ROD CUTTING
A “REAL” DYNAMIC PROGRAMMING EXAMPLE

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

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

\( n = 4 \)

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 \textit{recurse} on the remaining parts

- **Where** should we cut?

Income \( p_n \)

Income(Left) + Income(Right)

RECURSION RELATION

- Define \( M(k) = \text{maximum income for rod of length } k \)
- If we \textit{do not} cut the rod, max income is \( p_k \)
- If we \textit{do} cut a rod at \( i \)

\[
\begin{align*}
\text{max income is } & M(i) + M(k - i) \\
\text{Want to maximize this over all } & i \text{ for } 0 < i < k \\
\text{max } & \{ M(i) + M(k - i) \}
\end{align*}
\]

OPTIMAL SUBSTRUCTURE:

Max income from two rods w/sizes \( i \) and \( k - i \)

is max income we can get from the rod size \( i \) + max income we can get from the rod size \( k - i \)

COMPUTING SOLUTIONS BOTTOM-UP

- **Recurrence:** \( M(k) = \max\{ p_k, \max_{i=1}^{k-1} (M(i) + M(k - i)) \} \)
- **Compute** \textit{table} of solutions: \( M[1..n] \)
- **Dependencies:** \textit{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, $N(k) = \text{maximum income for rod of length } k$

Recurrence: $N(k) = \max(\{p_i + \max_{w_i \leq k} [N(k - w_i)]: 1 \leq i \leq n\})$

**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 $t[0]=t[1]=0$ in RibDP...

---

**DP SOLUTION TO 0-1 KNAPSACK**

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

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

- Suppose the optimal solution does not include $i$
  - Then with the $i$ must achieve the best possible value using only items $1 \ldots i-1$
- What if the camera is included in $O$?

**Subproblems:**
- Output max value for 5 kg out of these three items
- This is a smaller subproblem: reduced # of items

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

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

Note: $P[4, 4] = P[4, 7]$ is the optimal profit

- If $O$ does not include the camera, then $P[4, 7] = P[4, 4]$ but we can do with the first three items and weight limit 7kg
  - That is, $P[4, 7] = P[3, 7]$

**In general**

- Recall: $P[i, j] = \text{maximum profit using any subset of the items } 1 \ldots i, \text{ with weight limit } j$
- If $O$ does not include the camera, then $P[4, 7] = P[4, 4]$ but we can do with the first three items and weight limit 7kg
  - That is, $P[4, 7] = P[3, 7]$
- If $O$ includes the camera, then $P[4, 7] = P[4, 4]$ but we can do with the first three items and weight limit 7kg
  - That is, $P[4, 7] = P[3, 7]$

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

- Note that this $P[2, -1]$ is only valid if $i \geq 2$ and $m \geq w_i$
- What to do when $i = 1$ or $w_i < w_i$? These are special cases.

---

**Item 1**

- Has weight $w_1 = 1$, value $p_1 = 1$.

**Item 2**

- Has weight $w_2 = 2$, value $p_2 = 4$.

**Item 3**

- Has weight $w_3 = 3$, value $p_3 = 3$.

**Item 4**

- Has weight $w_4 = 5$, value $p_4 = 5$.

**Memoization**

- $P[i, j] = \text{maximum profit using any subset of the items } 1 \ldots i, \text{ with weight limit } j$
FILLING THE ARRAY:

Recurrence Relation:

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

**General case:** \(i \geq 2\) and \(m \geq w_i\)

Since \(m \geq w_i\), we can carry item 1.

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

**Special case 1:** \(i = 2\) and \(m < w_i\)

Since \(m < w_i\), we cannot carry item 1.

\[ P[i, m] = 0 \]

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

Since \(i = 1\), we can only use item 1.

\[ P[i, m] = p[i-1, m] \]

**Special case 3:** \(i = 1\) and \(m < w_i\)

Since \(i = 1\), we can only use item 1.

\[ P[i, m] = 0 \]

**Data dependency:**

To satisfy data dependencies, we can fill entries in the order:

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

To fill entries in the order:

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

**Exercises:**

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:

**m-axis (weight)**

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>13</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ P[3, 16] = ? \]

What do you think?
**EXERCISE**

\[ \max \{ P[i-1, m] + P[i-1, m - w_i] \} \quad \text{if } i \geq 2, m \geq w_i \]
\[ P[i-1, m] \quad \text{if } i \geq 2, m < w_i \]

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:

<table>
<thead>
<tr>
<th>(i)-axis</th>
<th>(w)-axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items</td>
<td>(weight)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ P[3, 16] = \max \{ P[2, 16], P[2, 11] + 3 \} = \max \{3, 3 + 3\} = 6. \]

---

**OUTPUTTING CONTENTS OF THE OPTIMAL KNAPSACK**

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>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

Best profit for remaining items + weight

**OUTPUTTING CONTENTS OF THE OPTIMAL KNAPSACK**

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].

