Complexity Theory
Part One
It may be that since one is customarily concerned with existence, [...] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a better-than-finite algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”
WELCOME TO THEORYLAND
It may be that since one is customarily concerned with existence, [...] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a better-than-finite algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”
It may be that since one is customarily concerned with existence, [...] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a better-than-finite algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”
It may be that since one is customarily concerned with existence, [...] decidability, and so forth, one is not inclined to take seriously the question of the existence of a better-than-decidable algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”
A Decidable Problem

- **Presburger arithmetic** is a logical system for reasoning about arithmetic.
  - $\forall x. x + 1 \neq 0$
  - $\forall x. \forall y. (x + 1 = y + 1 \rightarrow x = y)$
  - $\forall x. x + 0 = x$
  - $\forall x. \forall y. (x + y) + 1 = x + (y + 1)$
  - $(P(0) \land \forall y. (P(y) \rightarrow P(y + 1))) \rightarrow \forall x. P(x)$

- Given a statement, it is decidable whether that statement can be proven from the laws of Presburger arithmetic.

- Any Turing machine that decides whether a statement in Presburger arithmetic is true or false has to move its tape head at least $2^{cn}$ times on some inputs of length $n$ (for some fixed constant $c \geq 1$).
For Reference

- Assume $c = 1$. 
The Limits of Decidability

- The fact that a problem is decidable does not mean that it is *feasibly* decidable.
- In *computability theory*, we ask the question: What problems can be solved by a computer?
- In *complexity theory*, we ask the question: What problems can be solved *efficiently* by a computer?
- In the remainder of this course, we will explore this question in more detail.
Where We've Been

- The class $R$ represents problems that can be solved by a computer.
- The class $RE$ represents problems where “yes” answers can be verified by a computer.
Where We're Going

• The class $\mathbf{P}$ represents problems that can be solved *efficiently* by a computer.

• The class $\mathbf{NP}$ represents problems where “yes” answers can be verified *efficiently* by a computer.
Undecidable Languages

Regular Languages

CFLs

Efficiently Decidable Languages

Undecidable Languages
The Setup

• In order to study computability, we needed to answer these questions:
  • What is “computation?”
  • What is a “problem?”
  • What does it mean to “solve” a problem?
• To study complexity, we need to answer these questions:
  • What does “complexity” even mean?
  • What is an “efficient” solution to a problem?
Measuring Complexity

- Suppose that we have a decider $D$ for some language $L$.
- How might we measure the complexity of $D$?
Measuring Complexity

- Suppose that we have a decider \( D \) for some language \( L \).
- How might we measure the complexity of \( D \)?

Answer at PollEv.com/cs103 or text CS103 to 22333 once to join, then your answer.
Measuring Complexity

- Suppose that we have a decider $D$ for some language $L$.
- How might we measure the complexity of $D$?
  - Number of states.
  - Size of tape alphabet.
  - Size of input alphabet.
  - Amount of tape required.
  - Amount of time required.
  - Number of times a given state is entered.
  - Number of times a given symbol is printed.
  - Number of times a given transition is taken.
  - (Plus a whole lot more...)
Measuring Complexity

• Suppose that we have a decider $D$ for some language $L$.
• How might we measure the complexity of $D$?
  Number of states.
  Size of tape alphabet.
  Size of input alphabet.
  Amount of tape required.
• Amount of time required.
  Number of times a given state is entered.
  Number of times a given symbol is printed.
  Number of times a given transition is taken.
  (Plus a whole lot more...)
What is an efficient algorithm?
Searching Finite Spaces

• Many decidable problems can be solved by searching over a large but finite space of possible options.

• Searching this space might take a staggeringly long time, but only finite time.

• From a decidability perspective, this is totally fine.

