Computability

What Can A Computer Solve?

What Can't It Solve?

Today's Big Questions

Part 1: Unrecognizable Languages

- Are there problems no computer can ever solve?
- How do we prove something is impossible?

Part 2: The Halting Problem

- What is a concrete example of an unsolvable problem?
- Why can't we detect infinite loops?

Part 3: Reduction

- How do we prove other problems are unsolvable?
- What are the practical implications?

The Journey Ahead

Our path through impossibility:

1. **Counting Argument** → Prove unsolvable problems exist
2. **Concrete Example** → The halting problem
3. **Generalization** → Rice's Theorem and beyond

Key Theme: Understanding what computers *cannot* do is as important as understanding what they *can* do

Part 1: Unrecognizable Languages

Proving Impossible Problems Exist

Two Kinds of Infinity

Both sets are infinite, but are they the same "size"?

Natural numbers: {1, 2, 3, 4, ...}

Real numbers: [0, 1]

Surprising truth: Some infinities are strictly larger than others!

Key Concepts:

- **Countable** = Can enumerate systematically
- **Uncountable** = Cannot enumerate, strictly larger
- We can compare infinite sets meaningfully

What Makes a Set Countable?

Definition: A set is countable if it has a 1-to-1 correspondence with natural numbers

Example: Natural numbers

Even numbers

Natural	↔	Even
1	→	2
2	→	4
3	→	6
n	→	2n

Counterintuitive: Both sets have the "same size" even though evens are "half" of all

The Size of All Possible Strings

Question: How large is the set of all possible strings?

Answer: Countable!

Enumeration strategy:

1. Length 0: ε
2. Length 1: a, b, c, ...
3. Length 2: aa, ab, ac, ba, bb, bc, ...
4. Continue forever...

Key insight: Each length has finitely many strings, so we can list them all systematically without omission

The Size of All Turing Machines

Question: Since TMs are encoded as strings, how large is the set of all TMs?

Answer: Countable!

Reasoning:

- All possible strings = countable (just proved)
- TM encodings = subset of all strings
- Subset of countable set = countable

Implication: We can theoretically enumerate every possible algorithm!

Diagonalization: Cantor's Brilliant Trick

Claim: Infinite binary strings are uncountable

Proof by contradiction: Suppose we could list them all

n	f(n)
1	**0**110...
2	1**1**01...
3	01**0**0...
4	110**1**...

Construct x by taking opposites of the diagonal:

x = 1010... (opposite of diagonal)

The Diagonalization Contradiction

Our constructed string x = 1010...

Is x in our list?

- x ≠ f(1) because they differ at position 1
- x ≠ f(2) because they differ at position 2
- x ≠ f(n) for any n because they differ at position n

Contradiction! We claimed to list all infinite binary strings, but x isn't in the list

Conclusion: Infinite binary strings are **uncountable**

Think-Pair-Share

Question: Why can't we use diagonalization to prove that finite strings are uncountable?

Languages as Infinite Binary Strings

Every language corresponds to a unique infinite binary string

Example: Language a(a|b)* over {a, b}

All Strings	**ε**	**a**	**b**	**aa**	**ab**	**ba**	**bb**	**aaa**	**...**
In a(a|b)*?	No	Yes	No	Yes	Yes	No	No	Yes	...
Binary	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	...

Result: Language

↔️

0101100 11...

Since infinite binary strings are uncountable:

→ The set of all languages is **uncountable**!

The Critical Comparison

What we've established:

Set	Size
All possible strings	Countable ∞
All possible TMs	Countable ∞
All possible languages	**Uncountable ∞**

Mathematical fact: Uncountable > Countable

Therefore: # languages > # TMs

Unavoidable conclusion: Some languages have no TM to recognize them!

