Introduction

- So far, we have focused on problems with "efficient" algorithms
- i.e., problems with algorithms that run in polynomial time: \(O(n^c)\) for some constant \(c \geq 1\)
  - Side note: We call it efficient even if \(c\) is large, since it is likely that another, even more efficient, algorithm exists
  - Side note 2: Need to be careful to speak of polynomial in size of the input, e.g., size of a single integer \(k\) is \(\log k\), so time linear in \(k\) is exponential in size (number of bits) of input
- But, for some problems, the fastest known algorithms require time that is superpolynomial
  - Includes sub-exponential time (e.g., \(2^{n^{1/3}}\)), exponential time (e.g., \(2^n\)), doubly exponential time (e.g., \(2^{2^n}\)), etc.
- There are even problems that cannot be solved in any amount of time (e.g., the "halting problem")

P vs. NP

- Our focus will be on the complexity classes called \(P\) and \(NP\)
- Centers on the notion of a Turing machine (TM), which is a finite state machine with an infinitely long tape for storage
  - Anything a computer can do, a TM can do, and vice-versa
  - More on this in CSCE 428/828 and CSCE 424/824
- \(P = \) "deterministic polynomial time" = the set of problems that can be solved by a deterministic TM (deterministic algorithm) in polynomial time
- \(NP = \) "nondeterministic polynomial time" = the set of problems that can be solved by a nondeterministic TM in polynomial time
  - Can loosely think of a nondeterministic TM as one that can explore many, many possible paths of computation at once
  - Equivalently, \(NP\) is the set of problems whose solutions, if given, can be verified in polynomial time

P vs. NP Example

- Problem HAM-CYCLE: Does a graph \(G = (V,E)\) contain a Hamiltonian cycle, i.e., a cycle that visits every vertex in \(V\) exactly once?
  - This problem is in \(NP\), since if we were given a specific \(G\) plus the answer to the question plus a certificate, we can verify a "yes" answer in polynomial time using the certificate
  - What would be an appropriate certificate?
  - Not known if HAM-CYCLE \(\in P\)

P vs. NP Example (2)

- Problem EULER: Does a directed graph \(G = (V,E)\) contain an Euler tour, i.e., a cycle that visits every edge in \(E\) exactly once and can visit vertices multiple times?
  - This problem is in \(P\), since we can answer the question in polynomial time by checking if each vertex’s in-degree equals its out-degree
  - Does that mean that the problem is also in \(NP\)? If so, what is the certificate?

NP-Completeness

- Any problem in \(P\) is also in \(NP\), since if we can efficiently solve the problem, we get the poly-time verification for free
  - \(P \subseteq NP\)
- Not known if \(P \subseteq NP\), i.e., unknown if there a problem in \(NP\) that’s not in \(P\)
- A subset of the problems in \(NP\) is the set of NP-complete (NPC) problems
  - Every problem in NPC is at least as hard as all others in NPC
  - These problems are believed to be intractable (no efficient algorithm), but not yet proven to be so
  - If any NPC problem is in \(P\), then \(P = NP\) and life is glorious
Proving NP-Completeness

- Thus, if we prove that a problem is NPC, we can tell our boss that we cannot find an efficient algorithm and should take a different approach
  - E.g., Approximation algorithm, heuristic approach

- How do we prove that a problem \( A \) is NPC?
  1. Prove that \( A \in \text{NP} \) by finding certificate
  2. Show that \( A \) is as hard as any other NP problem by showing that if we can efficiently solve \( A \) then we can efficiently solve all problems in NP

- First step is usually easy, but second looks difficult
- Fortunately, part of the work has been done for us...

Reductions

- We will use the idea of a reduction of one problem to another to prove how hard it is
  - A reduction takes an instance of one problem \( A \) and transforms it to an instance of another problem \( B \) in such a way that a solution to the instance of \( B \) yields a solution to the instance of \( A \)
  - Example: How did we prove lower bounds on convex hull and BST problems?
  - Time complexity of reduction-based algorithm for \( A \) is the time for the reduction to \( B \) plus the time to solve the instance of \( B \)

Decision Problems

- Before we go further into reductions, we simplify our lives by focusing on decision problems
  - In a decision problem, the only output of an algorithm is an answer "yes" or "no"
  - I.e., we're not asked for a shortest path or a hamiltonian cycle, etc.

