ROD CUTTING
A “REAL” DYNAMIC PROGRAMMING EXAMPLE

- **Input:**
  - \( n \): length of rod
  - \( p_1, \ldots, p_n \): \( p_i \) = price of a rod of length \( i \)

- **Output:**
  - Max *income* possible by cutting the rod of length \( n \) into any number of *integer* pieces (maybe no cuts)

**Example output:** 10

<table>
<thead>
<tr>
<th>length ( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>price ( p_i )</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

All ways of cutting a rod of length 4

Example output: 10
DYNAMIC PROGRAMMING APPROACH

- **High level idea** *(can just think recursively to start)*
  - Given a rod of length $n$
  - Either make no cuts,
    or make a cut and **recurse** on the remaining parts

- **Where** should we cut?

\[ \text{Income } p_n \]

\[ \text{Income(Left) + Income(Right)} \]
DYNAMIC PROGRAMMING APPROACH

- Try **all ways** of making that cut
  - i.e., try a cut at positions 1, 2, ..., \( n - 1 \)
  - In each case, recurse on two rods \([0, i]\) and \([i, n]\)
- Take the max income over **all possibilities** (each \( i \) / no cut)

\[
\begin{align*}
  i &= 1 \\
  i &= 2 \\
  i &= 3 \\
  \ldots \\
  i &= n - 1 \\
\end{align*}
\]

Optimal substructure: Max income from two rods w/sizes \( i \) and \( n - i \)

... is max income we can get from the rod size \( i \)

+ max income we can get from the rod size \( n - i \)
RECURRENCE RELATION

- Define \( M(k) = \) maximum income for rod of length \( k \)
- If we do \textbf{not} cut the rod, max income is \( p_k \)
- If we \textbf{do} cut a rod at \( i \)
  - max income is \( M(i) + M(k - i) \)
  - Want to maximize this \textbf{over all} \( i \)
    - \( \max_i \{ M(i) + M(k - i) \} \) (for \( 0 < i < k \))
- \( M(k) = \max \{ p_k, \max_{1 \leq i \leq k-1} \{ M(i) + M(k - i) \} \} \)

Critical step! Must define what \( M(k) \) means, semantically!
Recurrence: $M(k) = \max\{p_k, \max_{1 \leq i \leq k-1} \{M(i) + M(k - i)\}\}$

Compute table of solutions: $M[1..n]$

Dependencies: entry $k$ depends on
- $M[i] \rightarrow M[1..(k - 1)]$
- $M[k - i] \rightarrow M[1..(k - 1)]$

All of these dependencies are $< k$

So we can fill in the table entries in order $1..n$
Recall, semantically, $M(k) =$ maximum income for rod of length $k$

**Recurrence:**

$$M(k) = \max\{p_k, \max_{1 \leq i \leq k-1} (M(i) + M(k-i))\}$$

```plaintext
RodCutting(n, p[1..n])
M = new array[1..n]

// compute each entry M[k]
for k = 1..n
    M[k] = p[k] // current best = no cuts

// try each cut in 1..(k-1)
for i = 1..(k-1)
    M[k] = max(M[k], M[i] + M[k-i])

return M[n]
```

**Time complexity (unit cost)?**  $\Theta(n^2)$
MISCELLANEOUS TIPS

- Building a table of results bottom-up is what makes an algorithm DP

- There is a similar concept called memoization
  - But, for the purposes of this course, we want to see bottom-up table filling!

- Base cases are critical
  - They often completely determine the answer
  - Try setting \( f[0]=f[1]=0 \) in FibDP...
DP SOLUTION TO
0-1 KNAPSACK
Suppose the optimal solution $O$ does not include this.

Then with the $O$ must achieve the best possible value using only items 1-3.

**Subproblem:** output max value for $\leq 7kg$ out of these three items.

Problem: output maximum value one can get from taking $\leq 7kg$, out of these four items.

This is a smaller subproblem: reduced # of items.

