Overview of Theory of Computation

• What is Theory of Computation?
• Formal Languages
• Finite Automata and Regular Expressions
• Turing Machines
• $P \neq NP$?
• Undecidable Problems
What does computation mean?

The Antikythera Mechanism
An ancient (2 c. BCE) bronze clockwork device used to predict the location of the sun and moon, including eclipses.

Original at left, modern replica below.

When did computation begin?
What does computation mean?

Where is it going?
What is Theory of Computation?

- In the 1930's there was great interest in the field of mathematical logic, around foundational questions like these:
  - What kinds of mathematical claims can or cannot be proved? What, if any, are the limits of proof itself?
- Key names: Gottlob Frege, David Hilbert, Giuseppe Peano, Bertrand Russell, Kurt Gödel.
- Their work, and others, inspired and led to formal, mathematical concepts of computation, unrelated to any physical implementation.
What is Theory of Computation?

- Theory of Computation concerns the mathematical fundamentals of the questions, *What is a computer? What can computers do?*

  - **Essence.** From a mathematical perspective, what are the essential characteristics that define a computer (independent of physical realization)?
    - Are there different *levels*? E.g., primitive to advanced
  
  - **Abilities and Limits.** What kinds of problems can be solved with a computer? What kinds *cannot*?
  
  - **Speed.** What kinds of problems can be solved *efficiently*, in some sense? What kinds cannot?
What is Theory of Computation?

• To explore the question of “what is a computer,” people have developed mathematical *models* of computation:
  
  – **Alan Turing** (at age 24) published an abstract “machine” that accepts input, executes a stored program, and reads writes memory: it has the same essential elements of a modern computer.
  
  – **Alonzo Church** developed a function-like system called “lambda calculus,” an inspiration for LISP.
  
  – **Emil Post** published something a lot like the Turing machine, a few months later than Turing (too bad).

They all published these ideas in 1936, independently!
What is Theory of Computation?

- **Basic results:**
  - **Church-Turing thesis:** the “Turing machine” concept is at least as capable as any system that can do computation.
  - **Limits:** the Turing machine does have limits: there are *undecidable* problems out there, beyond the capability of any computing machine. (Turing, ’36)
  - **Levels:** in the 1940's, researchers investigated more primitive systems (finite automata), giving a *variety of models* of computation, of differing power.
  - **Speed:** there is still a huge *open question* in this area: “Is it true that $P \neq NP$?”
Overview of Theory of Computation

- What is Theory of Computation?
- **Formal Languages**
- $P \neq NP$?
- Finite Automata and Regular Expressions
- More about Turing Machines
- Undecidable Problems
Formal Languages

• Main idea:
  – All interesting computational problems can be rephrased as yes-or-no questions of the form
    \[ x \in L \]
    ... where \( x \) is a string and \( L \) is a "language" (a set of strings).

• Let's unpack this idea.
Languages and Computation

- We are accustomed to seeing “computation” as reading input, doing some work, and producing some interesting output. But that's just habit.

Example: Multiplication (customary view)

(5,7) \[\rightarrow\] Multiply \[\rightarrow\] 35

This machine represents a program which can multiply two integers.

It has input, and it produces output.
Languages and Computation

- We could turn the problem around and imagine a (conceptual) computer that verifies its input and produces “yes” or “no” as output.

This machine represents a program which can **CHECK** to see if a given multiplication was performed correctly.

Its only output is a boolean answer: TRUE or FALSE.
Languages and Computation

Viewing Computation as Decider

(5,7),33

Multiply-Check

Multiply-Check

Multiply-Check
Languages and Computation

Viewing Computation as Decider

(5,7),33 → Multiply-Check

(5,7),34 → Multiply-Check

→ Multiply-Check
Languages and Computation

Viewing Computation as Decider

(5,7),33 \rightarrow Multiply-Check

(5,7),34 \rightarrow Multiply-Check

(5,7),35 \rightarrow Multiply-Check
These two machines have the same computational power. One is more efficient...but they can answer the same questions, given enough time.
Languages and Computation

