Module 7: Turing Machines and Computability

This module represents a pivotal moment in our exploration of the theoretical limits of computation. Having studied finite automata (with no memory beyond their current state) and pushdown automata (with a single stack), we now introduce the **Turing Machine (TM)**, a theoretical model of computation that is considered the most powerful and general model conceived. The Turing Machine provides a formal, abstract representation of an algorithm and serves as the bedrock for the modern theory of computability. We will meticulously define its components and operations, demonstrate its functionality with solved examples, discuss the profound implications of the **Church-Turing Hypothesis**, and delineate the crucial concepts of decidability and Turing recognizability, which define the boundaries of what computers can and cannot do. Finally, we will examine the closure properties of these language classes, illustrating with further examples where appropriate.

Modeling Computation using Turing Machines (TM)

While finite automata could model simple pattern recognition and pushdown automata could handle hierarchical, nested structures, both possessed inherent limitations related to memory. Finite automata had no auxiliary memory, and pushdown automata were restricted to a single, last-in-first-out (LIFO) stack. These limitations mean they cannot solve problems that require arbitrary amounts of sequential memory or the ability to read and rewrite anywhere in that memory.

The **Turing Machine (TM)**, conceived by Alan Turing in 1936, overcomes these limitations by introducing an infinitely long tape that serves as its memory. This simple yet powerful addition allows the TM to simulate any algorithmic process. It is not intended as a practical model for building computers, but rather as a theoretical abstraction to understand the fundamental capabilities and limitations of computation itself.

A **Turing Machine (TM)** is formally defined as a 7-tuple $(Q,\Sigma,\Gamma,\delta,q0,qaccept,qreject)$, where:

- **Q (States):** A finite, non-empty set of internal states. These states represent the TM's current configuration or phase of computation, similar to states in automata.
    - *Example:* `q_start`, `q_read`, `q_write`, `q_found_match`.
- **Σ (Input Alphabet):** A finite, non-empty set of input symbols. These are the symbols that can appear in the initial input string placed on the tape.
    - *Example:* {0,1}, {a,b,c}. Crucially, the blank symbol _ is *never* part of the input alphabet.
- **Γ (Tape Alphabet):** A finite, non-empty set of symbols that can be written onto the tape. This set includes all input symbols and a special **blank symbol**.
    - *Requirement:* $\Sigma \subseteq \Gamma$.
    - *Special Symbol:* _ (blank symbol). This symbol is a member of Γ but not Σ. It represents an empty cell on the tape. The tape is initially filled with blanks everywhere except where the input string is written.
- **δ (Transition Function):** This is the core of the TM's operation. It dictates the TM's behavior at each step. Unlike DFAs or NFAs, the TM's transition depends on the

current state and the symbol *under the tape head*. For a given (current state, tape symbol under head) pair, it specifies:
    - A new state to transition to.
    - A symbol to write onto the current tape cell (replacing the symbol just read).
    - A direction to move the tape head (Left or Right).
    - Formally, $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
        - L means move the tape head one cell to the left.
        - R means move the tape head one cell to the right.
    - *Deterministic:* This definition describes a **deterministic** Turing Machine. For any given (state, symbol) pair, there is exactly one possible action.
- **q0 (Start State):** The unique initial state from Q where the TM begins its computation.
- **qaccept (Accept State):** A designated state from Q. If the TM enters this state, it immediately halts and accepts the input string.
- **qreject (Reject State):** A designated state from Q. If the TM enters this state, it immediately halts and rejects the input string.
    - *Important:* qaccept and qreject must be distinct states (qaccept=qreject). If a TM reaches either of these states, it halts. If it never reaches either, it runs forever (loops).

Components of a Basic Turing Machine:

1. **Tape:** An infinitely long strip, divided into cells. Each cell can hold exactly one symbol from the tape alphabet Γ. The tape extends infinitely to the right (and often conceptualized as infinite to the left as well, or at least arbitrarily extendable). Initially, the input string occupies the leftmost portion of the tape, and all other cells are filled with the blank symbol _.
2. **Tape Head:** A mechanism that can read a symbol from a cell, write a symbol to a cell, and move left or right one cell at a time. It always points to a single cell on the tape.
3. **Control Unit:** The "brain" of the TM. It is in one of a finite number of states. Based on its current state and the symbol read by the tape head, it consults the transition function δ to decide:
    - What symbol to write onto the tape.
    - What direction to move the tape head.
    - What its next internal state will be.

Basic Operation (Step-by-Step Execution):

1. **Initialization:** The input string is placed on the leftmost portion of the infinite tape. All other tape cells are filled with the blank symbol _. The tape head is positioned at the leftmost symbol of the input string. The control unit is in the start state q0.
2. **Execution Cycle (Loop):** The TM repeatedly performs the following actions:
    - **Read:** The tape head reads the symbol currently in the cell it is pointing to.
    - **Consult Transition Function:** The control unit takes its current state and the symbol just read as input to the transition function δ.

