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.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

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 index $i < j$ such that job $i$ is compatible with $j$.

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

<table>
<thead>
<tr>
<th>j</th>
<th>p(j)</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.** \( OPT(j) = \) 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 \( \{ p(j) + 1, p(j) + 2, \ldots, j - 1 \} \)
  - 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, ..., \( j-1 \)

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

Brute force recursive algorithm.

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

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

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

Compute-Opt(\( j \)) {
    if (\( j = 0 \))
        return 0
    else
        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 ⇒ exponential time.

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

\[
\begin{array}{cccccc}
& & & 5 & 4 & 3 \\
& & 4 & 3 & 2 & 2 \\
& 3 & 2 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 \\
p(1) = 0, p(j) = j-2
\end{array}
\]
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 {
    \[\text{OPT}[0] = 0\]
    \[\text{for } j = 1 \text{ to } n\]
     \[\text{OPT}[j] = \max(v_j + \text{OPT}[p(j)], \text{OPT}[j-1])\]
}

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

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

Timing: Easy. Main loop is \( O(n) \); sorting 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 index $i < j$ such that job $i$ 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>$opt_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

\[
\begin{array}{ccc}
  j & p_j & v_j & \max(v_j + \text{opt}[p(j)], \text{opt}[j-1]) = \text{opt}[j] \\
  0 & - & - & - & 0 \\
  1 & 0 & 2 & \max(2+0, 0) = 2 \\
  2 & 0 & 3 & \max(3+0, 2) = 3 \\
  3 & 0 & 1 & \max(1+0, 3) = 3 \\
  4 & 1 & 6 & \max(6+2, 3) = 8 \\
  5 & 0 & 9 & \max(9+0, 8) = 9 \\
  6 & 2 & 7 & \max(7+3, 9) = 10 \\
  7 & 3 & 2 & \max(2+3, 10) = 10 \\
  8 & 5 & ? & \max(\?+9, 10) = ?
\end{array}
\]

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”

```plaintext
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).
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.
- 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: fill knapsack so as to maximize total value.

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

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

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

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

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

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

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

**Input:** $n, w_1, \ldots, w_n, v_1, \ldots, v_n$

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

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

return OPT[n, W]
```

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

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

if (w_i > w)
  OPT[i, w] = OPT[i-1, w]
else
  OPT[i, w] = max\{OPT[i-1,w],v_i+OPT[i-1,w-w_i]\}

<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>

W = 11
Knapsack Problem: Running Time

Running time. \( \Theta(n W) \).
- \textit{Not} polynomial in input size!
- "Pseudo-polynomial."
- Knapsack is NP-hard. [Chapter 8]

Knapsack approximation algorithm. There exists a polynomial time algorithm that produces a feasible solution that has value within 0.01% (or any other desired factor) of optimum. [Section 11.8]