• From a complexity perspective, this may be totally unacceptable.
A Sample Problem

4 3 11 9 7 13 5 6 1 12 2 8 0 10
A Sample Problem

Goal: Find the length of the longest increasing subsequence of this sequence.
A Sample Problem

Goal: Find the length of the longest increasing subsequence of this sequence.
A Sample Problem

| 4 | 3 | 11 | 9 | 7 | 13 | 5 | 6 | 1 | 12 | 2 | 8 | 0 | 10 |

Goal: Find the length of the longest increasing subsequence of this sequence.
Longest Increasing Subsequences

- **One possible algorithm:** try all subsequences, find the longest one that's increasing, and return that.
- There are $2^n$ subsequences of an array of length $n$.
  - (Each subset of the elements gives back a subsequence.)
- Checking all of them to find the longest increasing subsequence will take time $O(n \cdot 2^n)$.
- Nifty fact: the age of the universe is about $4.3 \times 10^{26}$ nanoseconds old. That's about $2^{85}$ nanoseconds.
- Practically speaking, this algorithm doesn't terminate if you give it an input of size 100 or more.
Longest Increasing Subsequences

- **Theorem:** There is an algorithm that can find the longest increasing subsequence of an array in time \(O(n \log n)\).

- The algorithm is *beautiful* and surprisingly elegant. Look up *patience sorting* if you're curious.

- This algorithm works by exploiting particular aspects of how longest increasing subsequences are constructed. It's not immediately obvious that it works correctly.
Another Problem

A -- B -- C
|     |     |
|     |     |
D -- E -- F
Another Problem
Another Problem

Goal: Determine the length of the shortest path from A to F in this graph.
Shortest Paths

• It is possible to find the shortest path in a graph by listing off all sequences of nodes in the graph in ascending order of length and finding the first that's a path.

• This takes time $O(n \cdot n!)$ in an $n$-node graph.

• For reference: 29! nanoseconds is longer than the lifetime of the universe.
Shortest Paths

- **Theorem:** It's possible to find the shortest path between two nodes in an $n$-node, $m$-edge graph in time $O(m + n)$.
- **Proof idea:** Use breadth-first search!
- The algorithm is a bit nuanced. It uses some specific properties of shortest paths and the proof of correctness is nontrivial.
For Comparison

- **Longest increasing subsequence:**
  - Naive: $O(n \cdot 2^n)$
  - Fast: $O(n^2)$

- **Shortest path problem:**
  - Naive: $O(n \cdot n!)$
  - Fast: $O(n + m)$.
Defining Efficiency

• When dealing with problems that search for the “best” object of some sort, there are often at least exponentially many possible options.

• Brute-force solutions tend to take at least exponential time to complete.

• Clever algorithms often run in time $O(n)$, or $O(n^2)$, or $O(n^3)$, etc.
Polynomials and Exponentials

• An algorithm runs in \textit{polynomial time} if its runtime is some polynomial in \( n \).
  • That is, time \( O(n^k) \) for some constant \( k \).
• Polynomial functions “scale well.”
  • Small changes to the size of the input do not typically induce enormous changes to the overall runtime.
• Exponential functions scale terribly.
  • Small changes to the size of the input induce huge changes in the overall runtime.
The Cobham-Edmonds Thesis

A language $L$ can be \textit{decided efficiently} if there is a TM that decides it in polynomial time.

Equivalently, $L$ can be decided efficiently if it can be decided in time $O(n^k)$ for some $k \in \mathbb{N}$.

Like the Church-Turing thesis, this is \textit{not} a theorem!

It's an assumption about the nature of efficient computation, and it is somewhat controversial.
The Cobham-Edmonds Thesis

According to the Cobham-Edmonds thesis, how many of the following runtimes are considered efficient?

\[ 4n^2 - 3n + 137 \]
\[ 10^{500} \]
\[ 2^n \]
\[ 1.000000000000001^n \]
\[ n^{1,000,000,000,000} \]
\[ n^{\log n} \]

Answer at PollEv.com/cs103 or text CS103 to 22333 once to join, then a number.
The Cobham-Edmonds Thesis

- Efficient runtimes:
  - \(4n + 13\)
  - \(n^3 - 2n^2 + 4n\)
  - \(n \log \log n\)
- “Efficient” runtimes:
  - \(n^{1,000,000,000,000}\)
  - \(10^{500}\)

- Inefficient runtimes:
  - \(2^n\)
  - \(n!\)
  - \(n^n\)
- “Inefficient” runtimes:
  - \(n^{0.0001 \log n}\)
  - \(1.000000001^n\)
Why Polynomials?

