Dynamic Programming

“divide problem into small number of sub-problems and memoize solution to avoid redundant computation”

Example: Weighted Interval Scheduling

input:
- $n$ jobs $J = \{1, \ldots, n\}$
- $s_i =$ start time of job $i$
- $f_i =$ finish time of job $i$
- $v_i =$ value of job $i$

output: Schedule $S \subseteq J$ of compatible jobs with maximum total value.

Recall Greedy: “earliest finish time”

1
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>--------</td>
</tr>
<tr>
<td>--------</td>
</tr>
</tbody>
</table>

Idea: job $i$ is either in $\text{OPT}(J)$ or not.

1. let $J'$ = jobs compatible with $i$ in $J$.
2. let $V =$ value of $\text{OPT}$ if “$i \in \text{OPT}(j)$”.

1
\[
= v_i + \text{OPT}(J')
\]

3. Let \( V' = \text{value of OPT if } i \notin \text{OPT}(j) \)

\[
= \text{OPT}(J \setminus \{i\}).
\]

4. Return \( \text{OPT}(J) = \max(V, V') \).

Note: subproblems: schedule \( J' \) and \( J \setminus \{i\} \).

Recurrence: \( T(n) = 2T(n - 1) + 1 \)

Challenge 1: redundant computation

Example:

\[
\begin{array}{ccc}
1 & 3 \\
\text{|---| |---|} \\
\text{|---------|}
\end{array}
\]

\[
\begin{array}{ccc}
2 \\
\text{|--------|}
\end{array}
\]

Note: \( \text{OPT}\{3\} \) called twice!

Solution: memoize

“when computing the value of a subproblem save the answer to avoid computing it again”

Result: runtime = \# of subproblems \times cost to combine.

\[
T(n) = O(2^n)
\]
**Challenge 2:** could have too many subproblems.
(could be exponential!)

**Solution:** require “succinct description” of subproblems.

**Idea:** for interval scheduling, process jobs in order of start time so subproblems suffixes of order.

- sort jobs by increasing start time, \( s_1 \leq s_2 \leq \cdots \leq s_n \).
- let next\([i]\) denote job with earliest start time after \( i \) finishes. (if none, set next\([i]\) = \( n+1 \))
- subproblems when processing job 1:
  - \( J' = \{\text{next}[i], \text{next}[i]+1, \ldots, n\} \)
  - \( J \setminus \{i\} = \{2, 3, \ldots, n\} \)
- suffix \{j,\ldots,n\} is succinctly described by “j”.

**Algorithm:** Weighted Interval Scheduling:
1. sort jobs by increasing start time.
2. initialize array next\([i]\).
3. initialize memo\([i]\) = \( \emptyset \) for all \( i \).
4. initialize memo\([n+1]\] = 0.
5. compute OPT(1).

**Subroutine:** OPT\((i)\)
1. if memo\([i]\) \( \neq \emptyset \), return memo\([i]\).
2. memo\([i]\) \( \leftarrow \max(v_i + \text{OPT}(\text{next}[i]), \text{OPT}(i + 1)) \).
3. return memo\((i)\).

**Correctness**

“OPT\((i)\)” is correct (by induction on \( i \))

**Runtime Analysis**

- \( n \) subproblems
- constant time to combine
- initialization: sorting & precomputing next array

Runtime: \( O(n) + \text{initialization} = O(n \log n) \)
Key Ideas of Dynamic Programming

Subproblems must be:

1. succinct
   (only a polynomial number of them)
2. efficiently combinable.
3. partially ordered (avoid infinite loops),
   e.g.,
   • process elements “once and for all”
   • “measure of progress/size”.

Comparison to Divide and Conquer

<table>
<thead>
<tr>
<th>Div&amp;Conq</th>
<th>Dynamic Prog.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree structure</td>
<td>DAG structure</td>
</tr>
</tbody>
</table>

Iterative DPs

“fill in memoization table from bottom to top”

Algorithm: iterative weighted interval scheduling

1. memo[n + 1] = 0.
2. for i = n down to 1.
   
   memo[i] = max(v_i + memo[next(i)], memo[i + 1]).

Finding Optimal Schedule

“traverse memoization table to find schedule”

Algorithm: schedule

\[ i = 1 \]

while \( i < n \)

\[ \text{if } \text{memo}[i + 1] < v_i + \text{memo}[\text{next}(i)] \]

\[ \text{schedule } i; i ← \text{next}(i). \]

\[ \text{else} \]

\[ i ← i + 1. \]

endwhile

Suggested Approach

I. identify subproblem in english

\[ \text{OPT}(i) = \text{“optimal schedule of } \{i, \ldots, n\} \text{ (sorted by increasing start time)“} \]

II. specify subproblem recurrence

\[ \text{OPT}(i) = \max(\text{OPT}(i + 1), v_i + \text{OPT(\text{next}(i)))} \]

III. identify base case

\[ \text{OPT}(n + 1) = 0 \]

IV. write iterative DP. 