BASIC COMPUTABILITY THEORY

The Concept of Computability

Computability
A foundational concept in computer science that determines whether a problem can be solved using an algorithm, a finite sequence of well-defined steps.

In simple terms, it asks the question: *Is there a method that a computer can follow to always produce a correct answer and eventually stop?* It is not concerned with how *efficiently* a problem is solved but rather whether a solution exists at all. It provides the theoretical foundation for algorithm design, program correctness, and artificial intelligence.

The study of computability is essential because it defines the boundaries of what computers can and cannot do. Before designing or optimizing an algorithm, it must first be established that the problem itself is solvable algorithmically. Computability focuses on the existence of solutions rather than their efficiency.

Key Properties of Computability

- **Definiteness**: Every step in the algorithm must be precisely defined.
- **Finiteness**: The algorithm must terminate after a finite number of steps.
- **Input and Output**: Every algorithm takes input data and produces output results.
- **Effectiveness**: Each step must be basic enough to be carried out by a mechanical process.

Computable and Non-computable
A problem is said to be **computable** if there exists an algorithm that can process any valid input, perform a finite sequence of operations, and produce a correct output in a finite amount of time.

Example: Simple Computable Problem
Checking whether a number is even or odd:

```
Input: 10
Process: 10 mod 2 = 0 → Even
Output: "Even"
```

This can be expressed as an algorithm that always terminates and gives a correct result for any integer input. Hence, the problem is **computable**.

Examples: Real-World Computable Tasks

- Sorting a list of student grades: Can be solved by algorithms like Merge Sort or Quick Sort.
- Searching for a name in a database Can be solved using Binary Search or Hash Tables.
- Encrypting data using an algorithm like AES: Computable because it follows deterministic steps.

Conversely, if no such algorithm exists, the problem is **non-computable**.

Example: Non-Computable Problem
Predicting whether any arbitrary program will run forever or eventually stop (known as the **Halting Problem**) cannot be solved algorithmically for all possible programs. Therefore, it is **non-computable**.

Examples: Real-World Non-Computable Tasks

- Determining whether two (2) arbitrary AI programs will always produce the same result
- Predicting the exact future weather conditions for all time
- Determining the “most beautiful” image: non-computable due to subjectivity and lack of algorithmic definition

The Turing Machine Model

Alan Turing, in 1936, proposed an abstract model of computation known as the **Turing Machine**, a theoretical model capable of simulating any computer algorithm. A function or problem is **Turing-computable** if such a machine can solve it. This concept remains the standard definition of computability in modern computer science.

Thus, any problem that a real computer can solve can also be solved by a Turing Machine.

If a Turing Machine can solve a problem, the problem is **Turing-computable**. If no such machine can exist, the problem is **non-computable**.

Decidable and Undecidable Problems

Computability leads to the classification of problems into two broad categories: **Decidable** and **Undecidable**. This classification helps computer scientists understand whether a problem can be algorithmically determined with a definite answer.

Decidable Problems

A decidable problem is one for which an algorithm exists that can provide a correct "yes" or "no" answer for all inputs and halts after a finite number of steps. These problems are computable because a mechanical procedure can always solve them.

Examples:

- Determining whether a number is even or odd
- Checking if a string is a valid email address
- Determining if a path exists between two (2) nodes in a graph

Expounding on the example of whether a number is even or odd (prime):

Algorithm:

1. Input a number n.
2. Divide n by all integers from 2 to √n.
3. If no divisor is found = "Prime." Otherwise = "Not Prime."

   This algorithm halts for all inputs — therefore, the problem is **decidable**.

Another example of checking balanced parentheses:

Given an expression such as `(a + b) * (c + d)`, an algorithm using a **stack data structure** can verify if every opening parenthesis has a closing match. Since the process always finishes and yields a correct result, the problem is **decidable**.

These problems can always be solved by deterministic algorithms that guarantee termination and correctness.

Undecidable Problems

An undecidable problem is one for which no algorithm can be constructed that always provides a correct "yes" or "no" answer for every input. In other words, there exist cases where the algorithm would either run forever or fail to give a definitive result.

Examples of Undecidable Problems:

- **The Halting Problem**: Determining whether a given program will eventually stop or run indefinitely.

  "Given a program and an input, can we determine whether the program will eventually stop or run forever?"

  Alan Turing proved that there can be no universal algorithm capable of solving this problem for all programs. This means that even with unlimited computing power, certain outcomes cannot be predicted algorithmically.

- **Program Equivalence**: Determining if two (2) arbitrary programs will produce the same output for all inputs.

  "Given two arbitrary programs, can we determine whether they always produce the same output for every input?"

  This is also undecidable because it would require solving the Halting Problem as a substep.

- **Post Correspondence Problem** (PCP): Checking if a sequence of string pairs can be arranged to form matching concatenations.

  Involves determining whether a set of string pairs can be concatenated to form identical strings. There is no general algorithm that can decide this for all cases, making it another **undecidable** problem.

Real-World Example

Suppose you are asked to design an AI model that can automatically determine whether *any given software code* will ever crash or loop infinitely. No such algorithm can exist, because this task is equivalent to solving the **Halting Problem** — hence **undecidable**.

Reductions

Reduction

A method used to prove that a problem is **as hard as** or **harder than** another problem. It is a logical technique that transforms one problem into another in such a way that solving the second problem automatically provides a solution to the first.

