Topic 17: Languages, P vs. NP, Decidability

- P vs. NP
- Languages
- Finite Automata
- Regular Expressions
- 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?
P vs. NP

Why is it important?

Public-Key Cryptography is based on the assumption that

\[ P \neq NP \]

If this assumption is wrong, then all encrypted traffic on the Internet can be cracked in polynomial time.
P vs. NP

- In Computer Science, a **class of problems** is a set of questions whose solutions have similar runtimes.

- In this slide deck, we only care about two major groups:
  - Polynomial: $O(n^k)$
  - Exponential: $O(2^n)$ (or worse)
Class $P$ is the problems whose solutions can be found in polynomial time

Examples
- Sorting
- Integer multiplication and division
- Many types of searching
- Finding combinations (when the count is fixed to size $k$)
Worse than P?

- Many problems appear to be **outside of P** (that is, no polynomial time algorithm is possible).
  - Integer factoring
  - Searching among all possible subsets
  - Finding combinations (when the count is unlimited)

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

- Problems that *appear* to be outside P typically have exponential-time solutions.
- If you can *prove* that it's exponential, then it's *definitely* outside of P (by definition).
- But for most, we don't yet know – perhaps a polynomial-time algorithm exists.
NP

- Class NP (non-deterministic polynomial) 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)
  - Traveling salesman
The Question is:

Does $P = NP$?

That is:

Does every problem that has an inexpensive checking algorithm have an inexpensive solution algorithm – or not?
So Why “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
**Class Exercise:**

Let's assign the following prime numbers, one to each student:

| 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 | 23 | 29 | 31 | 37 |

Now, each student will check to see if their prime # is a factor of the following numbers:
Non-Deterministic Computation

Class Exercise:

Let's assign the following prime numbers, one to each student:

2  3  5  7  11  13
17  19  23  29  31  37

Now, each student will check to see if their prime # is a factor of the following numbers:

123456
Class Exercise:

Let's assign the following prime numbers, one to each student:

2  3  5  7  11  13  
17  19  23  29  31  37

Now, each student will check to see if their prime # is a factor of the following numbers:

123456 = 2^6 x 3 x 643
Non-Deterministic Computation

Class Exercise:
Let's assign the following prime numbers, one to each student:
2 3 5 7 11 13
17 19 23 29 31 37

Now, each student will check to see if their prime # is a factor of the following numbers:

7654321
Non-Deterministic Computation

Class Exercise:

Let's assign the following prime numbers, one to each student:

2  3  5  7  11  13  17  19  23  29  31  37

Now, each student will check to see if their prime # is a factor of the following numbers:

7654321 = 19 x 402859
Class Exercise:

Let's assign the following prime numbers, one to each student:

\[\begin{array}{cccccccc}
2 & 3 & 5 & 7 & 11 & 13 \\
17 & 19 & 23 & 29 & 31 & 37 \\
\end{array}\]

Now, each student will check to see if their prime # is a factor of the following numbers:

9997999
Class Exercise:

Let's assign the following prime numbers, one to each student:

2  3  5  7  11  13
17 19 23 29 31 37

Now, each student will check to see if their prime # is a factor of the following numbers:

9997999 = 11 x 908909
Class Exercise:

Let's assign the following prime numbers, one to each student:

2  3  5  7  11  13  17  19  23  29  31  37

Now, each student will check to see if their prime # is a factor of the following numbers:

1234567
Class Exercise:

Let's assign the following prime numbers, one to each student:
2  3  5  7  11  13
17  19  23  29  31  37

Now, each student will check to see if their prime # is a factor of the following numbers:

$$1234567 = 127 \times 9721$$
Non-Deterministic Computation

No student reported that they knew of a factor of this number.

But a factor existed – we simply never checked it.

A Non-deterministic computer needs an unbounded number of CPUs in order to check all possible solutions. Any limited solution will fail when the problems get large enough.

\[1234567 = 127 \times 9721\]
Non-Deterministic Computation

• Of course, non-deterministic computers don't exist!

