Dynamic Programming

CSci 4041: Algorithms and Data Structures

Instructor: Arindam Banerjee

March 12, 2015

Dynamic Programming

- Applied for optimization problems
  - Finding a solution vs. optimum solution
- Partition the problem in terms of subproblems
  - Subproblems may have shared subsubproblems
  - Divide-and-conquer will recompute things
  - Dynamic programming solves each subproblem once
- Developing dynamic programming algorithms
  - Characterize the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the value of an optimalsolution
    - Typically in a bottom-up fashion
  - Construct an optimal solution

Rod Cutting

- Given a rod of length n
  - Cut it into k smaller pieces, $1 \leq k \leq n$
  - Size of each piece is $i_1, i_2, \ldots, i_k$

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

- Price table gives price of every piece
  - Piece of size i has price p_i
  - Total revenue for a cut of sizes $i_1, i_2, \ldots, i_k$ is

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

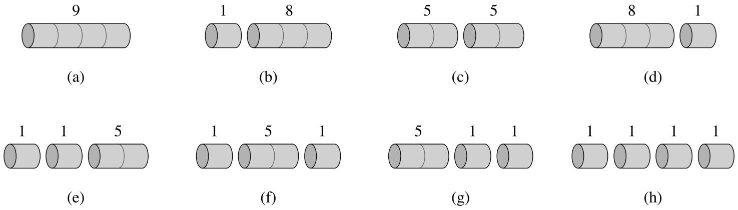
- How do we cut the rod to maximize revenue?

Example: Rod Cutting

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

- Price table for rod cutting
- Prices p_i for each length i
- Optimizing cuts for a given n

Example: Rod Cutting with $n = 4$



- Consider all possible cuts
- Total number of options is 2^{n-1}
- Decide on cutting or not at distance i from the left end

Example: Rod Cutting by Inspection

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

n	Cut	r_n
1	1 (no cuts)	1
2	2 (no cuts)	5
3	3 (no cuts)	8
4	2 + 2	10
5	2 + 3	13
6	6 (no cuts)	17
7	1 + 6	18
8	2 + 6	22
9	3 + 6	25
10	10 (no cuts)	30

Rod Cutting as Optimization

- Express r_n in terms of optimal revenue from shorter rods
  - Initial cut into two pieces of size $(i, n-i), i = 1, 2, \ldots, n-1$
  - Also consider the 'no cut' scenario

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

- Optimal substructure
  - Optimal solution includes optimal solution to smaller subproblems
  - After first cut, we have two smaller problems of the same type
- Alternative simpler approach for recursive structure
  - Cut the first piece of size i
  - Optimize over the remainder of size $(n-i)$
  - The first piece cannot be cut further, the remainder can be

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

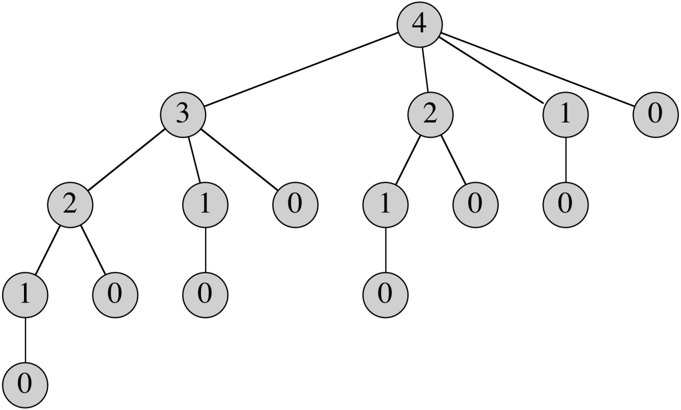
Algorithm: Cut-Rod

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

- Induction on n shows correctness
- Really slow algorithm
- $T(n)$: total number of calls made to CUT-ROD

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \quad \Rightarrow \quad T(n) = 2^n$$

Issues with CUT-ROD



- Repeatedly solves the same subproblem

Dynamic Programming for Rod Cutting

- Idea: Solve every sub-problem only once
- Time-memory tradeoff
  - Use extra memory to store solutions of subproblems
  - Each subproblem needs to be solved once
  - Subsequent calls is just a look-up
- From exponential to polynomial time
  - Number of distinct subproblems needs to be polynomial
- Two approaches to dynamic programming
  - Top-down with memoization: Recursive, but stores subproblem solutions
  - Bottom-up: Solve smaller subproblems first

Algorithm: MEMOIZED-CUT-ROD

MEMOIZED-CUT-ROD(p, n)

let $r[0\,..\,n]$ be a new array
for $i = 0$ **to** n
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX(p, n, r)

- Use array r to store solutions
- Same subproblem need not be solved twice

Algorithm: Memoized-Cut-Rod-Aux

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

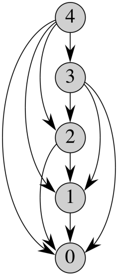
- If $r[n] \geq 0$, subproblem has been solved before
- Otherwise solve and store in $r[n]$
- Overall approach has complexity $\Theta(n^2)$

Algorithm: Bottom-Up-Cut-Rod

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

- No recursions, solve the smaller subproblems first
- When $r[j-i]$ is called, it has been computed
- Complexity of $\Theta(n^2)$, good constants

Subproblem Graphs



- Directed edge implies subproblem dependency
- For Bottom-Up, solve subproblems by ‘reverse topological sort’
- Number of subproblems is equal to the number of vertices
- Typically, time for a subproblem is proportional to outdegree
- Typically, complexity is linear in number of edges and nodes

Algorithm: EXTENDED-BOTTOM-UP-CUT-ROD

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

- Array s keeps track of first piece to cut
- $s[j]$ is the optimal size i for problem of size j

Printing A Solution

PRINT-CUT-ROD-SOLUTION(p, n)

$(r, s) = $ EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while $n > 0$
    print $s[n]$
    $n = n - s[n]$

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

- For $(p, 10)$ will just print 10
- For $(p, 7)$, will print 1, 6

Review: Matrix Multiplication

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
    for i = 1 to A.rows
        for j = 1 to B.columns
            c_ij = 0
            for k = 1 to A.columns
                c_ij = c_ij + a_ik · b_kj
    return C
```

- Assume A is $p \times q$, B is $q \times r$
- Number of columns of A = number of rows of B
- C is $p \times r$, total pqr scalar multiplications

Matrix Chain Multiplication

- Matrix multiplication is associative: $(A_1 A_2) A_3 = A_1 (A_2 A_3)$
- Example: $A_1 A_2 A_3 A_4$ can be parenthesized in 5 ways

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

Matrix Chain Multiplication: Problem Formulation

- Fully parenthesize the product: $A_1 A_2 \cdots A_n$
  - Each A_i has dimensions $p_{i-1} \times p_i$
  - Find parenthesization with minimum number of scalar multiplications
- Different parenthesizations have different complexity
- Consider $A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3)$
  - A_1 is 10×100, A_2 is 100×5, A_3 is 5×50
  - $((A_1 A_2) A_3)$ takes a total of $5000 + 2500 = 7500$ scalar multiplications
  - $(A_1 (A_2 A_3))$ takes a total of $25,000 + 50,000 = 75,000$ scalar multiplications

Total Number of Parenthesizations

- $P(n)$ denote the total number of parenthesis

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

- Similar recurrence as Catalan numbers, which grows as $\Omega(4^n/n^{3/2})$
- Can show that the recurrence grows as $\Omega(2^n)$