    - **Write, Move, Change State:** Based on the output of δ(current_state,symbol_read)=(new_state,symbol_to_write,direction_to_move):
        - The `symbol_to_write` is written onto the current tape cell.
        - The tape head moves one cell in the specified `direction_to_move` (L or R).
        - The control unit transitions to the `new_state`.
    - **Check for Halting:** If the `new_state` is qaccept or qreject, the TM halts.
3. **Halting Conditions:**
    - **Acceptance:** If the TM enters state qaccept, it halts and the input string is considered **accepted**.
    - **Rejection:** If the TM enters state qreject, it halts and the input string is considered **rejected**.
    - **Looping:** If the TM never enters qaccept or qreject, it continues to run indefinitely (loops). In this case, the input string is neither accepted nor rejected; it simply causes the machine to "hang."

Solved Question 1: TM for L={0n1n∣n≥1}

Problem: Design a Turing Machine that recognizes the language consisting of strings with an equal number of 0s followed by an equal number of 1s, where n≥1. Example strings: `01`, `0011`, `000111`.

Strategy: The TM will repeatedly "check off" a `0` and a `1`.

1. Scan right, find the leftmost `0`. Mark it with an `X`.
2. Scan right past all `0`s and `X`s, find the leftmost `1`. Mark it with a `Y`.
3. If a `0` was found but no `1` (or vice versa), reject.
4. Scan left to find the rightmost `X`. Move one cell right to find the first unmarked symbol.
5. Repeat until all `0`s and `1`s are marked.
6. After marking all `0`s and `1`s, scan the tape to ensure only `X`s, `Y`s, and blanks remain (i.e., no unmatched `0`s or `1`s). If so, accept.

Formal Definition: Q={q0,q1,q2,q3,q4,qaccept,qreject} Σ={0,1} Γ={0,1,X,Y,_} q0: Start state qaccept: Accept state qreject: Reject state

Transition Function δ (Rules):

- **From q0 (Initial state, finding leftmost `0`):**
    - δ(q0,0)=(q1,X,R) : If `0` is read, mark it `X`, move right to find a `1`, go to q1.
    - δ(q0,Y)=(q4,Y,R) : If `Y` is read, it means all `0`s have been matched and marked `X`. Now check if only `Y`s and blanks remain (i.e., balanced). Move right, go to q4.
    - δ(q0,_)=(qreject,_,R) : If blank is read, it implies empty string or no `0`s to start, which is not 0n1n for n≥1. Reject.

    - δ(q0,1)=(qreject,1,R) : Cannot start with 1. Reject.
- **From q1 (Found 0, now looking for 1):**
    - δ(q1,0)=(q1,0,R) : Skip over 0s.
    - δ(q1,Y)=(q1,Y,R) : Skip over already marked 1s (Ys).
    - δ(q1,1)=(q2,Y,L) : If 1 is read, mark it Y, move left to find the X (to return to start of next 0), go to q2.
    - δ(q1,_)=(qreject,_,R) : If blank is read, found 0s but no matching 1s. Reject.
- **From q2 (Found 1, returning to find next 0):**
    - δ(q2,0)=(q2,0,L) : Skip over 0s while moving left.
    - δ(q2,Y)=(q2,Y,L) : Skip over Ys while moving left.
    - δ(q2,X)=(q0,X,R) : If X is read, found the marked 0, move right to start the next iteration (find next unmatched 0). Go to q0.
- **From q3 (Error/Reject state - not explicitly used in this simplified direct path, implies implicit rejection if rule not defined):**
    - (Any other unlisted (state, symbol) pair from q0,q1,q2 implicitly leads to qreject or causes the TM to halt if no transition is defined for that state-symbol pair)
- **From q4 (All 0s matched, verifying no 1s remain unmarked):**
    - δ(q4,Y)=(q4,Y,R) : Skip over Ys (marked 1s).
    - δ(q4,_)=(qaccept,_,R) : If blank is read, means all 0s and 1s were perfectly matched. Accept.
    - δ(q4,0)=(qreject,0,R) : Found an unmatched 0. Reject.
    - δ(q4,1)=(qreject,1,R) : Found an unmatched 1. Reject.

