CS 341: ALGORITHMS
Trevor Brown

trevor.brown@uwaterloo.ca

DC 2338, Office hour M3-4pm
THIS TIME

- Announcement: course evaluations
- Intractability (hardness of problems)
  - Complexity class NP
  - Polynomial transformations
Algorithm Solving a Decision Problem: An algorithm $A$ is said to solve a decision problem $\Pi$ provided that $A$ finds the correct answer ("yes" or "no") for every instance $I$ of $\Pi$ in finite time.

Polynomial-time Algorithm: An algorithm $A$ for a decision problem $\Pi$ is said to be a polynomial-time algorithm provided that the complexity of $A$ is $O(n^k)$, where $k$ is a positive integer and $n = \text{Size}(I)$.

The Complexity Class $\mathbf{P}$ denotes the set of all decision problems that have polynomial-time algorithms solving them. We write $\Pi \in \mathbf{P}$ if the decision problem $\Pi$ is in the complexity class $\mathbf{P}$. 
### Travelling Salesperson Problems

<table>
<thead>
<tr>
<th>Problem 7.5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TSP-Optimization</strong></td>
</tr>
<tr>
<td><strong>Instance:</strong> A graph $G$ and edge weights $w : E \to \mathbb{Z}^+$.</td>
</tr>
<tr>
<td><strong>Find:</strong> A hamiltonian cycle $H$ in $G$ such that $w(H) = \sum_{e \in H} w(e)$ is minimized.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem 7.6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TSP-Optimal Value</strong></td>
</tr>
<tr>
<td><strong>Instance:</strong> A graph $G$ and edge weights $w : E \to \mathbb{Z}^+$.</td>
</tr>
<tr>
<td><strong>Find:</strong> The minimum $T$ such that there exists a hamiltonian cycle $H$ in $G$ with $w(H) = T$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem 7.7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TSP-Decision</strong></td>
</tr>
<tr>
<td><strong>Instance:</strong> A graph $G$, edge weights $w : E \to \mathbb{Z}^+$, and a target $T$.</td>
</tr>
<tr>
<td><strong>Question:</strong> Does there exist a hamiltonian cycle $H$ in $G$ with $w(H) \leq T$?</td>
</tr>
</tbody>
</table>

---

Last time we saw all three of these are **equivalent** (modulo polynomial factors). If one can be solved in poly-time then all can.
COMPLEXITY CLASS \textbf{NP}

NP: Non-deterministic polynomial time
Suppose I give you a certificate consisting of an array of numbers, and claim it represents such a subset. Of course, I might lie and give you a subset that does not sum to zero…

If I’m telling the truth, then we call this a yes-certificate. It is essentially a proof that “yes” is the correct output.

Can you use a yes-certificate to solve the problem efficiently?

Can you efficiently determine whether I am lying?

Finding such a subset can be extremely difficult.

Can you use a yes-certificate to solve the problem efficiently?

Of course, I might lie and give you a subset that does not sum to zero…

I could even give you numbers that are not in the input…

EXAMPLE: SUBSET-SUM PROBLEM

• Suppose we are given some integers, -7, -3, -2, 5, 8

• Does some subset of these sum to zero?
  • In this case, yes: (-3) + (-2) + 5 = 0
Certificates

Certificate: Informally, a certificate for a yes-instance \( I \) is some “extra information” \( C \) which makes it easy to verify that \( I \) is a yes-instance.

Certificate Verification Algorithm: Suppose that \( Ver \) is an algorithm that verifies certificates for yes-instances. Then \( Ver(I, C) \) outputs “yes” if \( I \) is a yes-instance and \( C \) is a valid certificate for \( I \). If \( Ver(I, C) \) outputs “no”, then either \( I \) is a no-instance, or \( I \) is a yes-instance and \( C \) is an invalid certificate.

Polynomial-time Certificate Verification Algorithm: A certificate verification algorithm \( Ver \) is a polynomial-time certificate verification algorithm if the complexity of \( Ver \) is \( O(n^k) \), where \( k \) is a positive integer and \( n = \text{Size}(I) \).

For example, for subset-sum, a correct \( Ver(I, C) \) should return “yes” only if \( C \subseteq I \) and \( \text{sum}(C) = 0 \)

It can be hard to define a certificate for a no-instance... E.g., how to create a certificate that proves no subset sums to 0?
**SUBSET-SUM: ALGORITHM VIA VERIFYING CERTIFICATES**

1. `SubsetSum(X[1..n])`
2.   for every possible subset $S$ of $X$
3.     if `sumsToZero(S)` then return true
4. return false

If any certificate $S$ sums to zero, it is a **yes-certificate** (a proof that the answer to the decision problem is “true”), and we return true.

Generating all certificates is expensive; exponential time!

But **verifying one** certificate is fast; runtime is $O(|S|)$

This is polynomial in the input size

A certificate that does **not** sum to zero doesn’t really prove anything (would need to know that all certificates sum to non-zero)

What does this brute force solution have to do with **NP**?