Let $P[i, m] =$ maximum profit using any subset of the items 1..$i$, with weight limit $m$.

Note: $P[n, M] (= P[4, 7])$ is the optimal profit.

If $O$ does not include the camera, then $P[4, 7] = \text{best we can do with the first three items and weight limit } 7kg$.


What if the camera IS included in $O$?

Goal: create recurrence relation to describe optimal solution in terms of subproblems.

Item 4
- Camera
- Weight: 1 kg
- Value: 1000$8

Item 3
- Laptop
- Weight: 3 kg
- Value: 2000$8

Item 2
- Necklace
- Weight: 4 kg
- Value: 4000$8

Item 1
- Vase
- Weight: 5 kg
- Value: 4500$8

This diagram shows the items and their weights and values, along with the problem statement and subproblems.
Suppose the optimal solution $O$ includes this subproblem:

Output max value for $\leq 6\text{kg}$ out of these three items.

Problem: Output maximum value one can get from taking $\leq 7\text{kg}$, out of these four items.

This is a smaller subproblem: reduced weight and # of items.

Recall: $P[i, m] = \text{maximum profit using any subset of the items } 1 \ldots i, \text{ with weight limit } m$.

If $O$ includes the camera, then

$P[4, 7] = p_4 + \text{best we can do with the first three items and weight limit } 7\text{kg} - w_4 = 6\text{kg}$


How to evaluate both possibilities: in & not in $O$?

Then with the remaining $7\text{kg} - w_4 = 6\text{kg}$, and items 1-3, $O$ must achieve the best possible value.
Recall: $P[i, m] =$ maximum profit using any subset of the items $1 \ldots i$, with weight limit $m$

If O does not include the camera, then $P[4, 7] =$ best we can do with the first three items and weight limit 7kg

<table>
<thead>
<tr>
<th>If O does not include the camera, then</th>
<th>$P[4, 7] = P[3, 7]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[4, 7] = P[3, 7]$</td>
<td>$P[i, m] = P[i - 1, m]$</td>
</tr>
</tbody>
</table>

If O includes the camera, then $P[4, 7] = p_4 +$ best we can do with the first three items and weight limit 7kg – $w_4 = 6kg$

<table>
<thead>
<tr>
<th>If O includes the camera, then</th>
<th>$P[4, 7] = p_4 + P[3, 7 - w_4]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[4, 7] = p_4 + P[3, 7 - w_4]$</td>
<td>$P[i, m] = p_i + P[i - 1, m - w_i]$</td>
</tr>
</tbody>
</table>

Try both and take the better result! (How?)

<table>
<thead>
<tr>
<th>Try both and take the better result! (How?)</th>
<th>$P[4, 7] = \max{P[3, 7], p_4 + P[3, 7 - w_4]}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[4, 7] = \max{P[3, 7], p_4 + P[3, 7 - w_4]}$</td>
<td>$P[i, m] = \max{P[i - 1, m], p_i + P[i - 1, m - w_i]}$</td>
</tr>
</tbody>
</table>

Note that $\max\{P[i - 1, m], p_i + P[i - 1, m - w_i]\}$ is only valid if $i \geq 2$ and $m \geq w_i$

What to do when $i = 1$ or $m < w_i$? These are special cases.
**General case:** $i \geq 2$ and $m \geq w_i$

Since $m \geq w_i$, we **can carry item** $i$.

$P[i,m] = \max\{P[i-1,m], p_i + P[i-1,m-w_i]\}$

---

**Special case 1:** $i \geq 2$ and $m < w_i$

Since $m < w_i$, we **cannot carry item** $i$.

So, $P[i,m] = P[i-1,m]$.

---

**Special case 2:** $i = 1$ and $m \geq w_i$

Since $i \leq 1$, we **can only use item** 1.

Since $m \geq w_i$, we **can carry item** 1.

So, $P[i,m] = p_i$.

