Knapsack:

Total capacity $B$

and $n$ objects with

integer weights $w_1, \ldots, w_n$

& integer values $v_1, \ldots, v_n$

Goal: Find subset $S$ of objects

that (a) fits in the backpack

(so total weight is $\leq B$)

& (b) maximizes the total value

In other words, find subset $S$ of objects where

(a) $\sum_{i \in S} w_i \leq B$

& (b) maximizes $\sum_{i \in S} v_i$

Application: Scheduling jobs with limited resources

More importantly, dynamic programming approach will be used frequently.
Two versions:

1) without repetition: one copy of each object (so \( S \) is a set).
2) with repetition: unlimited supply of each object (so \( S \) is a multiset).

Today: without repetition — from now on assume one copy per object.

Next class: with repetition.

First off, what about a greedy approach?

Example:

<table>
<thead>
<tr>
<th>Object</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Weights</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ B = 22 \]

Sort objects by \( R_i = \frac{V_i}{W_i} \), value per unit of weight.

\( R_1 > R_2 > R_3 > R_4 \)

Greedy: objects 1 & 4, for total value = 16

Optimal: 2 & 3, total value = 18
Dynamic Programming approach:

First, define the subproblem

Initial idea, consider prefixes of objects

\[ K(j) = \text{max value achievable using a subset of objects } 1, \ldots, j. \]

Try to write recurrence for \( K(j) \) in terms of \( K(1), \ldots, K(j-1) \).

Take \( K(j-1) \), how do we know whether we can add object \( j \) to the solution for \( K(j-1) \)?

Need to know how much weight is available.

Want to know for each weight \( \leq 6 \) the best solution for objects \( 1, \ldots, j-1 \).
Thus, for $b \leq j$ where $0 \leq b \leq B$ and $0 \leq j \leq n$

let $K(b, j) =$ max value achievable using a subset of objects $1, \ldots, j$ & total capacity $b$.

Goal: compute $K(B, n)$.

For recurrence for $K(b, j)$, either:

(i) use object $j$:

- gains value $v_j$ & available capacity goes down by $w_j$ so we want optimal solution for objects $1, \ldots, j-1$ with capacity $b-w_j$.

Hence, in this case, max value is $v_j + K(b-w_j, j-1)$. 

(ii) Don't use object \( j \):

- Use optimal solution for \( 1 \ldots j-1 \) objects
- with capacity \( b \)

Hence, in this case, max value is \( K(b, j-1) \).

Therefore,

if \( w_j \leq b \),

then \( K(b, j) = \max \{ m_j + K(b - w_j, j-1), K(b, j-1) \} \)

else \( K(b, j) = K(b, j-1) \)

Base cases:

\( K(b, 0) = 0 \)
\( K(0, j) = 0 \).
Recurrence for $K(\cdot,j)$ uses $K(\cdot,j-1)$
Hence fill table from $j=0 \rightarrow j=n$.

So fill column by column

Knapsack No Repeat $(B, w_1, \ldots, w_n, v_1, \ldots, v_n)$
for $j=0 \rightarrow n$, $K(0,j) = 0$
for $b=0 \rightarrow B$, $K(b,0) = 0$
for $j=1 \rightarrow n$
    for $b=1 \rightarrow B$
        if $w_j \geq b$
            then $K(b,j) = K(b,j-1)$
        else $K(b,j) = \max \{ K(b-1,j), v_j + K(b-w_j, j-1) \}$
return $(K(B,n))$
Running time: $O(nB)$

Is this algorithm have running time that is polynomial in the input size?

It's $\text{poly}(n)$

but to write $B$ only takes $O(\log B)$ bits, hence it's exponential & we'd like an algorithm with running time $\text{Poly}(n, \log B)$.

As we'll see later, that's unlikely because Knapsack is NP-complete.