**Type of a certificate:**
- set of integers

Generate every subset certificate $S$

Verify certificate $S$
- (valid + sums to zero)
Given such an oracle, this algorithm would **solve** subset-sum in poly-time

```plaintext
SubsetSumWithOracle(I)
  C = Oracle(I)
  return verify(I, C)
```

```plaintext
verify(I, C)
  if C not subset of I then return false
  return (sum(C) == 0)
```

Suppose instead of generating every possible subset, there exists a poly-time **non-deterministic oracle** which magically returns a **subset that sums to 0** if one exists and otherwise returns **any set of integers**.

Non-deterministic is the N in NP: “Non-deterministic polynomial time”

The “non-deterministic” part of the oracle is how it “magically returns” a yes-certificate if one exists.

Otherwise, either C is not a subset of the input (return false), or C sums to a non-zero value (return false).
**GENERALIZING BEYOND SUBSET-SUM**

- You can solve **any decision problem** in non-deterministic poly-time if you have:
  1. a poly-time non-deterministic **oracle**, and
  2. a poly-time **verify** algorithm
- Such that:
  - If $I$ is a **yes**-instance, then the oracle returns a **yes**-certificate $C$ (i.e., a “proof” the answer is “yes”) and $verify(I, C)$ returns **true**
  - If $I$ is a **no**-instance, then $verify(I, C)$ returns **false** for all $C$ (i.e., it must be impossible to fool $verify$ into returning **true**)
- The algorithm:
  ```
  1. SolveAnyProblemWithOracle(I)
  2. C = Oracle(I)
  3. return verify(I, C)
  ```

Our definition of NP will **not** explicitly involve non-deterministic oracles. But it is based on **certificate verification**, which really only makes sense if you think of such oracles.

Could you “fool” the subset-sum verify function?
**DEFINING NP**

Intuition: For a yes-instance, there must exist some certificate that verify would accept (and, if one exists, the oracle would find it, solving the problem). For a no-instance, verify must always reject.

- A decision problem $\Pi$ is **solved** by a poly-time $\text{verify}$ alg. if:
  - for every **yes**-instance $I$, there exists a certificate $C$ such that $\text{verify}(I, C)$ returns true, and
  - for every **no**-instance $I$, $\text{verify}(I, C)$ returns $\text{false}$ for every $C$
  - The complexity class **NP** denotes the set of all decision problems that can be solved by poly-time $\text{verify}$ algorithms
  - Note: it is **not** necessary to be able to implement an oracle for a problem to be in NP. We can simply **assume** an oracle exists, and show a poly-time $\text{verify}$ algorithm exists.
Always keep the following in mind: finding a certificate can be much more difficult than verifying a given certificate.

As a rough analogy, finding a proof for a theorem can be much harder than verifying the correctness of someone else's proof.
MECHANICS OF SHOWING A PROBLEM IS IN NP

• **Recall:** A decision problem \( \Pi \) is **solved** by a poly-time **verify** alg. if:
  
  • for every **yes**-instance \( I \), **there exists** a certificate \( C \) such that \( \text{verify}(I, C) \) returns true, and
  
  • for every **no**-instance \( I \), \( \text{verify}(I, C) \) returns false for **every** \( C \)

• How to show \( \Pi \in NP \)
  
  1. Define a class of certificates (e.g., sets of integers)
  2. Design a poly-time \( \text{verify}(I, C) \) algorithm
  3. Prove it is correct (case 1): Let \( I \) be any yes-instance; Find \( C \) such that \( \text{verify}(I, C) = true \)
  4. Case 2: Let \( I \) be any no-instance, and \( C \) be any certificate; Show \( \text{verify}(I, C) = false \)
Let's show that this problem is in NP!

Have to find a poly-time verify algorithm...

Type of certificate? Array of nodes (which may or may not represent a Hamiltonian cycle)

How to verify that a given array of nodes represents a cycle?

How about a Hamiltonian cycle?

Problem 7.2

Hamiltonian Cycle

Instance: An undirected graph $G = (V, E)$.

Question: Does $G$ contain a hamiltonian cycle?

A hamiltonian cycle is a cycle that passes through every vertex in $V$ exactly once.
This is a verify algorithm that we imagine being called on the certificate $X$ produced by $oracle(G)$.

If $G$ is a no-instance of the problem, then every certificate should cause this procedure to return false.

If $G$ is a yes-instance of the problem, then must show exists some certificate $X$ for which this procedure returns true.

Easier to prove contrapositive: if we return true, then $G$ is a yes-instance.

Yes-instance implies there is a Hamiltonian cycle. Suppose $X$ is a sequence of $n$ consecutive nodes on that cycle. Then we return true!

So, this problem is in NP.
HOW ARE P AND NP RELATED?

