CSE 421

Dynamic Programming

Yin Tat Lee
Weighted Interval Scheduling
Interval Scheduling

- Job $j$ starts at $s(j)$ and finishes at $f(j)$ and has weight $w_j$.
- Two jobs compatible if they don’t overlap.
- Goal: find maximum weight subset of mutually compatible jobs.
Unweighted Interval Scheduling: Review

**Recall**: Greedy algorithm works if all weights are 1:
- Consider jobs in ascending order of finishing time
- Add job to a subset if it is compatible with prev added jobs.

**Observation**: Greedy ALG fails spectacularly if arbitrary weights are allowed:

![Diagram showing scheduling with different weights]
Weighted Job Scheduling by Induction

Suppose $1, \ldots, n$ are all jobs. Let us use induction:

**IH**: Suppose we can compute the optimum job scheduling for $< n$ jobs.

**IS**: Goal: For any $n$ jobs we can compute OPT.

**Case 1**: Job $n$ is not in OPT.
-- Then, just return OPT of $1, \ldots, n - 1$.

**Case 2**: Job $n$ is in OPT.
-- Then, delete all jobs not compatible with $n$ and recurse.

Q: Are we done?
A: No, How many subproblems are there? Potentially $2^n$ all possible subsets of jobs.
Sorting to Reduce Subproblems

Sorting Idea: Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$
IS: For jobs 1, ..., $n$ we want to compute OPT

Case 1: Suppose OPT has job $n$.
- So, all jobs $i$ that are not compatible with $n$ are not OPT
- Let $p(n) =$ largest index $i < n$ such that job $i$ is compatible with $n$.
- Then, we just need to find OPT of 1, ..., $p(n)$
Sorting to Reduce Subproblems

Sorting Idea: Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$
IS: For jobs 1, ..., $n$ we want to compute OPT

Case 1: Suppose OPT has job $n$.
- So, all jobs $i$ that are not compatible with $n$ are not OPT
- Let $p(n) = \text{largest index } i < n \text{ such that job } i \text{ is compatible with } n$.
- Then, we just need to find OPT of 1, ..., $p(n)$

Case 2: OPT does not select job $n$.
- Then, OPT is just the OPT of 1, ..., $n - 2$

Q: Have we made any progress (still reducing to two subproblems)?
A: Yes! This time every subproblem is of the form 1, ..., $i$ for some $i$
So, at most $n$ possible subproblems.
Weighted Job Scheduling by Induction

**Sorting Idea:** Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$

Def $OPT(j)$ denote the weight of OPT solution of $1, \ldots, j$

To solve $OPT(j)$:

**Case 1:** $OPT(j)$ has job $j$.
- So, all jobs $i$ that are not compatible with $j$ are not $OPT(j)$.
- Let $p(j)$ = largest index $i < j$ such that job $i$ is compatible with $j$.
- So $OPT(j) = OPT(p(j)) + w_j$.

**Case 2:** $OPT(j)$ does not select job $j$.
- Then, $OPT(j) = OPT(j - 1)$.

The most important part of a correct DP; It fixes IH

$$OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o. w.}
\end{cases}$$
**Algorithm**

**Input:** $n$, $s(1),...,s(n)$ and $f(1),...,f(n)$ and $w_1,...,w_n$.

Sort jobs by finish times so that $f(1) \leq f(2) \leq \cdots f(n)$.

Compute $p(1),p(2),...,p(n)$

$OPT(j)$ {
    if $(j = 0)$
        return 0
    else
        return $\max (w_j + OPT(p(j)), OPT(j - 1))$.
}
Recursive Algorithm Fails

Even though we have only $n$ subproblems, if we do not store the solution to the subproblems

- we may re-solve the same problem many many times.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence

$$p(1) = 0, p(j) = j - 2$$

$\begin{align*}
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
\end{array}
\end{align*}$

$\begin{align*}
\begin{array}{cccccc}
1 & 2 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\end{align*}$
Algorithm with Memoization

Memorization. Compute and Store the solution of each sub-problem in a cache the first time that you face it. lookup as needed.

**Input:** $n$, $s(1), ..., s(n)$ and $f(1), ..., f(n)$ and $w_1, ..., w_n$.

Sort jobs by finish times so that $f(1) \leq f(2) \leq \cdots f(n)$.