- The multiply-checker machine could be viewed as testing **set membership**, for the following set of strings:

\[ M = \{ '(x,y) z' : x, y, z \in \mathbb{Z} \land xy = z \} \]

- '(5,7)42' \( \in M \)?
- '(5,7)35' \( \in M \)?
- '(20484, 21079)431782236' \( \in M \)?
- 'woof woof' \( \in M \)?
- '(20168, 22099)455692632' \( \in M \)?
Languages and Computation

The multiply-checker machine could be viewed as testing **set membership**, for the following **set of strings**:

\[ M = \{ '(x,y)\ z' : x, y, z \in \mathbb{Z} \land xy = z \} \]

- \'(5,7) 42' \notin M
- \'(5,7) 35' \in M
- \'(20484, 21079) 431782236' \in M
- 'woof woof' \notin M
- \'(20168, 22099) 455692632' \notin M
Formal Languages

- In T.O.C., we focus our attention on sets of strings, which we call “languages.”

- An **alphabet** is a finite set of symbols.
  - Could be the Latin alphabet, but think abstractly: it could be anything. It's just a **set**.
  - Some theorists prefer to use the alphabet \{0, 1\}.

- A **string** is a finite sequence of symbols from a given alphabet.

- A **language** is merely a set of strings (**words**).
Formal Languages

• To investigate the abilities and limits of an abstract computational model, we restrict our models to produce only one bit of output, "yes" or "no," answering the question,

\[ x \in L ? \]

... for some interesting language \( L \).

• So we will describe all computational problems as "language membership" tests.
But Why Formal Languages?

• By restricting our abstract computational models this way, they only need to produce only **one bit** of output ("yes" or "no").

• Sounds like a stifling restriction! Not really.

• It keeps our abstract models simple.
  
  – If you can get one bit of output, you could in principle get any finite number of bits of output.
  
  – Good enough to tell **solvable from unsolvable** problems.
  
  – Even good enough for a coarse idea of **speed**.
Languages and Computation

Thus, we usually do not say this:

Instead, we usually say it like this:

- "Easy / difficult / interesting computational problem."
- "Easy / difficult / interesting language."
Example: suppose you want to test membership in the "language" of all possible legal Java source code file contents:

- Call this set $J = \{\text{Perfectly valid Java programs}\}$.

Given string $x$, it may be hard to test whether $x \in J$.

- You would need to check for syntax errors, check all identifiers, object types, inheritance, interfaces, etc.
- Seems like you would basically want a Java compiler, modified to produce just one bit of output.
- The task is almost as challenging as compilation itself.
- Conclusion: $J$ is decidable, but the code is nontrivial.
Example: suppose you want to test membership in the "language" of all possible legal Java source code file contents:

- Call this set $J = \{\text{Perfectly valid Java programs}\}$.

Given string $x$, it may be hard to test whether $x \in J$.

- You would need to check for syntax errors, check all identifiers, object types, inheritance, interfaces, etc.
- Seems like you would basically want a Java compiler, modified to produce just one bit of output.

- The task is almost as challenging as compilation itself.
- Conclusion: $J$ is decidable, but the code is nontrivial.
Example: suppose you want to test membership in the "language" of all possible legal Java source code file contents:

- Call this set $J = \{\text{Perfectly valid Java programs}\}$. Given string $x$, it may be hard to test whether $x \in J$.
  - You would need to check for syntax errors, check all identifiers, object types, inheritance, interfaces, etc.
  - Seems like you would basically want a Java compiler, modified to produce just one bit of output.
    - The task is almost as challenging as compilation itself.
    - Conclusion: $J$ is decidable, but the code is nontrivial.
Formal Languages: summary so far

• In Theory of Computation, we don't need models with fancy output. One bit of output is challenge enough.

• We reframe computational challenges as language decision problems:
  – You and I agree on an interesting language $L$.
  – You hand me a string $x$.
  – My model (I hope) decides whether or not $x \in L$. 
Famous Families of Languages

- Languages come in various difficulty levels. (Difficulty of deciding membership, that is.)
  