---

**Special case 3:** $i = 1$ and $m < w_i$

Since $i \leq 1$, we **can only use item** 1.

Since $m < w_i$, we **cannot carry item** 1.

So, $P[i,m] = 0$.

---

**Recurrence Relation:**

$$P[i,m] = \begin{cases} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2, m \geq w_i \\ P[i-1,m] & \text{if } i \geq 2, m < w_i \\ p_1 & \text{if } i = 1, m \geq w_1 \\ 0 & \text{if } i = 1, m < w_1. \end{cases}$$
FILLING THE ARRAY:

\[ P[i, m] = \begin{cases} 
\max\{P[i-1, m], p_i + P[i-1, m-w_i]\} & \text{if } i \geq 2, m \geq w_i \\
\max\{P[i-1, m], p_i\} & \text{if } i \geq 2, m < w_i \\
p_1 & \text{if } i = 1, m \geq w_1 \\
0 & \text{if } i = 1, m < w_1.
\]

**i-axis**
(can use items in 1..i)

**m-axis** (remaining weight limit)

No data dependencies on any other array cells.

Suppose item 1 does not fit until this m value (m = w_1)
FILLING THE ARRAY:

\[ P[i, m] = \begin{cases} 
\max\{P[i-1, m], p_i + P[i-1, m-w_i]\} & \text{if } i \geq 2, \ m \geq w_i \\
P[i-1, m] & \text{if } i \geq 2, \ m < w_i \\
p_1 & \text{if } i = 1, \ m \geq w_1 \\
0 & \text{if } i = 1, \ m < w_1.
\end{cases} \]

Suppose \( m < w_2 \) from here

**i-axis**
(can use items in 1..i)

Data dependency: need cell **above** to be computed **already**

\( m \)-axis (remaining weight limit)
FILLING THE ARRAY:

\[ P[i, m] = \begin{cases} 
0 & \text{if } i = 0, \text{ or } m < w_i, \\
p_i & \text{if } i = 1, m \geq w_1, \\
p_i + P[i-1, m-w_i] & \text{if } i > 1, m \geq w_i, \\
p_i + P[i-1, m-w_i] & \text{if } i > 1, m < w_i. 
\end{cases} \]

\[ \text{Entry } [i - 1, m] \]

- **i-axis** (can use items in 1..i)
- **m-axis** (remaining weight limit)

Where is slot \([i - 1, m - w_i]\)?

Consider this entry where \(m \geq w_2\)

Data dependency: need this to be computed already

So, what value should be stored in this entry?

\[ \max\{p_1, p_2 + 0\} \]
FILLING THE ARRAY:

\[
P[i, m] = \begin{cases} 
\max\{P[i - 1, m], p_i + P[i - 1, m - w_i]\} & \text{if } i \geq 2, m \geq w_i \\
P[i - 1, m] & \text{if } i \geq 2, m < w_i \\
p_1 & \text{if } i = 1, m \geq w_1 \\
0 & \text{if } i = 1, m < w_1.
\end{cases}
\]

**i-axis**
(can use items in 1..i)

We only ever look at the previous row!

To satisfy data dependencies, we can fill entries in the order:
for \((i = 1..n)\), for \((m = 0..M)\)

Depending how many zeros we have in the top row, and how far back we're looking, might start to get cells containing \(\max\{p_1, p_2 + p_1\}\)

Would the following fill-order work?
for \((i = 1..n)\), for \((m = M..0)\)

**m-axis** (remaining weight limit)
EXERCISE

\[
\begin{align*}
\text{max}\{P[i - 1, m], p_i + P[i - 1, m - w_i]\} & \quad \text{if } i \geq 2, \ m \geq w_i \\
P[i - 1, m] & \quad \text{if } i \geq 2, \ m < w_i
\end{align*}
\]

Suppose we have profits 1, 2, 3, 5, 7, 10, weights 2, 3, 5, 8, 13, 16, and capacity 30.

