Decidable | Semi-decidable | Undecidable

THEORY OF FORMAL LANGUAGES · LECTURE 13

Decidability & Computability

What problems can an algorithm always solve, which ones can it only half-solve, and which ones admit no algorithm at all?

Prof. K. R. Chowdhary

Professor of CS

ROADMAP

What We'll Cover

1. **Recursively enumerable (RE) languages**
   The broadest class a Turing machine can recognize — and how it differs from recursive languages.

2. **Three kinds of problems**
   Decidable, undecidable, and the in-between case: semi-decidable problems you can verify but not always solve.

3. **Decision problems, formally**
   What it means for a procedure to be effective — complete, mechanistic, and deterministic.

4. **RE sets and a worked TM example**
   Generating sets by effective procedures, then a concrete machine that recognizes even numbers.

Recursively Enumerable (RE) Languages

DEFINITION 13.1

An RE language is a formal language for which some Turing machine will halt and accept any string in the language — but on a string NOT in the language, it may halt and **reject, or loop forever.**

i **IN PLAIN ENGLISH**

"Recursively enumerable" means the accepted strings can be listed out, one by one, by some mechanical process — hence the other names for this class: partially decidable, or Turing-recognizable. If w truly belongs to the language, the TM is guaranteed to eventually say so; if it doesn't, you might simply wait forever without an answer.

KEY FACTS ABOUT RE LANGUAGES

1. RE is type-0 in the Chomsky hierarchy — the broadest class a Turing machine can recognize.
2. Every RE language is countably infinite.
3. Regular, context-free, context-sensitive, and recursive languages are all recursively enumerable.

WHY THE LOOP-FOREVER CASE MATTERS

While M is still running, we cannot tell if it will eventually accept, or run forever. That ambiguity is exactly why RE languages **are also called partially decidable.**

RE vs. Recursive Languages

RECURSIVELY ENUMERABLE

Halts and accepts if w is in the language.

If w is NOT in the language, the TM may reject — or may simply never halt at all.

RECURSIVE

Halts — and gives an answer — no matter what.

The stronger guarantee: every recursive language is RE, but not the reverse.

i

IN PLAIN ENGLISH

Both kinds of machine recognize the same “yes” strings correctly. The difference only shows up on a “no” string: a recursive language's TM always comes back and tells you “no”; an RE language's TM might just leave you waiting, unable to distinguish “no” from “not done yet.”

RE languages, together with their complement (co-RE), form the basis of the arithmetical hierarchy — a finer classification of undecidable problems we'll return to later.

What Problems Are Solvable?

"What problems are solvable?" is too vague to answer directly. Here are eight concrete decision problems to sharpen it — each with a Yes/No answer:

1. Is a natural number n prime?
2. Is a propositional formula θ satisfiable?
3. Is a propositional formula θ a tautology?
4. Is a predicate formula ψ provable, i.e. $\Gamma \vdash \psi$?
5. Does $P(x_1,\ldots,x_n) = 0$ have integer solutions?
6. Does C program P halt on some given input?
7. Does C program Q halt on every input?
8. Do n consecutive 3's occur in the digits of π?

i **IN PLAIN ENGLISH**

Some instances of each are trivial to solve by hand — the real question is whether a single, general algorithm handles every instance.

Decidable Problems

THE GENERAL PRINCIPLE

A problem is decidable (solvable) if an effective procedure gives the correct **outcome for every possible instance.**

i

IN PLAIN ENGLISH

“Decidable” isn't about any one input being easy — it's about having ONE general-purpose recipe that correctly handles every instance, without exception, forever.

A KEY TEST: PRIMALITY

A fixed algorithm checks any n for primality and prints Yes/No — a completely mechanical procedure.

SATISFIABILITY (θ SAT?)

Build the truth table for θ. If any row comes out True, θ is satisfiable.

TAUTOLOGY (θ ALWAYS TRUE?)

Same truth table, opposite check: θ is a tautology only if EVERY row comes out True.

Undecidable Problems

For predicate logic provability, no recipe is available: after failing to find a natural-deduction proof of ψ, we can't tell whether ψ just isn't provable, or whether natural deduction simply isn't strong enough to find it.

THE CRUCIAL DISTINCTION

It might not be that we lack a good algorithm — there may be NO algorithm at all. When that's provably the case, the **problem's solution is undecidable.**

i

IN PLAIN ENGLISH

Contrast with the π-digit search: there, we simply don't yet have a procedure. For predicate-logic provability, the decision procedure is impossible as a matter of fact — a much stronger claim.

A SUBTLE COMPANION CASE

Consecutive 3's in π

Generate digits of π until n consecutive 3's appear. This looks exactly like the predicate-logic search — keep trying, and stop only on success.

But the two differ in kind: for ψ, no procedure can possibly exist. For π, we simply don't currently have one — it may or may not be decidable at all.