  \{w : w \text{ ends with the suffix } "izzle"\}
  
  \{w : w \text{ contains an odd number of } 1\text{'s}\}
  
  \{w : w \text{ is a palindrome}\}
  
  \{<G, T> : G \text{ represents a connected, weighted graph and } T \text{ is its minimum spanning tree}\}
  
  \{<G, S> : G \text{ represents a connected weighted graph, } S \text{ is the shortest path through every vertex}\}
  
  \{w : w \text{ represents Java code that will run error-free, and never enter an infinite loop}\}
Famous Families of Languages

- Languages come in various difficulty levels. (Difficulty of deciding membership, that is.)

**VERY EASY** ("regular")
\{w : w ends with the suffix "izzle"\}
\{w : w contains an odd number of 1's\}

**EASY** ("context-free")
\{w : w is a palindrome\}

**MEDIUM*** ("P")
\{<G, T> : G represents a connected, weighted graph and T is its minimum spanning tree\}

**HARD*** ("NP")
\{<G, S> : G represents a connected weighted graph, S is the shortest path through every vertex\}

**IMPOSSIBLE** ("Undecidable")
\{w : w represents Java code that will run error-free, and never enter an infinite loop\}

* a bit oversimplified
Overview of Theory of Computation

• What is Theory of Computation?
• Formal Languages
• **Finite Automata and Regular Expressions**
• Turing Machines
• \( P \neq NP \) ?
• Undecidable Problems
Finite Automata: DFAs, NFAs

- Some languages are so easy to decide, that we do not need a full-powered computer.
- We can use an extremely restricted model called a deterministic finite automaton:
  - Represented by a digraph with vertices = "states"
    - Model is always in one of its states.
  - Reads the input character-by-character, single-pass
  - Chooses its next state based on input character.
  - Final state indicates yes/no decision.
Example DFA

- Example DFA that decides this language:
  \[ \{ w : w \text{ is a bit string with an odd number of 1's} \} \]

- Notation: if final state has double-border, decision is 'yes' \((x \in L)\), otherwise, decision is 'no' \((x \notin L)\).
DFAs

- A **Deterministic Finite Automaton (DFA)** is a model for a very primitive computer.
  - **Cannot** do everything that full computer can do.
  - But has more power than you might think!
  - Can be implemented super fast in circuitry.

- Simple state machine
  - Its only memory is to keep track of “current state.”
  - Follows one link per input symbol.
  - If used as a language decider, its binary output depends on its final state when the input ends.
DFAs

A Simple DFA

Start State (can only be one)

Edges have labels to tell you where to go, based on the next symbol.

Accept State (could be many)
If, after reading the last letter in the input, we are in one of the accept states, then the machine accepts the input.

Otherwise, it rejects the input.
Example Input:

aabbbaaabbbbaaabba
Example Input:

aabbbbbbaaabba

We always start at the start state, before we have read any input.
DFAs

Example Input:

aabbabbaaabba

We read the first letter in the input, and follow a link out of the start state.
Example Input: 

aabbbaabaaaaabba

As we read each letter, the edge tells us where to go next.
DFAs

**Example Input:**

aabbbbbbaaabba

As we read each letter, the edge tells us where to go next.
Class Exercise:
What **words** does this machine **accept**?

That is, what **language** does it decide?
DFAs

Answer:
This machine accepts all words which have a string of three (or more) 'a's in a row, anywhere in the string.
DFAs

- Machines like this can count, and they can even do a little arithmetic: integer addition.
  - That is, they can test for correct addition.
- But, they cannot do multiplication. (Provable!)
  - That is, they cannot test for correct multiplication.
NFAs

- A **Nondeterministic Finite Automaton (NFA)** is a DFA, plus nondeterministic behavior
  - Nondeterminism = there are multiple things it might do in any given situation.
  - One view: it prophetically "knows" which choice to make, in order to accept.
  - 2nd view: parallel versions make both choices.
  - Both views are weird, but the 2nd is easy to simulate. Let's explore that view further.

