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?
**Unweighted Interval Scheduling Review**

**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:** \( p(8) = 5, p(7) = 3, p(2) = 0. \)
Dynamic Programming: Binary Choice

Notation. $OPT(j)$ = 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

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

Brute force recursive algorithm.

Input: n, s_1, ..., s_n, f_1, ..., f_n, v_1, ..., v_n

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

Compute p(1), p(2), ..., p(n)

Compute-Opt(j) {
    if (j = 0)
        return 0
    else
        return max(v_j + Compute-Opt(p(j)), 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.

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

Bottom-up dynamic programming. Unwind recursion.

**Input:** $n, s_1, ..., s_n, f_1, ..., f_n, v_1, ..., v_n$

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

Compute $p(1), p(2), ..., p(n)$

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

Output OPT[$n$]

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

**Ex:** \( p(8) = 5, p(7) = 3, p(2) = 0. \)
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 → 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(\text{?}+9, 10) = \text{?} )</td>
</tr>
</tbody>
</table>

Exercise: What values of \( v_8 \) cause it to be in/excluded from \( \text{opt} \)?
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”

\[
\begin{align*}
\text{Run } & \ M-\text{Compute-Opt}(n) \\
\text{Run } & \ Find-\text{Solution}(n) \\
\text{Find-Solution}(j) & \{ \\
\text{if } & \ (j = 0) \\
& \text{output nothing} \\
\text{else if } & \ (v_j + \text{OPT}[p(j)] > \text{OPT}[j-1]) \\
& \text{print } j \\
& \text{Find-Solution}(p(j)) \\
\text{else } & \text{Find-Solution}(j-1) \\
\} \\
\end{align*}
\]

- The condition determining the max when computing OPT[ ]
- The relevant sub-problem

\# of recursive calls \(\leq n \Rightarrow O(n)\).
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 $j$.

<table>
<thead>
<tr>
<th>j</th>
<th>$p_j$</th>
<th>$v_j$</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>2</td>
<td>$\max(2+9, \ 10) = 11$</td>
</tr>
</tbody>
</table>

V8 = 2 is included; opt solution is v8+v5
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 $j$.

<table>
<thead>
<tr>
<th>j</th>
<th>$p_j$</th>
<th>$v_j$</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>$\max(2+0, 0) = 2$</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>.1</td>
<td>$\max(0.1+9, 10) = 10$</td>
</tr>
</tbody>
</table>

$V_8 = 0.1$ is excluded; opt solution is $v_6 + v_2$.
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.

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

- **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, W

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>

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

if \(w_i > w\)
    \(\text{OPT}[i, w] = \text{OPT}[i-1, w]\)
else
    \(\text{OPT}[i, w] = \max\{\text{OPT}[i-1,w], v_i+\text{OPT}[i-1,w-w_i]\}\)
Knapsack Problem: Running Time

Running time. $\Theta(n W)$.

- 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.