---

**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

\[ \text{time} = \log_2 M \cdot n + \sum_{i=1}^{n} \log_2 w_i \cdot \sum_{j=1}^{i-1} \log_2 p_j. \]

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?

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

So the DP step 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.

DP takes \(\Theta(nM)\) time, which could be \(\Theta(n^2)\) for huge \(M\).

Note that \(M\) must be very small.

A recursive algorithm would take \(\Theta(2^n)\) time.
SIMPLIFYING BASE CASES

For \( i \geq 1 \) and \( m \geq w_i \):

- \( P[i, m] = P[i-1, m-w_i] \)

For \( i \geq 1 \) and \( m < w_i \):

- \( P[i, m] = P[i-1, m] \)

For \( i = 0 \) and \( m < w_1 \):

- \( P[0, m] = 0 \)

For \( i = 1 \):

- \( P[1, m] = \max(P[0, m], p_1 + P[0, m-w_1]) \)

We get much simpler code!

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

There is a denomination with unit value!

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 \( t \).

Exploring: some sensible base case(s)?

General case:
What are the different ways we could use coin denomination \( d_i \)?

What subproblems / solutions should we use?

Final recurrence relation
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 $d_i = 1$, we immediately have $N[i, t] = t$ for all $i$.

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 $i$.

Final recurrence relation

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:

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

Compute $\min(\cdots)$ over $j = 0 \ldots \lfloor t/d_i \rfloor$.

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

Consider cell $N[i, t]$.
If $t = 0$, we only look at the previous $i$-row!
If $t > 0$, we only look at the previous $i$-row!

\[\begin{array}{cccccccccccc} \hline i & t & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline \text{i-axis (coin type)} & \text{i-axis (coin type)} & \text{i-axis (coin type)} & \text{i-axis (coin type)} & \text{i-axis (coin type)} & \text{i-axis (coin type)} & \text{i-axis (coin type)} & \text{i-axis (coin type)} & \text{i-axis (coin type)} & \text{i-axis (coin type)} & \text{i-axis (coin type)} \\ \text{| recall: N[i,t] uses coin types 1...i | recall: N[i,t] uses coin types 1...i | recall: N[i,t] uses coin types 1...i | recall: N[i,t] uses coin types 1...i | recall: N[i,t] uses coin types 1...i | recall: N[i,t] uses coin types 1...i | recall: N[i,t] uses coin types 1...i | recall: N[i,t] uses coin types 1...i | recall: N[i,t] uses coin types 1...i | recall: N[i,t] uses coin types 1...i | recall: N[i,t] uses coin types 1...i } \\ \hline \end{array} \]
OUTPUTTING OPTIMAL SET OF COINS

```
1 CoinChangerGP_coins(d, n, k):
2     counts = new array[2][n]
3     t = T
4     for i = 1 to n:
5         counts[i] = j[i][t]
6     t = t - counts[i] * d[i]
7     return counts
```

Exercise for later: compute the correct output without using \( \sigma \) and \( t \) (i.e., using only \( k, d, T \))

POLYNOMIAL TIME

An algorithm runs in (worst case) **polynomial time** iff its runtime \( R(T) \) on every input is upper bounded by a polynomial in the input size \( S \)

i.e., \( R(T) \in O(c_k + c_2d^2 + c_3d^3 + \cdots + c_3d^3) \)

... so is \( O(nT^2) \) polynomial in our input size \( S \)?

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 \( S \in O(n \log T) \) 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!

INPUT SIZE

\( S = \text{bits}(T) + \text{bits}(d_i) + \cdots + \text{bits}(d_n) \)

- It takes \( \log_2(T) \) bits to store \( T \)
- It takes \( \log_2(d_i) \) 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 \( \log_2(d_i) \in O(\log T) \)
  - So, \( S \in O(n \log T) \)

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 \( R(I) \in O(n^2) \)
  - Note \( O(n^2) \) 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, \( R(I) \in O(n(\log n)^k) \) and \( S \in O(n \log n) = O(n \log n) \)
- And \( R(I) \in O(n^{k+1}) \subset O((n \log n)^{k+1}) = O(S^{k+1}) \)