CSE 421

Dynamic Programming

Yin Tat Lee
Announcement

• No Homework due this week.

• Office hour is both on Zoom and in person this and next week.
  • (As requested by some student.)

• My OH is on Monday. (Sorry that it was not clear in the website before)

• We haven’t graded the midterm. It will be done this week.
Jeremy Lin has created a time machine. Now, he knows exactly the price of $GME$ for the next $n$ days, which are $p_1, p_2, \ldots, p_n$.

Give an algorithm for Jeremy to finds the best days to buy and sell the stocks.
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.
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 the impact of weight on the greedy algorithm](image_url)
Weighted Job Scheduling by Induction

Suppose 1, ..., \( n \) 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, ..., \( n - 1 \).

**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, \( \ldots, n \) 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 optimal schedule for jobs 1, \( \ldots, p(n) \)

Why can’t we order by start time?
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 in $OPT$
- Let $p(n) =$ largest index $i < n$ such that job $i$ is compatible with $n$.
- Then, we just need to find $OPT$ of 1, ..., $p(n)$

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

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 that are not compatible with $j$ are not in $OPT(j)$.
- 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, $OPT(j) = OPT(j - 1)$.

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

The most important part of a correct DP; It fixes IH
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 \( p(1), p(2), \ldots, p(n) \)

\[
OPT(j) \ {\{} \\
\quad \text{if} \ ( \ j = 0 \ ) \\
\quad \quad \text{return} \ 0 \\
\quad \text{else} \\
\quad \quad \text{return} \ \max (w_j + OPT(p(j)), OPT(j - 1)) . \\
\{ 
\]
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

\[ p(1) = 0, p(j) = j - 2 \]
**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:** \( n, s(1), \ldots, s(n) \) and \( f(1), \ldots, f(n) \) and \( w_1, \ldots, w_n \).

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

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

for \( j = 1 \) to \( n \)

1. \( M[j] = \) empty
2. \( M[0] = 0 \)

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

In practice, you may get \( \text{stack overflow} \) 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**: $n$, $s(1), ..., s(n)$ 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 $p(1), p(2), ..., p(n)$

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

} 

Output $M[n]$

**Claim**: $M[j]$ is value of $OPT(j)$

**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 \( j \).

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

\[
\begin{array}{|c|c|c|c|}
\hline
j & w_j & p(j) & OPT(j) \\
\hline
0 & & & 0 \\
1 & 3 & 0 & \\
2 & 4 & 0 & \\
3 & 1 & 0 & \\
4 & 3 & 1 & \\
5 & 4 & 0 & \\
6 & 3 & 2 & \\
7 & 2 & 3 & \\
8 & 4 & 5 & \\
\hline
\end{array}
\]
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 \( j \).

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

Time

0 1 2 3 4 5 6 7 8 9 10 11

<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>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>
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: $f(1) \leq \cdots \leq f(n)$.

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

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

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

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 $j$.

<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></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></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 $j$.

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

We can simply maintain the solution.

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

\[
\text{OPT}(j) \{
    M[0] = 0 \\
    S[0] = \{\}
    \text{for } j = 1 \text{ to } n \\
    \quad \text{if } w_j + M[p(j)] > M[j-1] \\
    \quad \quad M[j] = w_j + M[p(j)]. \\
    \quad S[j] = \{j\} \cup S[p(j)] \quad O(1) \text{ time}
    \quad \text{else} \\
    \quad \quad M[j] = M[j-1] \\
    \quad S[j] = S[j-1] \quad O(1) \text{ time}
    \}
\]

Output \( M[n] \) and \( S[n] \)

What is the runtime of this new algorithm?

Each \( S[j] \) points to some vertices of a tree.

\( \leftarrow \) We add leaf \( j \) with its parent \( S[p(j)] \).
Quiz

Jeremy Lin has created a time machine. Now, he knows exactly the price of $GME$ for the next $n$ days, which are $p_1, p_2, \ldots, p_n$.

Somehow, Jeremy doesn’t want to be labeled as greedy.

So, can you use dynamic programming to help Jeremy instead?
Let $w_k$ be the Jeremy Lin's net worth on the $k$-th day. Then, we have

\[
\begin{align*}
    w_k &= w_{k-1} \times \frac{p_k}{p_{k-1}} \\
    w_k &= \max(w_{k-1} \times \frac{p_k}{p_{k-1}}, w_{k-1}) \\
    w_k &= \max(w_{k-1} + p_k - p_{k-1}, w_{k-1}) \\
    w_k &= w_{k-1} + p_k - p_{k-1} \\
    w_k &= w_{k-1} \times \max(p_k/p_{k-1}, 0)
\end{align*}
\]
Life is not easy.
Robinhood doesn’t want someone to hold $GME to the moon 🚀 🚀

Now, Jeremy can only hold $GME for at most 2 consecutive days.

So, what is the formula for $w_k$?

$$w_k = \max \left( w_{k-1}, w_{k-2} \frac{p_k}{p_{k-1}}, w_{k-3} \frac{p_k}{p_{k-2}} \right).$$
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.

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,

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

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

<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
### DP for Knapsack

#### Table

<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>

#### Dynamic Programming

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

## Table

<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>

## Dynamic Programming

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

\[W = 11\]
DP for Knapsack

\[ M[i, w] = \begin{cases} 
M[i-1, w] & \text{if } w_i > w \\
\max \{M[i-1, w], v_i + M[i-1, w-w_i]\} & \text{else}
\end{cases} \]

<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>

\( W = 11 \)
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>

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>

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

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

\( W = 11 \)
Life is not easy.
Robinhood doesn’t want someone to hold $GME to the moon 🚀 🚀

Now, Jeremy can only hold $GME for at most 2 consecutive days. and can only trade $GME for at most $t$ times.

So, what is the best trading?
Let $w_{k,t}$ be the network at $k$-th day using $t$ trades.

$$w_{k,t} = \max \left( w_{k-1,t}, w_{k-2,t-1} \frac{p_k}{p_{k-1}}, w_{k-3,t-1} \frac{p_k}{p_{k-2}} \right).$$
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 $\text{Poly}(n)$.  

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 may have to carry more information to find the Optimum.

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