CS 435: Algorithms

Dynamic Programming:

Dynamic Programming - Basics

Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances

- ***Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS***

- ***"Programming" here means "planning"***

- ***Main idea:***
  - ***set up a recurrence relating a solution to a larger instance to solutions of some smaller instances***
  - ***solve smaller instances once***
  - ***record solutions in a table***
  - ***extract solution to the initial instance from that table***

Dynamic Programming - Basics

- ***Similar to Divide and Conquer***

- ***However, Divide and Conquer uses non-overlapped sub-problems whereas DP uses Overlapped sub-problem.***

- ***Use of Divide & Conquer will result many unnecessary computations as sub-problems are overlapped.***

- ***DP saves computations in each iteration which can be used in the next iteration. – thus saves computation time BUT at the cost of more memory!***

Key Steps in DP

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

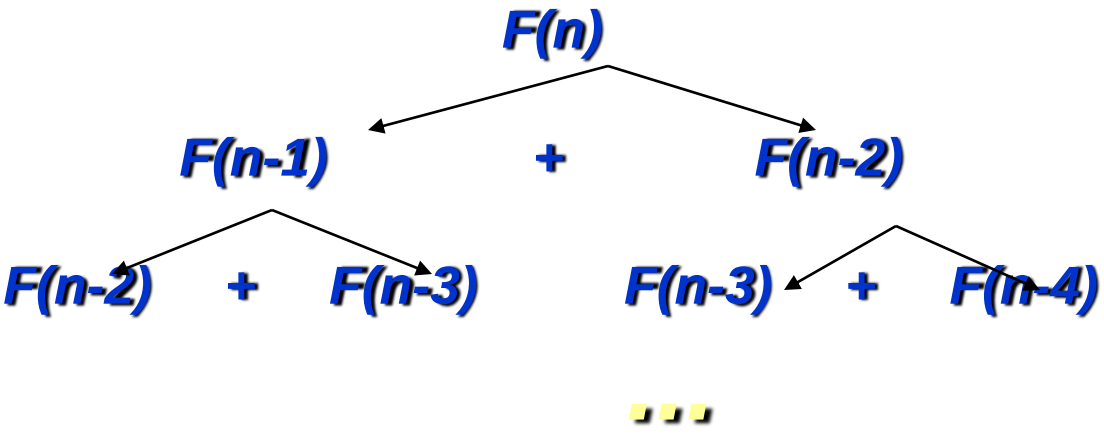
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Example: Fibonacci numbers

- ***Recall definition of Fibonacci numbers:***

  F(n) = F(n-1) + F(n-2)
  F(0) = 0
  F(1) = 1

- ***Computing the n^{th} Fibonacci number recursively (top-down):***



Example: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

F(0) = 0
F(1) = 1
F(2) = 1+0 = 1
…
F(n-2) =
F(n-1) =
F(n) = F(n-1) + F(n-2)

$0 \quad 1 \quad 1 \quad . \; . \; . \quad F(n\text{-}2) \quad F(n\text{-}1) \quad F(n)$

Efficiency:
- time n
- space n

Examples of DP algorithms

- ***Computing a binomial coefficient***
- ***Longest common subsequence***
- ***Warshall's algorithm for transitive closure***
- ***Floyd's algorithm for all-pairs shortest paths***
- ***Constructing an optimal binary search tree***
- ***Some instances of difficult discrete optimization problems:***
  - ***traveling salesman***
  - ***knapsack***

