Weighted Interval Scheduling

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \)

Each job has a weight (or value) \( v_i > 0 \)

\( i \) and \( j \) are compatible if \([s_i, f_i]\) and \([s_j, f_j]\) are disjoint

**Output:** a maximum-weight subset of mutually compatible jobs

---

![Diagram of weighted interval scheduling with job weights and start/finish times]
**Weighted Interval Scheduling**

**Input:** $n$ jobs, job $i$ with start time $s_i$ and finish time $f_i$

- Each job has a weight (or value) $v_i > 0$

- $i$ and $j$ are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

**Output:** a **maximum-weight** subset of mutually compatible jobs

![Diagram of intervals and weights](image)

Optimum value = 220
Q: Which job is safe to schedule?
Q: Which job is safe to schedule?

- Job with the earliest finish time?
Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights
Hard to Design a Greedy Algorithm

**Q:** Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights
- Job with the largest weight? No, when weights are equal, this is the shortest job
Hard to Design a Greedy Algorithm

**Q:** Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights
- Job with the largest weight? No, we are ignoring times
Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights.
- Job with the largest weight? No, we are ignoring times.
- Job with the largest \( \frac{\text{weight}}{\text{length}} \)?
Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights.
- Job with the largest weight? No, we are ignoring times.
- Job with the largest \( \frac{\text{weight}}{\text{length}} \) ?
  - No, when weights are equal, this is the shortest job.
Q: Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights.
- Job with the largest weight? No, we are ignoring times.
- Job with the largest \( \frac{\text{weight}}{\text{length}} \)?

No, when weights are equal, this is the shortest job.

![Diagram showing job schedule]

0 1 2 3 4 5 6 7 8 9
Designing a Dynamic Programming Algorithm

Sort jobs according to non-decreasing order of finish times.

\[ \text{opt}[i] : \text{optimal value for instance containing jobs } \{1, 2, \ldots, i\} \]

\[
\begin{array}{cccc}
1 & 180 \\
2 & 100 \\
3 & 100 \\
4 & 105 \\
5 & 150 \\
6 & 170 \\
7 & 185 \\
8 & 220 \\
9 & 220 \\
\end{array}
\]
Sort jobs according to non-decreasing order of finish times
- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$
Designing a Dynamic Programming Algorithm

- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$opt[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Sort jobs according to non-decreasing order of finish times

\( opt[i] \): optimal value for instance only containing jobs \( \{1, 2, \ldots, i\} \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( opt[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
Sort jobs according to non-decreasing order of finish times

$opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$
Sort jobs according to non-decreasing order of finish times

$opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$
**Designing a Dynamic Programming Algorithm**

- Sort jobs according to non-decreasing order of finish times
- \(opt[i]\): optimal value for instance only containing jobs \(\{1, 2, \ldots, i\}\)

<table>
<thead>
<tr>
<th></th>
<th>(opt[i])</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Designing a Dynamic Programming Algorithm

- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \ldots, i\}$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$opt[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>105</td>
</tr>
<tr>
<td>5</td>
<td>150</td>
</tr>
<tr>
<td>6</td>
<td>170</td>
</tr>
<tr>
<td>7</td>
<td>185</td>
</tr>
<tr>
<td>8</td>
<td>220</td>
</tr>
<tr>
<td>9</td>
<td>220</td>
</tr>
</tbody>
</table>
Focus on instance \( \{1, 2, 3, \cdots, i\} \),

\[\text{opt}[i]: \text{optimal value for the instance}\]
Focus on instance \( \{1, 2, 3, \cdots, i\} \),

- \( \text{opt}[i] \): optimal value for the instance

Assume we have computed \( \text{opt}[0], \text{opt}[1], \cdots, \text{opt}[i-1] \)
Focus on instance \(\{1, 2, 3, \ldots, i\}\).

- \(opt[i]\): optimal value for the instance
- assume we have computed \(opt[0], opt[1], \ldots, opt[i - 1]\)

**Q:** The value of optimal solution that does not contain \(i\)?
Focus on instance \( \{1, 2, 3, \ldots, i\} \),

- \( opt[i] \): optimal value for the instance
- assume we have computed \( opt[0], opt[1], \ldots, opt[i - 1] \)

**Q:** The value of optimal solution that does not contain \( i \)?

**A:** \( opt[i - 1] \)
Focus on instance \( \{1, 2, 3, \cdots, i\} \),

- \( opt[i] \): optimal value for the instance
- assume we have computed \( opt[0], opt[1], \cdots, opt[i-1] \)

Q: The value of optimal solution that does not contain \( i \)?

A: \( opt[i - 1] \)

Q: The value of optimal solution that contains job \( i \)?
Focus on instance \( \{1, 2, 3, \cdots, i\} \).