- An NFA accepts an input if **any one of its parallel versions** accepts the input.
This NFA has two accept states; it will accept the input if **any one of its copies** is in **any one of the accept states** when the input ends.
NFAs

Some states in an NFA will have multiple links with the same symbol. This is why we say the machine is non-deterministic; there are multiple possible paths to follow.
NFAs

Not all of the states have links for all possible symbols.

In both DFAs and NFAs, if you cannot find a link to follow, then the machine automatically rejects.

But of course, an NFA can have multiple machines running; one may die, while others continue on.
NFAs

**Example Input:**
ab

Just like a DFA, an NFA starts at a single defined start state.
NFAs

**Example Input:**
ab

This state has two links which can handle the 'a' input...
NFAs

Example Input:

This state has two links which can handle the 'a' input...
...so we split into two parallel machines – one for each state.
NFAs

Example Input: ab

But notice that neither machine can parse the next letter of the input!

So this NFA rejects the input.
NFAs

Example Input: aaa

Let's start over, but with a different input.
NFAs

Example Input:

aaa

This state has two links which can handle the 'a' input...
NFAs

Example Input:

aaa

This state has two links which can handle the 'a' input...
...so we split into two parallel machines – one for each state.
NFAs

Example Input: aaa

Both of the active machines are able to parse the next letter in the input, so we proceed.
NFAs

**Example Input:**

`aaa`

One of the two machines has reached the accept state, but this doesn't matter, as there is still one more character to read from input.

In this case, the machine in the accept state will die on the next step.
Example Input:

aaa

One of the machines died, but that doesn't matter; another one survived.

The second machine reached the accept state right as the input ended.

This machine ACCEPTS the input.
NFAs

Class Exercise:
What language does this NFA recognize?
NFAs

Answer:
This NFA recognizes only two strings:
“aa”, “aaa”
Finite Automata have limited abilities

- Many languages simply cannot be decided by any DFA or NFA.

**Examples:**

- “A sequence of a's, followed by the same number of b's”
- “Same number of a's in the word as b's”
- Palindromes

What is common between these three problems?

They all require **arbitrarily large** memories.
Regular Expressions

• **Regular expressions** are a convenient way to express certain simple languages
  – A way to describe string patterns.
  – The pattern defines a set of strings: any string matching the pattern is, by definition, "in" the set.
  – Very useful in the real world.

• Three basic operations:
  – Concatenation
  – Alternation
  – Repetition
Regular Expressions

- The most basic element of a regular expression is a **single letter**. You can **concatenate** any number of these together to match a part of a word.

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Language It Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>{ “abc” }</td>
</tr>
<tr>
<td>def</td>
<td>{ “def” }</td>
</tr>
<tr>
<td>jkl0123</td>
<td>{ “jkl0123” }</td>
</tr>
</tbody>
</table>
## Regular Expressions

- **Alternation** (choosing between options) is expressed by the `|` operator.

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Language It Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>`abc</td>
<td>def`</td>
</tr>
<tr>
<td>`a</td>
<td>aa</td>
</tr>
<tr>
<td>`jkl</td>
<td>0123`</td>
</tr>
</tbody>
</table>
Regular Expressions

- **Repetition** is expressed by the * operator, known as the **Kleene star**. The expression will match zero, one, or many copies of the term.

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Language It Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>a*</td>
<td>{ &quot;&quot;, &quot;a&quot;, &quot;aa&quot;, ... }</td>
</tr>
<tr>
<td>aa*</td>
<td>{ &quot;a&quot;, &quot;aa&quot;, &quot;aaa&quot;, ... }</td>
</tr>
<tr>
<td>a<em>b</em></td>
<td>Zero or more 'a's, followed by zero or more 'b's</td>
</tr>
</tbody>
</table>
Regular Expressions

- **Parentheses** are used to group terms together.

