Announcements
1. Homework X is available on the web page (has been since Friday).
2. Homework X is due Wednesday.
3. Revised office hours for the last two weeks:
   - TA hours 7-10 Monday and 8-10 Tuesday.
   - My hours 2-4 Monday and Tuesday.

Quick Review
1. Last time, we encountered a number of key definitions including:
   - Definition: Let $f$ and $g$ be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = O(g(n))$ if for some positive integers $c$ and $n_0$
     $$f(n) < cg(n)$$
     for all $n \geq n_0$. We say that $g(n)$ is an asymptotic upper bound for $f(n)$.
   - Definition: Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the time complexity class, $TIME(t(n))$ to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.
   - Definition: $P$ is the class of language that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words
     $$P = \bigcup_k TIME(n^k)$$
   - Definition: $NP$ is the class of language that are decidable in polynomial time on a nondeterministic single-tape Turing machine. In other words
     $$NP = \bigcup_k NTIME(n^k)$$
     or equivalently
     - Definition: $NP$ is the class of language that are polynomial verifiable.

2. Many common algorithms correspond to languages in $P$:
   - Parsing relative to a CFG.
   - Searching and sorting

3. We also began talking about the class of problems that could be solved using nondeterministic Turing machines whose computation branches were all limited to be of polynomial length as a function of the input size.
   - Definition: $NTIME(t(n)) = \{ L \mid L$ is a language decided by an $O(t(n))$ time nondeterministic Turing machine $\}$.
   - Definition: A verifier for a language $A$ is a deterministic Turing machine $V$ where
     $$A = \{ w \mid V$ accepts $\langle w, c \rangle$ for some string $c \}$$
     We say that $V$ is a polynomial time verifier if it runs in polynomial time in the length of $w$. In this case, we say that $A$ is polynomial verifiable.
   - Definition: $NP$ is the class of language that are polynomial verifiable.

4. We also discussed two problems that are clearly in $NP$ but probably not in $P$:
   - Subset Sum Given a list of numbers like:
     $$17, 24, 5, 9, 11, 24, 57, 4, 39, 40, 84, 11, 19$$
     Is there some sublist of these numbers that adds up to exactly $N$? (for some given $N$)
Maximal 3-dimensional Matching  Given three sets of items, $X$, $Y$, and $Z$, and a set of allowable triples, $A \subseteq X \times Y \times Z$, we say that $M \subseteq A$ is a 3-dimensional matching if for any pair of distinct triples $(x_1, y_1, z_1), (x_2, y_2, z_2) \in M$, $x_1 \neq x_2$, $y_1 \neq y_2$, and $z_1 \neq z_2$. The problem is to determine whether we can find an $M \subseteq A$ that includes all elements of $X, Y,$ and $Z$ (this only makes sense when $|X| = |Y| = |Z|$).

5. Finally, we saw that we could reduce the Maximal 3-dimensional Matching problem to the Subset Sum problem in polynomial time by carefully encoding the elements of the set of allowable matchings as integers.

6. We refer to such a transformation as a polynomial-time reduction:

Definition: We say that $A$ is polynomial-time reducible to $B$ (written $A \leq_p B$) if and only if there exists a polynomial time function $f : A \rightarrow B$ such that $w \in A$ if and only if $f(w) \in B$.

7. In particular, we now can say:

3-dim matching $\leq_p$ subset sum

8. Just as many-to-one reducibility allowed us to draw conclusions about whether a language was decidable or recognizable, polynomial reducibility allows us to make statements about membership in P (and eventually NP).

Theorem: If $A \leq_p B$ and $B \in P$ then $A \in P$.

and conversely

Theorem: If $A \leq_p B$ and $A \notin P$ then $B \notin P$.

9. Thus, we now know that if subset sum is in P then 3-dimensional matching (at least looking for complete matchings) is in P. Also, if 3-dimensional matching is not in P then subset sum must not be in P either.

