NP-Completeness:
Transforming 3-SAT to SUM-OF-SUBSETS in polynomial time

Chapter 9.4.2: Consider \( n \) boolean variables, \( x_1, \ldots, x_n \). A literal is any \( x_i \) or \( \sim x_i \).

A boolean formula in 3-CNF form (conjunctive normal form) is the AND of clauses, where each clause is the OR of 3 literals. (Conjunction means AND, and disjunction means OR.)

The 3-SAT problem is to determine if a formula in 3-CNF form be satisfied; that is, can the boolean variables be set so that the formula is true? For example,

\[
(x_1 \lor x_2 \lor \sim x_4) \land (\sim x_1 \lor x_3 \lor x_4)
\]

can be satisfied by setting \( x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 1 \) (among many other ways), so the answer is “yes.” However,

\[
(x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_2 \lor \sim x_3) \land (x_1 \lor \sim x_2 \lor x_3) \land (x_1 \lor \sim x_2 \lor \sim x_3) \\
\land (\sim x_1 \lor x_2 \lor x_3) \land (\sim x_1 \lor x_2 \lor \sim x_3) \land (\sim x_1 \lor \sim x_2 \lor x_3) \land (\sim x_1 \lor \sim x_2 \lor \sim x_3)
\]

cannot be satisfied, because no matter how the variables are set, one of the clauses will be false and all the others will be true, so the AND will be false.

3-SAT is in NP: if you are told “set the \( x \)’s to these values to make the formula true,” you could plug them into your 3-CNF formula, and if it’s true, you will confirm it in polynomial time. It’s also in co-NP: if plugging in those values makes it false, you will know in polynomial time. Currently, there are no examples of decision problems that have been proven to be in NP but not co-NP, or vice-versa.

Chapter 5.4: The SUM-OF-SUBSETS problem takes \( n \) numbers \( w_1, \ldots, w_n \) and a goal \( W \), and asks whether there is a subset of the numbers that adds up to \( W \) exactly. It is a variation of the Knapsack problem, and can be solved by similar tree-search algorithms.

This is in NP and co-NP: if you are told “this specific subset adds up to \( W \),” you can add together those numbers in linear time and check whether they do or don’t sum to \( W \).

An input to either problem can be converted to an input giving the same yes/no answer to the other problem in polynomial time. (Beyond the yes/no answer, knowing the settings of the \( x \)’s in the 3-SAT problem will tell you which \( w \)’s to take in the version of the problem converted to SUM-OF-SUBSETS, and vice-versa.) This implies that if either problem can be solved in polynomial time, then both can. In fact, both problems are NP-complete: inputs to all problems in NP can be transformed to inputs to these problems in polynomial time, so solving either one of these problems in polynomial time would solve all NP problems in polynomial time. There are over 10000 known NP-complete problems to date.

On the next page there is a demonstration of a conversion of a 3-SAT instance to a SUM-OF-SUBSETS instance. A conversion in the other direction is also possible.

There are many details in proving this conversion works. Starting with a 3-CNF formula \( f \), you must show that the inputs to the SUM-OF-SUBSETS problem can be produced in polynomial time, and that these inputs give the same yes or no answer that 3-SAT returns for \( f \). Full details are in

Lagoudakis, Michail G., “The 0-1 Knapsack Problem: An Introductory Survey,”
http://citeseer.nj.nec.com/151553.html
Here is a conversion of a 3-SAT problem instance to a SUM-OF-SUBSETS problem instance. Consider the following input to 3-SAT:

\[
(x_1 \lor x_2 \lor \sim x_3) \land (\sim x_1 \lor x_3 \lor x_4)
\]

\[(n = 4 \text{ variables, } m = 2 \text{ clauses)}
\]

This table can be produced in polynomial time:

<table>
<thead>
<tr>
<th>clause 2</th>
<th>clause 1</th>
<th>\sim x_1 \lor x_3 \lor x_4</th>
<th>\sim x_1 \lor x_3 \lor x_4</th>
<th>x_3</th>
<th>x_2</th>
<th>x_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The format of the table is