- Not as restrictive as it may seem: Rather than asking for the weight of a shortest path from \( i \) to \( j \), just ask if there exists a path from \( i \) to \( j \) with weight at most \( k \)

- Such decision versions of optimization problems are no harder than the original optimization problem, so if we show the decision version is hard, then so is the optimization version

- Decision versions are especially convenient when thinking in terms of languages and the Turing machines that accept/reject them

Reductions (2)

- What is a reduction in the NPC sense?
  - Start with two problems \( A \) and \( B \), and we want to show that problem \( B \) is at least as hard as \( A \)
  - Will reduce \( A \) to \( B \) via a polynomial-time reduction by transforming any instance \( \alpha \) of \( A \) to some instance \( \beta \) of \( B \) such that
    1. The transformation must take polynomial time (since we're talking about hardness in the sense of efficient vs. inefficient algorithms)
    2. The answer for \( \alpha \) is "yes" if and only if the answer for \( \beta \) is "yes"
  - If such a reduction exists, then \( B \) is at least as hard as \( A \) since if an efficient algorithm exists for \( B \), we can solve any instance of \( A \) in polynomial time
  - Notation: \( A \leq_p B \), which reads as "\( A \) is no harder to solve than \( B \), modulo polynomial time reductions"

Reductions (3)

<table>
<thead>
<tr>
<th>instance ( \alpha ) of ( A )</th>
<th>polynomial-time reduction algorithm</th>
<th>instance ( \beta ) of ( B )</th>
<th>polynomial-time algorithm to decide ( B )</th>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reductions (4)

- But if we want to prove that a problem \( B \) is NPC, do we have to reduce to it every problem in NP?
  - No we don’t:
    - If another problem \( A \) is known to be NPC, then we know that any problem in NP reduces to it
    - If we reduce \( A \) to \( B \), then any problem in NP can reduce to \( B \) via its reduction to \( A \) followed by \( A \)'s reduction to \( B \)
  - We then can call \( B \) an \textit{NP-hard} problem, which is NPC if it is also in \textit{NP}
  - Still need our first NPC problem to use as a basis for our reductions
Our first NPC problem: **CIRCUIT-SAT**

- An instance is a boolean combinational circuit (no feedback, no memory)
- Question: Is there a satisfying assignment, i.e., an assignment of inputs to the circuit that satisfies it (makes its output 1)?

To prove **CIRCUIT-SAT** to be NPC, need to show:
1. **CIRCUIT-SAT** ∈ **NP**; what is its certificate that we can confirm in polynomial time?
2. That any problem in NP reduces to **CIRCUIT-SAT**

We’ll skip the NP-hardness proof, save to say that it leverages the existence of an algorithm that verifies certificates for some NP problem.

**CIRCUIT-SAT (2)**

- Satisfiable
- Unsatisfiable

**Other NPC Problems**

- We’ll use the fact that **CIRCUIT-SAT** is NPC to prove that these other problems are as well:
  - SAT: Does boolean formula \( \phi \) have a satisfying assignment?
  - 3-CNF-SAT: Does 3-CNF formula \( \phi \) have a satisfying assignment?
  - CLIQUE: Does graph \( G \) have a clique (complete subgraph) of \( k \) vertices?
  - VERTEX-COVER: Does graph \( G \) have a vertex cover (set of vertices that touch all edges) of \( k \) vertices?
  - HAM-CYCLE: Does graph \( G \) have a hamiltonian cycle?
  - TSP: Does complete, weighted graph \( G \) have a hamiltonian cycle of total weight \( \leq k \)?
  - SUBSET-SUM: Is there a subset \( S \) of finite set \( S \) of integers that sum to exactly a specific target value \( t \)?
- Many more in Garey & Johnson’s book, with proofs
The Satisfiability problem (SAT) is in NPC:

1. **SAT is in NP:** The assigning assignment certifies that the answer is "yes" and this can be easily checked in poly time.
2. **SAT is NP-hard:** Will show CIRCUIT-SAT ≤_P SAT by reducing from CIRCUIT-SAT to SAT.
3. In reduction, need to map any instance (circuit) \( C \) of CIRCUIT-SAT to some instance (formula) \( \phi \) of SAT such that \( C \) has a satisfying assignment if and only if \( \phi \) does.
4. Further, the time to do the mapping must be polynomial in the size of the circuit (number of gates and wires), implying that \( \phi \)’s representation must be polynomially sized.

