SOUTHEAST UNIVERSITY

Meeting the Challenges of Time

Assignment

Course Code: CSE441.3

Course Name: Theory of Computing

Submitted By:

MD. Aayat Hossain Mridha

Roll No: 2023100000457

Batch:64

Submitted To:

Mr. Nahid Hasan

Lecturer

Department of CSE

Southeast University

1. Decidable Languages

Definition:

A decidable language is a language for which there exists a Turing machine that halts on *every input* and correctly accepts or rejects depending on whether the input is in the language.

Example:

The language where some number of a is followed by equal number of b. A Turing machine can scan the input, match each ‘a’ with a ‘b’, and halt with accept or reject.

Importance:

They represent all problems that can be completely solved algorithmically.

---

2. Undecidability of the Halting Problem

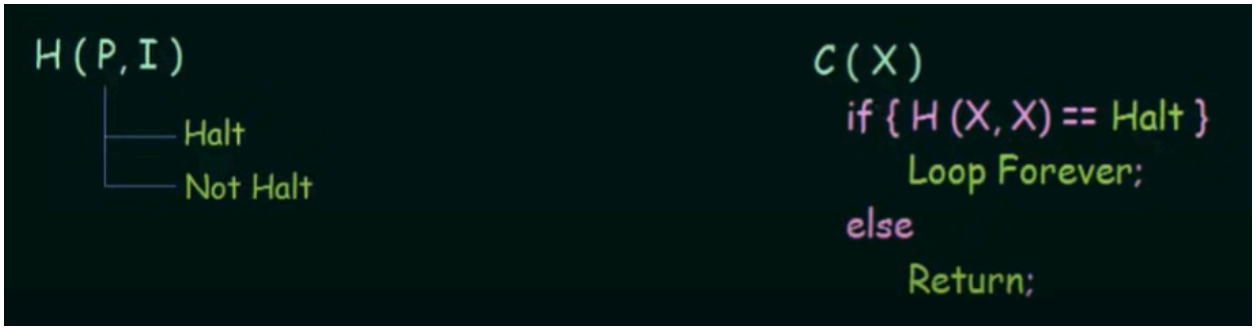
Definition:

The Halting Problem asks:
“Given a Turing machine P and input I, will P halt on I?” This is undecidable and it is proved below:

Proof:

a. Let us imagine there exists a program H(P,I) that can predict whether any program P halts when run on input I. If P halts, H says “Halt.” If P does not halt, H says “Not Halt.”

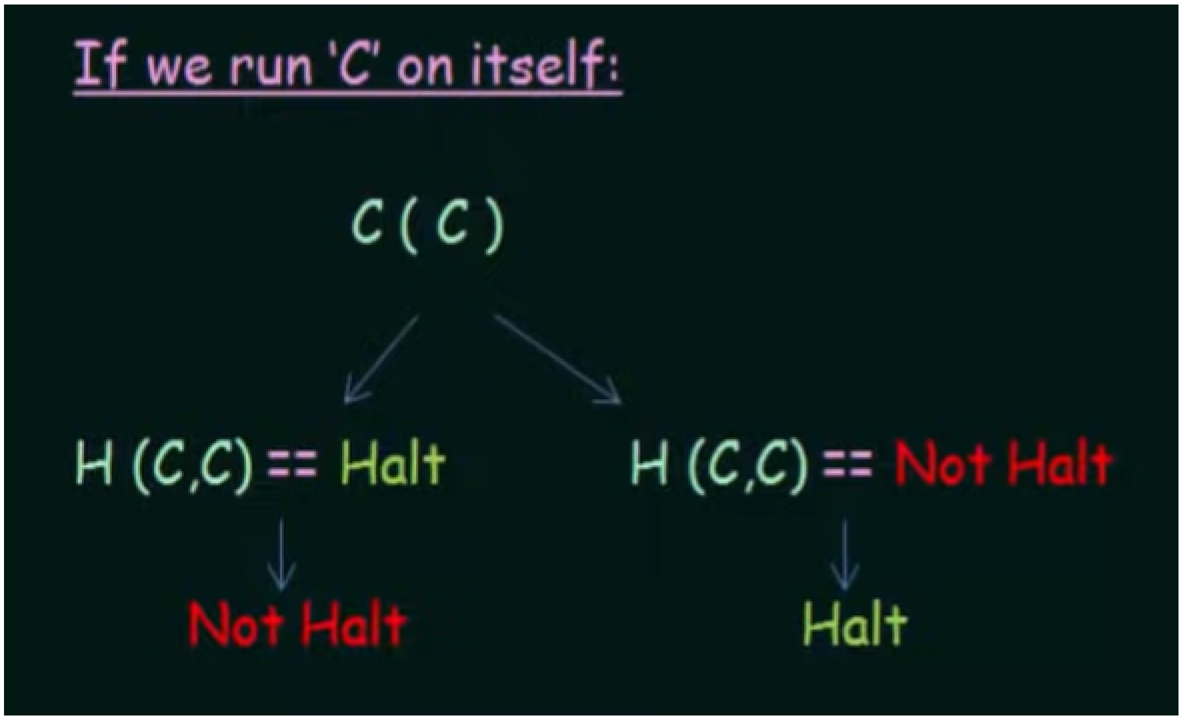
b. Now, using this program H, we build another program C(X,X). For an input X, program C consults H to see what X does when run on itself. If H claims that X halts on input X, then C enters an infinite loop. If H claims that X does not halt on input X, then C halts immediately.



```
H(P,I)
  ├── Halt
  └── Not Halt
```

```
C(X)
  if { H(X,X) == Halt }
      Loop Forever;
  else
      Return;
```