• Typically, the only known algorithm is to explore all possible solutions
  - Exponentially many!

• So NP problems typically take exponential time.
Solution as a “Witness”

• A solution for a problem in NP gives you a “witness” of the solution.
  – Instead of running non-deterministically, run a single computation (following the witness)
  – If it's really a solution, then you will prove that in polynomial time

• Thus:
  – Exponential to find
  – Polynomial to check
NP-complete

- **NP-complete** problems are problems which can **simulate** any other NP problem
  - Full explanation requires that we understand Turing Machines (later in this slide deck)

- Two key points:
  - If any NP-complete problem is in P, then all NP problems are in P
  - If P != 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 salesman
- Sudoku (!)
NP-complete

- In computer science:

  NP-complete = “seems simple...but awfully hard to do quickly”
Topic 17: Languages, P vs. NP, Decidability

- P vs. NP
- Languages
- Finite Automata
- Regular Expressions
- Turing Machines
- Undecidable Problems
Languages

- An **alphabet** is a set of symbols
  - We'll often use English alphabet, but it could be anything. It's just a **set**.

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

- A **language** is a set of words
Languages

Example Languages:

{ “aa”, “bb” }
All words that end in “x”
All words that are 10 symbols long
All palindromes
All words with the same number of “a”s as “b”s
Languages

More Complex Languages:

- Any word from language L, repeated any number of times
- Any sequence of words from language L
- Any word from languages L or M
- Any word that is in both L and M
- Any word from L, followed by a word from M
Languages and Data Structures

• Languages can represent **data structures** of arbitrary complexity
  - Use symbols to encode complex ideas
  - Have syntax and internal semantics

Examples:
  - All valid C programs
  - All prime numbers, encoded as bits
  - All pictures of the Mona Lisa, encoded as png
Languages and Data Structures

• Anything that a computer can store can be viewed as a language!

Actually, here's a language:

“All encodings of 64 billion bits which represent a data structure of type X”

… in other words …

“What I can store in 8 GB of memory.”
Languages and Computation

• Languages can be viewed as \textbf{computation}:

\textbf{Example:}

All encodings of problem $x$, where program $\gamma$ will give an affirmative answer on that input.

All encodings of problem $x$ and solution $x'$, where program $\gamma$ will agree that $x'$ is the solution of problem $x$. 
Languages and Computation

Viewing Computation as Input and Output

This machine represents a program which can multiply two integers.

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

Viewing Computation as Decider

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.
We can simulate the first machine by running the second machine many times.
Languages and Computation

Viewing Computation as Decider

(5,7), 33  →  Multiply-Check

→  Multiply-Check

→  Multiply-Check

→  Multiply-Check
Languages and Computation

Viewing Computation as Decider

(5,7),33

(5,7),34
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

We will often be fuzzy with our language.

We will talk about machines having “input and output,” but our formal machines will usually be deciders.
Languages and Computation

Another Example:

All words $PB$, where $P$ is a program written in the C language, and $B$ is the compiled binary generated from that program.

My point?

Arbitrarily complex programs can be viewed as languages, with “deciders” identifying the correct output for the program.
Topic 17: Languages, P vs. NP, Decidability

- P vs. NP
- Languages
- **Finite Automata**
- Regular Expressions
- Turing Machines
- Undecidable Problems
DFAs

- A **Deterministic Finite Automaton (DFA)** is a model for a very simple computer
  - **Cannot** do everything that full computer can do
  - But has more power than you might think!

- Simple state machine
  - Only memory is to keep track of “current state”
  - Follows one link per input symbol
  - Accepts if it is in an “accept state” at the end
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)
DFAs

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:

aabbbaaabbbaaabbba
DFAs

Example Input:

aabbbaaaaabba

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

Example Input:

aabbbaabbbaaaabbaaabba

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

Example Input:

aabbabbbbaaabba

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

Example Input:

aabbbbbaaaabba

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 **recognize**?
DFAs

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

- A **Nondeterministic Finite Automaton (NFA)** is a DFA, except nondeterministic
  - Nondeterministic = there are multiple things it might do in any given situation
  - Effectively, we have many, many DFAs running in parallel

- An NFA accepts an input if **any one of its copies** 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.
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.
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:
ab

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.
Example Input: aaaa

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.
NFAs

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.