Trace for input 0011: Tape content: _0011___ (head on first 0, state q0)

1. (q0,0)→(q1,X,R) _X011___ (head on second 0, state q1)
2. (q1,0)→(q1,0,R) _X011___ (head on first 1, state q1)
3. (q1,1)→(q2,Y,L) _X0Y1___ (head on 0, state q2)
4. (q2,0)→(q2,0,L) _X0Y1___ (head on X, state q2)
5. (q2,X)→(q0,X,R) _X0Y1___ (head on 0, state q0)
6. (q0,0)→(q1,X,R) _XXY1___ (head on Y, state q1)
7. (q1,Y)→(q1,Y,R) _XXY1___ (head on 1, state q1)
8. (q1,1)→(q2,Y,L) _XXYY___ (head on Y, state q2)
9. (q2,Y)→(q2,Y,L) _XXYY___ (head on X, state q2)
10. (q2,X)→(q0,X,R) _XXYY___ (head on Y, state q0)
11. (q0,Y)→(q4,Y,R) _XXYY___ (head on Y, state q4)
12. (q4,Y)→(q4,Y,R) _XXYY___ (head on _, state q4)
13. (q4,_)→(qaccept,_,R) **Accept!**

This example clearly shows how the TM uses its tape to mark symbols and move back and forth to keep track of its computation, a capability beyond that of PDAs.

Equivalent Models of Computation

The simple, single-tape, deterministic Turing Machine described above is remarkably powerful. So powerful, in fact, that numerous other theoretical models of computation, seemingly more powerful or different in structure, have been shown to be **equivalent** to the basic Turing Machine. This means that any computation that can be performed by one of these alternative models can also be performed by a standard TM, and vice-versa. This robust equivalence lends significant weight to the Turing Machine as the definitive model of general computation.

Here are some common equivalent models and why they don't surpass the basic TM:

1. **Multi-Tape Turing Machine:**
    - **Description:** Instead of one tape, a multi-tape TM has several independent tapes, each with its own read/write head. At each step, the control unit reads symbols from all heads, makes a transition, writes symbols on all tapes, and moves all heads independently.
    - **Equivalence:** A single-tape TM can simulate a multi-tape TM.
        - **Simulation Idea:** The single tape can be thought of as having multiple "tracks" for each of the multi-tape TM's tapes. For example, if a multi-tape TM has k tapes, the single tape could be divided into 2k tracks: k tracks for the content of each tape, and k tracks to mark the head positions on each of those k tapes.
        - To simulate a step: The single-tape TM scans its tape from left to right, remembering the symbols under each of the k simulated heads and their positions (stored on a separate "head position" track or by marking). Once all k symbols are read, the single-tape TM determines the multi-tape TM's next state, symbols to write, and head movements. Then, it makes another pass (or multiple passes) over its single tape to update the symbols on the respective content tracks and move the head markers according to the multi-tape TM's rules. While this simulation is slower (polynomially slower, but not fundamentally less powerful), it proves equivalence.
2. **Multi-Track Turing Machine:**
    - **Description:** A multi-track TM has a single tape, but each tape cell is divided into several "tracks" or channels. The tape head reads/writes all symbols on all tracks simultaneously at a given cell.
    - **Equivalence:** This is trivially equivalent to a standard single-tape TM.
        - **Simulation Idea:** If a multi-track TM has k tracks, a single-tape TM can treat each symbol on a tape cell as an ordered k-tuple of symbols. For example, if a multi-track TM has symbols `(a, x, P)` on a cell, the single-tape TM's tape alphabet can simply include `(a, x, P)` as a single composite symbol. The transition function is then defined over these composite symbols. No change in fundamental power.
3. **Non-Deterministic Turing Machine (NTM):**
    - **Description:** Unlike a deterministic TM, an NTM's transition function δ can specify multiple possible next configurations for a given (state, symbol) pair. If multiple choices exist, the NTM "forks" into parallel computational paths, exploring all possibilities simultaneously. An NTM accepts if at least one of its computational paths leads to an accept state.