What Unrecognizable Means

Unrecognizable Language:

- No algorithm can verify that every string is in the language
- No TM exists to recognize it
- Not about difficulty—about fundamental impossibility

Computational Problem Equivalent:

- The problem cannot be solved by any computer
- No program, however clever, can solve it
- This is mathematical impossibility, not engineering limitation

Quick Poll

Which of these is countable or uncountable?

1. All Java programs
2. All real numbers between 0 and 1
3. All finite DNA sequences
4. All possible Python functions

Part 1 Summary

What we proved:

- Infinite sets can be compared (countable vs uncountable)
- **TMs = countable**
- **Languages = uncountable**
- Therefore: # Languages > # TMs

Implication: Some problems are mathematically impossible to solve

But which problems? Let's find out...

Part 2: The Halting Problem

A Concrete Unsolvable Problem

The Dream Function

Wouldn't this be amazing?

```
// Returns whether f(x) halts
public static boolean halts(String javaCodeForF, String x) {
    // Magic here...
}
```

Benefits:

- Catch infinite loops before running
- Verify program termination
- Ensure reliability

Question: Can we write this function?

Three Paths to the Truth

We'll prove the halting problem is unsolvable using three approaches:

1. **Java formulation** - Intuitive and practical
2. **Turing Machine formulation** - Formal and precise
3. **Diagonalization formulation** - Elegant and visual

Pick your favorite—they all prove the same thing!

The answer: No, we cannot build `halts`

Approach 1: Java Formulation

Assume `halts` exists:

```
public static boolean halts(String javaCodeForF, String x) {
    // Returns true if f(x) halts, false otherwise
}
```

Now create the contrarian function:

```
public static void contrarian(String javaCodeForF) {
    if (halts(javaCodeForF, javaCodeForF)) {
        while (true) {}  // Loop forever
    }
    // Otherwise halt immediately
}
```

Behavior: Does opposite of what `halts` predicts!

The Contrarian Paradox

What happens with this call?

```
contrarian(javaCodeForContrarian);
```

Case 1: Suppose `contrarian(javaCodeForContrarian)` halts

- Then `halts(javaCodeForContrarian, javaCodeForContrarian)` returns true
- So contrarian enters the infinite loop
- **Contradiction!** It doesn't halt

Case 2: Suppose `contrarian(javaCodeForContrarian)` doesn't halt

- Then `halts(javaCodeForContrarian, javaCodeForContrarian)` returns false
- So contrarian skips the loop and halts
- **Contradiction!** It does halt

The Inescapable Conclusion

Both cases lead to contradictions!

The only resolution:
Our assumption that `halts` exists must be **false**

Proof technique: Proof by contradiction

> The function `boolean halts(String f, String x)` cannot exist

This isn't a limitation of Java—it applies to any programming language or computational model!

Approach 2: Turing Machine Formulation

Define the language H:

H = {⟨M, w⟩ | M is a TM and M halts on input w}

Question: Is H decidable?

Assume yes (this leads to contradiction):

- Then there exists a TM called D that decides H
- D(⟨M, w⟩) accepts if M(w) halts
- D(⟨M, w⟩) rejects if M(w) doesn't halt

Now construct contrarian TM C...

The Contrarian TM

Define TM C using D as a subroutine:

C on input ⟨M⟩:

1. Run D(⟨M, ⟨M⟩⟩)
2. If D accepts → loop forever
3. If D rejects → accept

Behavior:

- C(⟨M⟩) loops if M(⟨M⟩) halts
- C(⟨M⟩) halts if M(⟨M⟩) doesn't halt

Now run: C(⟨C⟩)

The TM Contradiction

What happens with C(⟨C⟩)?