• $P \subseteq NP$
  • Consider a problem $\Pi \in P$
  • We show there exists a poly-time $verify(I, C)$ such that:
    • For every yes-instance $I$ of $\Pi$, $verify(I, C) = true$ for some $C$
    • For every no-instance $I$ of $\Pi$, $verify(I, C) = false$ for all $C$
  • By definition, there is a poly-time algorithm $A$ to solve $\Pi$
    • Implement $verify(I, C)$ by simply running $A(I)$
    • Regardless of what $C$ is, $verify(I, C)$ satisfies the above
• How about $NP \subseteq P$? Million dollar question. We think not.
POLYNOMIAL TRANSFORMATIONS

Formally defining poly-time reductions

Used for NP-completeness and impossibility results
POLYNOMIAL TRANSFORMATIONS

For a decision problem $\Pi$, let $\mathcal{I}(\Pi)$ denote the set of all instances of $\Pi$. Let $\mathcal{I}_{\text{yes}}(\Pi)$ and $\mathcal{I}_{\text{no}}(\Pi)$ denote the set of all yes-instances and no-instances (respectively) of $\Pi$.

Suppose that $\Pi_1$ and $\Pi_2$ are decision problems. We say that there is a polynomial transformation from $\Pi_1$ to $\Pi_2$ (denoted $\Pi_1 \leq_P \Pi_2$) if there exists a function $f : \mathcal{I}(\Pi_1) \rightarrow \mathcal{I}(\Pi_2)$ such that the following properties are satisfied:

- $f(I)$ is computable in polynomial time (as a function of $\text{size}(I)$, where $I \in \mathcal{I}(\Pi_1)$)
- if $I \in \mathcal{I}_{\text{yes}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\text{yes}}(\Pi_2)$
- if $I \in \mathcal{I}_{\text{no}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\text{no}}(\Pi_2)$

To argue there is a polynomial transformation, you must give $f(I)$, show it runs in poly-time, and show it satisfies these properties.

So, after transforming $\Pi_1$'s input, you can run a solution to $\Pi_2$ and just return the result!
A polynomial transformation can be thought of as a (simple) special case of a polynomial-time Turing reduction, i.e., if $\Pi_1 \leq_P \Pi_2$, then $\Pi_1 \leq^T_P \Pi_2$.

Given a polynomial transformation $f$ from $\Pi_1$ to $\Pi_2$, the corresponding Turing reduction is as follows:

Given $I \in \mathcal{I}(\Pi_1)$, construct $f(I) \in \mathcal{I}(\Pi_2)$.

Given an oracle for $\Pi_2$, say $A$, run $A(f(I))$.

We transform the instance, and then make a single call to the oracle.

Very important point: We do not know whether $I$ is a yes-instance or a no-instance of $\Pi_1$ when we transform it to an instance $f(I)$ of $\Pi_2$.

To prove the implication “if $I \in \mathcal{I}_{\text{no}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\text{no}}(\Pi_2)$”, we usually prove the contrapositive statement “if $f(I) \in \mathcal{I}_{\text{yes}}(\Pi_2)$, then $I \in \mathcal{I}_{\text{yes}}(\Pi_1)$.”

This can help when it is hard to precisely characterize certificates for no-instances (or when such certificates don’t prove much).

Also known as Karp reductions and many-one reductions

We haven’t solved the problem yet, so we don’t know much about the input...

We saw one instance where a contrapositive was easier to prove when we discussed Hamiltonian cycles.
SUMMARIZING
THE MORE CONVENIENT DEFINITION

• Let $\Pi_1$ and $\Pi_2$ be decision problems
• $\Pi_1 \leq_P \Pi_2$ iff there exists $f : I(\Pi_1) \rightarrow I(\Pi_2)$ such that:
  • $f(I)$ is computable in poly-time, for all $I \in I(\Pi_1)$
  • If $I \in I_{yes}(\Pi_1)$ then $f(I) \in I_{yes}(\Pi_2)$
  • If $f(I) \in I_{yes}(\Pi_2)$ then $I \in I_{yes}(\Pi_1)$

This is the contrapositive. Was previously:
If $I \in I_{no}(\Pi_1)$ then $f(I) \in I_{no}(\Pi_2)$

Note: this is the same as saying
$(I \in I_{yes}(\Pi_1)) \iff (f(I) \in I_{yes}(\Pi_2))$
**Example Polynomial Transformation**

**Problem 7.8**

**Clique**

**Instance:** An undirected graph $G = (V, E)$ and an integer $k$, where $1 \leq k \leq |V|$.

**Question:** Does $G$ contain a clique of size $\geq k$? (A **clique** is a subset of vertices $W \subseteq V$ such that $uv \in E$ for all $u, v \in W$, $u \neq v$.)

**Problem 7.9**

**Vertex Cover**

**Instance:** An undirected graph $G = (V, E)$ and an integer $k$, where $1 \leq k \leq |V|$.

**Question:** Does $G$ contain a vertex cover of size $\leq k$? (A **vertex cover** is a subset of vertices $W \subseteq V$ such that $\{u, v\} \cap W \neq \emptyset$ for all edges $uv \in E$.)

---

**Every edge** must touch a **node** in $W$.

These $k$ nodes touch **every edge** in $G$.

Actual reduction will be done **next time**.