Semi-Decidable Problems

THE KEY IDEA

Some unsolvable problems are still easy to VERIFY: given a candidate answer, checking **it is straightforward — even if finding it isn't.**

THE GENERATE-AND-CHECK STRATEGY

1. Systematically generate candidate proofs (or solutions) one by one, forever.
2. Check each candidate mechanically — if it's valid, stop and report success.
3. If ψ truly has a proof, this eventually finds it. If not, the search never ends — and never tells you so.

i

IN PLAIN ENGLISH

This asymmetry — confirm easily, search forever if wrong — is exactly the signature of a semi-decidable problem.

EXAMPLE — FERMAT-STYLE EQUATION

$$z^n = x^n + y^n$$

Known to have solutions for n = 2. Whether any solution exists for n ≥ 3 was open for centuries — easy to check a guess, hard to decide in general.

EXAMPLE — DOES PROGRAM P HALT?

Run P and wait. If it halts, you have your answer. If it doesn't halt yet, you can **never be sure it won't halt later.**

NOT THE SAME AS: DOES Q HALT ON EVERY INPUT?

That's a different, harder problem — infinitely many inputs to check, not just one.

Three Kinds of Problems, Side by Side

DECIDABLE

An effective procedure always halts with the correct Yes or No.

EXAMPLE

Primality, SAT, tautology checking.

SEMI-DECIDABLE

A procedure confirms every “Yes” instance, but may search forever on a “No.”

EXAMPLE

Does program P halt on this input?

UNDECIDABLE

Provably no algorithm exists — not just that we haven't found one yet.

EXAMPLE

Predicate-logic provability, in general.

IN PLAIN ENGLISH

These form a strict hierarchy: every decidable problem is also semi-decidable (the same algorithm just always finishes), but not every semi-decidable problem is decidable — and undecidable problems aren't even guaranteed the one-sided confirmation semi-decidable problems offer.

Decision Problems, Formally

DEFINITION 13.2

The decision problem for a set is that of effectively determining, for an arbitrary given positive integer, whether it is or is **not in the set.**

A DECISION PROBLEM P, UNPACKED

P is a set of related questions p_i, each with a Yes/No answer. P is decidable exactly **when some algorithm answers every p_i.**

i **IN PLAIN ENGLISH**

Every decision problem is really an infinite family of yes/no questions, one per instance. “Solving P” doesn't mean answering one question — it means having a single method that correctly answers all of them, forever, without needing to be redesigned for each new instance.

WORKED EXAMPLE — IS N A PERFECT SQUARE?

p_0: Is 0 a perfect square? → Yes

p_1: Is 1 a perfect square? → Yes

p_2: Is 2 a perfect square? → No

⋮

To test n = 20: generate $1^2, 2^2, 3^2, 4^2 = 16$, then $5^2 = 25$. Since 25 > 20 with no exact match along the way, 20 is not a perfect **square — the search can stop for good.**

Each p_i is one instance of the problem; the perfect-square test itself is what makes the whole family decidable.

Effective Procedures

DEFINITION 13.3

A procedure that is complete, mechanistic, **and deterministic is an effective procedure.**

C **Complete**
Gives the correct answer for every instance in the problem's domain — no exceptions, no selective coverage.

M **Mechanistic**
Built from finitely many elementary steps — add 1, discard an argument, test for zero — each done without insight or guesswork.

D **Deterministic**
Every step has exactly one unambiguous outcome; identical input always triggers identical computation.

TM AS AN EFFECTIVE PROCEDURE

A standard TM is mechanistic and deterministic by construction. It's complete only when it halts on every input — which is exactly the recursive/decidable case from earlier.

USING A TM TO SOLVE A DECISION PROBLEM

Encode each problem instance as a string (its representation). The TM accepts and halts for Yes, rejects and halts for No.

THE CHURCH–TURING THESIS (DECISION-PROBLEM FORM)

A TM can solve a decision problem exactly when it's solvable by an effective procedure.

Two Roads to “Computable”

Effectively computable functions can be studied from two directions that turn out to describe exactly the same class:

THE MATHEMATICAL ROAD

Start from a few simple, intuitively computable base functions. Build more complex ones using recursion — the result is called a recursive function.

Worked instance: a polynomial

$P(x_1,\ldots,x_n)$ with non-negative integer coefficients is effectively calculable: plug in positive integers, and ordinary addition and multiplication compute its value.

THE PRACTICAL ROAD

Build an idealized computing framework — unlimited time and memory, but simple rules — and study what it can do. The Turing machine is the best-known such framework.

Other equivalent frameworks

The register machine (computes exactly the recursive functions) and the λ-calculus both model computation too — different machinery, same computational power.

i **IN PLAIN ENGLISH**

Different starting points, same destination: recursive functions, TMs, register machines, and λ-calculus all define the same class of computable functions.

Recursively Enumerable Sets

Not every set is recursive. The sets whose elements can be effectively generated — one by one, forever — are the recursively enumerable (RE) sets.