### NPC Problem: 3-CNF Satisfiability (3-CNF-SAT)

1. **Given:** A boolean formula that is in 3-conjunctive normal form (3-CNF), which is a conjunction of clauses, each a disjunction of 3 literals, e.g.
   \[
   (x_1 \lor \neg x_3 \lor \neg x_4) \land (x_3 \lor x_2 \lor x_5) \land (x_7 \lor x_1 \lor x_3) \land (x_9 \lor x_6 \lor x_2) \land (x_10 \lor \neg x_7 \lor \neg x_8) \land (\neg x_5 \lor x_2 \lor x_4) \land \neg x_2 \land \neg x_2 \land \neg x_4 \land x_4 \lor x_5 \lor x_1.
   \]
2. **Question:** Is there an assignment of boolean values to \( x_1, \ldots, x_n \) to make the formula evaluate to \( 1 \)?

### 3-CNF-SAT is NPC

1. **3-CNF-SAT is in NP:** The satisfying assignment certifies that the answer is "yes" and this can be easily checked in poly time.
2. **3-CNF-SAT is NP-hard:** Will show SAT ≤_P 3-CNF-SAT.
3. **Again, need to map any instance \( \phi \) of SAT to some instance \( \phi'' \) of 3-CNF-SAT.**
   1. Parenthesize \( \phi \) and build its parse tree, which can be viewed as a circuit.
   2. Assign variables to wires in this circuit, as with previous reduction, yielding \( \phi' \), a conjunction of clauses.
   3. Use the truth table of each clause \( \phi'_i \) to get its DNF, then convert it to CNF \( \phi''_i \).
   4. Add auxiliary variables to each \( \phi''_i \) to get three literals in it, yielding \( \phi'''_i \).
   5. Final CNF formula is \( \phi''' = \bigwedge_i \phi'''_i \).
Building the Parse Tree

\( \phi = \left( (x_1 \rightarrow y_2) \vee \neg ((\neg x_1 \leftrightarrow x_2) \vee x_4) \right) \wedge \neg x_2 \)

Might need to parenthesis \( \phi \) to put at most two children per node

Convert Each Clause to CNF

- Consider first clause \( \phi_1' = (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \)
- Truth table:

<table>
<thead>
<tr>
<th>( y_1 )</th>
<th>( y_2 )</th>
<th>( x_2 )</th>
<th>( (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Can now directly read off DNF of negation:

\( \neg \phi_1' = (y_1 \wedge \neg y_2 \wedge y_2) \vee (y_1 \wedge \neg y_2 \wedge \neg y_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2) \)

- And use DeMorgan’s Law to convert it to CNF:

\( \phi_1'' = (\neg y_1 \vee \neg y_2 \vee \neg y_2) \wedge (\neg y_1 \vee y_2 \wedge \neg y_2) \wedge (\neg y_1 \vee \neg y_2 \wedge \neg y_2) \wedge (y_1 \vee \neg y_2 \wedge y_2) \)

Assign Variables to wires

\( \phi' = y_1 \wedge (y_2 \leftrightarrow (y_2 \wedge \neg x_2)) \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \wedge (y_4 \leftrightarrow (y_5 \vee x_4)) \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) \)

Add Auxiliary Variables

- Based on our construction, \( \phi = \phi'' \wedge \phi''', \) where each \( \phi'' \) is a CNF formula each with at most three literals per clause.
- But we need to have exactly three per clause!
- Simple fix: For each clause \( C_j \) of \( \phi'' \):
  1. If \( C_j \) has three distinct literals, add it as a clause in \( \phi''' \)
  2. If \( C_j = (t_1 \lor t_2 \lor t_3) \) for distinct literals \( t_1 \) and \( t_2 \), then add to \( \phi''' \)
    \( (t_1 \lor t_2 \lor t_3) \wedge (t_1 \lor t_2 \lor \neg p) \)
  3. If \( C_j = (t_1) \), then add to \( \phi''' \)
    \( (t_1 \lor p \lor q) \lor (t_2 \lor \neg p \lor \neg q) \lor (t_3 \lor \neg p \lor \neg q) \)
- \( p \) and \( q \) are auxiliary variables, and the combinations in which they’re added result in a logically equivalent expression to that of the original clause, regardless of the values of \( p \) and \( q \)

Proof of Correctness of Reduction

- \( \phi \) has a satisfying assignment iff \( \phi''' \) does
  1. CIRCUIT-SAT reduction to SAT implies satisfiability preserved from \( \phi \) to \( \phi' \)
  2. Use of truth tables and DeMorgan’s Law ensures \( \phi'' \) equivalent to \( \phi' \)
  3. Addition of auxiliary variables ensures \( \phi''' \) equivalent to \( \phi'' \)
- Constructing \( \phi''' \) from \( \phi \) takes polynomial time
  1. \( \phi' \) gets variables from \( \phi \), plus at most one variable and one clause per operator in \( \phi \)
  2. Each clause in \( \phi' \) has at most 3 variables, so each truth table has at most 8 rows, so each clause in \( \phi'' \) yields at most 8 clauses in \( \phi''' \)
  3. Since there are only two auxiliary variables, each clause in \( \phi''' \) yields at most 4 in \( \phi''' \)
  4. Thus size of \( \phi''' \) is polynomial in size of \( \phi \), and each step easily done in polynomial time

NPC Problem: Clique Finding (CLIQUE)

- Given: An undirected graph \( G = (V, E) \) and value \( k \)
- Question: Does \( G \) contain a clique (complete subgraph) of size \( k \)?

