CSc 473
Automata, Grammars and Languages

The Greatest Hits
What is Computation?

What makes a computer like a person?

Are there fundamental limits?

Is there a general model?
What is Computation?

Group Discussion:

In what ways is a computer algorithm like a person who is thinking?

What sorts of tasks do computers do well?

Where do they struggle?

Off limits: AI, Science Fiction
"I think you should be more explicit here in step two."
What is Computation?

Algorithms Need:
- Clear inputs and outputs
- Precise steps
- Limited duration (terminates)

Important Details:
- Precision
  - Accuracy, Irrational Numbers
- Representation / Encoding
What is Computation?

Insight:

- Every computer program has a precise mathematical description
- Every mathematical algorithm or proof can be encoded in a computer program

Thus:

Program = Mathematical Process
Machine = Abstract “Computation”
What is Computation?

Not Just Electronics

10,000 Domino Computer
https://www.youtube.com/watch?v=OpLU__bhu2w

16-bit ALU in Minecraft
https://www.youtube.com/watch?v=LGkkyKZVzug
What is Computation?

Not Just Electronics

Conway's Game of Life
https://www.youtube.com/watch?v=-FaqC4h5Ftg

Babbage's Difference Engine
https://www.youtube.com/watch?v=BlbQsKpq3Ak
What is Computation?

Not Just Electronics

Turing Machines
https://www.youtube.com/watch?v=dNRDvLACg5Q

Langston's Ant
https://www.youtube.com/watch?v=1X-gtr4pEBU
What is Computation?
Not Just Electronics

3SAT

\[(x_1 \lor x_2 \lor \overline{x_3}) \land (x_2 \lor x_3 \lor \overline{x_4}) \land (x_1 \lor \overline{x_2} \lor x_4)\]
What is Computation?

The Problem:
- How do we model computation in general?
- Is it possible to build a single model, which models all possible computations?
What is Computation?

2 Great Results (both 1936):

**Turing Machines**  
Alan Turing  
Abstract computer, based on discrete states and memory cells.

**Lambda Calculus**  
Alonzo Church  
Recursive mathematical formulae.  
Foundation of LISP.
What is Computation?

Church-Turing thesis:

These models are equivalent. Either of them can be used to model any computation.
What is Computation?

In this slide deck...
- Explore various “automata”
- Turing Machines are the most complex, most complete

Key questions:
- What questions can we ask?
- How fast are our algorithms?
- Are there any things that we cannot answer?
Greatest Hits of 473

What is a Formal Language?

Alphabets and strings

Strings as input data

Languages as computation 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.
• 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) → Multiply → 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

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
Languages and Computation

Viewing Computation as Decider

(5,7),33 → Multiply-Check

(5,7),34 → Multiply-Check

Multiply-Check

Multiply-Check

Multiply-Check
Languages and Computation

Viewing Computation as Decider

(5,7),33

(5,7),34

(5,7),35
These two machines have the **same computational power**. One is more efficient...but they can answer the same questions, given enough time.
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?
Formal 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 a set of strings.
Formal Languages

Simple Languages:

- All strings over the alphabet \{0, 1\}.
- \{1, 01, 101, 1101, 101101\}
- All strings that end in "111"
- All strings that are ten symbols long.
- All *palindromes*.
- All strings with equal numbers of 1s and 0s.
Suppose you have languages $L$ and $M$. We can derive new languages from them:

- Any word from language $L$ or from $M$.
- Any word that is common to both $L$ and $M$.
- Any word from language $L$, repeated any number of times.
- Any word from $L$, followed by a word from $M$.
- Any sequence of words from language $L$. 

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

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

<table>
<thead>
<tr>
<th>Difficulty Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERY EASY</td>
<td>({w : w \text{ ends with the suffix &quot;izzle&quot;}})</td>
</tr>
<tr>
<td></td>
<td>({w : w \text{ contains an odd number of 1's}})</td>
</tr>
<tr>
<td>EASY</td>
<td>({w : w \text{ is a palindrome}})</td>
</tr>
<tr>
<td>MEDIUM*</td>
<td>({&lt;G, T&gt; : G \text{ represents a connected, weighted graph and } T \text{ is its minimum spanning tree}})</td>
</tr>
<tr>
<td>HARD*</td>
<td>({&lt;G, S&gt; : G \text{ represents a connected weighted graph, } S \text{ is the shortest path through every vertex}})</td>
</tr>
<tr>
<td>IMPOSSIBLE</td>
<td>({w : w \text{ represents Java code that will run error-free, and never enter an infinite loop}})</td>
</tr>
</tbody>
</table>

* a bit oversimplified
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.
Greatest Hits of 473

