Instructor: Richard Peng

In the week before the break, we introduced the notion of NP-hardness, then discussed ways of showing that a problem $\Pi$ is NP-complete:

1. Showing that it’s in NP, aka. it has a polynomial time verifier.

2. Showing that for some problem $\hat{\Pi}$, we have $\hat{\Pi} \rightarrow \Pi$, where $\rightarrow$ represents a poly-time reduction.

We then discussed ways of approximating the solution using very simple schemes such as greedy. In particular, we showed how to get a 2-approximation for minimum vertex cover.

In this and the next lecture, we will give the same treatment to the knapsack problem. Recall the problem was given a set of objects, with weights $w_i$ and prices $p_i$, we want to find a subset whose weights do not exceed $W$, and the price is maximized. To convert this to a decision problem, we introduce a 'goal' $g$, and ask whether the total price can be made at least $g$ without exceeding the capacity $W$.

This problem is in NP since a set of objects can be represented using a length $n$ bit mask, and its total capacity can be verified in linear time.

We also talked about an $O(nW)$ time dynamic programming algorithm for this problem. However, if $W$ is represented as binary numbers, we can have some very large numbers, ones that’s exponential in its length. In those settings, the $O(nW)$ bound is actually exponential in the input size.

In the rest of this lecture, we will show that if we do allow inputs with up to $n$ bits in $W$, the problem is in fact NP complete. We will first show a more restrictive version, where we need to exactly meeting the budget. This problem is known as SUBSET-SUM, and asks whether we can exactly make up a total of $W$, where $W$ is the weight limit.
We will show
\[ 3\text{-SAT} \rightarrow \text{SUBSET-SUM} \rightarrow \text{KNAPSACK}. \]

First we show the simpler reduction,

\[ \text{SUBSET-SUM} \rightarrow \text{KNAPSACK} \]

Here we simply keep the \( w_i \)'s the same, but set
\[ p_i \leftarrow w_i, \]
where \( W \) is the limit of the weights. Since knapsack seeks to maximize the profit, it will pick the largest weight that does not exceed the limit, and output \( W \) if it indeed can be achieved. So setting \( g = W \) completes the transformation.

So it remains to show:

\[ 3\text{-SAT} \rightarrow \text{SUBSET-SUM}. \]

The main idea is to treat the digits of \( W \) independently: this can be achieved by adding a large number of 0s between the digits of \( W \), or working with a large base, e.g. \( n \).

We will let each of these digits represent a clause or a variable. Then for each variable \( x_i \), we can create two items: one that corresponds to taking it, and one that corresponds to not taking it. For the item corresponding to \( x_i \), we set its weight by putting a 1 in each digit that corresponds to a clause that contains \( x_i \), and for the item corresponding to \( \overline{x}_i \), we set its weight by putting a 1 in each digit that contains \( \overline{x}_i \). To make sure that we only take one of \( x_i \) or \( \overline{x}_i \), we use a digit whose limit is 1, and has 1 in both \( x_i \) and \( \overline{x}_i \).

We can use a similar construct to make sure that each clause has a true literal. Here things are slightly trickier because clauses can have between 1 to 3 satisfying elements. To account for this flexibility, we require a sum of 9 on the digit corresponding to a clause, and allow ‘dummy’ entries of 6, 7 or 8 on it instead. Note that we need to choose large dummy values (instead of 0, 1, or 2) because we don’t want to use two of them to make up the total instead.

As an example, consider the 3-SAT instance from Figure 8.8 of the textbook:
\[
(x \lor y \lor z) \land (x \lor \overline{y} \lor z) \land (x \lor y \lor z) \land (\overline{x} \lor \overline{y}).
\]

The numbers that we will create in the absence of carries would be:
\[
W = 1119999,
\]
since there are 4 clauses, and the entries corresponding to the variables are:
\[
x : 1000110.
\]
\[
\overline{x} : 1001001.
\]
\[
y : 0101010.
\]
\( \overline{y} : 0100101. \)

\( z : 0010110. \)

\( \overline{z} : 0011000. \)

The satisfying assignment of \( x = F, y = T, Z = T \) gives:

\[
1001001 + 0101010 + 0010110 = 1112121,
\]

to which we then add \( 0007000 + 0000800 + 0000070 + 0000008 \) to make 1119999.

Putting these together then gives

\[
3\text{-SAT} \rightarrow \text{SUBSET-SUM} \rightarrow \text{KNAPSACK},
\]

which completes the proof that KNAPSACK is NP complete.