    - **Equivalence:** A deterministic single-tape TM can simulate an NTM.
        - **Simulation Idea:** The simulating DTM systematically explores all possible computation paths of the NTM using a breadth-first search (BFS) strategy. It can use a multi-tape setup (which we know is equivalent to a single-tape TM): one tape for the original input, one tape to store the current configuration of the NTM being simulated, and one tape to store a "list" of possible choices made so far (or alternative configurations to explore). It systematically tries all branches until it finds an accepting path or exhausts all possibilities. While exponentially slower in worst-case time, it can still *simulate* any NTM computation, demonstrating equivalence in terms of *what can be computed*, not necessarily *how fast*. This is a crucial result, as it shows non-determinism does not increase the power of TMs (unlike with finite automata where NFA > DFA in terms of design convenience, but not theoretical power, and with pushdown automata where non-determinism *does* increase power).
4. **Turing Machines with Stay-Option:**
    - **Description:** The tape head can move Left, Right, or Stay (S) at the current cell.
    - **Equivalence:** Easily simulated by a standard TM.
        - **Simulation Idea:** A "Stay" move can be simulated by two moves: one move right (R) and then one move left (L). The symbols written/read remain consistent. So, a transition $\delta(q,a)=(q',b,S)$ can be replaced by two transitions in the standard TM: $\delta(q,a)=(q_{intermediate},b,R)$ and $\delta(q_{intermediate},X)=(q',X,L)$ for all $X \in \Gamma$.
5. **Turing Machines with Semi-Infinite Tape:**
    - **Description:** The tape extends infinitely only to the right, having a fixed leftmost cell.
    - **Equivalence:** A two-way infinite tape TM can be simulated by a semi-infinite tape TM.
        - **Simulation Idea:** A semi-infinite tape TM can divide its single tape into two conceptual tracks. One track simulates the original TM's right half of the tape, and the other track simulates the original TM's left half (but reversed, so the leftmost cell of the original left half is now on the right side of the track, closest to the "fold"). The head of the semi-infinite tape TM then needs to be able to "jump" between these two tracks to simulate movement across the conceptual mid-point. Each symbol in the semi-infinite tape's alphabet would be an ordered pair of symbols, one for each track. When the original TM would move left from its initial position, the simulating TM moves to the right on its second track (the reversed left half).

These equivalences are profoundly important because they suggest that the concept of "computability" is robust and independent of minor variations in the computational model. They all converge on the same set of computable functions.

Church-Turing Hypothesis

The **Church-Turing Hypothesis** (also known as the Church-Turing Thesis) is a fundamental, widely accepted, but unprovable assertion at the heart of computer science and mathematics. It provides the crucial link between the informal, intuitive notion of an "algorithm" or "effective procedure" and the formal, mathematical model of a Turing Machine.

The Hypothesis States:

"Any function that can be computed by an algorithm (an effective procedure) can be computed by a Turing Machine."

What it means:

- **Formalizing "Algorithm":** Before Turing Machines, the concept of an "algorithm" was intuitive but lacked a precise mathematical definition. Turing's work, along with independent work by Alonzo Church on lambda calculus (another equivalent computational model), provided this formal definition. The hypothesis proposes that the Turing Machine (or any of its equivalent models) perfectly captures what an "algorithm" truly is.
- **The Limit of Computability:** If the Church-Turing Hypothesis is true (and all evidence strongly suggests it is), then the capabilities of a Turing Machine define the ultimate limits of what can be computed by *any* form of computation, whether by a human following a step-by-step procedure, a mechanical device, or any future supercomputer. No matter how clever an algorithm you devise, if it cannot be simulated by a Turing Machine, then it is not truly an algorithm in the sense of being effectively computable.
- **Universality:** This hypothesis supports the idea of universal computation. If a Turing Machine can simulate any algorithm, then a Universal Turing Machine (a TM that can simulate any other TM given its description as input) can, in principle, compute anything that any computer can compute. This is the theoretical basis for modern programmable computers.
- **Unprovable Nature:** The Church-Turing Hypothesis is a hypothesis, not a theorem, because the informal concept of "algorithm" cannot be mathematically defined. It's an assertion that a formal model (Turing Machine) accurately captures an intuitive concept (algorithm). We cannot logically prove it, but we can gather overwhelming evidence for it by showing that all other proposed models of computation (lambda calculus, recursive functions, random access machines, cellular automata, quantum computers, etc.) are equivalent to Turing Machines.

Implications of the Church-Turing Hypothesis:

- **Foundation of Computer Science:** It underpins the entire field of theoretical computer science. When we talk about what is "computable" or "uncomputable," we are implicitly referring to what a Turing Machine can or cannot do.
- **Impossibility Results:** If a problem can be formally proven to be unsolvable by a Turing Machine (e.g., the Halting Problem, which asks whether an arbitrary program will halt on a given input), then the Church-Turing Hypothesis implies that *no algorithm whatsoever* can solve that problem, regardless of how powerful future computers become. This is the basis for proving absolute limits on computation.

- **Practical Equivalence:** It explains why all general-purpose programming languages and computing machines are fundamentally equivalent in terms of what they can compute (though they differ vastly in efficiency, ease of programming, etc.). A program written in Python, C++, or Java can, in principle, be translated into a set of Turing Machine instructions and simulated by a TM, and vice-versa.

The Church-Turing Hypothesis is a cornerstone that defines the very scope of computability, allowing us to reason rigorously about what is computable and what is inherently beyond the reach of any algorithm.

Decidability and Turing Recognizability

The power of Turing Machines allows us to classify problems (or more precisely, the languages that represent those problems) into different categories based on whether a Turing Machine can solve them and how it solves them. This leads to the fundamental concepts of decidability and Turing recognizability.

