Algorithm Design Techniques

- Greedy
- Divide and Conquer
- Dynamic Programming
- Network Flows

Dynamic Programming Schedule

- Today: Intro + Scheduling and Packing
- Thursday: Sequence Alignment + Biology problems
- 10/25: Graph problems
- 10/27: AI + Statistics problems

Divide and Conquer Recipe

- Devise recursive form for solution
- Implement recursion

**Example** Compute sum of leaf weights for each internal node in \( k \)-ary tree. (From practice exam)

- Recursive form \( w(v) = \sum_{u \text{ child of } v} w(u) \).

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.

Example (From HW1)

**Problem.** Given array \( A \) of length \( n \), compute matrix \( B \) with \( B[i, j] = A[i] + \ldots + A[j] \) for \( i < j \).

```plaintext
for i = 1, 2, \ldots, n do
    for j = i + 1, \ldots, n do
    end for
end for
```

Running time: \( \Theta(n^2) \).
Example (From HW1)


$$B[i, j] = \begin{cases} B[i, j - 1] + A[j] & \text{if } j > i \\ 0 & \text{if } j \leq i \end{cases}$$

for $i = 1, 2, \ldots, n$ do
  $B[i, i] = 0$
  for $j = i + 1, \ldots, n$ do
  end for
end for

Running time: $O(n^2)$

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?**

Recursive Form

Order shows by finish time $f_1 \leq f_2, \ldots, f_n$.
Compute $p(i) = \max\{j : f_j \leq s_i\}$ for all $i$.

- Suppose $O$ is an optimal solution ($O = \text{OPT}(n)$).
  - If $n \in O$, then $O = \text{OPT}(p(n)) \cup \{n\}$.
  - If $n \notin O$ then $O = \text{OPT}(n - 1)$.
- Define $V = \text{VAL}(n)$ to be the optimal value.
  - If $n \in O$, then $V = \text{VAL}(p(n)) + v_n$.
  - If $n \notin O$, then $V = \text{VAL}(n - 1)$.

**Recurrence** $\text{VAL}(n) = \max(\text{VAL}(p(n)) + v_n, \text{VAL}(n - 1))$.

Unrolling recurrence?

$\text{Val}(j)$:
If $j = 0$ return 0.
Return $\max\{\text{Val}(p(j)) + v_j, \text{Val}(j - 1)\}$.

$\text{Val}(n)$ can require $2^n$ calls in the worst case.
Only $n + 1$ values to compute $\Rightarrow$ redundancy!

Memoized approach

**Idea.** Save the output of recursive calls when you do them.

Array $M[0..n] = \text{null}$.

$M\text{-Val}(j)$:
If $j = 0$ return 0.
If $j \neq \text{null}$, return $M[j]$.
$M[j] \leftarrow \max(v_j + M\text{-Val}(p(j)), M\text{-Val}(j - 1))$.
Return $M[j]$.

Running time: $O(n)$.

Iterative approach

**Idea.** Work from $0 \rightarrow n$ computing array entries only once.

Array $M[0..n] = \text{null}$.

$I\text{-All-Val}(n)$:
$M[0] = 0$.
for $j = 1, \ldots, n$ do
  $M[j] \leftarrow \max(v_j + M[p(j)], M[j - 1])$.
end for

Running time: $O(n)$.
Finding the optimum set

- Suppose \( O \) is an optimal solution \( (O = OPT(n)) \).
  - If \( n \in O \), then \( O = OPT(p(n)) \cup \{n\} \).
  - If \( n \notin O \) then \( O = OPT(n - 1) \).

**Weighted-IS(n)**

Sort by finish time \( f_j \), compute \( p(j) \).

1. **MΩI-All-Val(n)**
   - # Compute \( M \) array
     - \( S \leftarrow \{ \}, j = n \)
     - while \( j \neq 0 \) do
       - if \( M[p(j)] + v_j \geq M[j - 1] \), \( S \leftarrow S \cup \{j\}, j \leftarrow p(j) \).
       - else \( j \leftarrow j - 1 \).
     - end while
     - Return \( S \).

Subset Sum

**Problem.** Given \( n \) jobs where job \( i \) requires \( w_i \) minutes of time and a budget \( W \).

- Find subset \( S \) that maximizes \( \sum_{i \in S} w_i \) and has \( \sum_{i \in S} w_i \leq W \).
- Greedy? Divide and Conquer?

**Solution Recurrence**

Let \( O \) be the optimal solution.

- If \( n \notin O \) then \( O \) is optimal solution using \( \{1, \ldots, n - 1\} \).
- If \( n \in O \) then \( O \) is optimal solution using \( \{1, \ldots, n - 1\} \) and budget \( W - w_n \).

\[
VAL(j, W) = \max\{VAL(j - 1, W), w_j + VAL(j - 1, W - w_j)\}
\]

Unless \( W < w_j \), then \( VAL(j, W) = VAL(j - 1, W) \).

Need to track both jobs and remaining budget.

**Example**

\( w_1 = 2, w_2 = 2, w_3 = 1, W = 4 \)

\[
M[j, w] \leftarrow \max\{M[j - 1, w], w_j + M[j - 1, w - w_j]\}
\]

\[
\begin{array}{cccccc}
  w & 0 & 1 & 2 & 3 & 4 \\
  j = 0 & 0 & 0 & 0 & 0 & 0 \\
  j = 1 & 0 & 0 & 2 & 2 & 2 \\
  j = 2 & 0 & 2 & 2 & 4 & 4 \\
  j = 3 & 3 & 1 & 2 & 3 & 4 \\
\end{array}
\]
Another Example

\[ w_1 = 2, w_2 = 2, w_3 = 3, W = 4 \]

\[ M[j, w] \leftarrow \max\{ M[j - 1, w], w_j + M[j - 1, w - w_j] \} \]

<table>
<thead>
<tr>
<th></th>
<th>( w = 0 )</th>
<th>( w = 1 )</th>
<th>( w = 2 )</th>
<th>( w = 3 )</th>
<th>( w = 4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j = 3 )</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>( j = 2 )</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>( j = 1 )</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>( j = 0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Finding Optimal Solution

- Similar to weighted interval scheduling.
- Walk table from \( M[n, W] \), following the entry you are based on.

\[ w_1 = 2, w_2 = 2, w_3 = 3, W = 4 \]

<table>
<thead>
<tr>
<th></th>
<th>( w = 0 )</th>
<th>( w = 1 )</th>
<th>( w = 2 )</th>
<th>( w = 3 )</th>
<th>( w = 4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j = 3 )</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>( j = 2 )</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>( j = 1 )</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>( j = 0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Running Time

- Table has \( O(nW) \) entries, each entry requires \( O(1) \) computation.
- Finding optimal solution takes \( O(n) \) time with table.
- Not polynomial in size of the input, since \( W \) can be specified in \( \log_2 W \) bits. \( Pseudo-polynomial \ time \)

Dynamic Programming Takeways

- Identify recurrence for solution.
- Often easier to compute optimal value.
- Build DP table.
- Extract optimal solution from table.
- Analyze running time.