Dynamic Programming:
Interval Scheduling and Knapsack
6.1 Weighted Interval Scheduling
Weighted Interval Scheduling

Weighted interval scheduling problem.

- Job \( j \) starts at \( s_j \), finishes at \( f_j \), and has weight or value \( v_j \).
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

How?
- Divide & Conquer?
- Greedy?
Recall. Greedy algorithm works if all weights are 1.
- Consider jobs in ascending order of finish time.
- Keep job if compatible with previously chosen jobs.

Observation. Greedy fails spectacularly with arbitrary weights.

Exercises: by “density” = weight per unit time? Other ideas?
Weighted Interval Scheduling

**Notation.** Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).

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

“\( p \)” suggesting (last possible) “predecessor”

**Ex:** \( p(8) = 5 \), \( p(7) = 3 \), \( p(2) = 0 \).

<table>
<thead>
<tr>
<th>j</th>
<th>p(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</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>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
Dynamic Programming: Binary Choice

Notation. \( \text{OPT}(j) = \text{value of optimal solution to the problem consisting of job requests 1, 2, ..., j.} \)

- Case 1: Optimum selects job \( j \).
  - can't use incompatible jobs \( \{ p(j) + 1, p(j) + 2, ..., j - 1 \} \)
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., \( p(j) \)

- Case 2: Optimum does not select job \( j \).
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., \( j-1 \)

key idea: binary choice

\[
\text{OPT}(j) = \begin{cases} 
  0 & \text{if } j = 0 \\
  \max \left\{ v_j + \text{OPT}(p(j)), \text{OPT}(j-1) \right\} & \text{otherwise}
\end{cases}
\]
Weighted Interval Scheduling: Brute Force Recursion

Brute force recursive algorithm.

Input: $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

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

$$\text{Compute-Opt}(j) \{$$
  $$\text{if} \ (j = 0)$$
  $$\text{return} \ 0$$
  $$\text{else}$$
  $$\text{return} \ \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))$$
$$\}$$
Weighted Interval Scheduling: Brute Force

**Observation.** Recursive algorithm is correct, but spectacularly slow because of redundant sub-problems $\Rightarrow$ exponential time.

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

\[ p(1) = p(2) = 0; \quad p(j) = j-2, \quad j \geq 3 \]
Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

Input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

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

Iterative-Compute-Opt 
\[
\text{OPT}[0] = 0 \\
\text{for } j = 1 \text{ to } n \\
\quad \text{OPT}[j] = \max(v_j + \text{OPT}[p(j)], \text{OPT}[j-1])
\]

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

Claim: \( \text{OPT}[j] \) is value of optimal solution for jobs 1..j

Timing: Loop is \( O(n) \); sort is \( O(n \log n) \); what about \( p(j) \)?
Weighted Interval Scheduling

**Notation.** Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).

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

**Ex:** \( p(8) = 5, p(7) = 3, p(2) = 0 \).

<table>
<thead>
<tr>
<th>j</th>
<th>( v_j )</th>
<th>( p_j )</th>
<th>( \text{opt}_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Weighted Interval Scheduling Example

Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$. $p(j) =$ largest $i < j$ s.t. job $i$ is compatible with $j$.

Exercise: try other concrete examples:
If all $v_j = 1$: greedy by finish time $\rightarrow 1, 4, 8$
what if $v_2 > v_1$?, but $v_1 + v_4$?
$v_2 > v_1 + v_4$, but $v_2 + v_6 < v_1 + v_7$, say? etc.

<table>
<thead>
<tr>
<th>j</th>
<th>pj</th>
<th>vj</th>
<th>$\max(v_j + \text{opt}[p(j)], \text{opt}[j-1]) = \text{opt}[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>$\max(2+0, 0) = 2$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>$\max(3+0, 2) = 3$</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>$\max(1+0, 3) = 3$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>6</td>
<td>$\max(6+2, 3) = 8$</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>9</td>
<td>$\max(9+0, 8) = 9$</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>7</td>
<td>$\max(7+3, 9) = 10$</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>$\max(2+3, 10) = 10$</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>?</td>
<td>$\max(8+9, 10) = ?$</td>
</tr>
</tbody>
</table>

Exercise: What values of $v_8$ cause it to be in/excluded from opt?
Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?
A. Do some post-processing – “traceback”

