What’s a computer?

A small suan pan (Chinese abacus)
*Photo courtesy of the Computer History Museum*

Hand-cranked Curta calculator
*Photos courtesy of the Computer History Museum*
Some kinds of computers have more computational power than others.

We can abstract devices of the "same kind" to produce models of computers and ask what kinds of problems can be solved under a particular model.

A computer is not dependent so much on technology as on ideas.

Present-day computers are built out of transistors and wires. They could be built, according to the same principles, from valves and water pipes or from sticks and strings.
One of the most remarkable things about computers is that their essential nature transcends technology.

Why do we need theory?

Why study this stuff?

Theoretical computer science has many fascinating big ideas, but also many small (and sometimes dull) details.

The more you learn, the more interesting it becomes.

*Our goal:* Be exposed to the exciting aspects of computer theory without getting bogged down in drudgery.

Theory is relevant to practice

Provides conceptual tools that practitioners use in computer engineering

- Design a new programming language for a special application — need (context-free) grammars!
- String searching and pattern matching — use *finite automata* and *regular expressions*!
When developing solutions to real problems, we often confront the limitations of what software can do:

*Undecidable things* – no program whatever can do it
*Intractable things* – there are programs, but no fast programs

Theory gives you the tools.

He who loves practice without theory is like the sailor who boards ship without a rudder and compass and never knows where he may cast.

Leonardo da Vinci, “Prolegomena and General Introduction to the Book on Painting”

**Theory shows the more elegant side of computers**

We usually think of computers as complicated machines.

The best computer designs and applications are conceived with elegance in mind.

A theoretical course can heighten your aesthetic sense and help you build more elegant systems.

In computer science, elegance is not a dispensable luxury, but a matter of life and death.

Edsger Dijkstra
Theory expands your mind

Computer technology changes quickly.

Since studying theory enables you to understand the underlying models of all computation, not just technical details that become outdated in a few years.

Studying theory trains you in abilities with lasting value:

Think and express yourself clearly and precisely.
Solve problems — and know when you haven’t solved a problem.

Automata, computability, and complexity

Linked by the question: What are the fundamental capabilities and limitations of computers?

Each area interprets the question differently:

Automata theory: Definitions and properties of mathematical models of computation.
Computability: Is a given problem solvable or unsolvable?
Complexity: Is a given problem easy or hard?

How could we talk about computation?

We start with automata theory:

Theories of computability and complexity require a precise definition of a computer.

Automata theory allows practice with formal definitions of computation as it introduces concepts relevant to other, non-theoretical areas of computer science.
The central idea in the theory of computation is that of a *universal computer*, a computer powerful enough to simulate any other computing device.

Most computers we encounter in everyday life are universal computers.
With the right software – and enough time and memory – they can simulate any other type of computer…

**Universal computers**
The idea of a universal computer was recognized and described in 1937 by Alan Turing.¹

He called it a “universal machine” since at the time, “computer” still meant “a person who performs computations”.

¹ Poor Alonzo Church is a footnote. Where’s his movie?

The central idea in the theory of computation is that of a **universal computer**, a computer powerful enough to simulate any other computing device.

Most computers we encounter in everyday life are universal computers.

With the right software – and enough time and memory – they can simulate any other type of computer… or – as far as we know – any other device at all that processes information.

**Hypothesis:** *Any* computing device – made of transistors, sticks and strings, or neurons – can be simulated by a universal computer.

This suggests that making a computer think like a brain is just a matter of programming it correctly!

While a universal computer can compute anything that can be computed any other computing device, there are some things that are just impossible to compute.
Vaguely defined questions

E.g., “What is the meaning of life?”

Lack data

“What is the winning number in tomorrow’s lottery?”

But there are also flawlessly defined computational problems that are impossible to solve.

We call these problems noncomputable.

What exactly are the limits to what a computer can do?

We'll work to an answer of this over the semester!

This will take us through the philosophically interesting topics of nondeterminism, Turing machines, computability, Gödel's incompleteness theorem, and – time allowing – quantum computing.
Because computers can do some things that seem very like human thinking, people worry that they threaten our unique position as rational beings.

Some people seek reassurance in mathematical proofs of the limits of computers.

There have been analogous controversies in human history.

Andrew Borde, 
The First Book of the Introduction of Knowledge, 1542

"Far out in the uncharted backwaters of the unfashionable end of the Western spiral arm of the galaxy lies a small unregarded yellow sun. Orbiting this, at a distance of roughly ninety million miles is an utterly insignificant little blue-green planet..."
These philosophical crises are based on a misplaced judgment of the source of human worth, and the current philosophical discussions about the limits of computers are based on a similar misjudgment.

Course overview

Study categories of languages and machines:
- Regular languages and finite automata
- Context-free languages and pushdown automata
- Unrestricted languages and Turing machines

Study solvability and efficiency:
- The Halting Problem
- NP-completeness
Prerequisites

CMPU 102: Data Structures and Algorithms

CMPU 145: Foundations of Computer Science

Math?
Sets

Group of objects represented as a unit

May contain any type of object: numbers, symbols, other sets, ...

- **Set membership**: $\in$
- **Non-membership**: $\notin$
- **Subset**: $\subseteq$
- **Proper subset**: $\subset$
- **Empty set**: $\emptyset$

Infinite set contains infinitely many elements, e.g., the set of integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$

Cartesian product and relations

*Cartesian product* builds a set consisting of ordered pairs from two or more existing sets:

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

$$A^3 = A \times A \times A$$

A **binary relation** on sets $A$ and $B$ is a subset of $A \times B$.