c. The paradox arises when we run C on itself. If H predicts that C halts on C, then by design C will loop forever — contradicting H’s prediction. If H predicts that C does not halt on C, then C halts at once — again contradicting H.



This self-contradiction means our initial assumption about the existence of H was wrong. No program can universally decide whether any other program halts. This is the classic proof that the **Halting Problem is undecidable**.

Importance:

It proves there are **fundamental limits to what algorithms can decide**.

---

3. Diagonalization and Reduction

Definitions:

- **Diagonalization**: A proof technique used to show that some languages cannot be decided by any Turing Machine, by constructing a language that differs from the output of every Turing Machine in a hypothetical list at a “diagonal” position, ensuring it cannot be recognized by any machine in the list.

- **Reduction**: A technique used to show that the difficulty of one problem can be transferred to another by transforming instances of the first problem into instances of the second in a way that preserves the answer, often used to prove undecidability or computational hardness.

Example:

- **Diagonalization**: Cantor’s proof that the real numbers are uncountable; Turing’s proof that some languages are not Turing-recognizable.
- **Reduction**: Suppose we transform the Halting Problem into the problem “Does the program enter an infinite loop?” If we could solve the “infinite loop problem,” we could solve the Halting Problem too. Since Halting is undecidable, the “infinite loop problem” is also undecidable.

Importance:

They are central tools for proving undecidability and computational hardness.

---

4. P, NP, and NP-Complete (NPC) Class Problems

Definition:

- **P**: Class of decision problems solvable by a deterministic Turing machine in polynomial time.

- **NP**: Class of decision problems whose solutions can be verified in polynomial time by a deterministic Turing machine or solved by a nondeterministic Turing machine in polynomial time.

- **NP-Complete (NPC)**:

  A decision problem is NP-Complete if it satisfies two conditions:

  1. **It belongs to NP:** Any proposed solution can be verified in polynomial time by a deterministic Turing machine.

  2. **It is NP-hard:** Every problem in NP can be reduced to it in polynomial time.

Example:

- **P – Merge Sort:** Sorting a list can be done efficiently in polynomial time, O(nlogn).

- **NP – Hamiltonian Path:** Given a graph, we can quickly verify if a proposed path visits every vertex exactly once; solving takes O(n!) in general, verification = O(n).

- **NP-Complete – Boolean Satisfiability (SAT):** Given a Boolean formula, is there an assignment of true/false to its variables that makes it true? We can verify a satisfying assignment in O(n), but finding one is as hard as any NP problem. SAT was the first proven NP-complete problem. Solving complexity: Brute force is $O(2^n)$; no polynomial-time algorithm is known.

Importance:

P, NP, and NP-Complete form the core of computational complexity. P represents problems that can be solved efficiently, NP represents problems whose solutions can be verified quickly, and NP-Complete represents the hardest problems in NP.