Has a clique of size \( k = 6 \), but not of size 7
CLIQUE is NPC

- CLIQUE is in NP: A list of vertices in the clique certifies that the answer is "yes" and this can be easily checked in poly time
- CLIQUE is NP-hard: Will show 3-CNF-SAT ≤P CLIQUE by mapping any instance \( \phi \) of 3-CNF-SAT to some instance \( (G, k) \) of CLIQUE
  - Seems strange to reduce a boolean formula to a graph, but we will show that \( \phi \) has a satisfying assignment iff \( G \) has a clique of size \( k \)
  - Caveat: the reduction merely preserves the iff relationship; it does not try to directly solve either problem, nor does it assume it knows what the answer is

CLIQUE by mapping any instance \( \phi \) of 3-CNF-SAT to some instance \( (G, k) \) of CLIQUE

The Reduction

- Let \( \phi = C_1 \land \cdots \land C_s \) be a 3-CNF formula with \( s \) clauses
- For each clause \( C_r = (x'_r \lor x'_s \lor x'_t) \) put vertices \( v'_r, v'_s, \) and \( v'_t \) into \( V \)
- Add edge \((v'_r, v'_t)\) to \( E \) if:
  1. \( r \neq s \), i.e., \( v'_r \) and \( v'_s \) are in separate triples
  2. \( x'_r \) is not the negation of \( x'_s \)
- Obviously can be done in polynomial time

The Reduction (2)

\[ \phi = (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_2 \lor x_3) \land (x_2 \lor x_3) \]

Satisfied by \( x_2 = 0, x_3 = 1 \)

The Reduction (3)

- If \( \phi \) has a satisfying assignment, then at least one literal in each clause is true
- Picking corresponding vertex from a true literal from each clause yields a set \( V' \) of \( k \) vertices, each in a distinct triple
- Since each vertex in \( V' \) is in a distinct triple and literals that are negations of each other cannot both be true in a satisfying assignment, there is an edge between each pair of vertices in \( V' \)
- \( V' \) is a clique of size \( k \)
- If \( G \) has a size-\( k \) clique \( V' \), can assign 1 to corresponding literal of each vertex in \( V' \)
- Each vertex in its own triple, so each clause has a literal set to 1
- Will not try to set both a literal and its negation to 1
- Get a satisfying assignment

NPC Problem: Vertex Cover Finding (VERTEX-COVER)

- A vertex in a graph is said to cover all edges incident to it
- A vertex cover of a graph is a set of vertices that covers all edges in the graph
- Given: An undirected graph \( G = (V, E) \) and value \( k \)
- Question: Does \( G \) contain a vertex cover of size \( k \)?

Has a vertex cover of size \( k = 2 \), but not of size 1

VERTEX-COVER is NPC

- VERTEX-COVER is in NP: A list of vertices in the vertex cover certifies that the answer is "yes" and this can be easily checked in poly time
- VERTEX-COVER is NP-hard: Will show CLIQUE ≤P VERTEX-COVER by mapping any instance \( (G, k) \) of CLIQUE to some instance \( (G', k') \) of VERTEX-COVER
- Reduction is simple: Given instance \( (G = (V, E), k) \) of CLIQUE, instance of VERTEX-COVER is \( (\overline{G}, |V| - k) \), where \( \overline{G} = (V, \overline{E}) \) is \( G \)'s complement:
  \[ \overline{E} = \{(u, v) : u, v \in V, u \neq v, (u, v) \notin E\} \]
- Easily done in polynomial time
VERTEX-COVER is NPC (2)

![Graph](image)