In computability theory, reductions are used to:

- Demonstrate the **relationship between problems**
- Prove **undecidability** by reducing a known undecidable problem to a new one.

How Reduction Works

Suppose there are two (2) problems, A and B. If it can be shown that any instance of A can be converted (or reduced) to an instance of B, and if B is already known to be undecidable, then A must also be undecidable.

Formally, a **reduction** converts an instance of one problem (A) into another problem (B) in such a way that solving B provides a solution to A.

Why Reductions Matter

- They provide a proof technique in theoretical computer science
- They show that if one problem is hard or unsolvable, another related one likely is too
- They are also used in complexity theory to identify NP-hard and NP-complete problems

Example:
The **Halting Problem** is often used as a reference for reductions. Many other problems are shown to be undecidable by reducing the Halting Problem to them.

Consider the **Halting Problem (H)** and another problem **P**, which asks:

"Given a program and input, will it ever print the word 'Hello'?"

H can be reduced to **P** by constructing a version of the program that prints "Hello" right before halting. If **P** can be solved, then **H** can also be solved, but since **H** is undecidable, **P** must be undecidable too.

Types of Reductions

Many-One Reduction (Mapping Reduction)
This is the most direct type of reduction. It involves constructing a **computable function** that transforms instances of one problem (A) into instances of another problem (B) such that the answer to both problems remains the same.

It can be said that $\mathbf{A \leq_m B}$ (read as *"A is many-one reducible to B"*) if there exists a computable function **f** such that for every instance *x*:

$$x \in A \iff f(x) \in B$$

This means that solving B is sufficient to solve A.

Purpose:
To prove that problem A is *no harder* than problem B.
If B is known to be undecidable, then A must also be undecidable.

Example:
Let A be the problem "Does program P halt on input x?" (the Halting Problem).
Let B be the problem "Does program P eventually print the word 'STOP'?"

If a computable function can be constructed that converts each instance of A into an instance of B — such that answering B's question also answers A's — then $A \leq_m B$.

Since the Halting Problem is undecidable, B must also be undecidable.

Key Properties:
- The reduction must be computable
- The reduction preserves truth: the answer to A and B must be logically equivalent
- Used primarily to prove undecidability or NP-completeness

Analogy
The analogy here is like transforming a puzzle into another puzzle with the same solution; if the second puzzle cannot be solved, the first cannot either.

Turing Reduction
This allows an algorithm solving one problem (A) to use another problem (B) as a **subroutine** or **oracle**. Unlike many-one reduction, it does not require the entire transformation to be done in one step. Instead, the algorithm solving A can query the "oracle" for B multiple times during its execution.

It can be said that $\mathbf{A \leq_t B}$ (read as *"A is Turing reducible to B"*) if A can be decided by a Turing Machine that has access to an oracle capable of deciding B instantly.

Example:
Suppose access to an oracle that can decide the Halting Problem (even though impossible in reality) exists. This oracle can be used to decide whether a program halts before printing "Hello." In this case, "Halting before printing" is **Turing reducible** to the Halting Problem.

This is the most direct type of reduction. It involves constructing a **computable function** that transforms instances of one problem (A) into instances of another problem (B) such that the answer to both problems remains the same.

Key Properties:

- Allows multiple queries to the oracle for problem B
- More flexible than the many-one reduction
- Used to define **Turing degrees**, a hierarchy that measures the relative difficulty of undecidable problems
- If $A \leq_t B$ and B is decidable, then A is also decidable

Analogy:
A student (Problem A) can ask a teacher (Problem B) questions during an exam. If the teacher can answer instantly, the student can solve their own problem, but the student's ability depends on the teacher's knowledge.

Polynomial-Time Reduction (P-Time Reduction)
In **computational complexity**, where time efficiency matters, **polynomial-time reductions** are used. They show that one problem can be transformed into another in polynomial time — meaning the transformation can be computed efficiently.

$A \leq_p B$ if there exists a polynomial-time computable function **f** that transforms any instance of A into an instance of B such that:

$$x \in A \iff f(x) \in B$$

Purpose:
Used primarily to classify problems as **P**, **NP**, **NP-hard**, or **NP-complete**. If a known NP-hard problem can be reduced to a new problem in polynomial time, the new problem is also NP-hard.

Example:
Reducing the **3-SAT problem** to the **Clique problem** in graph theory:

- The 3-SAT problem involves determining if a Boolean formula in conjunctive normal form (CNF) can be satisfied.
- The Clique problem asks whether a graph contains a complete subgraph of size k. A polynomial-time algorithm can transform an instance of 3-SAT into an instance of Clique. Since 3-SAT is NP-complete, Clique is also NP-complete.

Key Properties:

- Used for **complexity analysis**, not decidability proofs
- Ensures that the reduction process is computationally efficient (runs in polynomial time)
- Forms the basis for NP-completeness proofs

Analogy:
If solving Puzzle A can be quickly turned into solving Puzzle B using minimal effort, and Puzzle B is extremely hard, then Puzzle A is just as hard.

References:

GeeksForGeeks. (2025). *Oracle Turing machine*. Retrieved on October 28, 2025, from https://www.geeksforgeeks.org/theory-of-computation/oracle-turing-machine/

OpenDSA. (n.d.a.). *Reductions*. Retrieved on October 28, 2025, from https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Reduction.html

Oracle. (n.d.a.). *Reductions*. Retrieved on October 28, 2025, from https://docs.oracle.com/en/database/oracle/property-graph/23.2/spgdg/reductions.html