Chapter 6
Dynamic Programming
6.1 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.
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.
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.** \( \text{OPT}(j) = \text{value of optimal solution to the problem consisting of job requests } 1, 2, \ldots, j. \)

- **Case 1:** \( \text{OPT} \) 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 \( 1, 2, \ldots, p(j) \)

- **Case 2:** \( \text{OPT} \) does not select job \( j \).
  - must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, j-1 \)

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

Brute force recursive algorithm.

Input: n, s₁,...,sₙ, f₁,...,fₙ, v₁,...,vₙ

Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.

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 fails spectacularly because of redundant sub-problems ⇒ exponential algorithms.

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

\[ p(1) = 0, \quad p(j) = j-2 \]
Weighted Interval Scheduling: Memoization

**Memoization.** Store sub-problem results in a cache; lookup as needed.

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

for \( j = 1 \) to \( n \)

\[
M[j] = \text{empty} \quad \leftarrow \text{global array}
\]

\( M[0] = 0 \)

\[
\text{M-Compute-Opt}(j) \{
\quad \text{if} \ (M[j] \text{ is empty})
\quad \quad M[j] = \max(w_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))
\quad \text{return} \ M[j]
\}
\]

Main() {
    ???
}

Weighted Interval Scheduling: Running Time

**Claim.** Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n)$ after sorting by start time.

- $M$-Compute-Opt$(j)$: each invocation takes $O(1)$ time and either
  - (i) returns an existing value $M[j]$,
  - (ii) fills in one new entry $M[j]$ and makes two recursive calls

- Progress measure $\Phi$ = # nonempty entries of $M[]$.
  - Initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases $\Phi$ by 1 ⇒ at most $2n$ recursive calls.

- Overall running time of $M$-Compute-Opt$(n)$ is $O(n)$.

**Remark.** $O(n)$ if jobs are pre-sorted by start and finish times.
**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** {
  
  \[ M[0] = 0 \]
  
  for \( j = 1 \) to \( n \)
  
  \[ M[j] = \max(v_j + M[p(j)], M[j-1]) \]
  
}

Output \( M[n] \)

**Claim:** \( M[j] \) is value of optimal solution for jobs 1..\( j \)

**Timing:** Easy. Main loop is \( O(n) \); sorting is \( O(n \log n) \)
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>( \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>1</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>0</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>
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 + M[p(j)] > M[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 \textit{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, \textit{any} of the \(2^n\) possible subsets might be relevant)

Don’t believe me? Think about the analogous problem for weighted \textit{rectangles} instead of intervals… (I.e., pick max weight non-overlapping subset of a set of axis-parallel rectangles.) Same problem for 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: 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>
</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>

\( W = 11 \)

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

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

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

- **Case 2:** $OPT$ 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. \( \text{OPT}(i, w) = \text{max profit subset of items } 1, \ldots, i \text{ with weight limit } w. \)

- **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 this 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

Knapsack. Fill up an n-by-W array.

**Input:** n, \(w_1,...,w_N, v_1,...,v_N\)

for \(w = 0\) to W  
   \(M[0, w] = 0\)

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

**return** \(M[n, W]\)
## Knapsack Algorithm

### Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\phi)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{1}</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>19</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>{1, 2, 3, 4}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>22</td>
<td>24</td>
<td>28</td>
<td>29</td>
<td>29</td>
<td>40</td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>22</td>
<td>28</td>
<td>29</td>
<td>34</td>
<td>34</td>
<td>40</td>
</tr>
</tbody>
</table>

### OPT

\[\text{OPT: } \{4, 3\}\]

\[\text{value} = 22 + 18 = 40\]

### Code

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

### W = 11

<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>
Knapsack Problem: Running Time

Running time. $\Omega(nW)$.
- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]