Case 1: D(⟨C, ⟨C⟩⟩) accepts

- This means C(⟨C⟩) halts
- But then C enters infinite loop
- **Contradiction!**

Case 2: D(⟨C, ⟨C⟩⟩) rejects

- This means C(⟨C⟩) doesn't halt
- But then C accepts and halts
- **Contradiction!**

Conclusion: The halting problem is **undecidable**

Approach 3: Diagonalization

List all TMs as rows, their encodings as columns

Table shows what decider D outputs:

D(⟨Mi, ⟨Mj⟩⟩)	**⟨M_1⟩**	**⟨M_2⟩**	**⟨M_3⟩**	**...**
M_1	**accept**	reject	accept	...
M_2	reject	**reject**	accept	...
M_3	accept	reject	**accept**	...
...	...	...	...	...

Diagonal: What each TM does when run on itself

The Diagonal Argument

Our contrarian C must be in this table somewhere:

D(⟨Mi, ⟨Mj⟩⟩)	**⟨M_1⟩**	**⟨M_2⟩**	**⟨M_3⟩**	**...**	**⟨C⟩**
M_1	**accept**	reject	accept	...	...
M_2	reject	**reject**	accept	...	...
M_3	accept	reject	**accept**	...	...
...	...	...	...	...	...
C	**reject**	**accept**	**reject**	...	**?**

C's row has opposite of diagonal values

But what about C on ⟨C⟩? It must be opposite of itself!

Active Learning: Explain It Back

Task: Explain the halting problem proof to a partner in your own words

Requirements:

1. Why we can't build a `halts` function
2. What role does the contrarian play?
3. Why both cases lead to contradictions

Why The Halting Problem Matters

Practical Implications:

- **No perfect debugger:** Can't detect all infinite loops
- **No complete program analyzer:** Can't verify all properties
- **No perfect optimizer:** Can't determine all optimizations
- **No ideal virus scanner:** Can't detect all malicious behavior

Fundamental Limit: Software verification has inherent boundaries

This affects real systems we build every day!

Part 2 Summary

What we proved:

- The halting problem is undecidable
- No algorithm can determine if arbitrary programs halt
- Three different proofs all reach same conclusion

Key Technique: Proof by contradiction using self-reference

The Question: Are there other unsolvable problems?

Answer: Yes! Many more. Let's see how to prove them...

Part 3: Reduction

Extending Unsolvability

The Reduction Concept

Definition: Convert one problem to another such that solving the second solves the first

Key Idea:

- If Problem A reduces to Problem B
- And Problem A is unsolvable
- Then Problem B must also be unsolvable

Why? If we could solve B, we could solve A (contradiction!)

Warm-up Example

Problem A: Find the maximum number in a list

- Input: [6, 4, 9, 2, 1]
- Output: 9

Problem B: Sort a list in ascending order

- Input: [6, 4, 9, 2, 1]
- Output: [1, 2, 4, 6, 9]

A reduces to B:

1. Solve B to get sorted list
2. Return last element (the maximum)

If B is unsolvable, A must be too!

Rice's Theorem

The Big Result:

> Determining whether a program has *any* non-trivial property is undecidable

Non-trivial property:

- Depends on program behavior (not syntax)
- Some programs have it, others don't

Examples of non-trivial properties:

- Computes function x^2
- Produces output
- Terminates in under 100 steps
- Returns even numbers

Rice's Theorem: The Setup

Assume we have this magic function:

```
public static boolean hasProperty(Predicate<String> f) {
    // Returns true if f has the specific property
}
```

Since property is non-trivial, there must exist:

```
Predicate<String> funcYes = ...; // Has the property
assertTrue(hasProperty(funcYes));

Predicate<String> funcNo = ...; // Doesn't have property
assertFalse(hasProperty(funcNo));
```

Without loss of generality, assume `funcNo` is non-halting.

We'll show: If `hasProperty` exists, we can solve halting!

Constructing the Reduction

Create a new function Mw:

```
public static Predicate<String> createMw(
    Predicate<String> M, String w) {

    return x -> {
        M.test(w);  // Run M on w

        // If M(w) doesn't halt, we never get here
        // So this predicate acts like funcNo

        // If M(w) halts, continue:
        return funcYes.test(x);
        // So this predicate acts like funcYes
    };
}
```

Key insight: Mw behaves differently depending on whether M(w) halts!

