Algorithmic Paradigms

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

**Divide-and-conquer.** Break up a problem into two or more 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

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:** \( p(8) = 5 \), \( p(7) = 3 \), \( p(2) = 0 \).
**Dynamic Programming: Binary Choice**

**Notation.** \( OPT(j) = \text{value of optimal solution to the problem consisting of job requests 1, 2, ..., j.} \)

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

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left\{ v_j + OPT(p(j)), \ OPT(j-1) \right\} & \text{otherwise}
\end{cases}
\]
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 \( p(1), p(2), \ldots, p(n) \)

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

for $j = 1$ to $n$
   $M[j] = \text{empty} \leftarrow$ global array

$M[j] = 0$

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

- $\text{M-Compute-Opt}(j)$: 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 $\Rightarrow$ at most $2n$ recursive calls.

- Overall running time of $\text{M-Compute-Opt}(n)$ is $O(n)$. ▪
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)?

Lisp (efficient)

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

Java (exponential)

```
static int F(int n) {
  if (n <= 1) return n;
  else return F(n-1) + F(n-2);
}
```
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 ≤ n ⇒ O(n).
Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

\begin{itemize}
\item \textbf{Input:} \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)
\item \textbf{Sort} jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).
\item \textbf{Compute} \( p(1), p(2), \ldots, p(n) \)
\end{itemize}

Iterative-Compute-Opt \{ 
\begin{align*}
M[0] & = 0 \\
\text{for } j = 1 \text{ to } n \\
M[j] & = \max(v_j + M[p(j)], M[j-1])
\end{align*}
\}


6.3 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?

![Graph showing goodness of fit vs. number of lines](image)

- ↑ goodness of fit
- ↑ number of lines
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 (penalty) function: \( E + cL \), for some constant \( c > 0 \).
Dynamic Programming: Multiway Choice

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

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

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

**INPUT:** n, p₁,…,pₙ, c

Segmented-Least-Squares() {
    M[0] = 0
    for j = 1 to n
        for i = 1 to j
            compute the least square error eᵢⱼ for the segment pᵢ,…, pⱼ

    for j = 1 to n
        M[j] = min₁≤ᵢ≤ⱼ (eᵢⱼ + c + M[i-1])

    return M[n]
}

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

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