**Proof of Correctness**

- Assume $G$ has a size-$k$ clique $V' \subseteq V$
- Consider edge $(x, v) \in E$
- If it’s in $\overline{G}$, then $(x, v) \notin E$, so at least one of $x$ and $v$ (which cover $(z, v)$) is not in $V'$, so at least one of them is in $V \setminus V'$
- This holds for each edge in $\overline{G}$, so $V \setminus V'$ is a vertex cover of $\overline{G}$ of size $|V| - k$
- Assume $\overline{G}$ has a size-$(|V| - k)$ vertex cover $V'$
- For each $(z, v) \in E$, at least one of $x$ and $v$ is in $V'$
- By contrapositive, if $u, v \notin V'$, then $(u, v) \in E$
- Since every pair of nodes in $V \setminus V'$ has an edge between them, $V \setminus V'$ is a clique of size $|V| - |V'| = k$

NPC Problem: Subset Sum (SUBSET-SUM)

- Given: A finite set $S$ of positive integers and a positive integer target $t$
- Question: Is there a subset $S' \subseteq S$ whose elements sum to $t$?
- E.g. $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$ and $t = 138457$ has a solution $S' = \{1, 2, 7, 98, 2409, 17206, 117705\}$

SUBSET-SUM is NPC

- SUBSET-SUM is in NP: The subset $S'$ certifies that the answer is "yes" and this can be easily checked in poly time
- SUBSET-SUM is NP-hard: Will show 3-CNF-SAT $\leq_p$ SUBSET-SUM by mapping any instance $\phi$ of 3-CNF-SAT to some instance $(S, t)$ of SUBSET-SUM
- Make two reasonable assumptions about $\phi$:
  1. No clause contains both a variable and its negation
  2. Each variable appears in at least one clause

The Reduction

- Let $\phi$ have $k$ clauses $C_1, \ldots, C_k$ over $n$ variables $x_1, \ldots, x_n$
- Reduction creates two numbers in $S$ for each variable $x_i$ and two numbers for each clause $C_j$
- Each number has $n + k$ digits, the most significant $n$ tied to variables and least significant $k$ tied to clauses
  1. Target $t$ has a 1 in each digit tied to a variable and a 4 in each digit tied to a clause
  2. For each $x_i$, $S$ contains integers $v_i$ and $v'_i$, each with a 1 in $x_i$’s digit and 0 for other variables. Put a 1 in $C_j$’s digit for $v_i$ if $x_i \in C_j$, and a 1 in $C_j$’s digit for $v'_i$ if $\neg x_i \in C_j$
  3. For each $C_j$, $S$ contains integers $s_j$ and $s'_j$, where $s_j$ has a 1 in $C_j$’s digit and 0 elsewhere, and $s'_j$ has a 2 in $C_j$’s digit and 0 elsewhere
- Greatest sum of any digit is 6, so no carries when summing integers
- Can be done in polynomial time

The Reduction (2)

$C_1 = \{x_1 \lor \neg x_2 \lor \neg x_3\}$, $C_2 = \{\neg x_1 \lor \neg x_2 \lor \neg x_3\}$, $C_3 = \{\neg x_1 \lor x_2 \lor x_3\}$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$x_1 = 0, x_2 = 0, x_3 = 1$
Proof of Correctness

⇒ If \( x_i = 1 \) in \( \phi \)'s satisfying assignment, SUBSET-SUM solution \( S' \) will have \( v_i \); otherwise \( v'_i \)
  ▷ For each variable-based digit, the sum of the elements of \( S' \) is 1
  ▷ Since each clause is satisfied, each clause contains at least one literal with the value 1, so each clause-based digit sums to 1, 2, or 3
  ▷ To match each clause-based digit in \( t \), add in the appropriate subset of slack variables \( s_i \) and \( s'_i \)

Proof of Correctness (2)

⇐ In SUBSET-SUM solution \( S' \), for each \( i = 1, \ldots, n \), exactly one of \( v_i \) and \( v'_i \) must be in \( S' \), or sum won’t match \( t \)
  ▷ If \( v_i \in S' \), set \( x_i = 1 \) in satisfying assignment, otherwise we have \( v'_i \in S' \) and set \( x_i = 0 \)
  ▷ To get a sum of 4 in clause-based digit \( C_j \), \( S' \) must include a \( v_i \) or \( v'_i \) value that is 1 in that digit (since slack variables sum to at most 3)
  ▷ Thus, if \( v_i \in S' \) has a 1 in \( C_j \)'s position, then \( x_i \) is in \( C_j \) and we set \( x_i = 1 \), so \( C_j \) is satisfied (similar argument for \( v'_i \in S' \) and setting \( x_i = 0 \))
  ▷ This holds for all clauses, so \( \phi \) is satisfied

In-Class Exercise

▷ OK, everything perfectly clear?
▷ Want a shot at extra credit?
▷ Put away your books (keep your notes), split into groups, and get ready for an in-class exercise!