Solving Halting with hasProperty

Now we can implement halts:

```
public static boolean halts(Predicate<String> M, String w) {
    Predicate<String> Mw = createMw(M, w);

    // If M halts on w: Mw = funcYes
    // If M doesn't halt: Mw = funcNo

    // So checking if Mw has the property
    // tells us if M halts on w!
    return hasProperty(Mw);
}
```

But halts can't exist!

Therefore: hasProperty can't exist either!

🎮

Active Learning: Find the Reduction

Scenario: Can we determine if a program outputs "Hello"?

Work in groups:

1. How would you reduce halting to this problem?
2. What function would you construct?
3. What's the key insight?

Hint: Create a program that runs M(w), then outputs "Hello" if it halts

Rice's Theorem Implications

What Rice's Theorem tells us is undecidable:

✗ Does this program compute prime numbers?
✗ Does this program ever output "error"?
✗ Does this program use recursion?
✗ Does this program allocate memory?
✗ Is this program a virus?

What we CAN decide:

✓ Syntactic properties (uses keyword "while")
✓ Trivial properties (true for all or no programs)

More Unsolvable Problems

Real-world impossible problems:

- **Optimal compression:** Can't determine if data is maximally compressed
- **Perfect compiler:** Can't guarantee full optimization of all programs
- **Program equivalence:** Can't determine if two programs compute same result
- **Perfect virus detection:** Can't identify all viruses
- **Memory management:** Can't determine if variables will be used again
- **Chaos prediction:** Can't determine if dynamical systems are chaotic

Why Unsolvability Matters

From Sedgewick & Wayne:

> "In a very real sense, unsolvability and Rice's theorem provide a foundation for understanding why ensuring reliability in software systems is so difficult."

Key Insights:

- We can't build programs that verify arbitrary properties
- Understanding impossibility guides realistic expectations
- Focus effort on tractable problems
- Design systems acknowledging fundamental limits

Reflection: Personal Connection

Consider:

1. How does knowing about unsolvable problems change your view of computing?
2. What are you NOT going to waste time trying to build?
3. How might this guide your career as a computer scientist?

Key Techniques Summary

1. Counting Arguments

- Compare sizes of infinite sets
- Prove existence without construction

2. Proof by Contradiction

- Assume solution exists
- Derive logical contradiction
- Conclude assumption was false

3. Reduction

- Convert one problem to another
- Transfer unsolvability
- Build on known results

Complete Picture: The Formal Language Hierarchy

From simplest to most complex:

1. **Regular** - DFA/NFA/Regex (always tractable)
2. **Context-Free** - PDA (often tractable)
3. **Decidable** - TM that always halts (may be intractable)
4. **Recognizable** - TM (may not halt on rejection)
5. **Unrecognizable** - No TM possible

We focused on the boundary between recognizable and unrecognizable

Practical Takeaways

For Software Engineering:

- Don't try to build perfect analyzers
- Accept limitations in testing/verification
- Design with impossibility in mind
- Focus on heuristics and practical solutions

For Computer Science:

- Understanding limits is as important as capabilities
- Impossibility results guide research directions
- Theoretical foundations impact practice

Summary: The Big Picture

Part 1: Existence

- Proved unsolvable problems must exist
- Used counting argument (∞ languages > ∞ TMs)

Part 2: Concrete Example

- The halting problem is undecidable
- Multiple proofs via self-reference

Part 3: Generalization

- Reduction extends unsolvability
- Rice's theorem covers all program properties

Looking Ahead: Complexity Theory

Next big question: Among decidable problems, which are practical?

Classes we'll explore:

- **P:** Solvable in polynomial time (efficient)
- **NP:** Verifiable in polynomial time
- **NP-Complete:** Hardest NP problems
- **P vs NP:** Biggest open question in CS

From computability to complexity: Moving from "can we?" to "can we efficiently?"