The following table is computed:

| i-axis (items) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 1             | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2             | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3             | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 4             | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] |
| 5             | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] |
| 6             | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] | \[\text{？}\] |

\[
P[3, 16] = \text{？}
\]

What do you think?
EXERCISE

\[
\begin{align*}
\text{max}\{P[i - 1, m], p_i + P[i - 1, m - w_i]\} & \quad \text{if } i \geq 2, \ m \geq w_i \\
P[i - 1, m] & \quad \text{if } i \geq 2, \ m < w_i
\end{align*}
\]

Suppose we have profits 1, 2, 3, 5, 7, 10, weights 2, 3, 5, 8, 13, 16, and capacity 30.

The following table is computed:

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 5 | 5 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 4 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 10 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| 5 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 6 | 7 | 7 | 8 | 8 | 9 | 10 | 10 | 11 | 11 | 11 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 6 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |

\[P[3, 16] = \text{max}\{P[2, 16], P[2, 11] + 3\} = \text{max}\{3, 3 + 3\} = 6.\]
Recall: To satisfy data dependencies, we can fill entries in the order: for \((i = 1..n)\), for \((m = 0..M)\)

\[
P[i, m] = \begin{cases} 
\max\{P[i-1, m], p_i + P[i-1, m-w_i]\} & \text{if } i \geq 2, m \geq w_i \\
P[i-1, m] & \text{if } i \geq 2, m < w_i \\
p_i & \text{if } i = 1, m \geq w_1 \\
0 & \text{if } i = 1, m < w_1.
\end{cases}
\]

Knapsack01(p[1..n], w[1..n], M)

1. P = new table[1..n][0..M]
2. // base cases where \(i=1\)
   for m = 0..M
     if m < w[1] then
       P[1][m] = 0
     else
       P[1][m] = p[1]
3. // general cases where \(i\geq2\)
   for i = 2..n
     for m = 0..M
       if m < w[i] then
         P[i][m] = P[i-1][m]
       else
         P[i][m] = \max(P[i-1][m], p[i] + P[i-1][m-w[i]])
4. return P[n][M]

Read & return optimal profit

How about the optimal items?
The optimal solution is computed by tracing back through the table. For the previous example, consisting of profits 1, 2, 3, 5, 7, 10, weights 2, 3, 5, 8, 13, 16, and capacity 30, the optimal solution is

<table>
<thead>
<tr>
<th>Items you can take</th>
<th>weight limit remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30</td>
</tr>
<tr>
<td>1</td>
<td>0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 2 2 3 3 3 4 5 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6</td>
</tr>
<tr>
<td>4</td>
<td>0 0 1 2 2 3 3 4 5 5 6 7 7 8 8 9 10 10 11 11 11 11 11 11 11 11 11 11 11 11 11</td>
</tr>
<tr>
<td>5</td>
<td>0 0 1 2 2 3 3 4 5 5 6 7 7 8 8 9 10 10 11 11 11 11 11 11 11 11 11 11 11 11 11</td>
</tr>
<tr>
<td>6</td>
<td>0 0 1 2 2 3 3 4 5 5 6 7 7 8 8 9 10 10 11 11 11 11 11 11 11 11 11 11 11 11 11</td>
</tr>
</tbody>
</table>

- **Items you can take**: 1, 2, 3, 4, 5, 6
- **8 > 6 so** O **must take item 4**
- **Same profit using items 1..4 or 1..5. So, there exists an optimal solution O that does not use item 5! Consider O.**
- **Best profit for remaining items + weight**
- **18 > 17, so any optimal solution must take item 6**
- **remaining weight = 14**
- **Start at optimal profit**
- **Exercise:** continue, and determine which other items are in O
The optimal solution is computed by tracing back through the table. For the previous example, consisting of profits 1, 2, 3, 5, 7, 10, weights 2, 3, 5, 8, 13, 16, and capacity 30, the optimal solution is [1, 1, 0, 1, 0, 1].

| Items you can take | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|--------------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1                  | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| 2                  | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  |
| 3                  | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  |
| 4                  | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 4 | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  |
| 5                  | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 4 | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  |
| 6                  | - | - | - | - | - | - | - | - | - | - | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | -  | 18 |
Runtime given $P$?