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Language It Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a</td>
<td>b) xyz)</td>
</tr>
<tr>
<td>((aa</td>
<td>bb))*</td>
</tr>
<tr>
<td>((ab<em>c))</em></td>
<td>Zero or more a-c pairs, with zero or more 'b's within each pair.</td>
</tr>
</tbody>
</table>
Surprising Equivalences

- **DFA = NFA = Regular Expression**
  - Any language easy enough to be decided by one of them may be decided by all of them.
  - These computational models are *equally capable*.
- Any NFA can be simulated with a DFA.
- Every Regular Expression can be converted to an NFA.
- Every NFA can be converted to a Regular Expression.
- **Proof:** (take CSc 473)
Turing Machines

- A **Turing Machine** is basically a DFA augmented with an infinite linear memory.
  - Memory is typically imagined as a “tape” divided into cells with a “read-write head” at one cell.

https://www.youtube.com/watch?v=E3keLeMwfHY
Turing Machines

• Instead of simply reading from input, the edges in a TM have a triple label:
  – Read the symbol from the cell under the head.
    • This replaces the simple “read from input” of a DFA.
  – Write a symbol into that cell.
  – Shift the head left or right by one cell.

• TM has special accept and reject states, too.

• Capabilities: thanks to its tape memory, the TM is a much more powerful model of computation than finite automata.
What can the Turing Machine do?

• In brief: it can compute anything *computable*.  
  - . . . which is somewhat circular – oh well.  
  - The **Church-Turing Hypothesis** states that any algorithm (mathematical or computer) can be executed using a Turing Machine.

• A programming language (or piece of hardware) is called **Turing-complete** if it has the ability to simulate a Turing Machine.  
  - (Except that you must assume infinite memory.)  
  - C, Java, Python, assembly are all Turing-complete.
Speed of a Turing Machine

- In T.O.C. we also use the TM as a model for broad questions of speed.
- Time cost measured in number of TM steps.
  - Since TMs do not exist in physical reality (due to infinite tape), this is the best we can do.
Classes $P$ and $NP$

- Languages can be categorized into "classes" of similar difficulty or shared characteristics.

![Diagram of Famous Families of Languages]

- Languages come in various difficulty levels.
  (Difficulty of deciding membership, that is.)

- VERY EASY ("regular")
  - $\{w : w$ ends with the suffix "izzle"\}$
  - $\{w : w$ contains an odd number of 1's\}$

- EASY ("context-free")
  - $\{w : w$ is a palindrome\}$

- MEDIUM ("$P$")
  - $\{<G, T> : G$ represents a connected, weighted graph and $T$ is its minimum spanning tree\}$

- HARD ("$NP$")
  - $\{<G, P> : G$ represents a connected weighted graph, $P$ is the shortest path through every vertex\}$

- IMPOSSIBLE ("Undecidable")
  - $\{w : w$ represents Java code that will run error-free, and never enter an infinite loop\}$
Classes $P$ and $NP$

- Languages can be categorized into "classes" of similar difficulty or shared characteristics.
- There are hundreds of such classes:
  - [https://complexityzoo.uwaterloo.ca/Complexity_Zoo](https://complexityzoo.uwaterloo.ca/Complexity_Zoo)
- Two classes you should know the names of:
  - $P$
  - $NP$
Classes $P$ and $NP$

- Languages can be categorized into "classes" of similar difficulty or shared characteristics.
- There are hundreds of such classes:
  - [https://complexityzoo.uwaterloo.ca/Complexity_Zoo](https://complexityzoo.uwaterloo.ca/Complexity_Zoo)
- Two classes you should know the names of:
  - $P$ – the set of languages that are decidable on a Turing Machine in a polynomial number of time steps.
  - $NP$ – the set of languages that are verifiable on a Turing Machine in a polynomial number of time steps.
Overview of Theory of Computation

- What is Theory of Computation?
- Formal Languages
- Finite Automata and Regular Expressions
- $P \neq NP$?
- More about Turing Machines
- Undecidable Problems
P vs. NP

• This is the most important open problem in Computer Science.
  – Probably in all of Mathematics!

• Formally:
  Is P equal to NP, or not?

