Dynamic Programming Recipe

- Devise recursive form for solution
- Observe that recursive implementation involves redundant computation. (Often exponential time)
- Design iterative algorithm that solves all subproblems without redundancy.

Comparison

<table>
<thead>
<tr>
<th>Formulate problem</th>
<th>Greedy</th>
<th>Divide and Conquer</th>
<th>Dynamic Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design algorithm</td>
<td>easy</td>
<td>hard</td>
<td>hard</td>
</tr>
<tr>
<td>Prove correctness</td>
<td>hard</td>
<td>easy</td>
<td>easy</td>
</tr>
<tr>
<td>Analyze running time</td>
<td>easy</td>
<td>hard</td>
<td>easy</td>
</tr>
</tbody>
</table>

Weighted Interval Scheduling

- Television scheduling problem: Given $n$ shows with start time $s_i$ and finish time $f_i$, watch as many shows as possible, with no overlap.
- A Twist: Each show has a value $v_i$ and want a set of shows $S$, with no overlap and maximum value $\sum_{i \in S} v_i$.
- Greedy? Example on board.
- Notation:
  - $s_j, f_j$: start and finish time of show (job) $j$
  - $v_j$: value of show $j$
  - Assume shows sorted by finishing time $f_1 \leq f_2 \leq \ldots \leq f_n$
  - Shows $i$ and $j$ are compatible if they don’t overlap

Weighted Interval Scheduling: Recursive Algorithm

- Observation: Let $O$ be the optimal solution. Either $n \in O$ or $n \notin O$. In either case, we can reduce the problem to a smaller instance of the same problem.
- Recursive algorithm to find value of optimal subset of first $j$ shows

\[
\text{Compute-Value}(j) \\
\text{Base case: if } j = 0 \text{ return } 0 \\
\text{Case 1: } j \in O \\
\text{Let } i < j \text{ be highest-numbered show compatible with } j \\
\text{val1} = v_j + \text{Compute-Value}(i) \\
\text{Case 2: } j \notin O \\
\text{val2} = \text{Compute-Value}(j - 1) \\
\text{return } \max(\text{val1}, \text{val2})
\]
Extracting the Solution

- Finding the solution itself is a simple modification of the same algorithm

Compute-Solution($j$)
- **Base case**: if $j = 0$ return $\emptyset$
- **Case 1**: $j \in O$
  Let $i < j$ be highest-numbered show compatible with $j$
  $O_1 = \{j\} \cup $ Compute-Solution($i$)
- **Case 2**: $j \notin O$
  $O_2 = $ Compute-Solution($j - 1$)
  return the solution $O_1$ or $O_2$ that has higher value
- **Advice**: first develop algorithm to compute optimal value; usually easy to modify it to compute the actual solution

Recurrence

- A recurrence is a mathematical way of expressing the value of an optimal solution.
- **Definitions**
  - $OPT(j) =$ value of optimal solution on first $j$ shows
  - $p_j$: highest-numbered show that is compatible with $j$
  - **Recurrence**
    
    $OPT(0) = 0$
    $OPT(j) = \max\{v_j + OPT(p_j), OPT(j - 1)\}$

Recursive Algorithm vs. Recurrence

- Compute-Value($j$)
  - If $j = 0$ return 0
  - $val1 = v_j + $ Compute-Value($p_j$)
  - $val2 = $ Compute-Value($j - 1$)
  - return $\max(val1, val2)$
- **Recurrence**
  
  $OPT(j) = \max\{v_j + OPT(p_j), OPT(j - 1)\}$

Running Time?

- **Board work**: running time of recursive solution
- **Recap**
  - **Recursion tree**
  - $\approx 2^n$ subproblems $\Rightarrow$ exponential time
  - Only $n$ unique subproblems. Save work by ordering computation to solve each problem once.

Iterative “Bottom-Up” Algorithm

- WeighedIS
  - Initialize array $M$ of size $n$ to hold optimal values
  - $M[0] = 0$ → Value of empty set
  - for $j = 1$ to $n$ do
    - $M[j] = \max(v_j + M[p_j], M[j - 1])$
  - end for
- **Example execution**
- **Comment**: usually direct “wrapping” of recurrence in appropriate for loop. Pay attention to dependence on previously-computed entries of $M$ to know which direction to iterate.

Review

- Recursive algorithm $\rightarrow$ recurrence $\rightarrow$ iterative algorithm
- Three ways of expressing value of optimal solution for smaller problems
  - Compute-Value($j$). Recursive algorithm—arguments identify subproblems.
  - $OPT(j)$. Mathematical expression. Write a recurrence for this that matches recursive algorithm.
Dynamic Programming Recipe

- Devise recursive form for solution. **Flavor:** make “first choice”, then recursively solve a smaller instance of same problem.
- Observe that recursive implementation involves redundant computation. (Often exponential time)
- Design iterative algorithm that solves all subproblems without redundancy.

Dynamic Programming

- First example: Weighted Interval Scheduling
  - Binary first choice: $j \in O$ or $j \notin O$?
- Next time: rod-cutting or segmented least squares
  - First choice has $n$ options