Simple Automata

DFAs

NFAs
Finite Automata: DFAs, NFAs

• Some languages are very easy.
  – Don't need a full TM

• DFAs:
  – State machine
  – Follow one link, each time you read a symbol
  – Terminate at end of input
  – Ask: Did we end in an “accept” state?
Example DFA

\{w : w \text{ is a bit string with an odd number of 1's}\}

Double-circled nodes indicate accept states.
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.
DFAs

Example Input:

aabbbbbbaaaabba
Example Input:

aabbbbbbaaabbba

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

Example Input:

aabbbaaaaaabba

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

aabbbbbbaaabba

As we read each letter, the edge tells us where to go next.
Example Input:

`aabbbbbaaabba`
DFAs

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.
  - That is, they cannot test for correct multiplication.
DFAs

What does DFA stand for?

Deterministic – makes simple, clear choices
Finite – has a limited amount of memory
Automaton – abstract computation machine
Determinism

- DFAs always make simple choices
  - Read one symbol
  - Choose one new state to go to

- Never backtrack
- Never explore alternatives

How could we explore 2 alternatives?
Nondeterminism

• A nondeterministic automaton explores all possible alternatives...

  ...at the same time!

• Next up:

  Nondeterministic Finite Automatons (NFAs)
NFAs

- NFAs: exactly like DFAs, except
  - A state can have 2 (or more) outgoing links, for a single symbol
  - Explore all options in parallel
  - Accept if any of the copies is in an accept state at the end

Also: NFAs allow epsilon links.

