Announcements

• HW 10 in
  • Due Monday 12/5 11:59pm

• HW 11 out
  • Due Monday 12/12 11:59pm

• HW 12
  • Out Tuesday 12/13
  • Due Monday 12/20 11:59pm
Last Time: Poly Time Complexity Class (P)

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

\[ P = \bigcup_k \text{TIME}(n^k). \]

• Corresponds to “realistically” solvable problems:
  • Problems in P
    • = “solvable” or “tractable”
  • Problems outside P
    • = “unsolvable” or “intractable”
Last Time: 3 Problems in $\mathbb{P}$

- A **Graph** Problem:
  \[ \text{PATH} = \{ (G, s, t) \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t \} \]
  
- A **Number** Problem:
  \[ \text{RELPRIME} = \{ (x, y) \mid x \text{ and } y \text{ are relatively prime} \} \]

- A **CFL** Problem:
  Every context-free language is a member of $\mathbb{P}$
Search vs Verification

- **Search** problems are often **unsolvable**
- But, **verification** of a search result is usually **solvable**

**Examples**

- **Factoring**
  - **Unsolvable:** Find factors of 8633
    - Must “try all” possibilities
  - **Solvable:** Verify 89 and 97 are factors of 8633
    - Just do multiplication

- **Passwords**
  - **Unsolvable:** Find my umb.edu password
  - **Solvable:** Verify whether my umb.edu password is ...
    - “correct horse battery staple”
The **PATH** Problem

\[ \text{PATH} = \{ \langle G, s, t \rangle | G \text{ is a directed graph that has a directed path from } s \text{ to } t \} \]

- It’s a **search** problem:
  - **Exponential time** (brute force) algorithm \((n^n)\):
    - Check all \(n^n\) possible paths and see if any connects \(s\) and \(t\)
  - **Polynomial time** algorithm:
    - Do a breadth-first search (roughly), marking “seen” nodes as we go \((n = \# \text{ nodes})\)

**Proof**

A polynomial time algorithm \(M\) for **PATH** operates as follows.

\(M = \) “On input \(\langle G, s, t \rangle\), where \(G\) is a directed graph with nodes \(s\) and \(t\):

1. Place a mark on node \(s\).
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of \(G\). If an edge \((a, b)\) is found going from a marked node \(a\) to an unmarked node \(b\), mark node \(b\).
4. If \(t\) is marked, accept. Otherwise, reject.”

\(O(n^3)\)
Verifying a $\textbf{PATH}$

$\text{PATH} = \{\langle G, s, t \rangle | G \text{ is a directed graph that has a directed path from } s \text{ to } t \}$

The verification problem:
• Given some path $p$ in $G$, check that it is a path from $s$ to $t$

• Let $m = \text{longest possible path} = \# \text{ edges in } G$

Verifier $V = \text{On input } \langle G, s, t, p \rangle, \text{ where } p \text{ is some set of edges:}$

1. Check some edge in $p$ has “from” node $s$; mark and set it as “current” edge
   • Max steps = $O(m)$

2. Loop: While there remains unmarked edges in $p$:
   1. Find the “next” edge in $p$, whose “from” node is the “to” node of “current” edge
   2. If found, then mark that edge and set it as “current”, else reject
   • Each loop iteration: $O(m)$
   • # loops: $O(m)$
   • Total looping time = $O(m^2)$

3. Check “current” edge has “to” node $t$; if yes accept, else reject

• Total time = $O(m) + O(m^2) = O(m^2)$ = polynomial in $m$

NOTE: extra argument $p$,
“Verifying” an answer requires having a potential answer to check!

$\text{PATH can be verified in polynomial time}$
Verifiers, Formally

\[ \text{PATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t \} \]

A **verifier** for a language \( A \) is an algorithm \( V \), where

\[ A = \{ w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c \} \]

We measure the time of a verifier only in terms of the length of \( w \), so a **polynomial time verifier** runs in polynomial time in the length of \( w \). A language \( A \) is **polynomially verifiable** if it has a polynomial time verifier.

**NOTE**: a cert \( c \) must be at most length \( n^k \), where \( n = \text{length of } w \)

- Why?