• Informally:
  Are there any problems in mathematics which are easy to check, but fundamentally hard to solve?
Class P is equivalent to the problems whose solutions can be found in polynomial time.

- Parameter $n$ represents the size of the input string.
- There exists some real $k$ such that $T(n) = O(n^k)$.
- Coarse: we don't even worry about the value of $k$.

Examples
- Sorting
- Integer multiplication and division
- Many types of searching
Worse than P?

- Many problems appear to be outside of P (a polynomial time algorithm seems impossible).
  - Integer factoring an $n$-bit integer
  - Searching among all possible subsets of $n$ items.

- Some have been proved to be outside of P (many have not).
Worse than P?

• Many problems that *appear* to be outside $P$ have exponential-time brute-force solutions. Examples:
  – Find the longest simple path in a graph.
  – Find shortest cycle for a traveling salesperson.

• If you could *prove* that its cost is $\Omega(2^n)$, then it's definitely outside of $P$ (by definition).

• But for lots and lots of problems, we don't know. Perhaps a polynomial-time algorithm exists!
Class NP (non-deterministic polynomial time) contains problems that can be checked in polynomial time, but no polynomial time solver is known.

Examples
- Factoring
- Boolean satisfiability
- Clique (a problem in graphs)
The Million Dollar Question is:

Does \( P = NP \) ?

That is:

Does every problem that has an inexpensive checking algorithm have an inexpensive solution algorithm – or not?

http://www.claymath.org/millennium-problems
N is for Non-Deterministic

- NP problems can be solved in polynomial time if you have a non-deterministic computer.

- A non-deterministic computer considers all possible solutions in parallel
  - Imagine having an infinite number of CPUs running in parallel
So Why “Non-Deterministic?”

- In an NP problem:
  - One of the solutions will succeed in polynomial time; or
  - All of the solutions will **fail** in polynomial time
TMs and P

• **P** can be formally defined as:
  - “The class of languages that have deciders that take polynomially many steps on an ordinary, deterministic Turing Machine.”

• **NP** can be alternately defined as:
  - “The class of languages that have deciders that take polynomially many steps on a nondeterministic Turing Machine.”
Non-Deterministic Computation

- Of course, non-deterministic computers don't exist!
- There's no great way to simulate nondeterminism except by brute force.
- That can take exponentially many steps!
- So NP problems often take exponential time when you solve them on deterministic computers.
• **NP-complete** problems are problems which can **simulate** any other NP problem.
  
  - Full explanation is weird and outside our scope.

• They are "the hardest" problems in NP:
  
  - If any NP-complete problem is in P, then **all** NP problems are in P.
  
  - If $P \neq NP$, then **no** NP-complete problem has a polynomial-time solution.
There are now **hundreds** of known NP-complete problems

- Boolean satisfiability (the first!)
- Clique
- Subset-sum
- Traveling salesperson
- Generalized Sudoku (!)
Decidable vs. Recognizable

- So far, we've thought of DFAs, NFAs, and TMs as "deciders." They run for finite time, and then answer ACCEPT or REJECT.

- Are there other models?
Decidable vs. Recognizable

- A **recognizer** is a program/automaton which will (eventually) **ACCEPT** every word in the language.
  - But it might loop forever if the word is **not** in the language, and thus never **REJECT**.

- If a recognizer is running for a long time, is it stuck in an infinite loop – or is it moving towards a future **ACCEPT** or **REJECT**?
The Halting Problem

Stated more generally:
If a program has been running for a long time, is it worthwhile to leave it running? Will it run forever, or will it eventually halt?

This is known as the Halting Problem.
The Halting Problem

• Turing famously proved that the Halting Problem is **undecidable**.
  - That is, you cannot in general decide (based on the source code) whether an arbitrary program is ever going to stop.
  - You can build a **recognizer** for the Halting Problem
    • Just simulate the program!
  - But a **decider** is a logical impossibility.
  - The proof is beautifully mindbending.

• For more, read Douglas Hofstadter, *Gödel, Escher and Bach, an Eternal Golden Braid*. 