We consider a **language L** to be a set of strings. A problem can be framed as a language recognition task: "Does this input string belong to the set of strings defined by the problem?"

1. **Turing Recognizable Languages (Recursively Enumerable Languages / RE):**
    - A language L is **Turing-recognizable** (also known as **recursively enumerable** or **RE**) if there exists a Turing Machine M such that for any input string w:
        - If $w \in L$, then M halts in the qaccept state.
        - If $w \in /L$, then M either halts in the qreject state *or* runs forever (loops).
    - **Intuition:** For strings in the language, the TM will eventually say "yes" (accept). For strings not in the language, it might say "no" (reject), or it might never give an answer. This means we can list or "enumerate" all strings in the language by systematically running the TM on all possible inputs.
    - *Analogy:* You send a detective to find a specific person. If the person exists, the detective will eventually find them and report "found." If the person doesn't exist, the detective might report "not found" (reject), or they might search forever without success (loop).
2. **Decidable Languages (Recursive Languages / R):**
    - A language L is **decidable** (also known as **recursive** or **R**) if there exists a Turing Machine M such that for any input string w:
        - If $w \in L$, then M halts in the qaccept state.
        - If $w \in /L$, then M halts in the qreject state.
    - **Crucial Difference:** A decidable language requires a Turing Machine that *always halts* for every input, whether the input is in the language or not. It provides a definite "yes" or "no" answer for every single input string.
    - **Relationship:** Every decidable language is also Turing-recognizable. (If a TM always halts, it certainly halts when it accepts.) However, not all Turing-recognizable languages are decidable.
    - *Analogy:* You ask a program to check if a number is prime. For any number, the program will eventually give you a definitive "yes, it's prime" or "no, it's not prime."

Key Differences and Importance:

- **Halting:** This is the core distinction. For decidable languages, the TM is guaranteed to halt. For Turing-recognizable languages, the TM is only guaranteed to halt (and accept) if the string is in the language; otherwise, it might loop.
- **Algorithm vs. Procedure:** A problem is **decidable** if there is an **algorithm** that solves it (always halts). A problem is **Turing-recognizable** if there is a **procedure** that solves it (may not halt for inputs not in the language). The term "algorithm" in computer science usually implies a guaranteed halt.
- **Practical Implications:**
    - **Decidable problems** are those that computers can genuinely "solve" in a finite amount of time for all valid inputs. Many practical problems in computer science fall into this category (e.g., checking if a string matches a regular expression, type-checking in a compiler, sorting an array).
    - **Turing-recognizable problems** are those for which we can write a program that will confirm "yes" if the answer is "yes," but might get stuck if the answer is "no." This is less ideal, but still represents a form of computability. The classic example of a Turing-recognizable but undecidable language is the **Halting Problem**. We can write a TM that halts and accepts if a given TM halts on a given input, but we cannot write one that always halts and rejects if it doesn't halt.
- **Unrecognizable Languages:** There exist languages that are not even Turing-recognizable. This means there's no Turing Machine at all that can even reliably say "yes" for strings in the language. These problems are considered fundamentally uncomputable.

The hierarchy of languages, based on the power of the recognizing machine, places these concepts in context:

- Regular Languages (recognized by Finite Automata) $\subset$ Context-Free Languages (recognized by Pushdown Automata) $\subset$ Decidable Languages (recognized by TMs that always halt) $\subset$ Turing-Recognizable Languages (recognized by TMs that may loop).

Solved Question 2: Decidability of the Language ADFA

Problem: Show that the language $ADFA=\{\langle D,w\rangle | D \text{ is a DFA and } D \text{ accepts } w\}$ is decidable. (Here, $\langle D,w\rangle$ represents an encoding of a DFA D and an input string w as a single string.)

Solution: To show that ADFA is decidable, we need to construct a Turing Machine M that decides ADFA. This means M must accept if D accepts w, and M must reject if D does not accept w, and importantly, M must **always halt** for any input $\langle D,w\rangle$.

Construction of Turing Machine M for ADFA:

The TM M will simulate the behavior of the given DFA D on the input string w.

1. **Input Setup:** M receives its input as a single string $\langle D,w\rangle$. This means D (its states, alphabet, transitions, start state, accept states) and w are all encoded as symbols on

M's tape. M can use multiple tracks or work tapes to conceptually separate D's description from w and its current state.

