Algorithm Design Techniques

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

Learning Goals

<table>
<thead>
<tr>
<th></th>
<th>Greedy</th>
<th>Divide and Conquer</th>
<th>Dynamic Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formulate problem</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Design algorithm</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Prove correctness</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analyze running time</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specific algorithms</td>
<td>✓</td>
<td></td>
<td>Bellman-Ford shortest paths</td>
</tr>
</tbody>
</table>

Sometimes Recursion is Easy... But Inefficient

Fibonacci sequence: $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$

Compute by straightforward recursion:

$$
\begin{array}{cccc}
F(5) & F(4) & F(3) \\
F(2) & F(1) & F(0) \\
F(1) & F(0) & \\
\end{array}
$$

Clicker Question

Consider the following function to compute Fibonacci numbers:

```python
fib(n):
    if n < 2 return n;
    return fib(n-1) + fib(n-2);
```

The complexity of `fib(n)` is

A. $\Theta(n^{\log_2 3})$
B. $\Theta(F(n))$
C. $\Theta(2^n)$
D. $\Theta(n!)$

Weighted Interval Scheduling

- TV 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 $v_i$. We want a set of shows $S$, with no overlap and maximum value $\sum_{i \in S} v_i$.
- Greedy? No longer optimal.
- Problem formulation
  - Show (job) $j$ has value $v_j$, start time $s_j$, finish time $f_j$
  - 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
  - Goal: subset of non-overlapping jobs with maximum value
Dynamic Programming Recipe

- **Step 1**: Devise simple recursive algorithm
  - Make one decision by trying all possibilities...
  - ... recursively solve subproblem to evaluate value of each
  - Problem: solve redundant subproblems, often exponential time
- **Step 2**: Write recurrence for optimal value
- **Step 3**: Design iterative algorithm

Clicker Question

Compute-Value(j)

- if j = 0 return 0
- Let i < j be highest-numbered show compatible with j
  - val1 = v_j + Compute-Value(i)
  - val2 = Compute-Value(j−1)
- return max(val1, val2)

The running time of this recursive solution is

A. O(n log n)
B. O(n^2)
C. O(1.618^n)
D. O(2^n)

Step 2: Recurrence

- Recurrence: directly expresses solution (optimal value) in terms of solutions for subproblems (recursive structure)
- Let OPT(j) be the value of optimal solution on first j shows
- Let p_j be the highest-numbered show that is compatible with j

Recurrence:

\[
\begin{align*}
OPT(0) &= 0 \\
OPT(j) &= \max\{v_j + OPT(p_j), OPT(j−1)\} \quad (\text{Case 1}) \quad (\text{Case 2})
\end{align*}
\]

Running Time?

- Recursion tree
  - \(\approx 2^n\) subproblems \(\Rightarrow\) exponential time
  - Only \(n\) unique subproblems. Save work by ordering computation to solve each problem once.

Recursive Algorithm vs. Recurrence

- Compute-Value(j)

  - If j = 0 return 0
  - Let i < j be highest-numbered show compatible with j
    - val1 = v_j + Compute-Value(i)
    - val2 = Compute-Value(j−1)
  - return max(val1, val2)

- Recurrence

  \[
  \begin{align*}
  OPT(j) &= \max\{v_j + OPT(p_j), OPT(j−1)\} \\
  OPT(0) &= 0
  \end{align*}
  \]

- Direct correspondence between the algorithm and recurrence
  - Tip: start by writing the recursive algorithm and translating it to a recurrence (replace method name by “OPT”)
  - After some practice, skip straight to the recurrence
**Step 3: Iterative “Bottom-Up” Algorithm**

**Memoization**

Intermediate approach: keep recursive function structure, but store value in array on first computation, and reuse it.

 Initialize array $M$ of size $n$ to hold optimal values

\[ M[0] = 0 \]

 for $j = 1$ to $n$
   \[ M[j] = \max(v_j + M[p_j], M[j-1]) \]

end for

**Example**

Running time? $O(n)$

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.

