# Notes, 11(b)

ECE 606

Weakly vs. Strongly **NP**-hard

KnapSack: given as input (i) $n$ items, each a value-weight pair, $\langle v_i, w_i \rangle$, each $v_i, w_i \in \mathbb{Z}^+$, (ii) a target value $V \in \mathbb{Z}^+$, and, (iii) the weight capacity of a knapsack, $W \in \mathbb{Z}^+$, does there exist a set of items, $J = \{j_1, \ldots, j_k\} \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in J} v_j \geq V$ and $\sum_{j \in J} w_j \leq W$?

**Claim 1.** KnapSack $\in$ ***NP****-complete.*

Proof for **NP**-hardness in your textbook is via a straightforward reduction from SubsetSum. So, assuming $\mathbf{P} \neq \mathbf{NP}$, no polynomial-time algorithm for KnapSack. And therefore no polynomial-time algorithm for the version in which $V$ is not specified, and we seek maximum value that can be achieved.

But! Knapsack has optimal substructure. Which we can leverage in an algorithm based on dynamic programming.

Suppose we perceive the items in some order, $\langle v_1, w_1 \rangle, \ldots, \langle v_n, w_n \rangle$. Now, we need to decide whether we add item $\langle v_n, w_n \rangle$ to the knapsack or not. Our decision is based on whichever choice maximizes our global payoff.

Let $M[i, c]$ be the maximum value we can achieve given items $\langle v_1, w_1 \rangle, \ldots, \langle v_i, w_i \rangle$, and knapsack capacity $c$. Then, we can write a recurrence for $M[i, c]$:

$$M[i,c] = \begin{cases} -\infty & \text{if } c < 0 \\ 0 & \text{if } i = 0 \text{ or } c = 0 \\ \max\{v_i + M[i-1, c-w_i], M[i-1, c]\} & \text{otherwise} \end{cases}$$

Thus our solution, i.e., maximum value we can achieve given all $n$ items and a knapsack capacity of $W$ is $M[n, W]$.

A bottom-up algorithm would look like something like the following.

```
MaxKnapSack-BottomUp({⟨v_1, w_1⟩, ..., ⟨v_n, w_n⟩}, W)
 1 M ← new array [0, ..., n] × [0, ..., W]
 2 foreach i from 0 to n do  M[i, 0] ← 0
 3 foreach c from 0 to W do  M[0, c] ← 0
 4 foreach i from 1 to n do
 5     foreach c from 1 to W do
 6         if c ≥ w_i then  M[i, c] ← max{v_i + M[i − 1, c − w_i], M[i − 1, c]}
 7 return M[n, W]
```

The running-time of the algorithm can be characterized meaningfully as $\Theta(nW)$.

Can that be deemed polynomial-time? The answer is: it depends. On the encoding we adopt. We first recall that "polynomial-time" stands for "in time polynomial in the size of the input."

If our computer uses unary encoding then the integer value $W$'s encoding takes size $\Theta(W)$, and the algorithm is then indeed polynomial- (quadratic-) time.

However, if we use binary, or base-10, or base-16 encoding, then the integer $W$'s encoding takes size $\Theta(\lg W)$. And the running-time is then exponential in the size of the encoding of $W$.

The algorithm MaxKnapSack-BottomUp is what is called a *pseudo polynomial-time* algorithm. And KnapSack is called weakly **NP**-hard, because there exists an encoding for which it is in **P**, but there also exists another encoding in which it is **NP**-hard.