2. **Simulation Steps:**
    - **Initialize DFA's State:** M records the current state of D. Initially, this is D's start state q0.
    - **Initialize DFA's Input Pointer:** M keeps track of the current position in w that D is reading. Initially, this is the first symbol of w.
    - **Loop for each symbol in w:** For each symbol in w (from left to right):
        1. M reads the current symbol from w (at the current input pointer).
        2. M looks up the transition in D's description: $\delta D(\text{current\_state\_of\_D}, \text{current\_symbol\_of\_w})$.
        3. M updates D's current state to the new state specified by δD.
        4. M advances the input pointer to the next symbol in w.
    - **Handle DFA halting (end of input):** After M has processed all symbols in w:
        1. M checks if the current state of D (after reading all of w) is one of D's accept states.
        2. If it is an accept state, M enters its own qaccept state and halts.
        3. If it is not an accept state, M enters its own qreject state and halts.

Why M always halts:

- **Finite Number of States:** The DFA D has a finite number of states.
- **Finite Input String:** The input string w has a finite length.
- **Deterministic Transitions:** DFA D's transitions are deterministic; for each (state, symbol) pair, there is exactly one next state.
- **Fixed Number of Steps:** M simulates exactly one step of D for each symbol in w. Since w has a finite length, M will perform a finite number of simulation steps.
- **Guaranteed Halting:** Once all symbols of w are processed, M makes a final decision based on D's final state and halts. There is no possibility of M looping indefinitely in this simulation.

Therefore, since M correctly decides whether D accepts w and always halts, ADFA is a decidable language. This example highlights how TMs can "understand" and execute the rules of other computational models.

Closure Properties of Decidable and Recognizable Languages

Just as with regular and context-free languages, understanding the closure properties of decidable and Turing-recognizable languages helps us understand their structural behavior and limitations.

1. Closure Properties of Decidable Languages (Recursive Languages / R):

Decidable languages are robust and closed under most common set operations. This means that if you perform these operations on languages that are known to be decidable, the resulting language will also be decidable.

- **Union:** If L1 and L2 are decidable languages, then $L1 \cup L2$ is decidable.

    - **Proof Idea (Construction):** Given DTM M1 for L1 and DTM M2 for L2 (both always halt). Construct a new DTM Munion that on input w:
        1. Simulate M1 on w.
        2. If M1 accepts w, then Munion accepts w. (Halt)
        3. If M1 rejects w, then Munion simulates M2 on w.
        4. If M2 accepts w, Munion accepts. (Halt)
        5. If M2 rejects w, Munion rejects. (Halt) Since M1 and M2 are guaranteed to halt for all inputs, Munion will always halt and correctly decide whether w is in L1 or L2 (or both).
- **Intersection:** If L1 and L2 are decidable languages, then $L1 \cap L2$ is decidable.
    - **Proof Idea (Construction):** Given DTM M1 for L1 and DTM M2 for L2. Construct a new DTM Mintersection that on input w:
        1. Simulate M1 on w.
        2. If M1 rejects w, then Mintersection rejects w. (Halt)
        3. If M1 accepts w, then Mintersection simulates M2 on w.
        4. If M2 accepts w, Mintersection accepts. (Halt)
        5. If M2 rejects w, Mintersection rejects. (Halt) Since M1 and M2 always halt, Mintersection will always halt and correctly decide $L1 \cap L2$.
- **Complement:** If L is a decidable language, then its complement $\overline{L}$ is decidable.
    - **Proof Idea (Construction):** Given DTM M for L. Construct a new DTM Mcomplement that on input w:
        1. Simulate M on w.
        2. If M accepts w, Mcomplement changes its state to qreject and halts.
        3. If M rejects w, Mcomplement changes its state to qaccept and halts. Since M is a decider, it always halts. Therefore, Mcomplement will always halt and correctly decide L. This is a very strong property, reflecting the power of decidable languages.
- **Concatenation:** If L1 and L2 are decidable languages, then L1L2 is decidable.
    - **Proof Idea (Construction):** Construct a DTM Mconcat that on input w:
        1. If w is the empty string, check if $\epsilon \in L1L2$. This is decidable by running M1 on ϵ and M2 on ϵ. If M1 accepts ϵ and M2 accepts ϵ, then ϵ is in L1L2. Accept or reject accordingly.
        2. If w is non-empty, systematically try all possible ways to split w into two substrings, w=xy, where x and y can be empty.
        3. For each possible split (x,y):
            - Simulate M1 on x.
            - If M1 accepts x:
                - Then simulate M2 on y.
                - If M2 also accepts y, then Mconcat accepts w and halts.
        4. If all possible splits (x,y) have been tried, and no combination of M1 accepting x and M2 accepting y was found, then Mconcat rejects w and halts. Since w has a finite length, there are a finite number of ways to split it. Since M1 and M2 are deciders, they always halt on any input string (including empty strings). Therefore, Mconcat will always halt.
- **Kleene Star:** If L is a decidable language, then $L*$ is decidable.
    - **Proof Idea (Construction):** Construct a DTM Mstar that on input w:
        1. If $w=\epsilon$, accept (as ϵ is always in $L*$ by definition).

