

C.A.T 2

ICS 3103 Automata Theory And Computability

1. **Turing Machines**: Introduction to Turing Machine, Formal Description, Instantaneous description, The language of a Turing machine

**Types of Turing machine:** Turing machines and halting

A **Turing Machine**, often called a TM, is a theoretical model of computation that was introduced by **Alan Turing in 1936**.

Its main purpose is to help us understand **what problems can be solved using algorithms**. In other words, it gives us a formal way to define what a computer can and cannot do.

Turing Machines are very important in **computability theory** and **automata theory** because they represent the most powerful kind of automaton we study in this course.

They are more powerful than **finite automata** or **pushdown automata**, because they can simulate any computation that a real computer can perform — as long as there is enough time and memory.

Even though it's a theoretical model, the Turing Machine helps us answer deep questions in computer science — like what is computable, and what the limits of computation are."

**Formal Description**

A Turing Machine is defined as a 7-tuple:  
**M = (Q, Σ, Γ, δ, q₀, q-accept, q-reject)**

|  |  |
| --- | --- |
| **Component** | **Description** |
| **Q** | **Finite set of states** |
| **Σ (Sigma)** | **Input alphabet (does not include the blank symbol □)** |
| **Γ (Gamma)** | **Tape alphabet (includes □ and all symbols in Σ)** |
| **δ (delta)** | **Transition function: Q × Γ → Q × Γ × {L, R}** |
| **q₀** | **Start state (q₀ ∈ Q)** |
| **q-accept** | **Accepting state (q\_accept ∈ Q)** |
| **q-reject** | **Rejecting state (q\_reject ∈ Q, and q\_accept ≠ q\_reject)** |

**Simple Example:**

**"Let me now explain a simple example of a Turing Machine that accepts strings of the form aⁿbⁿ, like ab, aabb, or aaabbb.  
These are strings where the number of as is equal to the number of bs, and all as come before the bs.**

**So, what is the Turing Machine actually doing?**

**First, remember that a Turing Machine has three key parts:**

1. **A tape, which is like a long strip of paper with symbols on it**
2. **A head, which reads and writes symbols and can move left or right**
3. **A set of rules or states that tell it what action to take based on what it reads**

**In this example, our machine has:**

* **A set of states: Q = {q₀, q₁, q₂, q\_accept, q\_reject}**
* **An input alphabet: Σ = {a, b}**
* **A tape alphabet: Γ = {a, b, X, Y, □}**
  + **Where X and Y are used as markers for a and b that have already been processed**
  + **And □ is a blank symbol that appears at the end of the tape**

**The machine starts in state q₀ and follows a set of instructions or transitions:**

**Step 1:  
If the machine is in state q₀ and sees an a,**

* **it replaces it with X (to mark it as used),**
* **moves one step to the right,**
* **and goes to state q₁.**

**We write this as:  
δ(q₀, a) = (q₁, X, R)**

**Step 2:  
Now, in state q₁, the machine moves right until it sees a b.**

* **It then replaces that b with Y,**
* **moves left,**
* **and goes to state q₂.**

**We write this as:  
δ(q₁, b) = (q₂, Y, L)**

**In state q₂, the machine moves back to the beginning of the tape to find the next unmatched a, and the process repeats.**

**What this machine is doing is:**

* **Matching each a with one b**
* **Marking matched letters with X and Y so it doesn’t reuse them**
* **Moving back and forth on the tape until all the as and bs are processed**

**Finally, the machine accepts the string if only Xs and Ys are left on the tape, which means every a had a matching b.**

**If there’s an unmatched symbol, or something out of order, the machine goes to the reject state.**

**Instantaneous Description (ID)**

**"Now I’ll explain something called the Instantaneous Description, or simply ID.  
This is used to show the current status of a Turing Machine during computation.**

**An ID includes three main things:**

1. **The current state the machine is in**
2. **The content of the tape**
3. **The position of the read/write head**

**The notation we use is:  
u q v, where:**

* ***u* is the portion of the tape to the left of the head,**
* ***q* is the current state,**
* ***v* is the symbol under the head and the rest of the tape to the right.**

**For example, let’s say the tape has: a a b □ □ …,  
and the head is currently reading the first b, while the machine is in state q₀.  
The ID would be written as: aa q₀ b□□.**

**This format helps us trace step-by-step what the machine is doing at any point in time."**

**✅ Language of a Turing Machine**

**"Now let’s talk about the language of a Turing Machine.**

**A Turing Machine accepts a string if, after processing it, the machine enters the accepting state, which we call q\_accept.**

**So, the language accepted by a Turing Machine is the set of all strings that cause it to eventually reach that accepting state.**

**This kind of language is called recursively enumerable, or RE.  
That means the machine will halt and accept valid strings, but for strings not in the language, it might either reject them or loop forever.**

**Let me give you an example.**

**Consider the language L = {aⁿbⁿ | n ≥ 1} — this includes strings like ab, aabb, aaabbb, and so on, where the number of a’s is equal to the number of b’s and all a’s come before b’s.**

**A Turing Machine that accepts this language would work like this:**

* **It scans the tape and replaces the first a with an X.**
* **Then it moves right to find the first b, and replaces it with a Y.**
* **It then goes back to the left and repeats the process.**

**If all a’s and b’s are matched properly, and no unmatched symbols remain, the machine reaches the accepting state.**

**This shows how Turing Machines can handle more complex patterns than simpler machines like finite automata or pushdown automata."**

**Types of Turing Machines**

**1. Deterministic Turing Machine (DTM)**

A DTM follows **only one path** for each step. It knows exactly what to do based on the current state and tape symbol.  
📌 *Use case:* It can check if a binary string is a **palindrome** by matching characters from both ends one by one.

**2. Non-deterministic Turing Machine (NTM)**

An NTM can take **multiple paths** from the same point. It explores all possibilities in parallel and accepts if **any path works**.  
📌 *Use case:* Solving puzzles where the machine tries all possible moves to see if it reaches a solution.

**3. Multi-tape Turing Machine**

This type uses **multiple tapes** with separate heads, allowing it to process more data at once and speed up operations.  
📌 *Use case:* It can compare two strings (like abc and abc) **faster** by reading both simultaneously from two tapes.

**4. Multi-track Turing Machine**

A Multi-track machine has **one tape**, but each cell holds **multiple symbols** on separate tracks.  
📌 *Use case:* It can track an input symbol and a marker together, which helps avoid moving back and forth across the tape.

**5. Universal Turing Machine**

A Universal TM can **simulate any other TM** by reading its description from the tape — just like a modern computer runs programs.  
📌 *Use case:* It can load and run different algorithms, making it a model of **programmable computation**.

**HALTING PROBLEM**

* "Now let’s talk about the **halting problem**, which is one of the most important concepts in the theory of computation.
* The halting problem asks a simple question:  
  If we give a Turing Machine some input, can we tell whether it will eventually stop or keep running forever?
* For example, imagine a machine T that is given another machine’s description and an input — like a program and its input. We ask: Will this program stop, or will it get stuck in an infinite loop?
* Alan Turing proved that **no machine** — not even a Turing Machine — can always answer that question correctly for all possible cases. This means that **no algorithm exists** that can check whether every program will halt.
* So, the halting problem is what we call **undecidable**. We can solve it in some cases, but there’s no general solution that works for all programs and all inputs.
* This is a big deal because it shows us that **computation has limits**. There are some questions that computers just can’t answer — no matter how powerful they are."