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.

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 job $j$.

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

Notation. $\text{OPT}(j)$ = 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 $1, 2, ..., j-1$

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

Weighted Interval Scheduling: Brute Force

Brute force algorithm.

Input: $n, s_1, s_2, ..., s_n, f_1, f_2, ..., f_n, v_1, v_2, ..., v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq ... \leq f_n$.

Compute $p(1), p(2), ..., p(n)$

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

Weighted Interval Scheduling: Memoization

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

Input: $n, s_1, s_2, ..., s_n, f_1, f_2, ..., f_n, v_1, v_2, ..., v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq ... \leq f_n$.

Compute $p(1), p(2), ..., p(n)$

for $j = 1$ to $n$
  $M[j] = \text{empty} --- \text{global array}$
  $M[0] = 0$

$\text{M-Compute-Opt}(j) \{$
  if ($M[j]$ is empty)
    $M[j] = \max(v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))$
  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 \(p()\): \(O(n)\) 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 \(\Rightarrow\) 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: 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 \(n\)
  \(M[j] = \max(v_j + M[p(j)], M[j-1])\)
}

---

**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)**
}

**Remark.** \(\#\) of recursive calls \(\leq n \Rightarrow O(n)\).
### 6.4 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.

**Greedy:** repeatedly add item with maximum ratio \(v_i / w_i\).

**Ex:** \{5, 2, 1\} achieves only value = 35 \(\Rightarrow\) greedy not optimal.

<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**

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

- Case 1: OPT does not select item *i*.
  - OPT selects best of \(\{1, 2, \ldots, i-1\}\) using weight limit *w*.

- Case 2: OPT selects item *i*.
  - new weight limit = \(w - w_i\)
  - OPT selects best of \(\{1, 2, \ldots, i-1\}\) using this new weight limit

\[
OPT(i, w) =\begin{cases} 
0 & \text{if } i = 0 \\
OPT(i-1, w) & \text{if } w_i > w \\
\max \{ OPT(i-1, w), v_i + OPT(i-1, w-w_i) \} & \text{otherwise}
\end{cases}
\]
Knapsack Problem: Bottom-Up

Input: \( n, w_1, \ldots, w_n, v_1, \ldots, v_n \)

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

Running time. \( \Theta(n W) \).
- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is \( \text{NP-complete}. \) [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]

String Similarity

How similar are two strings?
- occurrence
- occurrence

5 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps
Edit Distance

Applications:
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

- Gap penalty δ; mismatch penalty α_{mp}
- Cost = sum of gap and mismatch penalties.

Cost = \sum gap + \sum mismatch.

\[ OPT(i, j) = \min \begin{align*} 
& j \delta \\
& \alpha_{c_r, c_t} + OPT(i-1, j-1) \\
& \delta + OPT(i-1, j) \\
& i \delta 
\end{align*} \quad \text{if } i = 0 \\
\quad \text{min cost of aligning} \ x_i \ y_j
\]

Ex: CTACCG vs. TACATG.
Sol: M = x_1\ldots x_m, y_1\ldots y_n

Sequence Alignment

Goal: Given two strings X = x_1, x_2, \ldots, x_m and Y = y_1, y_2, \ldots, y_n find alignment of minimum cost.

Def. An alignment M is a set of ordered pairs x_i-y_j such that each item occurs in at most one pair and no crossings.

Def. The pair x_i-y_j and x_i'-y_j' cross if i < i', but j > j'.

\[ \text{cost}(M) = \sum \alpha_{c_r, c_t} + \sum \delta + \sum \delta \]

Ex: CTACCG vs. TACATG.
Sol: M = x_1\ldots x_m, y_1\ldots y_n

Sequence Alignment: Algorithm

\[
\text{Sequence-alignment}(m, n, \Delta, \alpha) \{ 
\text{for } i = 0 \text{ to } m 
\quad M[0, i] = \Delta i 
\text{for } j = 0 \text{ to } n 
\quad M[j, 0] = \Delta j 
\text{for } i = 1 \text{ to } m 
\quad \text{for } j = 1 \text{ to } n 
\quad \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \Delta \D