- Polynomial time *somewhat* captures efficient computation, but has a few edge cases.
- However, polynomials have very nice mathematical properties:
  - The sum of two polynomials is a polynomial. (Running one efficient algorithm, then another, gives an efficient algorithm.)
  - The product of two polynomials is a polynomial. (Running one efficient algorithm a “reasonable” number of times gives an efficient algorithm.)
  - The *composition* of two polynomials is a polynomial. (Using the output of one efficient algorithm as the input to another efficient algorithm gives an efficient algorithm.)
The Complexity Class $\mathbf{P}$

- The *complexity class* $\mathbf{P}$ (for *polynomial* time) contains all problems that can be solved in polynomial time.

- Formally:
  
  $\mathbf{P} = \{ L \mid \text{There is a polynomial-time decider for } L \}$

- Assuming the Cobham-Edmonds thesis, a language is in $\mathbf{P}$ if it can be decided efficiently.
Examples of Problems in $\mathbf{P}$

- All regular languages are in $\mathbf{P}$.
  - All have linear-time TMs.
- All CFLs are in $\mathbf{P}$.
  - Requires a more nuanced argument (the CYK algorithm or Earley's algorithm.)
- And a *ton* of other problems are in $\mathbf{P}$ as well.
  - Curious? Take CS161!
Undecidable Languages

Regular Languages

CFLs

P

R

Undecidable Languages
What *can't* you do in polynomial time?
How many simple paths are there from the start node to the end node?
How many subsets of this set are there?
An Interesting Observation

- There are (at least) exponentially many objects of each of the preceding types.
- However, each of those objects is not very large.
  - Each simple path has length no longer than the number of nodes in the graph.
  - Each subset of a set has no more elements than the original set.
- This brings us to our next topic...
What if you need to search a large space for a single object?
Verifiers – Again