THE FORMAL DEFINITION

A set of positive integers is RE if there's a recursive function f(x) whose values f(1), f(2), f(3), ... constitute exactly that **set — a recursive enumeration.**

i

IN PLAIN ENGLISH

If an element truly belongs to an RE set, running the generator eventually produces it — confirmed. If it doesn't belong, the generator just keeps running, and you can never be sure it won't show up later. This is exactly the semi-decidable pattern from earlier, now applied to sets instead of problems: RE sets are also called semi-decidable, or semi-recursive.

MODELS THAT COMPUTE THE SAME CLASS

Turing machines, register machines, and recursive functions all define the same computable class. Natural numbers appear in two roles here: as codes for functions, **and as arguments to those functions.**

WHERE RE SETS SHOW UP

Predicate-logic provability, integer solutions to polynomials, and “does program P halt?” are all RE: generate candidates, and confirm on a hit — exactly the earlier **semi-decidable examples.**

Three Examples of RE Sets

1 · SQUARES

$1^2, 2^2, 3^2, \ldots$

Generated directly by the recursive function f(x) = x^2. The simplest possible recursive enumeration.

Any given n's status can be checked by regenerating the sequence up to n — that's what "effective" buys us.

2 · SELF-REFERENTIAL TOWER

$1, 2, 2^{1+2}, \ldots$

Each new term raises 2 to the power of the sum of every term generated so far — e.g. the 3rd term is 2 raised to (1+2).

Post's theorem: the standard method for proving an enumerable set of enumerable sets is itself enumerable applies directly here.

3 · UNION OF POWER SERIES

$n^2, n^3, n^4, \ldots$

As each n is generated in order, spin up processes producing n^2, n^3, n^4, …. An enumerable union of enumerable sets is itself enumerable.

Every generated set of positive integers is recursively enumerable — the two ideas turn out to coincide.

Representing a Problem for a Turing Machine

To hand a decision problem to a TM, translate each instance p_i into an input string w_i. The machine's job is simply to answer Yes or No on every w_i:

Problem instance	TM input	Answer
p_1	w_1	Yes/No
p_2	w_2	Yes/No
…	…	…
p_i	w_i	Yes/No

i **IN PLAIN ENGLISH**

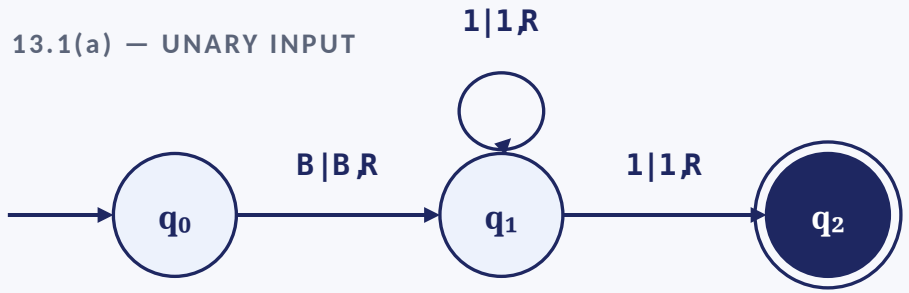
This table is the bridge between the abstract idea of a decision problem and an actual, runnable Turing machine: every instance becomes one row, one input string, and one halting computation.

ANSWERED BY ACCEPTING OR REJECTING

A problem instance gets a Yes when the TM accepts and halts, and a No when it rejects and halts — both are legitimate, definite **answers for a decidable problem.**

Worked Example — Recognizing Even Numbers

FIG. 13.1(a) — UNARY INPUT



Accepts 1^i exactly when reaching the trailing blank lands the head in accept state q_2 — i.e. when i is odd.

IN PLAIN ENGLISH

This lecture's unary code is n ↔ n+1 ones (same convention as earlier lectures): the string "1" encodes 0, "11" encodes 1, "111" encodes 2. So a string of ODD length 1^i encodes the EVEN number i−1 — which is exactly the accept condition above.

TABLE 13.2 — PROBLEM INSTANCES

Instance	Input	Even?
0	1	Yes
1	11	No
2	111	Yes
3	1111	No

FIG. 13.1(B) — BINARY INPUT, BRIEFLY

A similar machine handles binary numbers: it accepts exactly when the input string ends in the digit 0 — the ordinary test for **an even number in binary notation.**

SUMMARY

Key Takeaways

1. **RE is the broadest recognizable class**
   A TM always halts and accepts strings in the language, but may loop forever on strings outside it — recursive languages demand more: halting either way.

2. **Three tiers, one hierarchy**
   Decidable ⊆ semi-decidable, and undecidable problems don't even get semi-decidable's one-sided guarantee — the difference is provable, not just a gap in our algorithms.

3. **Effective means complete + mechanistic + deterministic**
   That's the intuitive bar a Turing machine has to clear to count as solving a decision problem at all.

4. **RE sets are generated, not just decided**
   An element in the set will eventually appear; an element outside it may simply never be ruled out.

Thank You

Questions on RE languages, decidability, or the worked TM example?

Prof. K. R. Chowdhary

Professor of CS · Theory of Formal Languages, Lecture 13