2. If w is non-empty, systematically try all possible ways to partition w into k non-empty substrings for all k from 1 to $|w|$: w=w1w2…wk.
3. For each partition:
    - Simulate M (the decider for L) on each wi.
    - If M accepts all wi in that partition, then Mstar accepts w and halts.
4. If all possible partitions have been tried and none resulted in all substrings being accepted by M, then Mstar rejects w and halts. Since w has a finite length, there are a finite number of ways to partition it. Since M is a decider, it always halts. Therefore, Mstar will always halt.

2. Closure Properties of Turing Recognizable Languages (Recursively Enumerable Languages / RE):

Turing-recognizable languages are also closed under several common operations, though not as comprehensively as decidable languages. The key challenge here is that the recognizing TM might loop for non-members.

- **Union:** If L1 and L2 are Turing-recognizable, then $L1 \cup L2$ is Turing-recognizable.
    - **Proof Idea (Construction):** Given TM M1 for L1 and TM M2 for L2 (which may loop). Construct a new TM Munion that on input w:
        1. **Simulate M1 on w and M2 on w in parallel.** This can be done by a multi-tape TM that alternates steps between M1's simulation and M2's simulation. For example, Munion would simulate one step of M1, then one step of M2, then two steps of M1, then two steps of M2, and so on (or more simply, one step of M1, then one step of M2, then one step of M1, etc.). This ensures both machines make progress.
        2. If either M1 *accepts* w or M2 *accepts* w at any point, then Munion immediately accepts w and halts. If $w \in L1 \cup L2$, then w is in at least one of the languages. The corresponding TM (M1 or M2) will eventually accept, causing Munion to accept. If $w \notin L1 \cup L2$, both M1 and M2 would either reject or loop, causing Munion to reject or loop.
- **Intersection:** If L1 and L2 are Turing-recognizable, then $L1 \cap L2$ is Turing-recognizable.
    - **Proof Idea (Construction):** Construct a new TM Mintersection that on input w:
        1. Simulate M1 on w.
        2. If M1 accepts w, then simulate M2 on w.
        3. If M2 also accepts w, then Mintersection accepts w and halts. If $w \in L1 \cap L2$, then M1 will eventually accept and halt, allowing M2 to run. Then M2 will also eventually accept and halt. So Mintersection accepts. If $w \notin L1 \cap L2$:
            - If $w \notin L1$, M1 will either reject or loop, so Mintersection will also reject or loop.
            - If $w \in L1$ but $w \notin L2$, M1 will accept, but M2 will reject or loop, so Mintersection will also reject or loop. This sequential approach works for intersection because both machines must accept.

- **Concatenation:** If L1 and L2 are Turing-recognizable, then L1L2 is Turing-recognizable.
    - **Proof Idea (Construction):** Construct a TM Mconcat that on input w:
        1. Systematically generate all possible ways to split w into two substrings x and y such that w=xy. (There are $|w|+1$ such splits, including empty x or y).
        2. For each split (x,y):
            - Simulate M1 on x and M2 on y **in parallel** (interleaving their execution steps).
            - If *both* M1 on x and M2 on y accept, then Mconcat accepts w and halts.
        3. If no such split leads to acceptance after trying all possibilities up to a certain simulation depth, Mconcat continues searching (by trying more simulation steps on the current splits, or moving to the next split if current ones rejected/looped). If $w \in L1L2$, a correct split will eventually be found and accepted by the parallel simulation. If $w \in /L1L2$, Mconcat may loop or reject.
- **Kleene Star:** If L is a Turing-recognizable language, then $L*$ is Turing-recognizable.
    - **Proof Idea (Construction):** Construct a TM Mstar that on input w:
        1. If $w=\epsilon$, accept.
        2. If w is non-empty, systematically generate all possible partitions of w into k non-empty substrings: $w=w1w2\ldots wk$, for k from 1 to $|w|$.
        3. For each partition:
            - Simulate M (the recognizer for L) on all wi simultaneously *in parallel* (interleaving their execution steps across multiple tapes).
            - If M accepts all wi for a given partition, then Mstar accepts w and halts. If $w \in L*$, a correct partition will eventually be found and verified by the parallel simulation, causing acceptance.