<table>
<thead>
<tr>
<th>clauses</th>
<th>\sim x_1 \lor x_3 \lor x_4</th>
<th>\sim x_1 \lor x_3 \lor x_4</th>
<th>x_3</th>
<th>x_2</th>
<th>x_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Region I.}</td>
<td>\text{Region II.}</td>
<td>\text{Region III.}</td>
<td>\text{Region IV.}</td>
<td>\text{Region V. all 3's}</td>
<td>\text{Region VI. all 1's}</td>
</tr>
</tbody>
</table>
| \text{selected, set} \text{w's} | \text{all 3's} | \text{e} \text{m} \text{1} \text{blank} | \text{d} \text{m} | \text{W} | | | 1 \text{blank} | \text{d} \text{1} | \text{e} \text{1} | \text{u} \text{1} | \text{u} \text{2} | \text{u} \text{3} | \text{u} \text{4}

| \text{u}_i = 010001 | \text{u}'_i = 100001 | \text{u}_i = 010010 | \text{u}'_i = 101000 | \text{u}_i = 101000 | \text{u}'_i = 001000 |
|                   |                   |                   |                   |                   |                   |
| \text{Region I.} | \text{Region II.} | \text{Region III.} | \text{Region IV.} | \text{Region V. all 3's} | \text{Region VI. all 1's} |
| \text{all 3's} | \text{all 3's} | \text{all 3's} | \text{all 3's} | \text{all 3's} | \text{all 3's} |

**Interpretation of this example:** Form a number in each row by filling in the blanks with 0's. The checked off numbers add up to \(W\):

\[u'_1 + u_2 + u_3 + u'_4 + d_1 + e_1 + d_2 = 331111 = W.\]

Checking off \(u_i\) indicates we should set \(x_i = 1\), and checking off \(u'_i\) indicates we should set \(x_i = 0\). We checked off \(u'_1, u_2, u_3, u'_4\), so it \((x_1, x_2, x_3, x_4) = (0, 1, 1, 0)\) to satisfy the original formula in 3-SAT.

Adding \(u'_1 + u_2 + u_3 + u'_4\) gives 211111. The first two digits mean there are 2 true literals in clause 2 and 1 true literal in clause 1 (out of 3 true literals possible per clause). The last 4 digits mean we selected \(u_i\) or \(u'_i\) but not both, for each \(i\). To compensate for one false literal in clause 2, we check off \(d_2\), and to compensate for two false literals in clause 1, we check off \(d_1\) and \(e_1\).

**Explanation in general:** A subset of the rows (excluding row \(W\)) is indicated by checking them off. If it adds up to \(W\), this is what it means, digit-by-digit:

In each right-hand column (labeled by variables \(x_i\)), the selected rows must add up to 1, forcing either \(u_i\) or \(u'_i\) to be selected, but not both. If \(u_i\) is selected, set \(x_i = 1\); if \(u'_i\) is selected, set \(x_i = 0\).

In each left-hand column (labeled by clauses \(c_j\)), the rules are designed so that if the selected rows add up to 3, the clause is true, and if they don’t add up to 3, it’s false. When they do add up to 3, up to two of the 1’s can come from Region III, so one to three of the 1’s must come from Region I (corresponding to the clause being true when between one and three of its literals are true). If all the literals in this clause are false, the \(x\)’s are not a solution to 3-SAT; further, the most this column sum could be is 2, so we also have not solved the SUM-OF-SUBSETS problem.

Note that we do not have to worry about carries. Even if we added ALL of the numbers above row \(W\) together, we would get \(5 \cdots 52 \cdots 2\), so as long as we work in base 10 (or any other base above 5), there will never be any carries for any subset.
Timing: It takes \(3m\) steps to fill in the nonzero entries of Region I, \(2n\) steps for Region II, \(2m\) for Region III, 0 for Region IV, \(m\) for Region V, and \(n\) for Region VI. The table has dimensions \((2m + 2n + 1) \times (m + n)\), so it takes another \((2m + 2n + 1)(m + n)\) steps to convert this all into the numbers that will be input to SUM-OF-SUBSETS. Let \(L\) be the length of the input (the 3-CNF formula). Since \(m, n \leq L\), the total time is \(O(L^2)\), so translating a problem instance of 3-SAT to one of SUM-OF-SUBSETS takes polynomial time.

Here’s a more precise relation of \(m, n, L\): there are \(2n\) literals \((x_1, \ldots, x_n\) and \(~x_1, \ldots, ~x_n)\), so it takes \([\lg(2n)]\) bits to represent a literal. Each clause has 3 literals, so the input size is about \(L = 3m \cdot [\lg(2n)]\) bits. For every variable to appear in at least one clause requires \(m \geq n/3\). The most clauses we can make from \(2n\) literals is \(\binom{2n}{3}\), so \(\binom{2n}{3} \geq m \geq n/3\).