Follow one for free, without reading any 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.
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:**

```
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.
NFAs

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 state 2 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 an 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"
NFAs

Class Exercise:

Devise an NFA which recognizes the following language:

“All strings over \{a,b,c\} that have an odd number of 'a' characters \textbf{or} an odd number of 'b' characters.”
DFAs = NFAs

Prepare to be Astonished:

DFAs and NFAs can recognize exactly the same languages!
DFAs = NFAs

• **Insight:**
  - DFAs and NFAs only remember their **current state**
    - (can be multiple for an NFA)
  - Cannot remember how they got there

• **Thus:**
  - If two “copies” in an NFA reach the same state, **merge them**
  - No need to keep redundant copies
DFAs = NFAs

Class Exercise:

Devise an NFA which recognizes the following language:

“All strings over \{a,b\} that contain at least three 'a' characters (not necessarily all in a row), OR contain at least three 'b' characters (not necessarily all in a row).”

Use as few states as possible.
Thought Experiment:

Suppose that an NFA has $k$ states. How many different possible combinations of states can be active at any given moment?

Could you represent each combination with a DFA state?

What then, would the DFA edges represent?
DFAs = NFAs

Class Exercise:

Devise an DFA (not an NFA!) which recognizes the following language:

“All strings over \{a,b,c\} that have an odd number of 'a' characters or an odd number of 'b' characters.”
NFA

DFA

(to simplify the picture, I omitted the self-links for c)
DFAs = NFAs

- Since every DFA is also an NFA...
- Since you can simulate any NFA with a (large) DFA...

- DFAs and NFAs recognize the same set of languages.

Use whichever one is convenient for your purpose.
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.
Greatest Hits of 473

Regular Expressions

- Pattern matching
- NFA equivalence
Regular Expressions

- Regular expressions are a convenient way to do pattern matching.
  - Matching patterns = deciding a language

\texttt{regexp:}

\[(\text{dog} | \text{cat}) \text{ food} | \text{doghouse}\]

\texttt{matches:}

\[\text{“dogfood”, “catfood”, “doghouse”}\]
regexp:
a*
matches:
"", "a", "aa", "aaa", ...

regexp:
(a|bc)*
matches:
"", "a", "bc", "aa", "abc", "bca", "bcbc",..., ...
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.
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

- Three basic operations:
  - Concatenation
  - Alternation
  - Repetition

- These are analogous to the basic operations of programming:
  - Sequencing, Selection, Iteration

https://www.101computing.net/sequencing-selection-iteration/
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>{ &quot;abc&quot; }</td>
</tr>
<tr>
<td>def</td>
<td>{ &quot;def&quot; }</td>
</tr>
<tr>
<td>jkl0123</td>
<td>{ &quot;jkl0123&quot; }</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><code>a*</code></td>
<td><code>{ &quot;&quot;, &quot;a&quot;, &quot;aa&quot;, ... }</code></td>
</tr>
<tr>
<td><code>aa*</code></td>
<td><code>{ &quot;a&quot;, &quot;aa&quot;, &quot;aaa&quot;, ... }</code></td>
</tr>
<tr>
<td><code>a*b*</code></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*c) *</td>
<td>Zero or more a-c pairs, with zero or more 'b's within each pair.</td>
</tr>
</tbody>
</table>
Class Exercise:

Devise regular expressions for each of the following languages (over the alphabet \{a,b,c\}):

“aa”, “aaa”

All strings containing three 'a's in a row, somewhere.

All strings containing \textbf{exactly} three 'a's, not necessarily in a row.

All strings containing an odd number of 'a's.

All strings containing an odd number of 'a's \textbf{OR} an odd number of 'b's.

All strings containing an odd number of 'a's \textbf{AND} an odd number of 'b's. (This is a lot harder.)
Converting a Regexp to an NFA

- We can simulate a regexp with an NFA
  - Start state represents beginning of expression
  - Accept state(s) represent valid end of expression

Class Exercise:

Devise NFAs which decide the same language as the following regexps:

a     a|ba     aa|aaa     a*     (a|b|c)xyz
Converting a Regexp to an NFA

Class Exercise:

Suppose you have two NFAs, each representing one regexp.

\[
\text{NFA } N_1 = \text{regexp } e_1
\]

\[
\text{NFA } N_2 = \text{regexp } e_2
\]

Build an NFA which represents concatenation \((e_1 e_2)\) of the two regexps.

What if the NFAs have multiple accept states???
Epsilon Links

We mentioned this in passing, earlier...

- NFAs support $\varepsilon$ (epsilon) links
  - Go from one state to another
  - But don't consume any input
Converting a Regexp to an NFA

\[
\text{NFA } N_1 = \text{regexp } e_1
\]

\[
\text{NFA } N_2 = \text{regexp } e_2
\]
Converting a Regexp to an NFA

Class Exercise:

Suppose you have two NFAs, each representing one regexp.

NFA $N_1 = \text{regexp } e_1$

NFA $N_2 = \text{regexp } e_2$

Build an NFA which represents alternation $(e_1 | e_2)$ between the two regexps.
Converting a Regexp to an NFA

\[ NFA_{N_1} = \text{regexp } e_1 \]

\[ NFA_{N_2} = \text{regexp } e_2 \]
Converting a Regexp to an NFA

Class Exercise:

Suppose you have one NFAs, representing a regexp.

\[ N_1 = \text{regexp } e_1 \]

Build an NFA which represents repetition \((e_1^*)\) of the regexp.
Converting a Regexp to an NFA

NFA \( N_1 \) = regexp \( e_1 \)
Surprising Equivalences

- You can also simulate any DFA with a regexp
  - Take 473 for the proof!

Thus:

DFA = NFA = Regular Expression
Greatest Hits of 473

Turing Machines

What if a DFA had memory?

What if a DFA could rewind through the data?
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.
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
Speed of a Turing Machine

- In Theory of Computation 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.

![Famous Families of Languages](image_url)

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

  - **VERY EASY** ("regular"): $\{ w : w $ ends with the suffix "izzle" $\}$
  - **EASY** ("context-free"): $\{ w : w $ contains an odd number of 1's $\}$
  - **MEDIUM** ("P"): $\{ w : w $ is a palindrome $\}$
  - **HARD** ("NP"): $\{ <G,T> : G $ represents a connected, weighted graph and $ T $ is its minimum spanning tree $\}$
  - **IMPOSSIBLE** ("Undecidable"): $\{ w : w $ represents Java code that will run error-free, and never enter an infinite loop $\}$

33
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
- 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.
Greatest Hits of 473

P vs NP

Two classic algorithm classes

Are they different or not?
P vs. NP

- This is the **most important open problem** in Computer Science.

- **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: $O(n^k)$ for some $k$
  - Exponential: $\Omega(2^n)$
Class P is 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!
NP

- **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.”
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
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 = 2^6 x 3 x 643
Class Exercise:

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

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
<td>31</td>
<td>37</td>
</tr>
</tbody>
</table>

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

7654321
## Class Exercise:

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

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
<td>31</td>
<td>37</td>
</tr>
</tbody>
</table>

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

\[ 7654321 = 19 \times 402859 \]
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:

9997999
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:

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:

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

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 x 9721
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.
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 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 ≠ NP, then no NP-complete problem has a polynomial-time solution.
NP-complete

- There are now hundreds of known NP-complete problems
  - Boolean satisfiability (the first!)
  - Clique
  - Subset-sum
  - Traveling salesperson
  - Generalized Sudoku (!)
NP-complete

• In computer science:

NP-complete =

“seems simple...but awfully hard to do quickly”
Greatest Hits of 473

Undecidability

Deciders vs. Recognizers

Undecidable languages
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**? (Or, is it going to loop forever?)
The Halting Problem

Stated more casually:

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*. 