So \( \text{PATH} \) is polynomially verifiable
The **HAMPATH** Problem

- A Hamiltonian path goes through **every** node in the graph

- The **Search** problem:
  - **Exponential time** (brute force) algorithm:
    - Check all possible paths and see if any connect $s$ and $t$ using all nodes
  - **Polynomial time** algorithm:
    - We **don’t know** if there is one!!

- The **Verification** problem:
  - Still $O(m^2)$!
  - **HAMPATH** is polynomially verifiable, but **not** polynomially decidable
The class \textbf{NP}

\begin{definition}
\textbf{NP} is the class of languages that have polynomial time \textit{verifiers}.
\end{definition}

\begin{itemize}
\item \textit{PATH} is in \textbf{NP}, and \textbf{P}
\item \textit{HAMPATH} is in \textbf{NP}, but it’s \textit{unknown} whether it’s in \textbf{P}
\end{itemize}
**NP** = **Nondeterministic polynomial time**

**NP** is the class of languages that have polynomial time verifiers.

**Theorem**

A language is in **NP** iff it is decided by some nondeterministic polynomial time Turing machine.

⇒ If a language is in **NP**, then it has a non-deterministic poly time decider
• **We know**: If a lang **L** is in **NP**, then it has a poly time verifier **V**
• **Need to**: create NTM deciding **L**:
  On input **w** =
  • Nondeterministically run **V** with **w** and all possible poly length certificates **c**

⇐ If a language has a non-deterministic poly time decider, then it is in **NP**
• **We know**: **L** has NTM decider **N**,
• **Need to**: show **L** is in **NP**, i.e., create polytime verifier **V**:
  On input <**w**, **c**> =
  • Convert **N** to deterministic TM, and run it on **w**, but take only one computation path
  • Let certificate **c** dictate which computation path to follow

**Note**: cert is usually a potential answer, but does not have to be (like here)

Certificate **c** specifies a path
NP

\[ \text{NP} = \bigcup_k \text{NTIME}(n^k) \]

\[ \text{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine} \}. \]
NP vs P

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$P = \bigcup_k \text{TIME}(n^k).$$

P = Deterministic polynomial time

$$\text{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$  

Also, NP = Deterministic polynomial time verification

NP = Nondeterministic polynomial time
More **NP** Problems

- **CLIQUE** = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}
  - A clique is a subgraph where every two nodes are connected
  - A $k$-clique contains $k$ nodes

- **SUBSET-SUM** = \{\langle S, t \rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \sum y_i = t\}
Theorem: **CLIQUE** is in NP

**Proof Idea**

The clique is the certificate.

**Proof**

The following is a verifier $V$ for CLIQUE.

$V = \text{"On input } \langle \langle G, k \rangle, c \rangle:\$  

1. Test whether $c$ is a subgraph with $k$ nodes in $G$.  
2. Test whether $G$ contains all edges connecting nodes in $c$.  
3. If both pass, accept; otherwise, reject.”

Let $n = \# \text{ nodes in } G$

$c$ is at most $n$

For each: node in $c$, check whether it’s in $G$

$O(n^2)$

For each: pair of nodes in $c$, check whether there’s an edge in $G$

$O(n^2)$

A **verifier** for a language $A$ is an algorithm $V$, where

$A = \{w | V \text{ accepts } \langle w, c \rangle \text{ for some string } c \}$

We measure the time of a verifier only in terms of the length of $w$, so a **polynomial time verifier** runs in polynomial time in the length of $w$. A language $A$ is **polynomially verifiable** if it has a polynomial time verifier.

