Algorithms

From Intuition to Turing Machines

What exactly is an algorithm?

Three Views of "Algorithm"

1. Intuitive Notion

A method to solve a problem

- Independent of programming language
- A general approach or strategy

2. Computer Science

An effective problem-solving method suitable for computer implementation

- Examples: sorting, searching, expression evaluation

3. Theory of Computation

A specific Turing Machine

Wait... What? 🤔

An algorithm is a Turing Machine?

This seems strange at first!

But remember:

- We need a **formal, mathematical** definition
- TMs provide **precise semantics**
- Enables **rigorous proofs** about what's computable

Let's build up to this definition...

Decidability: Three Possible Outcomes

When a TM processes an input string, three things can happen:

1. Accept ✓

- Reaches an **accept state**
- Computation halts
- Input is "in the language"

2. Reject ✕

- Reaches a **reject state**
- Computation halts
- Input is "not in the language"

3. Loop Forever ∞

- Never reaches accept or reject
- Computation **never halts**

Recognizing vs. Deciding

Recognizing a Language

> A TM **recognizes** language L if it accepts all strings in L

For strings NOT in L:

- May reject, OR
- May loop forever

Deciding a Language

> A TM **decides** language L if:

- It **accepts** all strings in L
- It **rejects** all strings not in L

Key difference: Always halts!

Example: The Incrementer TM

Language: All binary strings

Behavior:

- Input: any binary number
- Output: incremented binary number
- **Always halts** in accept state

Therefore: This TM **decides** the language of binary strings

Example: The Adder TM

Two versions possible:

With Reject State

- Valid inputs: `101+10`
- Invalid inputs: `101++10`
- **Rejects** invalid format
- **Decides** the language

Without Reject State

- Valid inputs: accepted
- Invalid inputs: may loop forever
  - Example: `101++10` loops in scan state
- **Only recognizes** the language

DFAs vs. TMs

Important distinction:

DFAs

- Always consume entire input
- Halt when input exhausted
- No need to distinguish "recognize" from "decide"
- If not in accept state → implicit reject

TMs

- Can loop indefinitely
- May never finish processing
- **Must** distinguish recognizing from deciding
- Need explicit reject states for decision

Quick Poll

Which of these always DECIDES its language?

A) Any DFA

B) Any NFA

C) Any TM

D) Both A and B

Poll Answer

Correct: D (Both A and B)

Explanation:

- DFAs always halt when input ends
- NFAs also halt when input ends
- Both implicitly reject non-accepted strings
- TMs might loop forever → may only recognize

Key insight: Finite automata are always decidable; TMs may not be!

Computability: Function Computation

So far: recognizing/deciding **languages** (sets of strings)

Now: computing **functions** (input → output)

Setup

- Input string **x** on tape
- TM processes
- When TM halts, tape contains **f(x)**

Definition

> Function f(x) is **computable** if there exists a TM that:

- Takes x as input
- Halts with f(x) on tape
- Always halts (for all valid inputs)

Computable Function Examples

Incrementer

- **Input:** binary number n
- **Output:** n + 1
- **Always halts:** ✓
- **Therefore:** Computable

Adder

- **Input:** $n_1 + n_2$
- **Output:** $n_1 + n_2$ (sum)
- **Always halts:** ✓
- **Therefore:** Computable

Pattern: Most "reasonable" mathematical functions are computable

The Formal Algorithm Definition

> A TM that **decides** some language or **computes** some function represents an **algorithm** for that task

What this means:

- Algorithm = TM that always halts
- Multiple TMs possible → different algorithms
- "TM" and "algorithm" become interchangeable terms
- Enables precise statements about computation

Why This Definition Matters

Before Formalization

- "Algorithm" was vague
- Hard to prove impossibility
- Couldn't compare computational models

After Formalization

- Precise mathematical object
- Can prove what's computable
- Can prove what's NOT computable
- Can compare different approaches

🤔

Think-Pair-Share

Scenario: Your friend says "I have an algorithm, but it sometimes runs forever on certain inputs."

Question: Is this actually an algorithm by our formal definition? Why or why not?

Multiple Algorithms for Same Task

Important insight:

> Many different TMs can solve the same problem

Examples:

- Different sorting algorithms (bubble, merge, quick)
- Different search algorithms (linear, binary)
- Trade-offs: speed, memory, complexity

This is why algorithm design matters!

From Theory to Practice

Theoretical Algorithm

- Specific TM construction
- Mathematical precision
- Proves possibility

Practical Algorithm

- Implementation in Java/Python/C++
- Optimized for real computers
- Considers actual resource constraints

Connection: Theory guarantees what's possible; practice makes it efficient

Example: Sorting

Theoretical View

- TM that decides: "Is this a sorted list?"
- TM that computes: list → sorted list
- Proves sorting is computable

Practical View

```
void quickSort(int[] arr) {
  // Implementation
}
```

- Specific algorithm choice
- Performance optimization
- Real-world constraints

💡

Application Exercise

Problem: For each task below, determine if it's decidable, recognizable, or computable

1. Checking if a string is a palindrome
2. Finding the largest prime number
3. Determining if a Java program will halt
4. Computing the factorial of n

Exercise Solutions

1. **Palindrome checking:** Decidable (TM can check and always halt)
2. **Largest prime:** Not computable (no largest prime exists!)
3. **Program halting:** Recognizable but not decidable (halting problem - we'll see this later)
4. **Factorial:** Computable (TM can multiply and halt)

Key insight: Not all problems have algorithmic solutions!

Key Vocabulary

Term	Definition
Recognize	TM accepts strings in language (may loop on others)
Decide	TM accepts in-language, rejects out-of-language (always halts)
Computable	Function that a TM can compute (always halting)
Algorithm	TM that decides a language or computes a function

Looking Ahead

Next lectures:

- **Church-Turing Thesis**: All computational models are equivalent
- **Uncomputability**: Problems no algorithm can solve
- **Halting Problem**: The most famous unsolvable problem
- **Complexity Theory**: Solvable but impractical problems

Foundation: Everything builds on this formal algorithm definition!