Compute $p(1), p(2), ..., p(n)$

```plaintext
for j = 1 to n
    M[j] = empty
M[0] = 0

OPT(j) {
    if (M[j] is empty)
        M[j] = max (w_j + OPT(p(j)), OPT(j - 1))
    return M[j]
}
```

In practice, you may get stack overflow if $n \gg 10^6$ (depends on the language).
Bottom up Dynamic Programming

You can also avoid recursion
• recursion may be easier conceptually when you use induction

Input: \( n, s(1), \ldots, s(n) \) and \( f(1), \ldots, f(n) \) and \( w_1, \ldots, w_n \).

Sort jobs by finish times so that \( f(1) \leq f(2) \leq \cdots f(n) \).

Compute \( p(1), p(2), \ldots, p(n) \)

\[ \text{OPT}(j) \{ \]
\[ \text{M}[0] = 0 \]
\[ \text{for j = 1 to n} \]
\[ \text{M}[j] = \max (w_j + \text{M}[p(j)], \text{M}[j - 1]). \]
\}[ \]

Output \( \text{M}[n] \)

Claim: \( \text{M}[j] \) is value of \( \text{OPT}(j) \)
Timing: Easy. Main loop is \( O(n) \); sorting is \( O(n \log n) \).
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j)$ = largest index $i < j$ such that job $i$ is compatible with $j$.

$OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max (w_j + OPT(p(j)), OPT(j - 1)) & \text{o.w.}
\end{cases}$

<table>
<thead>
<tr>
<th>$j$</th>
<th>$w_j$</th>
<th>$p(j)$</th>
<th>$OPT(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j$.

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max (w_j + OPT(p(j)), OPT(j-1)) & \text{o.w.}
\end{cases}
\]
Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.

$$OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o.w.}
\end{cases}$$
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.

$$OPT(j) = \begin{cases} 0, & \text{if } j = 0 \\ \max (w_j + OPT(p(j)), OPT(j-1)), & \text{o.w.} \end{cases}$$
**Example**

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).

\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o.w.}
\end{cases}
\]
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).

\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max\left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o.w.}
\end{cases}
\]

<table>
<thead>
<tr>
<th>( j )</th>
<th>( w_j )</th>
<th>( p(j) )</th>
<th>( OPT(j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).

\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max(w_j + OPT(p(j)), OPT(j - 1)) & \text{o.w.}
\end{cases}
\]
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.
$p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o.w.} \end{cases}$$

<table>
<thead>
<tr>
<th>$j$</th>
<th>$w_j$</th>
<th>$p(j)$</th>
<th>$OPT(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.
$p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j$.

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max(w_j + OPT(p(j)), OPT(j-1)) & \text{o.w.}
\end{cases}
\]

<table>
<thead>
<tr>
<th>j</th>
<th>$w_j$</th>
<th>p(j)</th>
<th>OPT(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j$.

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o.w.}
\end{cases}
\]

<table>
<thead>
<tr>
<th>$j$</th>
<th>$w_j$</th>
<th>$p(j)$</th>
<th>OPT($j$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
Dynamic Programming

• Give a solution of a problem using smaller (overlapping) sub-problems where
  the parameters of all sub-problems are determined in-advance

• Useful when the same subproblems show up again and again in the solution.
Knapsack Problem
Knapsack Problem

Given \( n \) objects and a "knapsack."

Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).

Knapsack has capacity of \( W \) kilograms.

**Goal:** fill knapsack so as to maximize total value.

**Ex:** OPT is \( \{ 3, 4 \} \) with value 40.

---

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

**W = 11**

**Greedy:** repeatedly add item with maximum ratio \( v_i/w_i \).

**Ex:** \( \{ 5, 2, 1 \} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.
Dynamic Programming: First Attempt

Let \( OPT(i) = \text{Max value of subsets of items } 1, \ldots, i \text{ of weight } \leq W. \)

Case 1: \( OPT(i) \) does not select item \( i \)
- In this case \( OPT(i) = OPT(i - 1) \)

Case 2: \( OPT(i) \) selects item \( i \)
- In this case, item \( i \) does not immediately imply we have to reject other items
- The problem does not reduce to \( OPT(i - 1) \) because we now want to pack as much value into box of weight \( \leq W - w_i \)

Conclusion: We need more subproblems, we need to strengthen IH.
Stronger DP (Strengthening Hypothesis)

Let $OPT(i, w) = \text{Max value of subsets of items } 1, \ldots, i \text{ of weight } \leq w$