**Clicker Question**

The asymptotic running time of the memoized algorithm is

A. the same as the initial recursive solution.

B. between the initial recursive solution and the iterative version.

C. the same as the iterative version.

**Epilogue: Recovering the Solution (1)**

Idea: modify the algorithm to save best choice for each subproblem

**Memoization**

Intermediate approach: keep recursive function structure, but store value in array on first computation, and reuse it.

 Initialize array $M$ of size $n$ to empty, $M[0] = 0$

 function $Mfun(j)$
   if $M[j]$ = empty
     $M[j] = \max(v_j + Mfun(p_j), Mfun(j-1))$
     return $M[j]$
   end if

Can help if we have recursive structure but unsure of iteration order
Or as intermediate step in converting to iteration

**Epilogue: Recovering the Solution (2)**

Then trace back from end and "execute" the choices

Use algorithm above to fill in $M$ and choose arrays

\[ O = \{\} \]

\[ j = n \]

while $j > 0$
  if choose($j$) == 1 then
    \[ O = O \cup \{j\} \]
    \[ j = p_j \]
  else
    \[ j = j - 1 \]
  end if
end while

**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.
Key Step: Identify Subproblems

- Finding solution means: make “first choice”, then recursively solve a smaller instance of same problem.
- First example: Weighted Interval Scheduling
  - Binary first choice: \( j \in O \) or \( j \notin O \)?
- Next example: rod cutting
  - First choice has \( n \) options

First decision?

Choose length \( i \) of first piece, then recurse on smaller rod

From Recurrence to Algorithm

\[
\text{OPT}(j) = \max_{1 \leq i \leq j} \{ p_i + \text{OPT}(j - i) \}
\]
\[
\text{OPT}(0) = 0
\]

What size memoization array \( M \)? What order to fill?
- \text{Cutrod}(:,), \text{OPT}(:,), and \( M[] \) have same argument: index \( j \) of unique subproblems
- Range of values of \( j \) determines size of \( M \). \( M[0..n] \)
- Fill \( M \) so RHS values are computed before LHS.
  Fill from 0 to \( n \)

Rod Cutting

- Formulate problem on board
- Problem Input:
  - Steel rod of length \( n \), can be cut into integer lengths
  - Price based on length, \( p(i) \) for a rod of length \( i \)
- Goal
  - Cut rods into lengths \( i_1, \ldots, i_k \) such that \( i_1 + i_2 + \ldots i_k = n \).
  - Maximize value \( p(i_1) + p(i_2) + \ldots + p(i_n) \)

Steps 1 and 2

Step 1: Recursive Algorithm.

\[
\text{CutRod}(j)
\]
- if \( j = 0 \) then return 0
- \( v = 0 \)
- for \( i = 1 \) to \( j \) do
  - \( v = \max (v, p[i] + \text{CutRod}(j - i)) \)
- end for
- return \( v \)

- Running time for \( \text{CutRod}(n) \)? \( \Theta(2^n) \)

Step 2: Recurrence

\[
\text{OPT}(j) = \max_{1 \leq i \leq j} \{ p_i + \text{OPT}(j - i) \}
\]
\[
\text{OPT}(0) = 0
\]

Step 3: Iterative Algorithm

\[
\text{CutRod-Iterative}
\]
- Initialize array \( M[0..n] \)
- Set \( M[0] = 0 \)
- for \( j = 1 \) to \( n \) do
  - \( v = 0 \)
  - for \( i = 1 \) to \( j \) do
    - \( v = \max (v, p[i] + M[j - i]) \)
  - end for
  - Set \( M[j] = v \)
- end for

- Running time? \( \Theta(n^2) \) Note: body of for loop identical to recursive algorithm, directly implements recurrence
Epilogue: Recover Optimal Solution

Trace back from end and reconstruct choices that lead to optimal value

Run previous algorithm to fill in $M$ array, but with the following modification: let $\text{first-cut}[i]$ be the index $i$ that leads to the largest value when computing $M[j]$.

cuts = {}
j = n

\[\begin{align*}
\text{while } & j > 0 \text{ do} \\
& j = j - \text{first-cut}[j] \\
& \text{cuts} = \text{cuts} \cup \{\text{first-cut}[j]\}
\end{align*}\]

end while