Satisfiability

1. Another interestingly difficult problem is satisfiability for boolean expressions.

- We restrict our attention to expressions involving boolean variables, ands ($\land$), ors ($\lor$), and negation ($\overline{}$) like:

$$\overline{x_1} \land ((x_2 \land \overline{x_3}) \lor (x_3 \land \overline{x_2})) \lor \overline{x_1} \land ((x_2 \land x_3) \lor (x_2 \lor x_3))$$

- We say such a formula is satisfiable if there is an assignment of the values true and false (equivalently 0 and 1) to the variables used in the formula that causes the formula to evaluate to true (1).

- For example,

$$\overline{x_1} \land ((x_2 \land \overline{x_3}) \lor (x_3 \land \overline{x_2})) \lor \overline{x_1} \land ((x_2 \land x_3) \lor (x_2 \lor x_3))$$

is an example of a boolean formula over three variables. If I got it right, its value is true exactly when the number of true variables is even (i.e., it is based on even parity). As a result this is an example of an easily satisfiable formula. (The truth table is shown below.)

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>ans</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

- On the other hand,

$$(x_1 \lor x_2) \land (x_1 \lor \overline{x_2}) \land (x_2 \lor \overline{x_1}) \land (\overline{x_1} \lor \overline{x_2})$$

is an example of a formula with no satisfying assignment.
2. The satisfiability problem is the problem of deciding membership in the language
   \[
   \text{SAT} = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable boolean formula} \}
   \]

3. We will also be interested in a version of the problem restricted to formulas written in a canonical form called 3-conjunctive normal form or 3-CNF.

4. 3-CNF is a special form of a more general special form called conjunctive normal form.
   - A formula is in conjunctive normal form (CNF) if it is a sequence of clauses all connected by conjunctions (ands) where each clause is a series of disjunctions (ors) or variables or their negations.
   - Given any boolean formula, we can rewrite it in conjunctive normal form.
   - A simple way to see that this is true is to visualize the truth table for the formula you want to express in CNF.
     - Your CNF formula will have one clause for each row of the truth table where the value of the function is false. Each clause should be true only if the variables do not have the values corresponding to that row.
     - Your formula collectively says the result should be true if the variable values do not match any row where the value would be false.

5. In 3-CNF, all formulas are conjunctions of clauses that are disjunctions of exactly 3 variables or negations of variables.
   - Given a formula in CNF, two tricks let us convert it into 3-CNF.
     - If the formula has less than three terms, duplicate one of its terms.
     - If the formula has more than three terms, add a new variable and break the formula up into two subparts such as:
       \[
       x_1 \lor x_2 \lor x_3 \lor x_4 = (x_1 \lor x_2 \lor x_{\text{new}}) \land (\overline{x_{\text{new}}} \lor x_3 \lor x_4)
       \]
   - Any satisfying assignment of the original problem will be a satisfying assignment of the 3-CNF form with added variables, and any assignment to the formula with added variables will reduce to a satisfying assignment of the original problem when the added variables are ignored.

6. The problem of determining if a boolean formula in 3-CNF is satisfiable is known as 3-SAT.
   \[
   \text{3-SAT} = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable boolean formula in 3-CNF} \}
   \]

7. Our ultimate goal is to show that any problem in NP can be polynomial reduced to satisfaction of a boolean formula. Before we do that, let’s see that satisfaction for 3-CNF terms can be reduced to subset sum.
   - As we did for the 3-dimensional matching problem, we will generate a peculiar set of number to use as an instance of the subset sum problem based on the 3-CNF formula we are given.
   - Let’s assume the formula has \( t \) clauses over \( v \) variables. We will assume the variables are numbered \( x_1, x_2, ..., x_v \) and we will number the clauses from 1 to \( t \). Each number in our collection will be \( 2k + v \) digits long (possibly with leading 0s). The digits of each number will be divided into three groups as shown below:

     | \( x_1 \) | \( x_2 \) | \( \ldots \) | \( x_v \) | \( c_1 \) | \( c_2 \) | \( \ldots \) | \( c_t \) | \( s_1 \) | \( s_2 \) | \( \ldots \) | \( s_t \) |
     |

