Dynamic Programming:
Weighted Interval Scheduling
KT (Ch.6 Intro, 6.1-6.2)
Section 15.3 (CLRS)
Algorithmic Paradigms

Greed. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
Dynamic Programming History

Belman. Pioneered the systematic study of dynamic programming in the 1950s.

Etymology.
• Dynamic programming = planning over time.
• Secretary of Defense was hostile to mathematical research.
• Bellman sought an impressive name to avoid confrontation.
  – "it's impossible to use dynamic in a pejorative sense"
  – "something not even a Congressman could object to"

Dynamic Programming Applications

Areas.
• Bioinformatics.
• Control theory.
• Information theory.
• Operations research.
• Computer science: theory, graphics, AI, systems, ....

Some famous dynamic programming algorithms.
• Viterbi for hidden Markov models.
• Unix diff for comparing two files.
• Smith-Waterman for sequence alignment.
• Bellman-Ford for shortest path routing in networks.
• Cocke-Kasami-Younger for parsing context free grammars.
DP relies are two important structural qualities:

- **Optimal substructure**: This property (sometimes called the principle of optimality) states that for the global problem to be solved optimally, each subproblem should be solved optimally. While this might seem intuitively obvious, not all optimization problems satisfy this property.
  - Example: it may be advantageous to solve one subproblem suboptimally in order to conserve resources so that another, more critical, subproblem can be solved optimally.

- **Overlapping Subproblems**: These subproblems overlap each other in such a way that the number of distinct subproblems is reasonably small, ideally polynomial in the input size.
Generating Subproblems

How to generate the solutions to these subproblems?

• **Top-Down**: A solution to a DP problem is expressed recursively. This approach applies recursion directly to solve the problem. However, due to the overlapping nature of the subproblems, the same recursive call is often made many times. An approach, called memoization, records the results of recursive calls, so that subsequent calls to a previously solved subproblem are handled by table look-up.

• **Bottom-up**: Although the problem is formulated recursively, the solution is built iteratively by combining the solutions to small subproblems to obtain the solution to larger subproblems. The results are stored in a table.
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 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.

![Diagram of weighted interval scheduling](image)
Weighted vs. Unweighted

optimal unweighted

\[ \text{opt} = 3 \]

(a)

optimal weighted

\[ \text{opt} = 14 \]

(b)
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: \( p(8) = 5, p(7) = 3, p(2) = 0 \).
### Weighted Input and P-Values

<table>
<thead>
<tr>
<th>$j$</th>
<th>intervals and values</th>
<th>$p(j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>
Weighted Interval Scheduling

- A: Weight 4, Duration 5
- B: Weight 5, Duration 2
- C: Weight 2, Duration 2
- D: Weight 1, Duration 1
- E: Weight 8, Duration 8
- F: Weight 4, Duration 4
- G: Weight 8, Duration 3
- H: Weight 3, Duration 8

Time: 0 1 2 3 4 5 6 7 8 9 10 11
Weighted Interval Scheduling
Weighted Interval Scheduling
Weighted Interval Scheduling

The diagram illustrates a scheduling problem with tasks labeled A to H, each having a specific duration and weight. The tasks are scheduled on a timeline from 0 to 11 units of time, with each task's start and end times indicated by bars. The weights are assigned to the bars, with weights of 1, 2, and 8 for tasks C, D, and E respectively. The diagram also includes a matrix labeled M, which shows a possible allocation of tasks to time slots, along with weights for each task.
Weighted Interval Scheduling
Weighted Interval Scheduling

A: 4
B: 5
C: 2
D: 1
E: 8
F: 4
G: 8
H: 3

M: 11 11 8 8 3
Weighted Interval Scheduling
Weighted Interval Scheduling

[Diagram showing intervals and weights for tasks A to H over time from 0 to 11.]

M

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Weighted Interval Scheduling

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M: 16 11 11 11 8 8 3
Weighted Interval Scheduling

Time

0 1 2 3 4 5 6 7 8 9 10 11

A B C D E F G H

M

16 11 11 11 11 8 8 3

+4
Weighted Interval Scheduling

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

M:

<table>
<thead>
<tr>
<th></th>
<th>16</th>
<th>16</th>
<th>11</th>
<th>11</th>
<th>11</th>
<th>8</th>
<th>8</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing intervals for tasks A to H with their respective weights.
Weighted Interval Scheduling

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Weights: M

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>8</td>
<td>8</td>
<td>3</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: \( \text{OPT} \) 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: \( \text{OPT} \) does not select job \( j \).
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., \( j-1 \)

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

**DP Selection Principle:**
When given a set of feasible options to choose from, try them all and take the best.
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\((j)\) {
    \( \text{if } (j = 0) \)
    \( \quad \text{return } 0 \)
    \( \text{else} \)
    \( \quad \text{return } \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \)
}
Weighted Interval Scheduling: Brute Force

**Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

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

$p(1) = 0, p(j) = j-2$
Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem 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 \( p(1), p(2), \ldots, p(n) \)

```plaintext
for j = 1 to n
    M[j] = empty
M[j] = 0

M-Compute-Opt(j) {
    if (M[j] is empty)
        M[j] = max(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
    return M[j]
}
```
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 $\Phi = \#\text{ nonempty entries of } M[\cdot]$.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases $\Phi$ by 1 $\Rightarrow$ 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.
Automated Memoization

Automated memoization. Many functional programming languages (e.g., Lisp) have built-in support for memoization.

Q. Why not in imperative languages (e.g., Java)?

```
(defun F (n)
 (if
   (<= n 1)
   n
   (+ (F (- n 1)) (F (- n 2)))))
```

Lisp (efficient)

```
static int F(int n) {
    if (n <= 1) return n;
    else return F(n-1) + F(n-2);
}
```

Java (exponential)

F(40)

F(39) ——— F(38)

F(38) ——— F(37)

F(37) ——— F(36)

F(36) ——— F(35)

F(35) ——— F(34)
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.

```
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 \( \leq n \Rightarrow O(n) \).
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 \( n \)
        \( M[j] = \max(v_j + M[p(j)], M[j-1]) \)
}

---

```java
bottom-up-WIS() {
    M[0] = 0
    for (i = 1 to n) {
        if (M[j-1] > v[j] + M[p[j]] ) {
            M[j] = M[j-1]; pred[j] = j-1;
        } else {
        }
    }
}
```
Example of Finding the Solution

Example of iterative construction and predecessor values. The final optimal value is 14. By following the predecessor pointers back from M[6] we see that the requests that are in the schedule are 5 and 2.

Computing Weighted Interval Scheduling Schedule

```plaintext
get-schedule() {
    j = n
    sched = (empty list)
    while (j > 0) {
        if (pred[j] == p[j]) {
            prepend j to the front of sched
        }
        j = pred[j]
    }
}
```
Next Time

• More Dynamic Programming Algorithms!