Does this Sudoku problem have a solution?
Verifiers – Again

Does this Sudoku problem have a solution?

```
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Verifiers – Again

| 9 | 3 | 11 | 4 | 2 | 13 | 5 | 6 | 1 | 12 | 7 | 8 | 0 | 10 |

Is there an ascending subsequence of length at least 7?
Verifiers – Again

Is there an ascending subsequence of length at least 7?
Verifiers – Again

Is there a simple path that goes through every node exactly once?
Verifiers – Again

Is there a simple path that goes through every node exactly once?
Verifiers

- Recall that a **verifier** for $L$ is a TM $V$ such that
  - $V$ halts on all inputs.
  - $w \in L$ iff $\exists c \in \Sigma^*$. $V$ accepts $\langle w, c \rangle$. 
Polynomial-Time Verifiers

• A *polynomial-time verifier* for $L$ is a TM $V$ such that
  • $V$ halts on all inputs.
  • $w \in L$ iff $\exists c \in \Sigma^*$. $V$ accepts $\langle w, c \rangle$.
  • $V$'s runtime is a polynomial in $|w|$ (that is, $V$'s runtime is $O(|w|^k)$ for some integer $k$)
The Complexity Class $\textbf{NP}$

- The complexity class $\textbf{NP}$ (\textit{nondeterministic polynomial time}) contains all problems that can be verified in polynomial time.
- Formally:

$$\textbf{NP} = \{ L \mid \text{There is a polynomial-time verifier for } L \}$$

- The name $\textbf{NP}$ comes from another way of characterizing $\textbf{NP}$. If you introduce \textit{nondeterministic Turing machines} and appropriately define “polynomial time,” then $\textbf{NP}$ is the set of problems that an NTM can solve in polynomial time.
- Although it’s not immediately obvious, $\textbf{NP} \subsetneq \textbf{R}$. Come talk to me after class if you’re curious why!
And now...
The Most Important Question in Theoretical Computer Science
What is the connection between \textbf{P} and \textbf{NP}?
\[ \mathbf{P} = \{ L \mid \text{There is a polynomial-time decider for } L \} \]

\[ \mathbf{NP} = \{ L \mid \text{There is a polynomial-time verifier for } L \} \]
\[ P = \{ L \mid \text{There is a polynomial-time decider for } L \} \]

\[ NP = \{ L \mid \text{There is a polynomial-time verifier for } L \} \]

\[ P \subseteq NP \]

- **Input string** (w)
- **Certificate** (c)

**Polynomial-Time Verifier for L**

- **yes!**
- **no!**

(ignored)
Which Picture is Correct?
Which Picture is Correct?
Does $P = NP$?
The $P \equiv NP$ question is the most important question in theoretical computer science.

With the verifier definition of $NP$, one way of phrasing this question is

*If a solution to a problem can be checked efficiently, can that problem be solved efficiently?*

An answer either way will give fundamental insights into the nature of computation.
Why This Matters

- The following problems are known to be efficiently verifiable, but have no known efficient solutions:
  - Determining whether an electrical grid can be built to link up some number of houses for some price (Steiner tree problem).
  - Determining whether a simple DNA strand exists that multiple gene sequences could be a part of (shortest common supersequence).
  - Determining the best way to assign hardware resources in a compiler (optimal register allocation).
  - Determining the best way to distribute tasks to multiple workers to minimize completion time (job scheduling).
  - *And many more.*

- If $P = NP$, *all* of these problems have efficient solutions.
- If $P \neq NP$, *none* of these problems have efficient solutions.
Why This Matters

• If $P = NP$:
  • A huge number of seemingly difficult problems could be solved efficiently.
  • Our capacity to solve many problems will scale well with the size of the problems we want to solve.

• If $P \neq NP$:
  • Enormous computational power would be required to solve many seemingly easy tasks.
  • Our capacity to solve problems will fail to keep up with our curiosity.
What We Know

- Resolving $P \neq NP$ has proven extremely difficult.
- In the past 45 years:
  - Not a single correct proof either way has been found.
  - Many types of proofs have been shown to be insufficiently powerful to determine whether $P \neq NP$.
  - A majority of computer scientists believe $P \neq NP$, but this isn't a large majority.
- Interesting read: Interviews with leading thinkers about $P \neq NP$:
  - [http://web.ing.puc.cl/~jabaier/iic2212/poll-1.pdf](http://web.ing.puc.cl/~jabaier/iic2212/poll-1.pdf)
The Million-Dollar Question

The Clay Mathematics Institute has offered a $1,000,000 prize to anyone who proves or disproves $P = NP$. 

CHALLENGE ACCEPTED
Do you think $P = NP$?
Time-Out for Announcements!
Please evaluate this course in Axess.
Your comments really make a difference.
Problem Set Nine

• Problem Set Nine is due this Friday at 2:30PM.
  • As a reminder, *no late submissions will be accepted*. Please budget enough time to get your submission in!
  • *Very smart idea*: submit at least three hours early.
• As always, feel free to ask questions in office hours or online via Piazza.
Final Exam Logistics

• Our final exam is Monday, March 19th from 3:30PM – 6:30PM, location Hewlett 200 & 201 (no special last name assignments).
  • Sorry about how soon that is – the registrar picked this time, not us. If we had a choice, it would be on the last day of finals week.
• The exam is cumulative. You’re responsible for topics from PS1 – PS9 and all of the lectures.
• As with the midterms, the exam is closed-book, closed-computer, and limited-note. You can bring one double-sided sheet of 8.5” × 11” notes with you to the exam, decorated any way you’d like.
• Students with OAE accommodations: if we don’t yet have your OAE letter, please send it to us ASAP.
Preparing for the Final

- On the course website you’ll find
  - *six* practice final exams, which are all real exams with minor modifications, with solutions, and
  - a giant set of 46 practice problems (EPP3), with solutions.
- Our recommendation: Look back over the exams and problem sets and redo any problems that you didn’t really get the first time around.
- Keep the TAs in the loop: stop by office hours to have them review your answers and offer feedback.
Practice Final Exam

- If you’re interested in attending a proctored practice final exam this Wednesday from 7PM - 10PM, please send us an email by the end of the evening.
- We can then book a space with enough room to hold everyone.
Back to CS103!
What do we know about $P \not= NP$?
Adapting our Techniques
A Problem

• The **R** and **RE** languages correspond to problems that can be decided and verified, *period*, without any time bounds.

• To reason about what's in **R** and what's in **RE**, we used two key techniques:
  
  • *Universality*: TMs can run other TMs as subroutines.
  
  • *Self-Reference*: TMs can get their own source code.

• Why can't we just do that for **P** and **NP**?
Theorem (Baker-Gill-Solovay): Any proof that purely relies on universality and self-reference cannot resolve $P \neq NP$.

Proof: Take CS154!
So how are we going to reason about $P$ and $NP$?
Next Time

- **Reducibility**
  - A technique for connecting problems to one another.

- **NP-Completeness**
  - What are the hardest problems in $\mathbf{NP}$?