COMPSCI 311: Introduction to Algorithms

Lecture 14: Dynamic Programming

Dan Sheldon

University of Massachusetts Amherst

Algorithm Design Techniques

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

Learning Goals

	Greedy	Divide and Conquer	Dynamic Programming
Formulate problem			
Design algorithm		✓	✓
Prove correctness	✓		
Analyze running time		✓	
Specific algorithms	Dijkstra, MST		Bellman-Ford

Weighted Interval Scheduling

- TV scheduling problem: n shows, can only watch one at a time. **New twist**: show j has value v_j. Want a set of shows S with no overlap and highest total value.
- Example on board
- 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 for *value* of optimal solution
  - Flavor: make "first choice", then recursively solve remaining part of the problem. (Problem: solve redundant subproblems $\rightarrow$ exponential time)
- Step 2: Write recurrence for optimal value
- Step 3: Design bottom-up iterative algorithm
- Epilogue: Recover optimal *solution*

Step 1: 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 compute value of optimal subset of first j shows

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
  val1 = v_j + Compute-Value(i)

  Case 2: $j \notin O$
  val2 = Compute-Value($j - 1$)

  return max(val1, val2)

Clicker

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 worst-case running time of this recursive solution is

A. $O(n \log n)$

B. $O(n^2)$

C. $O(1.618^n)$

D. $O(2^n)$

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.

Step 2: Recurrence

A **recurrence** expresses the optimal value for a problem of size j in terms of the optimal value of subproblems of size $i < j$.

Let OPT(j) = value of optimal solution on first j shows

$$\begin{aligned} \text{OPT}(0) &= 0 \\ \text{OPT}(j) &= \max\{\underbrace{v_j + \text{OPT}(p_j)}_{\text{Case 1}}, \quad \underbrace{\text{OPT}(j-1)}_{\text{Case 2}}\} \end{aligned}$$

- p_j: highest-numbered show $i < j$ that is compatible with j

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

$$\begin{aligned} \text{OPT}(j) &= \max\{v_j + \text{OPT}(p_j), \text{OPT}(j-1)\} \\ \text{OPT}(0) &= 0 \end{aligned}$$

- 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

Idea: compute the optimal value of every unique subproblem in order from smallest (base case) to largest (original problem). Use recurrence for each subproblem.

WeightedIS

```
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])
```

- Example

Step 3: Observations

WeightedIS
  Initialize array M of size n to hold optimal values
  $M[0] = 0$ ▷ Value of empty set
  for $j = 1$ to n **do**
    $M[j] = \max(v_j + M[p_j], M[j-1])$

- Iterative algorithm is a direct "wrapping" of recurrence in appropriate for loop.
- Pay attention to dependence on previously-computed entries of M to know in what order to iterate through array.
- Running time? $O(n)$

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* **then**
    M[j] $=$ max(v_j + Mfun(p_j), Mfun($j-1$))
  return M[j]

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

Clicker

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

WeigthedIS
  Initialize array $M[0 \ldots n]$ to hold optimal values
  Initialize array choose$[1 \ldots n]$ to hold choices
  $M[0] = 0$
  for $j = 1$ to n **do**
    $M[j] = \max(v_j + M[p_j], M[j-1])$
    Set choose$[j] = 1$ if first value is bigger, and 0 otherwise

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 $j > 0$ **do**
  if choose$(j) == 1$ **then**
    $O = O \cup \{j\}$
    $j = p_j$
  else
    $j = j - 1$

- Tip: first write algorithm to compute optimal value, then modify to compute actual solution

Review

- Recursive algorithm $\rightarrow$ recurrence $\rightarrow$ iterative algorithm
- Three ways of expressing value of optimal solutions of subproblems
  - Compute-Value(j). Recursive algorithm: arguments identify subproblems.
  - $\mathrm{OPT}(j)$. Used in recurrence; matches recursive algorithm.
  - $M[j]$. Array to hold optimal values for each distinct subproblem, filled in during iterative 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**

Rod Cutting

- **Input**: steel rod of length n, can be cut into integer lengths, get price $p(i)$ for piece of length i
- **Goal**: subdivide to maximize total value
- Example / problem formulation on board

First decision?

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

Step 1: Recursive Algorithm

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

- Running time for CutRod(n)? $\Theta(2^n)$

Step 2: Recurrence

$$\text{OPT}(j) = \max_{1 \le i \le j} \{p_i + \text{OPT}(j - i)\}$$
$$\text{OPT}(0) = 0$$

From Recurrence to Algorithm

$$\text{OPT}(j) = \max_{1 \le i \le j} \{p_i + \text{OPT}(j - i)\}$$
$$\text{OPT}(0) = 0$$

What size memoization array M? What order to fill? The recurrence provides all of the information needed to design an iterative algorithm.

- Cutrod($\cdot$), $OPT(\cdot)$, and $M[\cdot]$ 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

Step 3: Iterative Algorithm

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**
      $v = \max(v, p[i] + M[j-i])$
    Set $M[j] = v$

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

Epilogue: Recover Optimal Solution

Idea: Modify algorithm to record choices that lead to optimal value for each subproblem, then trace back from the end and "execute" the choices, starting with the largest problem.

Step 1: Run previous algorithm to fill in M array, but with the following modification: let first-cut[j] be the index i that leads to the largest value when computing $M[j]$.

Step 2: Trace back from end and execute choices.

  cuts = $\{\}$
  $j = n$ ▷ Remaining length
  while $j > 0$ **do**
    $j = j -$ first-cut$[j]$
    cuts = cuts $\cup$ $\{$first-cut$[j]\}$