- \( opt[i] \): optimal value for the instance
- Assume we have computed \( opt[0], opt[1], \cdots, opt[i-1] \)

**Q:** The value of optimal solution that does not contain \( i \)?

**A:** \( opt[i-1] \)

**Q:** The value of optimal solution that contains job \( i \)?

**A:** \( v_i + opt[p_i], \quad p_i = \text{the largest } j \text{ such that } f_j \leq s_i \)
Designing a Dynamic Programming Algorithm

- Focus on instance \( \{1, 2, 3, \cdots, i\} \).
- \( \text{opt}[i] \): optimal value for the instance
- Assume we have computed \( \text{opt}[0], \text{opt}[1], \cdots, \text{opt}[i-1] \)

**Q:** The value of optimal solution that does not contain \( i \)?

**A:** \( \text{opt}[i-1] \)

**Q:** The value of optimal solution that contains job \( i \)?

**A:** \( v_i + \text{opt}[p_i] \), \[ p_i = \text{the largest } j \text{ such that } f_j \leq s_i \]
Designing a Dynamic Programming Algorithm

Q: The value of optimal solution that does not contain \( i \)?
A: \( \text{opt}[i - 1] \)

Q: The value of optimal solution that contains job \( i \)?
A: \( v_i + \text{opt}[p_i], \quad p_i = \text{the largest } j \text{ such that } f_j \leq s_i \)
Q: The value of optimal solution that does not contain $i$?

A: $opt[i - 1]$

Q: The value of optimal solution that contains job $i$?

A: $v_i + opt[p_i]$, $p_i = \text{the largest } j \text{ such that } f_j \leq s_i$

Recursion for $opt[i]$: 

$$opt[i] = \max \{opt[i - 1], v_i + opt[p_i]\}$$
Recursion for $opt[i]$:

$$opt[i] = \max \{ opt[i-1], v_i + opt[p_i] \}$$

- $opt[0] = 0$
- $opt[1] = \max \{ opt[0], 80 + opt[0] \} = 80$
- $opt[2] = $
- $opt[3] = $
- $opt[4] = $
- $opt[5] = $
Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:  

\[ opt[i] = \max \{ opt[i - 1], v_i + opt[p_i] \} \]

- $opt[0] = 0$
- $opt[1] = \max \{ opt[0], 80 + opt[0] \} = 80$
- $opt[2] = $
- $opt[3] = $
- $opt[4] = $
- $opt[5] = $
Recursion for $opt[i]$:  
\[ opt[i] = \max \{ opt[i - 1], v_i + opt[p_i] \} \]

- $opt[0] = 0$
- $opt[1] = \max \{ opt[0], 80 + opt[0] \} = 80$
- $opt[2] = \max \{ opt[1], 100 + opt[0] \}$
- $opt[3] = $
- $opt[4] = $
- $opt[5] = $
Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$: $opt[i] = \max \{ opt[i - 1], v_i + opt[p_i] \}$

- $opt[0] = 0$
- $opt[1] = \max \{ opt[0], 80 + opt[0] \} = 80$
- $opt[2] = \max \{ opt[1], 100 + opt[0] \} = 100$
- $opt[3] = \max \{ \}
- $opt[4] = \max \{ \}$
- $opt[5] = \max \{ \}$
Designing a Dynamic Programming Algorithm

Recursion for \( \text{opt}[i] \):

\[
\text{opt}[i] = \max \{ \text{opt}[i - 1], v_i + \text{opt}[p_i] \}
\]

- \( \text{opt}[0] = 0 \)
- \( \text{opt}[1] = \max \{ \text{opt}[0], 80 + \text{opt}[0] \} = 80 \)
- \( \text{opt}[2] = \max \{ \text{opt}[1], 100 + \text{opt}[0] \} = 100 \)
- \( \text{opt}[3] = \max \{ \text{opt}[2], 90 + \text{opt}[0] \} \)
- \( \text{opt}[4] = \)
- \( \text{opt}[5] = \)
Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$: 

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = $
- $opt[5] = $
Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$: 

$$opt[i] = \max \{ opt[i-1], v_i + opt[p_i] \}$$

- $opt[0] = 0$
- $opt[1] = \max \{ opt[0], 80 + opt[0] \} = 80$
- $opt[2] = \max \{ opt[1], 100 + opt[0] \} = 100$
- $opt[3] = \max \{ opt[2], 90 + opt[0] \} = 100$
- $opt[5] = \max \{ opt[4], \}$

\[\]
Designing a Dynamic Programming Algorithm

Recursion for \( opt[i] \):

\[
opt[i] = \max \{ opt[i - 1], v_i + opt[p_i] \}
\]

- \( opt[0] = 0 \)
- \( opt[1] = \max \{ opt[0], 80 + opt[0] \} = 80 \)
- \( opt[2] = \max \{ opt[1], 100 + opt[0] \} = 100 \)
- \( opt[3] = \max \{ opt[2], 90 + opt[0] \} = 100 \)
- \( opt[4] = \max \{ opt[3], 25 + opt[1] \} = 105 \)
- \( opt[5] = \)
Designing a Dynamic Programming Algorithm

