Computational Complexity

Models, Classes, and Fundamental Limits

Khushraj Madnani

Department of Computer Science, Indian Institute of Technology Guwahati

What Is Complexity Theory About?

- Computability: what problems can be solved at all
- Complexity theory: how efficiently problems can be solved
- Efficiency measured using computational resources:
  - Time
  - Space (memory)
- Goal: classify problems by resource requirements

Problems vs Programs

- A **problem** is a mathematical object:

$$L \subseteq \Sigma^*$$

- A **program / algorithm** is a procedure that solves the problem
- Complexity is a property of the problem, not of a specific program

Time Complexity (Informal)

- Measures number of elementary steps as a function of input size
- Typically considers worst-case behavior
- Expressed asymptotically (Big-O notation)
- Examples:
  - Linear search: $O(n)$
  - Binary search: $O(\log n)$

Space Complexity (Informal)

- Measures amount of memory used during computation
- Includes auxiliary storage (work memory)
- Often excludes read-only input
- Space can be reused, unlike time

Why a Formal Model Is Needed

- Informal notions depend on machine details
- What is a "step"?
- What counts as "memory"?
- Need a simple, precise, universal model

Solution: Turing Machines

Turing Machines as a Model of Computation

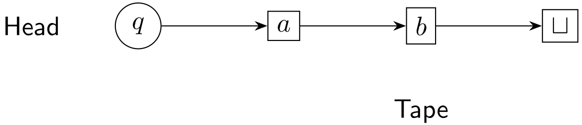
- Abstract model capturing the notion of algorithm
- Simple but computationally universal
- Any reasonable programming language can be simulated
- Forms the basis of computability and complexity theory

One-Tape Turing Machine

A Turing Machine consists of:

- Finite set of states
- A single infinite tape
- A tape head that can read, write, and move
- A transition relation (if the Turing machine is deterministic, the relation becomes a function.)

One-Tape Turing Machine Diagram



Single tape used as both input and memory

Formal Definition of a Turing Machine

A Turing Machine is a tuple:

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

where:

- Q: finite set of states
- Σ: input alphabet
- Γ: tape alphabet ($\Sigma \subseteq \Gamma$)
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- q_0: start state
- q_{acc}, q_{rej}: halting states

Configurations and Computation

- A configuration consists of:
  - Current state
  - Tape contents
  - Head position
- A computation is a sequence of configurations
- A machine halts when it reaches a halting state

Why Turing Machines Model Programs

- Tape corresponds to memory
- Head corresponds to instruction pointer
- States represent control flow
- Transition relation models instruction execution

Any algorithm can be simulated by a TM with polynomial overhead

The Halting Problem

Informal Question:
Given a program and an input, will the program eventually halt or run forever?

Formal Setting:

- Model of computation: Turing Machines
- Input: a Turing Machine M and a string w
- Question: Does M halt on input w?

Formal Definition

Definition (Halting Problem):

$$\text{HALT} = \{\langle M, w\rangle \mid M \text{ halts on input } w\}$$

Decision Problem:

$$\text{Is } \langle M, w\rangle \in \text{HALT?}$$

Decidability

Definition:
A language L is **decidable** if there exists a Turing Machine D such that:

- D halts on all inputs
- D accepts exactly the strings in L

Goal: Determine whether HALT is decidable.

Main Claim

Theorem:

The Halting Problem is undecidable.

That is, no Turing Machine can correctly decide HALT on all inputs.

Proof Strategy

Proof Technique: Contradiction

Steps:

1. Assume HALT is decidable
2. Construct a paradoxical Turing Machine
3. Derive a contradiction
4. Conclude HALT is undecidable

Assumption

Assume there exists a Turing Machine H such that:

$$H(\langle M, w \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ halts on } w \\ \text{reject} & \text{otherwise} \end{cases}$$

- H halts on all inputs
- H correctly decides HALT

Construction of a New Machine

Define a Turing Machine D as follows:

Input: $\langle M \rangle$

1. Run H on input $\langle M, M \rangle$
2. If H accepts, loop forever
3. If H rejects, halt and accept

Intuition Behind the Construction

Machine D:

- Uses H to predict its own behavior
- Then deliberately does the opposite

This creates a self-referential paradox.