**NP** is the class of languages that have polynomial time verifiers.
Proof 2: \textbf{CLIQUE} is in NP

\textit{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}

\begin{itemize}
    \item \(N = \text{``On input } \langle G, k \rangle, \text{ where } G \text{ is a graph:}"
    \item 1. Nondeterministically select a subset \(c\) of \(k\) nodes of \(G\).
    \item 2. Test whether \(G\) contains all edges connecting nodes in \(c\).
    \item 3. If yes, accept; otherwise, reject.''
\end{itemize}

To prove a lang \(L\) is in NP, create either a:
\begin{enumerate}
    \item Deterministic poly time verifier
    \item Nondeterministic poly time decider
\end{enumerate}

How to prove a language is in \textbf{NP}:
Proof technique \#2: create an NTM

\begin{itemize}
    \item A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.
\end{itemize}
More **NP** Problems

- **CLIQUE** = \{⟨G, k⟩| G is an undirected graph with a k-clique\}
  - A clique is a subgraph where every two nodes are connected
  - A k-clique contains k nodes

- **SUBSET-SUM** = \{⟨S, t⟩| S = \{x_1, \ldots, x_k\}, and for some \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, we have \(\sum y_i = t\)\}
  - Some subset of a set of numbers S must sum to some total t
  - e.g., \{4, 11, 16, 21, 27\}, 25 \in **SUBSET-SUM**
Theorem: \textit{SUBSET-SUM} is in NP

\[ \text{SUBSET-SUM} = \{ \langle S, t \rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \sum y_i = t \} \]

**Proof Idea** The subset is the certificate.

To prove a lang is in NP, create either:
1. Deterministic poly time verifier
2. Nondeterministic poly time decider

**Proof** The following is a verifier \( V \) for \textit{SUBSET-SUM}.

\[ V = \text{“On input } \langle \langle S, t \rangle, c \rangle \text{:} \]

1. Test whether \( c \) is a collection of numbers that sum to \( t \).
2. Test whether \( S \) contains all the numbers in \( c \).
3. If both pass, accept; otherwise, reject.”
Proof 2: **SUBSET-SUM** is in NP

**SUBSET-SUM** = \{\langle S, t \rangle \mid S = \{x_1, \ldots, x_k\}, and for some \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, we have \(\sum y_i = t\)\}

To prove a lang is in NP, create either:
1. Deterministic poly time verifier
2. Nondeterministic poly time decider

**ALTERNATIVE PROOF** We can also prove this theorem by giving a nondeterministic polynomial time Turing machine for **SUBSET-SUM** as follows.

\(N = \text{“On input } \langle S, t \rangle :\)\)
1. Nondeterministically select a subset \(c\) of the numbers in \(S\).
2. Test whether \(c\) is a collection of numbers that sum to \(t\).
3. If the test passes, accept; otherwise, reject.”

Runtime?
COMPOSITES = \{ x \mid x = pq, \text{ for integers } p, q > 1 \}

• A composite number is not prime

• COMPOSITES is polynomially verifiable
  • i.e., it’s in \text{NP}
  • i.e., factorability is in \text{NP}

• A certificate could be:
  • Some factor that is not 1

• Checking existence of factors (or not, i.e., testing primality) ...
  • ... is also poly time
  • But only discovered \text{recently} (2002)!
HW Question: Does P = NP?

How do you prove an algorithm doesn’t have a poly time algorithm? (in general it’s hard to prove that something doesn’t exist)
Implications if $P = NP$

- Every problem with a “brute force” solution also has an efficient solution
- I.e., “unsolvable” problems are “solvable”
- BAD:
  - Cryptography needs unsolvable problems
  - Near perfect AI learning, recognition
- GOOD: Optimization problems are solved
  - Optimal resource allocation could fix all the world’s (food, energy, space ...) problems?
Progress on whether $P = NP$?

• Some, but still not close

The Status of the P Versus NP Problem

By Lance Fortnow
Communications of the ACM, September 2009, Vol. 52 No. 9, Pages 78-86
10.1145/1562164.1562186

• One important concept discovered:
  • NP-Completeness
**NP-Completeness**

**DEFINITION**
A language $B$ is **NP-complete** if it satisfies two conditions:

1. $B$ is in NP, and
2. every $A$ in NP is polynomial time reducible to $B$.

**THEOREM**
If $B$ is NP-complete and $B \in \text{P}$, then $\text{P} = \text{NP}$.

• How does this help the $\text{P} = \text{NP}$ problem?
Flashback: Mapping Reducibility

Language $A$ is mapping reducible to language $B$, written $\overline{A} \leq_m \overline{B}$, if there is a computable function $f: \Sigma^* \rightarrow \Sigma^*$, where for every $w$,

$$w \in A \iff f(w) \in B.$$ 

The function $f$ is called the reduction from $A$ to $B$.