Non-Closure of Turing Recognizable Languages under Complement:

- **Complement:** If L is a Turing-recognizable language, its complement L is **not necessarily** Turing-recognizable.
    - **Crucial Result:** This is a fundamental result in computability theory. A language L is **decidable if and only if** both L AND its complement L are Turing-recognizable.
    - **Proof Idea (Demonstration):**
        - **Part 1: If L is decidable, then L and L are Turing-recognizable.** (This is straightforward: A decider for L is also a recognizer for L. And a decider for L can be easily modified to be a decider for L by swapping accept/reject states, making L also recognizable).
        - **Part 2: If L and L are Turing-recognizable, then L is decidable.**
            - Assume L is Turing-recognizable by M1, and L is Turing-recognizable by M2.
            - Construct a new TM Mdecider that on input w:
                - Simulate M1 on w and M2 on w **in parallel** (e.g., alternating one step of M1, then one step of M2).

        - If M1 accepts w, then Mdecider accepts w and halts.
        - If M2 accepts w, then Mdecider rejects w and halts.
      - **Why it works and always halts:** For any input string w, it must either be in L or in L.
        - If $w \in L$, then M1 will eventually accept. Mdecider will then accept and halt.
        - If $w \in L$, then M2 will eventually accept. Mdecider will then reject and halt.
      - Since one of M1 or M2 is guaranteed to accept (and halt) for any input w, Mdecider is guaranteed to halt for any input w. Thus, Mdecider is a decider for L, proving L is decidable.
  - **Consequence:** Since we know there exist Turing-recognizable languages that are *not* decidable (e.g., the Halting Problem, or any other undecidable problem that is proven to be RE), it logically follows from the above result that their complements cannot be Turing-recognizable. If they were, those languages would be decidable, which contradicts their known undecidability.

This distinction in closure under complement is a cornerstone of understanding the hierarchy of undecidable problems and the fundamental limits of computation. It means that for some problems, we can write a program that confirms "yes" answers, but we can't write a program that always confirms "no" answers.

Solved Question 3: The Halting Problem (Conceptual)

Problem: Define the Halting Problem and explain why it is undecidable.

Solution:

The **Halting Problem** is one of the most famous and fundamental problems in computer science, proven to be undecidable by Alan Turing in 1936.

Definition: The Halting Problem (denoted HALTTM) is the problem of determining, for an arbitrary Turing Machine M and an arbitrary input string w, whether M will halt (i.e., eventually stop, either accepting or rejecting) when run with input w.

Formally, the language representing the Halting Problem is: $HALT_{TM} = \{\langle M,w\rangle \mid M \text{ is a Turing Machine and } M \text{ halts on input } w\}$

Why it is Undecidable (Proof by Contradiction Sketch):

Assume, for the sake of contradiction, that the Halting Problem *is* decidable. This means there exists a Turing Machine, let's call it H, that decides $HALT_{TM}$. So, for any input $\langle M,w\rangle$:

- If M halts on w, H accepts $\langle M,w\rangle$.
- If M does not halt on w, H rejects $\langle M,w\rangle$. Crucially, H is a decider, so it *always halts*.

Now, let's construct a new Turing Machine, D, using H as a subroutine. **Turing Machine D operates as follows on input $\langle M\rangle$ (a description of a Turing Machine M):**

1. D receives input $\langle M\rangle$.

2. D creates a new input pair ⟨M,⟨M⟩⟩. (This means M will be run on its *own* description as input.)
3. D then simulates H on this new input ⟨M,⟨M⟩⟩.
4. Based on H's output:
    - If H **accepts** ⟨M,⟨M⟩⟩ (meaning M halts on ⟨M⟩), then D enters a state where it **loops forever**.
    - If H **rejects** ⟨M,⟨M⟩⟩ (meaning M does not halt on ⟨M⟩), then D enters qaccept and **halts and accepts**.

Now, let's analyze what happens when D is run on its *own* description, ⟨D⟩:

Consider D(⟨D⟩):

- **Case 1: Assume D halts on input ⟨D⟩.**
    - According to D's definition, if D halts on ⟨D⟩, it must be because H *rejected* ⟨D,⟨D⟩⟩.
    - But H rejecting ⟨D,⟨D⟩⟩ implies (by H's definition) that D *does not halt* on ⟨D⟩.
    - This is a contradiction: D halts AND D does not halt.
- **Case 2: Assume D does not halt (loops) on input ⟨D⟩.**
    - According to D's definition, if D loops on ⟨D⟩, it must be because H *accepted* ⟨D,⟨D⟩⟩.
    - But H accepting ⟨D,⟨D⟩⟩ implies (by H's definition) that D *halts* on ⟨D⟩.
    - This is a contradiction: D loops AND D halts.

In both cases, we reach a logical contradiction. Since our initial assumption (that HALTTM is decidable, meaning H exists) leads to a contradiction, that assumption must be false.

Conclusion: Therefore, the Halting Problem is **undecidable**. No algorithm, no matter how clever or powerful, can reliably determine whether an arbitrary program will halt on an arbitrary input. This is a fundamental limit of computation.