An **n-ary relation** on sets $S_1, S_2, \ldots, S_n$ is a subset of $S_1 \times S_2 \times \cdots \times S_n$
Functions

A **function** from a set $A$ to a set $B$ is a mapping of elements of $A$ to elements of $B$ such that each element of $A$ maps to **exactly one** element of $B$.

$f: A \rightarrow B$

- $A$ is the **domain** of $f$
- The **range** of $f: A \rightarrow B$ is the set $\{b \in B \mid b = f(a) \text{ for some } a \in A\}$.

A **total function** $f$ from $A$ to $B$ is a binary relation on $A \times B$ such that

For each $a \in A$, there is a $b \in B$ such that $(a, b) \in f$.

If $(a, b) \in f$ and $(a, c) \in f$, then $b = c$.

Countable and uncountable sets

We divide sets into classes:

**Countable**: finite or countably infinite

- A **countably infinite** set has the same number of elements as the set of natural numbers $\mathbb{N}$
- To show a set $A$ is countably infinite, we need to show a bijective function exists from $\mathbb{N}$ to $A$.

**Uncountable**: more elements than the set of natural numbers $\mathbb{N}$

Bijective functions

A function $f: A \rightarrow B$ is bijective if it is:

- **one-to-one**: each element of $A$ maps to a distinct element in the range $B$.
- **onto**: the range is the entire set $B$.

Uncountable sets

Prove a set $X$ is uncountable by showing that it’s impossible to sequentially list its members.

**Cantor’s diagonalization argument** is a proof by contradiction:

1. **Step 1**: Assume the set is countable and therefore its members can be exhaustively listed. This listing must contain all members of the set.
2. **Step 2**: Produce a member of the set that cannot occur anywhere in the listing, showing that such a listing cannot exist, and therefore the set is uncountable.
Recursive definitions of sets

Recursion provides a method for generating elements of a set by specifying:

- **basis elements** explicitly
  - a finite set of **operators**, used to construct the remaining elements of the set from the basis elements

Generation using a finite number of operations is a fundamental property of recursive definitions.

Example:

A recursive definition of $\mathbb{N}$, the set of natural numbers using the successor function, $s(n) = n + 1$:

- **Basis**: $0 \in \mathbb{N}$
- **Recursive step**: If $n \in \mathbb{N}$, then $s(n) \in \mathbb{N}$
  - $n \in \mathbb{N}$ only if it can be obtained from 0 by a finite number of applications of the recursive step.

Elements are:

- $0, \ s(0), \ s(s(0)), \ s(s(s(0))), \ ...$
- $0, \ 1, \ 2, \ 3, \ ...$

---

Languages

**Alphabet ($\Sigma$)**: any finite set of symbols, e.g.,

- Binary alphabet: $\Sigma = \{0, 1\}$
- ASCII: $\Sigma = \{a, b, c, ..., 0, 1, ..., !, @, #, ...\}$

**String** or **word**: a finite sequence of symbols chosen from some alphabet, e.g.,

- 01101
- *abracadabra*

**Convention**: Use lowercase letters from the beginning of the alphabet for symbols and lowercase letters from the end of the alphabet for strings.
The length of a word $w$, written as $|w|$, is the number of symbols in it e.g.,
\[ |abcde| = 5 \]
\[ |uv| = |u| + |v| \]

The empty string, denoted by $\varepsilon$, is the string with no symbols in it
\[ |\varepsilon| = 0 \]

String relations

A string $z$ is a substring of $w$ if $z$ appears consecutively in $w$, e.g.,
\[ \text{bat is a substring of } batman \]

Prefix and suffix: If $w = vu$, $v$ and $u$ are a prefix and a suffix of $w$, respectively, e.g.,
\[ \text{super is a prefix of } superman \]
\[ \text{man is a suffix of } superman \]

String operations

Reverse: $w^R$
E.g., if $w = abc$, then $w^R = cba$

Concatenation: Given strings $x$ and $y$, their concatenation is $xy$.
E.g., if $x = abc$, and $y = def$, then $xy = abcdef$

Languages

A language is a set of strings chosen from some alphabet, e.g.,

The set of all binary strings consisting of some number of 0s followed by an equal number of 1s, i.e.,
\[ \{\varepsilon, 01, 0011, 000111, \ldots\} \]

The C programming language, i.e.,
the set of all C programs that compile without syntax errors

English
More about languages

Languages can be **finite**, e.g.,
\[ L = \{a, aba, bba\} \]

or **infinite**, e.g.,
\[ L = \{\sigma^n \mid n > 0\} \]

Note: While a language can be infinite, the set of symbols it’s composed of (\( \Sigma \)) is always finite.

---

Powers:

\[ \Sigma^k = \text{set of all strings from alphabet } \Sigma \text{ with length } k \]
\[ \Sigma^0 = \{\varepsilon\} \]
\[ \Sigma^* = \text{set of all strings from alphabet } \Sigma \]
\[ \Sigma^+ = \Sigma^* - \{\varepsilon\} \]

---

We can **concatenate** languages:

\[ L_1 L_2 = \{xy \mid x \in L_1, y \in L_2\} \]
\[ L^n = L \text{ concatenated with itself } n \text{ times} \]
\[ L^0 = \{\varepsilon\} \]
\[ L^1 = L \]

**Star-closure**

\[ L^* = L^0 \cup L^1 \cup L^2 \cup ... \; (\text{note: } L^* = L^* - L^0) \]

---

Acknowledgments

This lecture incorporates material from:

Douglas Adams
W. Daniel Hillis
John Hopcroft
Nancy Ide
Michael Sipser