   - There will be two numbers in the set for each variable \( x_i \). One of these numbers will correspond to setting \( x_i \) to true, the other to setting \( x_i \) to false.
     - In both numbers the \( i \)th digit will be 1.
     - All of the remaining \( v + t \) leading digits will be 0.
     - The remaining digits, labeled \( s_j \) in the diagram above, will differ in the two numbers for a given variable. In one number \( s_j \) will be one if \( x_i \) appears in clause \( j \). In the other number, \( s_j \) will be one if \( \overline{x_i} \) appears in clause \( j \). Otherwise, \( s_j \) will be 0.
For example, given the formula:

\[(x_1 \lor x_2 \lor \overline{x_3}) \land (x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_3) \land (\overline{x_1} \lor \overline{x_2} \lor \overline{x_3})\]

We would generate the numbers:

<table>
<thead>
<tr>
<th></th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_3)</th>
<th>(c_1)</th>
<th>(c_2)</th>
<th>(c_3)</th>
<th>(s_1)</th>
<th>(s_2)</th>
<th>(s_3)</th>
<th>(s_4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(x_1)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(x_2)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(\overline{x_2})</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(x_3)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(x_3)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

• The goal of selecting these numbers in this way is to connect the set of numbers chosen to a possible truth assignment for the variables in such a way that the sum of the numbers chosen will indicate which clauses are satisfied by the corresponding truth assignment.

  - In the final subset sum problem, we will require that the first \(v\) digits of the sum be all ones. This will imply that in any solution to the subset sum problem we have chosen one of the numbers associated with each variable \(x_i\). The number chosen reflect whether that variable is set to true or false in our assignment.
  
  - If this is done, the each of the low order \(t\) digits of the sum will equal the number of literals in each clause that are true. If all of these digits are greater than 0, we know we have a satisfying assignment.

• To complete the reduction to the subset sum problem, we need more than knowing that the lower order digits must all be greater than 0. We need to be able to predict their exact values. To accomplish this we add 3 numbers to our subset sum problem for each of the clauses in the original formula.

  - In each of the numbers for clause \(j\), the \(j + v\)th digit of the number (i.e., \(c_j\))will be 1.
  - All of the remaining \(c_k\) digits will be 0.

  The value of \(s_j\) (i.e., the \(v + t + j\)th digit) will be 0 in one number, 1 in the second, and 2 in the third. All of the other \(s_k\) digits will be 0.

Given that our sample formula has 4 clauses we would generate the numbers:

<table>
<thead>
<tr>
<th></th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_3)</th>
<th>(c_1)</th>
<th>(c_2)</th>
<th>(c_3)</th>
<th>(s_1)</th>
<th>(s_2)</th>
<th>(s_3)</th>
<th>(s_4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clause 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clause 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clause 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clause 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clause 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clause 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clause 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clause 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>clause 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>clause 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clause 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clause 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

• If we have a satisfying assignment of the formula corresponding to the subset sum problem we have generated, then the number of literals that are not true in any clause of the formula must be 0, 1, or 2. By selecting the corresponding number for each clause, we can therefore ensure that the sum we obtain will be a number consisting of \(t + v\) 1s followed by \(t\) 3s.

• Thus, it should be clear that if there is a satisfying assignment, then the subset sum problem we have constructed has a solution.

• Conversely, if there is a solution to the subset sum problem, it must involve choosing either the number associated with \(x_i\) or \(\overline{x_i}\) for each \(i\) and the numbers chosen in this way must correspond to an assignment that satisfies the original boolean formula.

The Cook-Levin Theorem

1. In the early 70’s Stephen Cook and Leonid Levin independently showed that there are examples of problems in NP that are universal in the
sense that all problems in NP are polynomial-reducible to these problems. Such problems are said to be NP-complete. That is:

**Definition:** We say that a language $B$ is *NP-Complete* if $B \in \text{NP}$ and for all $A \in \text{NP}$, $A \leq_p B$.

2. Surprisingly (given the reductions we have looked at), Subset-sum was not the problem they showed to be NP-complete (although it is). Instead, they showed that satisfiability is NP-complete.

3. Now we are ready to consider how to prove the Cook-Levin Theorem:

**Theorem:** SAT is NP-Complete. That is, SAT $\in$ NP and for any $A \in \text{NP}$, $A \leq_p \text{SAT}$.