$\Theta(n)$

Is this linear time?

More on this soon...
Complexity of the Algorithm

Suppose we assume the unit cost model, so additions / subtractions take time $O(1)$.

The complexity to construct the table is $\Theta(nM)$.

Is this a polynomial-time algorithm, as a function of the size of the problem instance?

We have

$$
size(I) = \log_2 M + \sum_{i=1}^{n} \log_2 w_i + \sum_{i=1}^{n} \log_2 p_i.
$$

Note in particular that $M$ is exponentially large compared to $\log_2 M$. So constructing the table is not a polynomial-time algorithm, even in the unit cost model.

What would the complexity of a recursive algorithm be?

So the DP alg is faster when there are many item types, but small weight limit.

Huge $n$ is fine, but $M$ should be in $\text{poly}(n)$ to get an asymptotic improvement.

A recursive algorithm would take $\sim \Theta(2^n)$ time.

DP takes $\Theta(nM)$ time, which could be $\Theta(n2^n)$ for huge $M$.

$n$ must be very small.
Simplifying Base Cases

For $i = 1$, $m < w_i$, we have $P[i - 1, m]$, which is 0.

For $i = 1$, $m \geq w_i$, we have $p_i + P[i - 1, m - w_i]$ which is $p_i + 0$.

$i$-axis (can use items in $1 \ldots i$)

$m$-axis (remaining weight limit)

$P[i, m] = \begin{cases} 
\max\{P[i-1, m], p_i + P[i-1, m-w_i]\} & \text{if } i \geq 2, m \geq w_i \\
P[i-1, m] & \text{if } i \geq 2, m < w_i \\
p_i & \text{if } i = 1, m \geq w_1 \\
0 & \text{if } i = 1, m < w_1.
\end{cases}$
We get much simpler code!

Compare:

```java
Knapsack01(p[1..n], w[1..n], M)
    P = new table[0..n][0..M] containing zeros
    for i = 1..n
        for m = 0..M
            if m < w[i] then
                P[i][m] = P[i-1][m]
            else
                P[i][m] = max(P[i-1][m],
                               p[i] + P[i-1][m-w[i]])
    return P[n][M]
```
SAVING SPACE

We never look at $P[i-2][...].$ Just keep two arrays representing $P[i]$ and $P[i-1].$

Space complexity changes from $O(mn)$ to $O(m)$
COIN CHANGING
Coin Changing

Problem 5.2

Coin Changing
Instance: A list of coin denominations, \(1 = d_1, d_2, \ldots, d_n\), and a positive integer \(T\), which is called the target sum.
Find: An \(n\)-tuple of non-negative integers, say \(A = [a_1, \ldots, a_n]\), such that \(T = \sum_{i=1}^{n} a_i d_i\) and such that \(N = \sum_{i=1}^{n} a_i\) is minimized.

What subproblems should be considered?
What table of values should we fill in?

There is a denomination with unit value!

In 0-1 knapsack, we only considered two subproblems in our recurrence: taking an item, or not.
Here we can do more than use a coin denomination or not.
Let $N[i, t]$ denote the optimal solution to the subproblem consisting of the first $i$ coin denominations $d_1, \ldots, d_i$ and target sum $t$.

<table>
<thead>
<tr>
<th>Exploring: some sensible base case(s)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>General case:</td>
</tr>
<tr>
<td>What are the different ways we could use coin denomination $d_i$?</td>
</tr>
<tr>
<td>What subproblems / solutions should we use?</td>
</tr>
<tr>
<td>Final recurrence relation</td>
</tr>
</tbody>
</table>
Let $N[i, t]$ denote the optimal solution to the subproblem consisting of the first $i$ coin denominations $d_1, \ldots, d_i$ and target sum $t$. Since $d_1 = 1$, we immediately have $N[1, t] = t$ for all $t$.

