**DISCLAIMER:** These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

- Course/Instructor Opinion Survey (CIOS) is now available.
- This is the last lecture with required materials.
- Last time:
  1. 3-SAT → SUBSET-SUM → KNAPSACK.
  2. Why KNAPSACK ∈ NP.
  3. Turning SAT clauses into KNAPSACK entries, variables to digits.
  4. Do we need to know the definition of problems such as SUBSET-SUM for the test exam? No, definitions (including the NP-hard problem that you’ll be asked to reduce from) will be provided with the questions.

Last time we showed that KNAPSACK is NP-complete. The way we did it was to create separate digits for each variable/clause in 3-SAT, and then set costs to equal to weights. However, note that the weights that we created are quite different: if we only want a good approximation, enumerating the first few is sufficient.

In this lecture, we will formalize this approximation by showing an \(O(n^3/\epsilon)\) time algorithm for computing a solution whose prices are at least \(1 - \epsilon\) times the optimum.

Recall that the knapsack problem is given \(n\) items with weights \(w_1 \ldots w_n\) and a weight limit \(W\), along with prices \(p_1 \ldots p_n\), find a set \(S\) such that

\[
\sum_{i \in S} w_i \leq W
\]

while maximizing the total prices, \(\sum_{i \in S} p_i\).

The dynamic program that we showed ran in time \(O(nW)\) by having one state per weight. We can also do a dynamic program on prices, for each price that can be obtained, we look for the minimum weight that can do it. As there are no replacements, the dynamic program is

\[
\text{minWeight}(i, p) = \max \{\text{minWeight}(i - 1, p), \text{minWeight}(i, p - p_i) + w_i\}
\]
with the second case only happening if \( p \geq p_i \), and base case \( \text{minWeight}(0, p) = 0 \). Then the answer is simplest the largest \( p \) s.t. \( \text{minWeight}(n, p) \leq W \). This dynamic program can be evaluated in time
\[
O \left( n \sum p_i \right),
\]
and the sum of the \( p_i \)s can be large.

To get around this issue, we ‘round’ the \( p_i \)s: first we discard any item with \( w_i > W \), since it can never fit inside the knapsack. Then we find \( p_{\text{max}} = \max_i p_i \). Note that since this item fits, \( OPT \geq p_{\text{max}} \). This is very important later on when we want to upper bound the total error by \( \epsilon OPT \).

Then we set the ‘unit’ price be \( u = \frac{p_{\text{max}} \epsilon}{n} \), and convert each weight \( p_i \) to
\[
\hat{p}_i \leftarrow u \left\lfloor \frac{p_i}{u} \right\rfloor.
\]
Since \( p_i \leq p_{\text{max}} \), this means all the new prices are multiples of \( u \) between 0 and \( n/\epsilon \). So our dynamic programming states are just \( i \cdot u \) for integers \( i \) between \([0, n/\epsilon]\). This means the analog of \( \sum_i p_i \) is about \( O(n^2/\epsilon) \), giving a total runtime of \( O(n^3/\epsilon) \).

Now it remains to show that there exist a solution whose actual cost is at least \((1 - \epsilon)OPT\). Since the conversion scheme only decreased prices, it suffices to show that decrease cost of the optimum solution is at least \((1 - \epsilon)OPT\). Let this set be \( S^* \).

Note that for any \( x \) we have
\[
\lfloor x \rfloor \geq x - 1,
\]
which means
\[
\hat{p}_i \geq p_i - u.
\]
So the total decrease in price from these roundings is
\[
n \cdot u = \epsilon p_{\text{max}} \leq \epsilon OPT,
\]
so the price of \( S^* \) after this rounding is at least \((1 - \epsilon)OPT\). Since the dynamic program returns the maximum price solution.