The Critical Question

What happens when D is run on its own description?

$$D(\langle D \rangle)$$

We analyze two possible cases.

Case 1

Case 1: $H(\langle D, D\rangle)$ accepts.

- H predicts that D halts on input $\langle D\rangle$
- By definition of D, it loops forever

Contradiction: D does not halt.

Case 2

Case 2: $H(\langle D, D\rangle)$ rejects.

- H predicts that D does not halt
- By definition of D, it halts and accepts

Contradiction: D halts.

Contradiction

In both cases:

- H gives an incorrect answer
- This contradicts the assumption that H is correct

Therefore, such a machine H cannot exist.

Conclusion

HALT is undecidable

Why This Matters

- Fundamental limit of computation
- Basis for many undecidability results
- No general termination checker exists
- Not a complexity issue, but a logical impossibility

Key Takeaways

- Undecidability arises from self-reference
- Many programs do halt, but no algorithm decides all
- Undecidable $\neq$ inefficient

Time Complexity of a Turing Machine

Let M be a Turing Machine.

$$T_M(n) = \max_{|x|=n} \text{ number of steps before halting on } x$$

- Measures worst-case running time
- Depends only on input length

Space Complexity of a Turing Machine

$$S_M(n) = \max_{|x|=n} \text{ number of tape cells visited}$$

- Counts memory usage
- Excludes constant-size control
- Space can be reused

Why Complexity Is Defined Using TMs

- Machine-independent
- Mathematically precise
- Robust under reasonable model changes
- Captures intrinsic difficulty of problems

Complexity Classes

A complexity class is a set of problems solvable within given resource bounds.

- Time-bounded classes
- Space-bounded classes
- Deterministic and nondeterministic variants

Class P

$$\mathbf{P} = \{L \mid L \text{ decidable in polynomial time}\}$$

- Models efficient computation
- Considered tractable problems

Class NP

$$\mathbf{NP} = \{L \mid L \text{ has polynomial-time verifiable certificates}\}$$

- Equivalent to nondeterministic polynomial time
- Central open problem: $\mathbf{P} = \mathbf{NP}$?

Space Complexity Classes

- **PSPACE**: polynomial space
- **EXPSPACE**: exponential space
- Space allows reuse, making it powerful

Time Complexity Classes

- **EXPTIME**: exponential time
- 2-**EXPTIME** and beyond
- Arise naturally in games, logic, verification

Relations Between Classes

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{EXPSPACE}$$

- Savitch's Theorem: $\mathbf{PSPACE} = \mathbf{NPSPACE}$
- Space is generally more powerful than time

Elementary Functions

- Built from:
  - Addition
  - Multiplication
  - Exponentiation
- Bounded by fixed-height towers of exponentials
- Many classical complexity classes are elementary

Non-Elementary Functions

- Require unbounded exponential towers
- Arise in:
  - Higher-order logics
  - Certain verification problems
- No fixed stack of exponentials suffices

Complexity Hierarchy

From weakest to strongest:

- Regular
- Context-free
- **P**
- **NP**
- **PSPACE**
- **EXPTIME**
- **EXPSPACE**
- Non-elementary
- Undecidable

Undecidability

- Some problems cannot be solved by any algorithm
- No amount of time or space suffices
- These lie outside all complexity classes

The Halting Problem

Problem:

Given a Turing Machine M and input x, does M halt on x?

The Halting Problem is undecidable

Why the Halting Problem Is Undecidable

- Assume a decider exists
- Construct a self-referential machine
- Leads to contradiction via diagonalization

No algorithm can decide termination in general

Consequences of Undecidability

- No universal termination checker
- No general program verifier
- Fundamental limits of computation

Final Takeaway

- Turing Machines formalize computation
- Complexity theory measures resource usage
- Hierarchies classify problems by difficulty
- Undecidability marks the absolute boundary

Not everything computable is efficient, and not everything definable is computable.