General case:
What are the different ways we could use coin denomination $d_i$?
What subproblems / solutions should we use?

Also $N[i, 0] = 0$ for all $i$
Let $N[i, t]$ denote the optimal solution to the subproblem consisting of the first $i$ coin denominations $d_1, \ldots, d_i$ and target sum $t$.

Since $d_1 = 1$, we immediately have $N[1, t] = t$ for all $t$.

For $i \geq 2$, the number of coins of denomination $d_i$ is an integer $j$ where $0 \leq j \leq \lfloor t/d_i \rfloor$.

If we use $j$ coins of denomination $d_i$, then the target sum is reduced to $t - jd_i$, which we must achieve using the first $i - 1$ coin denominations.

Thus we have the following recurrence relation:

$$N[i, t] = \begin{cases} 
\min \{j + N[i - 1, t - jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor \} & \text{if } i \geq 2 \\
\min \{ t \} & \text{if } i = 1 \text{ OR } t = 0
\end{cases}$$
FILLING THE ARRAY

\[ N[1 \ldots n, 0 \ldots T] : \]

\[ N[i, t] = \begin{cases} 
\min\{j + N[i-1, t-jd_i] : 0 < j < |t/d_i|\} & \text{if } i > 2 \\
\frac{t}{d_i} & \text{if } i = 1. \\
0 & \text{OR } t = 0 
\end{cases} \]

No data dependencies on any other array cells.

\( i \)-axis (coin type)

(recall: \( N[i, t] \) uses coin types 1..i)
FILLING THE ARRAY $N[1 \ldots n, 0 \ldots T]$:

$N[i, t] = \begin{cases} \min\{j + N[i - 1, t - jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor\} & \text{if } i \geq 2 \\ t & \text{if } i = 1 \text{, OR } t = 0 \end{cases}$

<table>
<thead>
<tr>
<th>$t$-axis (target sum remaining)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>$\ldots$</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\ldots$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No data dependencies on any other array cells.

**i-axis** (coin type)

(recall: $N[i, t]$ uses coin types $1 \ldots i$)
FILLING THE ARRAY $N[1 \ldots n, 0 \ldots T]$:

$$N[i, t] = \begin{cases} \min\{j + N[i-1, t-jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor\} & \text{if } i > 2 \\ t & \text{if } i = 1. \end{cases}$$

OR $t = 0$

$i$-axis (coin type)

(recall: $N[i, t]$ uses coin types $1 \ldots i$)

$t$-axis (target sum remaining)

Consider cell $N[i, t]$

We only look at the previous $i$-row!

It is sufficient to fill:
row $i=1$ (base case), then
for $(i = 2 \ldots n)$, for $(t = 0 \ldots T)$

$d_i$
CoinChangingDP(d[1..n], T)

N = new table[1..n][0..T]
J = new table[1..n][0..T]

for t = 0..T  // base cases where i=1
    N[1][t] = t
    J[1][t] = t

for i = 2..n  // general cases
    for t = 0..T
        // initially best solution is 0 of d[i]
        N[i][t] = N[i-1][t]
        J[i][t] = 0

        // try j>0 coins of type d[i]
        for j = 1..floor(t / d[i])
            if j + N[i-1][t-j*d[i]] < N[i][t]
                N[i][t] = j + N[i-1][t-j*d[i]]
                J[i][t] = j // best is currently j of d[i]

return N[n][T]  // can also return N, J

\[ N[i,t] = \begin{cases} 
\min\{j + N[i-1,t-jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor\} & \text{if } i \geq 2 \\
    t & \text{if } i = 1.
\end{cases} \]
Exercise for later:
compute the correct output
without using $J[i, t]$
(i.e., using only $N, d, T$)
CoinChangingDP(d[1..n], T)
N = new table[1..n][0..T]
J = new table[1..n][0..T]

for t = 0..T  // base cases where i=1
    N[1][t] = t
    J[1][t] = t

for i = 2..n  // general cases
    for t = 0..T
        // initially best solution is 0 of d[i]
        N[i][t] = N[i-1][t]
        J[i][t] = 0