Rod Cutting Problem

The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Rod Cutting Problem

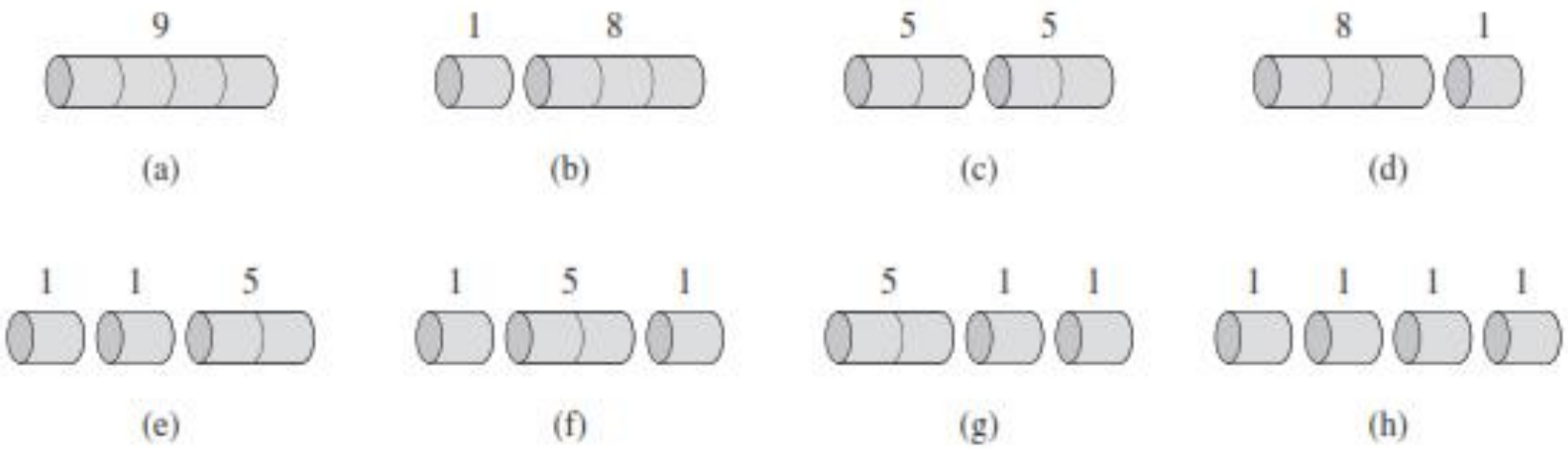


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Rod Cutting

- ***A decomposition is denoted into pieces using***

additive notation, so that $7 = 2 + 2 + 3$ indicates that a rod of length 7 is cut into three pieces—two of length 2 and one of length 3. If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$, then an optimal decomposition

$$n = i_1 + i_2 + \cdots + i_k$$

of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} .$$

Rod Cutting Problem

$$
\begin{aligned}
r_1 &= 1 \quad \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 \quad \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 \quad \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 \quad \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 \quad \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 \quad \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 \quad \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3, \\
r_8 &= 22 \quad \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 \quad \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 \quad \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}
$$

More generally, we can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1). \tag{15.1}$$

Rod Cutting

- ***The first argument means NO Cut at all.***
- ***The other arguments means cutting the rod into TWO pi eces of size I and n-i. AND then Optimally Cut ing those pieces further.***
- ***An equivalent scheme is to consider subdividing the rod into I and n-I with I the left decomposition and n-I the right decomposition. We then just subdivide the right side only (consider i=n etc). Thus solution with this is like***

no cuts at all as saying that the first piece has size $i = n$ and revenue p_n and that the remainder has size 0 with corresponding revenue $r_0 = 0$. We thus obtain the following simpler version of equation (15.1):

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}) \ . \qquad (15.2)$$

Rod Cutting

- ***Recursion Tree***

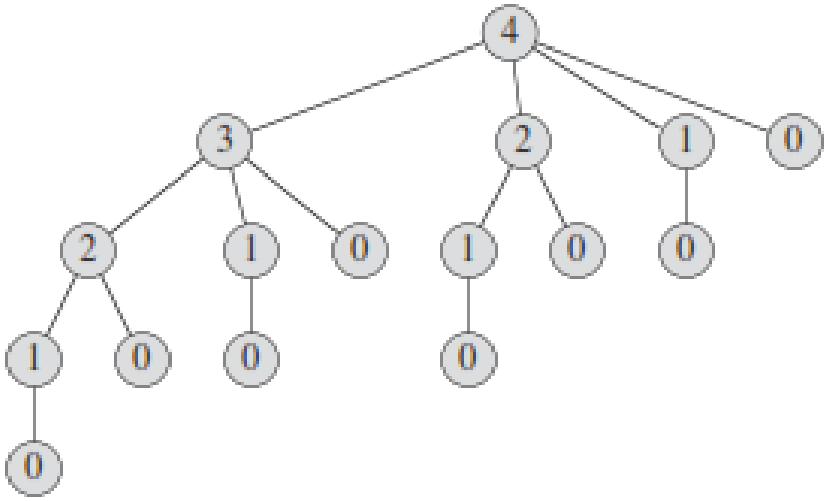


Figure 15.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

Rod Cutting

- ***Recursion Tree***

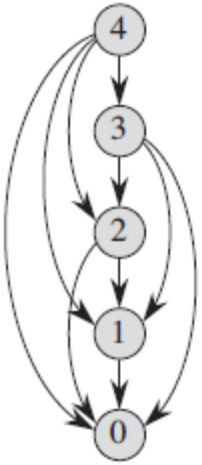


Figure 15.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that we need a solution to subproblem y when solving subproblem x. This graph is a reduced version of the tree of Figure 15.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

Rod Cutting – Running Time

- ***Running Time (Dynamic Programming will make it lot better)***

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) . \tag{15.3}$$

The initial 1 is for the call at the root, and the term $T(j)$ counts the number of calls (including recursive calls) due to the call CUT-ROD$(p, n-i)$, where $j = n-i$. As Exercise 15.1-1 asks you to show,

$$T(n) = 2^n , \tag{15.4}$$

and so the running time of CUT-ROD is exponential in n.

