangle | B \text{ is an NFA that accepts } w \} \]

Theorem: \( A_{\text{NFA}} \) is a decidable language.

On input \( \langle B, w \rangle \), where \( B \) is an NFA and \( w \) a string:

1. Convert the NFA \( B \) into equivalent DFA \( C \).
2. Run previous TM on input \( \langle C, w \rangle \).
3. If that TM accepts, accept, otherwise reject.

Note use of subroutine (2).
Decidability of Reg. Exp. Generation

Does a regular expression generate a string?

\[ A_{REX} = \{ (R, w) \mid R \text{ is a regular expression that generates } w \} \]
Decidability of Reg. Exp. Generation

Does a regular expression generate a string?

\[ A_{\text{REX}} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates } w \} \]

**Theorem:** \( A_{\text{REX}} \) is a decidable language.
Decidability of Reg. Exp. Generation

Does a regular expression generate a string?

\[ A_{\text{REX}} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates } w \} \]

**Theorem:** \( A_{\text{REX}} \) is a decidable language.

On input \( \langle R, w \rangle \), where \( R \) is a regular expression and \( w \) a string:

1. Convert regular expression \( R \) into DFA \( C \).
2. Run earlier TM on input \( \langle C, w \rangle \).
3. If that TM accepts, accept, otherwise reject.
Decidability of DFA Emptiness

Does a DFA accept the empty language, \( \emptyset \)?

Define \( E_{\text{DFA}} = \{ \langle A \rangle | A \text{ is a DFA and } L(A) = \emptyset \} \)
Decidability of DFA Emptiness

Does a DFA accept the empty language, $\emptyset$?

Define $E_{DFA} = \{ \langle A \rangle | A \text{ is a DFA and } L(A) = \emptyset \}$

Theorem: $E_{DFA}$ is a decidable language.
Decidability of DFA Emptiness

Does a DFA accept the empty language, $\emptyset$?

Define $E_{\text{DFA}} = \{ \langle A \rangle | A \text{ is a DFA and } L(A) = \emptyset \}$

**Theorem:** $E_{\text{DFA}}$ is a decidable language.

On input $\langle A \rangle$, where $A$ is a DFA:

1. Mark the start state of $A$.
2. Repeat until no new states are marked:
   3. Mark any state that has a transition coming into it from any already marked state.
3. If no accept state is marked, accept, otherwise reject.

This TM actually just tests whether any accepting state is reachable from initial state (a reachability problem in digraphs).
Decidability of DFAs Equivalence

\[ EQ_{\text{DFA}} = \{ \langle A, B \rangle \mid A, B \text{ are DFAs and } L(A) = L(B) \} \]
Decidability of DFAs Equivalence

\[ EQ_{\text{DFA}} = \{ \langle A, B \rangle \mid A, B \text{ are DFAs and } L(A) = L(B) \} \]

**Theorem:** \( EQ_{\text{DFA}} \) is a decidable language.
Decidability of DFAs Equivalence

\[ EQ_{\text{DFA}} = \{ \langle A, B \rangle \mid A, B \text{ are DFAs and } L(A) = L(B) \} \]

**Theorem:** \( EQ_{\text{DFA}} \) is a decidable language.

We construct a new DFA \( C \) from \( A \) and \( B \), such that \( C \) accepts only string accepted by \( A \) or by \( B \), but not both. In other words

\[ L(C) = \left( L(A) \cap \overline{L(B)} \right) \cup \left( \overline{L(A)} \cap L(B) \right). \]

- \( L(C) \) is *symmetric difference* of \( L(A) \) and \( L(B) \)
- use construction used for regular language theorems
- construction can be expressed as a TM
DFA Equivalence (continued)

**Theorem:** $EQ_{DFA}$ is a decidable language.
DFA Equivalence (continued)

Theorem: \( EQ_{DFA} \) is a decidable language.

On input \( \langle A. B \rangle \), where \( A, B \) are DFAs:
1. Construct DFA, \( C \), as described.
2. Run previous “emptyness” TM on input \( \langle C \rangle \).
3. If that TM accepts, accept, otherwise reject. ♣
Decidability of CFG Generation

Does a CFG generate a given string?

Define

\[ \mathcal{A}_{CFG} = \{ \langle G, w \rangle \mid \text{string } w \text{ is generated by CFG } G \} \]

**Theorem:** The language \( \mathcal{A}_{CFG} \) is decidable.
Decidability of CFG Generation (2)

Does a CFG generate a given string?
Decidability of CFG Generation (2)

Does a CFG generate a given string?

Initial Idea: Design a TM, $M$, to try all derivations.
Decidability of CFG Generation (2)

Does a CFG generate a given string?

Initial Idea: Design a TM, $M$, to try all derivations.

Problem: $M$ accepts, but does not decide. (why?)
Decidability of CFG Generation (3)

Lemma: If $G$ is in Chomsky normal form, $|w| = n$, and $w$ is generated by $G$, then $w$ has a derivation of length $2n - 1$ or less.

We won’t prove this (go ahead — try it at home!).

Algorithm’s idea:
Decidability of CFG Generation (3)

**Lemma:** If $G$ is in Chomsky normal form, $|w| = n$, and $w$ is generated by $G$, then $w$ has a derivation of length $2n - 1$ or less.

We won’t prove this (go ahead — try it at home!).

Algorithm’s idea:

- First, convert $G$ to Chomsky normal form.
Decidability of CFG Generation (3)

**Lemma:** If $G$ is in Chomsky normal form, $|w| = n$, and $w$ is generated by $G$, then $w$ has a derivation of length $2n - 1$ or less.

We won’t prove this (go ahead — try it at home!).

Algorithm’s idea:

- First, convert $G$ to Chomsky normal form.
- Now need only consider a finite number of derivations – those of length $2n - 1$ or less.
Decidability of $\text{CFG}$ Generation (3)

**Theorem:** $A_{\text{CFG}}$ is a decidable language.
Decidability of \textbf{CFG} Generation (3)

\textbf{Theorem: } $A_{\text{CFG}}$ is a decidable language.

On input $\langle G, w \rangle$, where $G$ is a grammar and $w$ a string,

1. Convert $G$ to Chomsky normal form.
2. List all derivations with $2n - 1$ steps, were $n = |w|$.
3. If any generates $w$, accept, otherwise reject.
Decidability of CFG Generation (3)

**Theorem:** $A_{\text{CFG}}$ is a decidable language.

On input $\langle G, w \rangle$, where $G$ is a grammar and $w$ a string,

1. Convert $G$ to Chomsky normal form.
2. List all derivations with $2n - 1$ steps, were $n = |w|$.
3. If any generates $w$, accept, otherwise reject.

**Remarks:**
- related to problem of compiling prog. languages
- would you want to use this algorithm at work?
- every theorem about CFLs is also about PDAs.