Algorithm Design Techniques

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

Recursion May Be Easy .. But Sometimes Inefficient

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

Compute by straightforward recursion:

```
        F(5)
         /   \
        F(4) F(3)
       /     /
      F(3) F(2) F(1)
     /     /   /   /
    F(2) F(1) F(0) F(1) F(0)
   /     /   /     /     /
  F(1) F(0) F(1) F(0) F(0)
```

Clicker Question 1

Consider the following function to compute Fibonacci numbers:

```python
def 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. $\Theta(n!)$

Dynamic Programming Recipe

- **Step 1:** Devise simple recursive algorithm
  - Usually for optimization: try all choices at one level, solve subproblems
  - But: subproblems often shared ⇒ redundant computation (may be exponential time)

- **Step 2:** Write recurrence for optimal value

- **Step 3:** Design iterative algorithm (avoids redundancy)
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? It worked for case without values

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

Step 1: Recursive Algorithm

- Observation: Let \( O \) be the optimal solution.
  Last interval \( n \) is either in \( O \) or it isn’t.
  In both cases, we get a **smaller instance** of the same problem.

  - Recursive algorithm: value of optimal subset of first \( j \) shows (going backwards from \( j \))

    - **Compute-Value** \( (j) \)
      - **Base case**: if \( j = 0 \) return 0
      - **Case 1**: \( j \in O \)
        - Let \( i < j \) be highest-numbered show compatible with \( j \)
        - \( v_{j} = v_{j} + \text{Compute-Value}(i) \)
      - **Case 2**: \( j \not\in O \)
        - \( v_{j} = \text{Compute-Value}(j - 1) \)
      - return max\( (v_{j}, v_{j}) \)

  - 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)
  - \( \text{OPT}(j) = \text{value of optimal solution on first } j \text{ shows} \)
  - \( p_j \): highest-numbered show that is compatible with \( j \)
  - Recurrence:
    \[
    \begin{align*}
    \text{OPT}(0) &= 0 \\
    \text{OPT}(j) &= \max\{v_j + \text{OPT}(p_j), \text{OPT}(j - 1)\}
    \end{align*}
    \]
    - Case 1
    - Case 2

Recursive Algorithm vs. Recurrence

- **Compute-Value** \( (j) \)
  - If \( j = 0 \) return 0
  - \( v_{j} = v_{j} + \text{Compute-Value}(p_j) \)
  - \( v_{j} = \text{Compute-Value}(j - 1) \)
  - return max\( (v_{j}, v_{j}) \)

- **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

WeightedIS
Initialize array $M$ of size $n$ to hold optimal values
$M[0] = 0$  \hspace{1cm} \triangleright \text{Value of empty set}
for $j = 1$ to $n$
$M[j] = \max(v_j + M[p_j], M[j - 1])$
end for

Usually we directly convert recurrence to appropriate for loop.
Pay attention to dependence on previously-computed array entries to know in which direction to iterate.

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 $M[j] = \text{empty}$
$M[j] = \max(v_j + Mfun(p_j), Mfun(j-1))$
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

Clicker Question 3

The asymptotic complexity of the memoized algorithm is
A: Same as the initial recurrence
B: Between the initial recurrence and the iterated version
C: Same as the iterated version

Epilogue: Recovering the Solution (1)

Idea: modify the algorithm to what choice is made at each iteration
WeightedIS
Initialize array $M[0 \ldots n]$ to hold optimal values
Initialize array $\text{choose}[1 \ldots n]$ to hold choices
$M[0] = 0$
for $j = 1$ to $n$
$M[j] = \max(v_j + M[p_j], M[j - 1])$
Set $\text{choose}[j] = 1$ if first value is bigger, and 0 otherwise
end for

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 $\text{choose}(j) == 1$
$O = O \cup \{j\}$
$j = p_j$
else
$j = j - 1$
end if
end while

Tip: first do algorithm to just compute optimal value; then modify it to compute the actual solution

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 choice?

- Greedy? Cut length with maximum price
- Or: cut piece with maximum price per length?
- Divide and Conquer:
  Break rod at some (integer) point. Recurse for pieces.
- Dynamic Programming:
  Choose length \( i \) of first piece, then recurse on rest

Step 3: Iterative Algorithm

- Array \( M[0..n] \) where \( M[i] \) holds value of \( \text{OPT}(i) \).
- Order to fill \( M \)? From 0 to \( n \).

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

Rod Cutting

- 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.

\[
\text{CutRod}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v, p[i] + \text{CutRod}(j-i) \} & \text{else}\end{cases}
\]

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

Step 2: Recurrence

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

\( \text{OPT}(0) = 0 \)

Epilogue: Recover Optimal Solution

- Run previous algorithm to fill in \( M \) array
- \( \text{cuts} = \{ \} \)
- \( j = n \)
- while \( j > 0 \) do
  - \( i^* = \text{null}, v = 0 \) \( \triangleright i^* \) is the selected cut, \( v \) is its value
  - for \( i = 1 \) to \( j \) do
    - if \( p[i] + M[j-i] > v \) then
      - \( i^* = i \)
      - \( v = p[i] + M[i] \)
    - end if
  - end for
  - \( j = j - i^* \)
  - \( \text{cuts} = \text{cuts} \cup \{ i^* \} \)
- end while