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
  if (j = 0)
    output nothing
  else if (v_j + OPT[p(j)] > OPT[j-1])
    print j
    Find-Solution(p(j))
  else
    Find-Solution(j-1)
}
```

- # of recursive calls ≤ n ⇒ O(n).

the condition determining the max when computing OPT[
the relevant sub-problem
Weighted Interval Scheduling Example

Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
$p(j) = \text{largest } i < j \text{ s.t. job } i \text{ is compatible with } j$.

<table>
<thead>
<tr>
<th>j</th>
<th>pj</th>
<th>vj</th>
<th>$\max(v_j + \text{opt}[p(j)], \text{opt}[j-1]) = \text{opt}[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>$\max(2+0, \quad 0) = 2$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>$\max(3+0, \quad 2) = 3$</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>$\max(1+0, \quad 3) = 3$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>6</td>
<td>$\max(6+2, \quad 3) = 8$</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>9</td>
<td>$\max(9+0, \quad 8) = 9$</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>7</td>
<td>$\max(7+3, \quad 9) = 10$</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>$\max(2+3, \quad 10) = 10$</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>2</td>
<td>$\max(2+9, \quad 10) = 11$</td>
</tr>
</tbody>
</table>

$V_8 = 2$ is included; opt solution is $v_8 + v_5$
Weighted Interval Scheduling Example

Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).
\[ p(j) = \text{largest } i < j \text{ s.t. job } i \text{ is compatible with } j. \]

\[
\begin{array}{cccccc}
\text{j} & \text{pj} & \text{vj} & \text{max(vj+opt[p(j)], opt[j-1])} = \text{opt[j]} \\
0 & - & - & - & 0 \\
1 & 0 & 2 & \text{max}(2+0, \ 0) = \ 2 \\
2 & 0 & 3 & \text{max}(3+0, \ 2) = \ 3 \\
3 & 0 & 1 & \text{max}(1+0, \ 3) = \ 3 \\
4 & 1 & 6 & \text{max}(6+2, \ 3) = \ 8 \\
5 & 0 & 9 & \text{max}(9+0, \ 8) = \ 9 \\
6 & 2 & 7 & \text{max}(7+3, \ 9) = \ 10 \\
7 & 3 & 2 & \text{max}(2+3, \ 10) = \ 10 \\
8 & 5 & .1 & \text{max}(0.1+9, \ 10) = \ 10 \\
\end{array}
\]

V8 = 0.1 is excluded; opt solution is v6+v2
Sidebar: why does job ordering matter?

It’s *not* for the same reason as in the greedy algorithm for unweighted interval scheduling.

Instead, it’s because it allows us to consider only a small number of subproblems (O(n)), vs the exponential number that seem to be needed if the jobs aren’t ordered (seemingly, *any* of the $2^n$ possible subsets might be relevant).

Don’t believe me? Think about the analogous problem for weighted rectangles instead of intervals… (i.e., pick max weight non-overlapping subset of a set of axis-parallel rectangles.) Same problem for squares or circles also appears difficult.
6.4 Knapsack Problem
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: maximize total value without overfilling knapsack

Ex: \{ 3, 4 \} has value 40.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
<th>$V/W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
<td>3.60</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
<td>3.66</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Greedy: repeatedly add item with maximum ratio $v_i / w_i$.
Ex: \{ 5, 2, 1 \} achieves only value = 35 \(\Rightarrow\) greedy not optimal.

[NB greedy is optimal for “fractional knapsack”: take #5 + 4/6 of #4]
Dynamic Programming: False Start

Def. \( \text{OPT}(i) = \text{max profit subset of items } 1, \ldots, i \).

- **Case 1:** \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \).

- **Case 2:** \( \text{OPT} \) selects item \( i \).
  - Accepting item \( i \) does not immediately imply that we will have to reject other items.
  - Without knowing what other items were selected before \( i \), we don't even know if we have enough room for \( i \).

**Conclusion.** Need more sub-problems!
Dynamic Programming: Adding a New Variable

Def. \( \text{OPT}(i, w) = \max \text{ profit subset of items } 1, \ldots, i \text{ with weight limit } w \).

- **Case 1:** \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using weight limit \( w \)

- **Case 2:** \( \text{OPT} \) selects item \( i \).
  - **new weight limit** = \( w - w_i \)
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using **new weight limit**

\[
\text{OPT}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{OPT}(i-1, w) & \text{if } w_i > w \\
\max \{ \text{OPT}(i-1, w), \ v_i + \text{OPT}(i-1, w-w_i) \} & \text{otherwise}
\end{cases}
\]
Knapsack Problem: Bottom-Up

OPT(i, w) = max profit from subset of items 1, ..., i with weight limit w.

**Input:** n, w₁,...,wₙ, v₁,...,vₙ

for w = 0 to W
    OPT[0, w] = 0

for i = 1 to n
    for w = 1 to W
        if (wᵢ > w)
            OPT[i, w] = OPT[i-1, w]
        else
            OPT[i, w] = max {OPT[i-1, w], vᵢ + OPT[i-1, w-wᵢ]}

return OPT[n, W]

(Correctness: prove it by induction on i & w.)
Knapsack Algorithm

<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 \( w_i > w \)

\[
\text{OPT}[i, w] = \text{OPT}[i-1, w]
\]

else

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

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

Value = 22 + 18 = 40

W = 11

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

Value = 22 + 18 = 40
Knapsack Problem: Running Time

Running time. $\Theta(nW)$.

- If $W$ is “small” this is fine, but in worst case…
- Not polynomial in input size! (“$W$” takes only $\log_2 W$ bits)
- Called "Pseudo-polynomial”
- Knapsack is NP-hard. [Chapter 8]

Knapsack approximation algorithm [Section 11.8].

Good News: There exists a polynomial time algorithm that produces a feasible solution (i.e., satisfies weight-limit constraint) that has value within 0.01% (or any other desired factor $\varepsilon$) of optimum.

Bad News: as $\varepsilon$ goes down, polynomial goes up.