ACCEPT!
NFAs

Class Exercise:
What language does this NFA recognize?
NFAs

Answer:
This NFA recognizes only two strings:
“aa”, “aaa”
Finite Automatons Are Limited

• A lot of simple languages cannot be recognized by DFAs or NFAs

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.
Topic 17: Languages, P vs. NP, Decidability

- P vs. NP
- Languages
- Finite Automata
- Regular Expressions
- Turing Machines
- Undecidable Problems
Regular Expressions

- Regular expressions are a convenient way to express certain simple languages
  - (Turns out, it's the exact same languages as DFAs/NFAs!)

- Three basic operations:
  - Sequence
  - Alternation
  - Repetition
Regular Expressions

• Regular expressions get used in the real world!

  - **PERL** *(Practical Extraction & Reporting Language)*
    Uses regular expressions for matching input strings
  - **grep** *(Global Regular Expression Print)*
    A *NIX tool for matching lines using regexps

  P.S. I used to think that grep was “Generalized Regular Expression Parser,”
  but it appears that I was wrong:
  Google: grep name origin
Regular Expressions

- The most basic element of a regular expression is a **single letter**. You can sequence 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><code>(ab* c) *</code></td>
<td>Zero or more a-c pairs, with zero or more 'b's within each pair.</td>
</tr>
</tbody>
</table>
Weird Equivalences

- DFA = NFA = Regular Expression
- 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
- They can decide the same languages!
Topic 17: Languages, P vs. NP, Decidability

- P vs. NP
- Languages
- Finite Automata
- Regular Expressions
- **Turing Machines**
- Undecidable Problems
Turing Machines

• A **Turing Machine** is a DFA with memory.
  - Typically imagined as a “tape” with a “read-write head” at one location

• Instead of simply reading from input, the links in a TM:
  - Read the current value on the tape
    • This replaces the simple “read from input” of a DFA
  - Write a new value
  - Move the head either direction
Turing Machines

- The **Church-Turing Hypothesis** states that any algorithm (mathematical or computer) can be executed using either:
  - A Turing Machine
  - Lambda Calculus (Church's invention)

- A programming language (or piece of hardware) is called **Turing-complete** if it has the ability to simulate a Turing Machine.
  - (If we assume that it has infinite memory)
Turing Machines

- A Turing Machine might be slow, but it's an OK model for a computer
  - Anything that a computer can do, the TM can do

- More importantly:
  - Any polynomial-time computer algorithm can be executed by a TM in polynomially-many steps
TMs and P

• **P** can be formally defined as:
  - “The class of problems which have deciders which take polynomially many steps on a Turing Machine.”

• **NP** can be formally defined as:
  - ???


TMs and P

• **P** can be formally defined as:
  - “The class of problems which have deciders which take polynomially many steps on a Turing Machine.”

• **NP** can be formally defined as:
  - “The class of problems which have deciders which take polynomially many steps on a nondeterministic Turing Machine.”
Topic 17: Languages, P vs. NP, Decidability

- P vs. NP
- Languages
- Finite Automata
- Regular Expressions
- Turing Machines
- Undecidable Problems
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**.
  - You can build a **recognizer** for the Halting Problem
    - Just simulate the program!
  - But a **decider** is a logical impossibility

- This problem ends up hitting us **everywhere**.
  - Professor Kececioglu once said in lecture:
    "Any non-trivial property of a program is undecidable."
The Halting Problem

That's a good end for 345.

There are many things we can know.

But there are some very simple, straightforward questions about programs, which no one can answer – unless they are truly omniscient.