COMPSCI 311: Introduction to Algorithms

Lecture 12: Dynamic Programming

Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon

Algorithm Design Techniques

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

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.

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

F(4) = F(3) + F(2) = (F(2) + F(1)) + (F(1) + F(0))
= ((F(1) + F(0))) + F(1)) + (F(1) + F(0))

Clicker Question 1

Consider computing the Fibonacci numbers by calling the function

```
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: $\Theta(1.618^n)$

C: $\Theta(2^n)$

D: $\Theta(n!)$

Comparison

	Greedy	Divide and Conquer	Dynamic Programming
Formulate problem	?	?	?
Design algorithm	easy	hard	hard
Prove correctness	hard	easy	easy
Analyze running time	easy	hard	easy

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
- Notation:
  - s_j, f_j: start and finish time of show (job) j
  - v_j = value of show 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

Weighted Interval Scheduling: 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: 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
val1 $= v_j$ + Compute-Value(i)

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

return max(val1, val2)

Extracting the Solution

- Finding the solution itself is a simple modification of the same algorithm

Compute-Solution(j)
Base case: if $j = 0$ return $\emptyset$

Case 1: $j \in O$
Let $i < j$ be highest-numbered show compatible with j
$O_1 = \{j\} \cup$ Compute-Solution(i)

Case 2: $j \notin O$
$O_2 =$ Compute-Solution($j - 1$)

return the solution O_1 or O_2 that has higher value

- Advice: first develop algorithm to compute optimal value; usually easy to modify it to compute the actual solution

Recurrence

- We've seen recurrences for running times (and various sequences in math)
- Here: recurrence expresses the value of an optimal solution.
- **Definitions**
  - OPT(j) = value of optimal solution on first j shows
  - p_j: highest-numbered show that is compatible with j
- Recurrence

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

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

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

- 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

Clicker Question 2

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

The running time of this recursive solution is

A: $O(n \log n)$

B: $O(n^2)$

C: $O(1.618^n)$

D: $O(2^n)$

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.

Iterative "Bottom-Up" Algorithm

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

Review

- Recursive algorithm → recurrence → iterative algorithm
- Three ways of expressing value of optimal solution for smaller problems
  - Compute-Value(j). Recursive algorithm—arguments identify subproblems.
  - $\text{OPT}(j)$. Mathematical expression. Write a recurrence for this that matches recursive algorithm.
  - $M[j]$. Array to hold optimal values. Entries filled in during iterative algorithm.

Dynamic Programming Recipe

- Devise recursive form for solution. **Flavor**: make "first choice", then recursively solve a smaller instance of same problem.
- Observe that recursive implementation involves redundant computation. (Often exponential time)
- Design iterative algorithm that solves all subproblems without redundancy.

Dividing into Problems

- 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

- **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 $p(i_1) + p(i_2) + \ldots + p(i_n)$

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

Steps 1 and 2

Step 1: Recursive Algorithm.

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

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

Step 2: Recurrence

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

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

Epilogue: Recover Optimal Solution

  Run previous algorithm to fill in M array
  cuts $= \{\}$
  $j = n$
  while $j > 0$ **do**
      $i^* =$ null, $v = 0$ ▷ 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
      $j = j - i^*$
      cuts $=$ cuts $\cup \{i^*\}$
  end while