**Case 1**: $OPT(i, w)$ selects item $i$
- In this case, $OPT(i, w) = v_i + OPT(i - 1, w - w_i)$

**Case 2**: $OPT(i, w)$ does not select item $i$
- In this case, $OPT(i, w) = OPT(i - 1, w)$.

Therefore,

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i - 1, w) & \text{if } w_i > w \\ \max(OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)) & \text{o.w.} \end{cases}$$

What is the ordering of item we should pick?
DP for Knapsack

Comp-OPT(i,w)

if M[i,w] == empty
  if (i==0)
    M[i,w]=0
  else if (w_i > w)
    M[i,w]= Comp-OPT(i-1,w)
  else
    M[i,w]= max {Comp-OPT(i-1,w), v_i + Comp-OPT(i-1,w-w_i)}
return M[i, w]

for w = 0 to W
  M[0, w] = 0
for i = 1 to n
  for w = 1 to W
    if (w_i > w)
      M[i, w] = M[i-1, w]
    else
      M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
return M[n, W]

recursive

Non-recursive
DP for Knapsack

\[
\begin{align*}
\text{if } (w_i > w) & \quad M[i, w] = M[i-1, w] \\
\text{else} & \quad M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}
\end{align*}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c}
\hline
\text{Item} & \text{Value} & \text{Weight} \\
\hline
1 & 1 & 1 \\
2 & 6 & 2 \\
3 & 18 & 5 \\
4 & 22 & 6 \\
5 & 28 & 7 \\
\hline
\end{array}
\]
## DP for Knapsack

### Knapsack Problem

We are given `n+1` items, each with a unique identifier from `1` to `n`. The value and weight of each item are given in the table below.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

Let `W` be the maximum weight we can carry. We need to find the maximum value that we can achieve with a knapsack of weight `W`.

### Dynamic Programming Table

<table>
<thead>
<tr>
<th>Item</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \phi )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{1}</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>{1,2}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1,2,3}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1,2,3,4}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1,2,3,4,5}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Recurrence Relation

- If \( w_i > w \):
  \[ M[i, w] = M[i-1, w] \]
- Else:
  \[ M[i, w] = \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \} \]

For `W = 11`, the final value is calculated as:

\[
M[5, 11] = \max \{ M[4, 11], v_5 + M[4, 11-w_5] \} = \max \{ 0, 28 + 0 \} = 28
\]

Thus, the maximum value that can be achieved with a knapsack of weight 11 is **28**.
DP for Knapsack

\[
W + 1
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
\phi & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\{1\} & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\{1, 2\} & 0 & 1 & 6 & 7 & & & & & & & \\
\{1, 2, 3\} & 0 & 1 & & & & & & & & & \\
\{1, 2, 3, 4\} & 0 & 1 & & & & & & & & & \\
\{1, 2, 3, 4, 5\} & 0 & 1 & & & & & & & & & \\
\end{array}
\]

**OPT:** \{4, 3\}
value = 22 + 18 = 40

\[
W = 11
\]

\[
\begin{array}{cccc}
\text{Item} & \text{Value} & \text{Weight} \\
1 & 1 & 1 \\
2 & 6 & 2 \\
3 & 18 & 5 \\
4 & 22 & 6 \\
5 & 28 & 7 \\
\end{array}
\]

if \(w_i > w\)
\[
M[i, w] = M[i-1, w]
\]
else
\[
M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}
\]
DP for Knapsack

if \((w_i > w)\)

\[
M[i, w] = M[i-1, w]
\]

else

\[
M[i, w] = \text{max} \{M[i-1, w], v_i + M[i-1, w-w_i]\}
\]

OPT: \{4, 3\}
value = 22 + 18 = 40

\[
W = 11
\]
DP for Knapsack

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

```
W = 11

OPT: \{4, 3\}
value = 22 + 18 = 40
```

```
if (w_i > w)
    M[i, w] = M[i-1, w]
else
    M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i]}
```
**DP for Knapsack**

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

**OPT:** \( \{ 4, 3 \} \)

value = \( 22 + 18 = 40 \)

\[
\text{if } (w_i > w) \\quad M[i, w] = M[i-1, w] \\
\text{else} \quad M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}
\]
Knapsack Problem: Running Time

**Running time:** $\Theta(n \cdot W)$

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.

**Knapsack approximation algorithm:**
There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum in time $\text{Poly}(n, \log W)$. 

UW Expert
DP Ideas so far

- You may have to define an ordering to decrease subproblems.

- You may have to strengthen DP, equivalently the induction, i.e., you have to carry more information to find the Optimum.

- This means that sometimes we may have to use two dimensional or three dimensional induction.