Recursion for \( \text{opt}[i] \):

\[
\text{opt}[i] = \max \{ \text{opt}[i - 1], v_i + \text{opt}[p_i] \}
\]

- \( \text{opt}[0] = 0 \)
- \( \text{opt}[1] = \max\{\text{opt}[0], 80 + \text{opt}[0]\} = 80 \)
- \( \text{opt}[2] = \max\{\text{opt}[1], 100 + \text{opt}[0]\} = 100 \)
- \( \text{opt}[3] = \max\{\text{opt}[2], 90 + \text{opt}[0]\} = 100 \)
- \( \text{opt}[4] = \max\{\text{opt}[3], 25 + \text{opt}[1]\} = 105 \)
- \( \text{opt}[5] = \max\{\text{opt}[4], 50 + \text{opt}[3]\} \)
Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max \{ opt[i - 1], v_i + opt[p_i] \}$$

- $opt[0] = 0$
- $opt[1] = \max\{ opt[0], 80 + opt[0] \} = 80$
- $opt[2] = \max\{ opt[1], 100 + opt[0] \} = 100$
- $opt[3] = \max\{ opt[2], 90 + opt[0] \} = 100$
- $opt[5] = \max\{ opt[4], 50 + opt[3] \} = 150$
Designing a Dynamic Programming Algorithm

Recursion for \( opt[i] \):

\[
opt[i] = \max \{ opt[i - 1], v_i + opt[p_i] \}
\]

- \( opt[0] = 0 \), \( opt[1] = 80 \), \( opt[2] = 100 \)
Recursion for $opt[i]$: 

$$opt[i] = \max \{ opt[i - 1], v_i + opt[p_i] \}$$

- $opt[0] = 0$, $opt[1] = 80$, $opt[2] = 100$
- $opt[7] = \max \{ opt[6], 80 + opt[4] \} = 185$
- $opt[8] = \max \{ opt[7], 50 + opt[6] \} = 220$
- $opt[9] = \max \{ opt[8], 30 + opt[7] \} = 220$
Dynamic Programming

1: sort jobs by non-decreasing order of finishing times
2: compute $p_1, p_2, \cdots, p_n$
3: $opt[0] \leftarrow 0$
4: for $i \leftarrow 1$ to $n$ do
5: $opt[i] \leftarrow \max\{opt[i-1], v_i + opt[p_i]\}$
Dynamic Programming

1. sort jobs by non-decreasing order of finishing times
2. compute $p_1, p_2, \cdots, p_n$
3. $opt[0] \leftarrow 0$
4. for $i \leftarrow 1$ to $n$ do
   5. $opt[i] \leftarrow \max\{opt[i - 1], v_i + opt[p_i]\}$

- Running time sorting: $O(n \lg n)$
- Running time for computing $p$: $O(n \lg n)$ via binary search
- Running time for computing $opt[n]$: $O(n)$
How Can We Recover the Optimum Schedule?

1: sort jobs by non-decreasing order of finishing times
2: compute $p_1, p_2, \cdots, p_n$
3: $opt[0] \leftarrow 0$
4: for $i \leftarrow 1$ to $n$ do
5:   if $opt[i-1] \geq v_i + opt[p_i]$ then
6:     $opt[i] \leftarrow opt[i-1]$
7:   else
8:     $opt[i] \leftarrow v_i + opt[p_i]$
9: 10:
How Can We Recover the Optimum Schedule?

1: sort jobs by non-decreasing order of finishing times
2: compute $p_1, p_2, \cdots, p_n$
3: $opt[0] \leftarrow 0$
4: **for** $i \leftarrow 1$ to $n$ **do**
5:     **if** $opt[i - 1] \geq v_i + opt[p_i]$ **then**
6:         $opt[i] \leftarrow opt[i - 1]$
7:         $b[i] \leftarrow N$
8:     **else**
9:         $opt[i] \leftarrow v_i + opt[p_i]$
10:     $b[i] \leftarrow Y$
How Can We Recover the Optimum Schedule?

1: sort jobs by non-decreasing order of finishing times
2: compute $p_1, p_2, \cdots, p_n$
3: $opt[0] \leftarrow 0$
4: for $i \leftarrow 1$ to $n$ do
5: \hspace{1em} if $opt[i - 1] \geq v_i + opt[p_i]$ then
6: \hspace{2em} $opt[i] \leftarrow opt[i - 1]$
7: \hspace{2em} $b[i] \leftarrow N$
8: \hspace{1em} else
9: \hspace{2em} $opt[i] \leftarrow v_i + opt[p_i]$
10: \hspace{2em} $b[i] \leftarrow Y$

1: $i \leftarrow n$, $S \leftarrow \emptyset$
2: while $i \neq 0$ do
3: \hspace{1em} if $b[i] = N$ then
4: \hspace{2em} $i \leftarrow i - 1$
5: \hspace{2em} else
6: \hspace{2em} $S \leftarrow S \cup \{i\}$
7: \hspace{2em} $i \leftarrow p_i$
8: return $S$