**IMPORTANT:** “if and only if” ...

To show mapping reducibility:
1. create computable fn
2. and then show forward direction
3. and reverse direction (or contrapositive of forward direction)

$A_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w \}$

$HALT_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \}$

... means $\overline{A} \leq_m \overline{B}$

A function $f: \Sigma^* \rightarrow \Sigma^*$ is a computable function if some Turing machine $M$, on every input $w$, halts with just $f(w)$ on its tape.
Polynomial Time Mapping Reducibility

Language $A$ is **mapping reducible** to language $B$ if there is a computable function $f : \Sigma^* \rightarrow \Sigma^*$,

$$w \in A \iff f(w) \in B.$$  

The function $f$ is called the **reduction** from $A$ to $B$.

Language $A$ is **polynomial time mapping reducible**, or simply **polynomial time reducible**, to language $B$, written $A \leq_P B$, if a polynomial time computable function $f : \Sigma^* \rightarrow \Sigma^*$ exists, where for every $w$,

$$w \in A \iff f(w) \in B.$$  

The function $f$ is called the **polynomial time reduction** of $A$ to $B$.

To show poly time mapping reducibility:
1. create **computable fn**
2. show **forward direction**
3. show **reverse direction** (or contrapositive of forward direction)
4. then show **computable fn runs in poly time**

Don’t forget: “if and only if” ...

poly time  

A function $f : \Sigma^* \rightarrow \Sigma^*$ is a **computable function** if some Turing machine $M$, on every input $w$, halts with just $f(w)$ on its tape.
**Flashback:** If $A \leq_m B$ and $B$ is decidable, then $A$ is decidable.

**Proof:** We let $M$ be the decider for $B$ and $f$ be the reduction from $A$ to $B$. We describe a decider $N$ for $A$ as follows.

$N =$ "On input $w$:
1. Compute $f(w)$.
2. Run $M$ on input $f(w)$ and output whatever $M$ outputs."

Language $A$ is *mapping reducible* to language $B$, written $A \leq_m B$, if there is a computable function $f: \Sigma^* \rightarrow \Sigma^*$, where for every $w$,

$$w \in A \iff f(w) \in B.$$ 

The function $f$ is called the *reduction* from $A$ to $B$. 

This proof only works because of the if-and-only-if requirement.
**Thm:** If \( A \leq_m B \) and \( B \in \mathsf{P} \), then \( A \in \mathsf{P} \). Because if \( B \in \mathsf{P} \), it is decidable, and \( A \leq_m B \) implies that \( A \) is reducible to \( B \). Since \( B \) is decidable, \( A \) is also decidable.

**Proof:** We let \( M \) be the decider for \( B \) and \( f \) be the reduction from \( A \) to \( B \). We describe a decider \( N \) for \( A \) as follows.

\[ N = \text{"On input } w:\text{"} \]

1. Compute \( f(w) \).
2. Run \( M \) on input \( f(w) \) and output whatever \( M \) outputs.

Language \( A \) is mapping reducible to language \( B \), written \( A \leq_m B \), if there is a computable function \( f : \Sigma^* \rightarrow \Sigma^* \), where for every \( w \),

\[ w \in A \iff f(w) \in B. \]

The function \( f \) is called the reduction from \( A \) to \( B \).
Thm: \textbf{If } A \leq_m B \textbf{ and } B \in P \textbf{ is decidable, then } A \in P \textbf{ is decidable.}

\textbf{PROOF} \quad \text{We let } M \text{ be the decider for } B \text{ and } f \text{ be the reduction from } A \text{ to } B. \text{ We describe a decider } N \text{ for } A \text{ as follows.}

\text{\textit{N = "On input } w:\n1. Compute } f(w).\n2. \text{ Run } M \text{ on input } f(w) \text{ and output whatever } M \text{ outputs."}
Next Time: 3SAT is polynomial time reducible to CLIQUE.
Check-in Quiz 12/6

On gradescope