Rod Cutting

- ***There are TWO ways with DP and associated space for memory (called Memoisation) – Top Down and Bottom Up***

- ***Key concept– solve the sub-problem once (not repeatedly as with Divide & Conquer) and store the result in a table.***

- ***Use the already computed result again etc.***

- ***First – the Top Down***

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Rod Cutting – Top down

- ***It is basically Memoised version of Cut-Rod***
-

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2        return r[n]
3  if n == 0
4        q = 0
5  else q = −∞
6        for i = 1 to n
7              q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

Rod Cutting – Bottom-up

- ***Bottom-up version is even simpler***
-

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

Rod Cutting – Reconstruction

- ***The solutions shown so far produces the optimized value but NOT the solution itself i.e. not the specific cuts that were used to get the optimal revenue or value.***

- ***Reconstruction will provide both value and the sizes for the optimal solutions.***

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

Rod Cutting – Reconstruction

- ***Extended Bottom-up solution***

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n − s[n]
```

In our rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Rod Cutting – Reconstruction

- ***Printed Solution***

A call to PRINT-CUT-ROD-SOLUTION$(p, 10)$ would print just 10, but a call with $n = 7$ would print the cuts 1 and 6, corresponding to the first optimal decomposition for r_7 given earlier.

Computing a binomial coefficient by DP

Binomial coefficients are coefficients of the binomial formula:

$$(a + b)^n = C(n,0)a^n b^0 + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)a^0 b^n$$

Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ *for* $n > k > 0$

$C(n,0) = 1, \quad C(n,n) = 1$ *for* $n \geq 0$

Value of C(n,k) can be computed by filling a table:

	0	1	2	. . .	k-1	k
0	1					
1	1	1				
.						
.						
.						
n-1					C(n-1,k-1)	C(n-1,k)

Computing $C(n,k)$: pseudocode and analysis

ALGORITHM *Binomial*(n, k)

```
//Computes C(n, k) by the dynamic programming algorithm
//Input: A pair of nonnegative integers n ≥ k ≥ 0
//Output: The value of C(n, k)
for i ← 0 to n do
    for j ← 0 to min(i, k) do
        if j = 0 or j = i
            C[i, j] ← 1
        else C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
return C[n, k]
```

Time efficiency: Θ(nk)

Space efficiency: Θ(nk)

Properties of a problem that can be solved with dynamic programming

- ***Simple Subproblems***
  - *We should be able to break the original problem to smaller subproblems that have the same structure*

- ***Optimal Substructure of the problems***
  - *The solution to the problem must be a composition of subproblem solutions (subproblem itself needs to be optimal)*

- ***Subproblem Overlap***
  - *Optimal subproblems to unrelated problems can contain subproblems in common*

Discovering Optimal Substructure

1. You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal

0-1 Knapsack Problem

Given a knapsack with maximum capacity W, and a set S consisting of n items
Each item i has some weight w_i and benefit value b_i (all w_i , b_i and W are integer values)

<u>Problem</u>: How to pack the knapsack to achieve maximum total value of packed items?

Knapsack Problem by DP

Given n items of

integer weights: w_1 w_2 ... w_n

values: v_1 v_2 ... v_n

a knapsack of integer capacity W

find most valuable subset of the items that fit into the knapsack

Consider instance defined by first i items and capacity j ($j \leq W$).

Let $V[i,j]$ be optimal value of such an instance. Then

Knapsack Problem by DP

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{if } j-w_i \geq 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

Initial conditions: $V[0,j] = 0$ and $V[i,0] = 0$

Knapsack Problem -Example

Item	a	b	c	d	e
Benefit	15	12	9	16	17
Weight	2	5	3	4	6

The maximum allowable total weight in the knapsack is Wmax = 12.

	1	2	3	4	5	6	7	8	9	10	11	12
a	0	15	15	15	15	15	15	15	15	15	15	15
b	0	15	15	15	15	15	27	27	27	27	27	27
c	0	15	15	15	24	24	27	27	27	36	36	36
d	0	15	15	16	24	31	31	31	40	40	43	43
e	0	15	15	16	24	31	31	32	40	40	43	48