        // try j>0 coins of type d[i]
        for j = 1..floor(t / d[i])
            if j + N[i-1][t-j*d[i]] < N[i][t]
                N[i][t] = j + N[i-1][t-j*d[i]]
                J[i][t] = j  // best is currently j of d[i]

return N[n][T]  // can also return N, J

Time complexity?

**Unit cost** computational model is reasonable here

Consider instance $I = (d, T)$

Runtime $R(I) \in O \left( \sum_{i=2}^{n} \sum_{t=0}^{T} \left\lfloor \frac{t}{d_i} \right\rfloor \right)$

\[
R(I) \in O \left( \sum_{i=2}^{n} \frac{1}{d_i} \sum_{t=0}^{T} t \right) \quad R(I) \in O \left( \sum_{i=2}^{n} \frac{1}{d_i} \frac{T(T + 1)}{2} \right)
\]

\[
R(I) \in O(DT^2)
\]

where $D = \sum_{i=2}^{n} \frac{1}{d_i} < n.$

If T is small, this is much better than brute force
POLYNOMIAL TIME

- An algorithm runs in (worst case) polynomial time IFF its runtime $R(I)$ on every input is upper bounded by a polynomial in the input size $S$
  - I.e., $R(I) \in O(c_0 + c_1 S + c_2 S^2 + c_3 S^3 + \cdots + c_k S^k)$
    for constants $k$ and $c_0, \ldots, c_k$
  - ... so is $O(nT^2)$ polynomial in our input size $S$?
INPUT SIZE

- $S = \text{bits}(T) + \text{bits}(d_1) + \cdots + \text{bits}(d_n)$
- It takes $\lceil \log_2 T \rceil$ bits to store $T$
- It takes $\lceil \log_2 d_i \rceil$ bits to store each $d_i$
- Assume $d_i \leq T$ (otherwise $d_i$ cannot be used at all, and should be omitted from the input)
  - Then we have $\lceil \log_2 d_i \rceil \in O(\log T)$
  - So, $S \in O(n \log T)$
COMPARING $T(I)$ TO $S$

- Recall $R(I) \in O(nT^2)$ and $S \in O(n \log T)$
- As an example, if $n$ is fixed at 10 and $T$ is allowed to vary, then $S \in O(\log T)$ and $R(I) \in O(T^2)$
  - In this case, $R(I)$ is exponential in $S$
- However, if $T$ is fixed at 10 and $n$ is allowed to vary, then $S \in O(n)$ and $R(I) \in O(n)$
  - In this case, $R(I)$ is linear in $S$
- So, large $n$ and small $T$ is where this DP solution shines!
A BIT MORE ANALYSIS

- Recall $R(I) \in O(nT^2)$ and $S \in O(n \log T)$
- If $T \in O(n)$, then $S \in O(n \log n)$ and $R(I) \in O(n^3)$
  - Note $O(n^3)$ is a **smaller** runtime than $O(S^3) = O(n^3 \log n)$
  - And $S^3$ is polynomial in $S$, so $O(n^3)$ is a **polynomial runtime**
- So, for some inputs with relatively small $T$, we can get polynomial runtimes!
  - In particular, for $T \in O(n^k)$ where $k$ is constant, $R(I) \in O\left(n(n^k)^2\right) = O(n^{2k+1})$ and $S \in O(n \log n^k) = O(n \log n)$
  - And $R(I) \in O(n^{2k+1}) \subseteq O\left((n \log n)^{2k+1}\right) = O(S^{2k+1})$