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

Bellman. 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.
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.

![Diagram of weighted interval scheduling]

- Jobs: a, b, c, d, e, f, g, h
- Time axis: 0 to 11

Weighted Interval Scheduling
Unweighted Interval Scheduling Review

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

![Diagram of weighted interval scheduling]
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 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 $p(1)$, $p(2)$, ..., $p(n)$

Compute-Opt($j$) {
    if ($j = 0$)
        return 0
    else
        return max($v_j + \text{Compute-Opt}(p(j))$, $\text{Compute-Opt}(j-1)$)
}
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 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 \( p(1), p(2), \ldots, p(n) \)

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

\[
\begin{align*}
M[j] &= \text{empty} \quad \text{— global array} \\
M[j] &= 0
\end{align*}
\]

\( M\text{-Compute-Opt}(j) \) {

if (\( M[j] \) is empty)

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

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.
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 $\leq n \Rightarrow O(n)$. 
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 {

\[
M[0] = 0
\]

\[
\text{for } j = 1 \text{ to } n
\]

\[
M[j] = \max(v_j + M[p(j)], M[j-1])
\]

}

Weighted Interval Scheduling: Bottom-Up
6.3 Segmented Least Squares
Segmented Least Squares

Least squares.

- Foundational problem in statistic and numerical analysis.
- Given n points in the plane: \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\).
- Find a line \(y = ax + b\) that minimizes the sum of the squared error:

\[
SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2
\]

Solution. Calculus \(\Rightarrow\) min error is achieved when

\[
a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}
\]
Segmented Least Squares

Segmented least squares.

Points lie roughly on a sequence of several line segments.

Given $n$ points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

Q. What's a reasonable choice for $f(x)$ to balance accuracy and parsimony?

<table>
<thead>
<tr>
<th>goodness of fit</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of lines</td>
</tr>
</tbody>
</table>

$y$

$x$
Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given $n$ points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes:
  - the sum of the sums of the squared errors $E$ in each segment
  - the number of lines $L$
- Tradeoff function: $E + cL$, for some constant $c > 0$. 
Dynamic Programming: Multiway Choice

Notation.
- $OPT(j)$ = minimum cost for points $p_1, p_{i+1}, \ldots, p_j$.
- $e(i, j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

To compute $OPT(j)$:
- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some $i$.
- Cost = $e(i, j) + c + OPT(i-1)$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{ e(i, j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$
Segmented Least Squares: Algorithm

**INPUT**: n, p_1, ..., p_N, c

```plaintext
Segmented-Least-Squares() {
  M[0] = 0
  for j = 1 to n
    for i = 1 to j
      compute the least square error e_ij for
      the segment p_i, ..., p_j

    for j = 1 to n
      M[j] = min_{1 ≤ i ≤ j} (e_ij + c + M[i-1])

  return M[n]
}
```

**Running time.** \(O(n^3)\). \(\checkmark\) can be improved to \(O(n^2)\) by pre-computing various statistics

\(\checkmark\) Bottleneck = computing e(i, j) for \(O(n^2)\) pairs, \(O(n)\) per pair using previous formula.