Algorithms 05

CS201

Kaustuv Nag

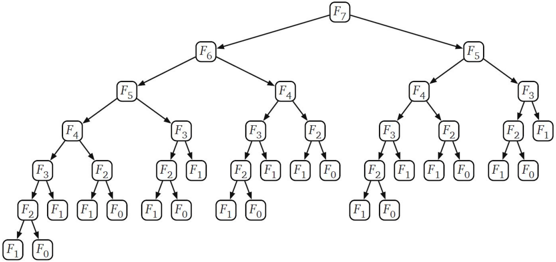
Dynamic Programming

- Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- Dynamic programming applies when the subproblems overlap, i.e., when subproblems share subsubproblems
  - A divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.
  - A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.
- We typically apply dynamic programming to optimization problems.

Fibonacci - version 1

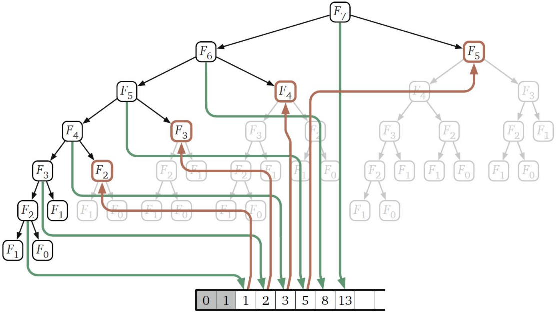
```
// fib -- compute Fibonacci(n)
function fib(integer n): integer
  assert (n >= 0)
  if n == 0: return 0 fi
  if n == 1: return 1 fi

  return fib(n - 1) + fib(n - 2)
end
```



Fibonacci - version 2

```
// fib -- compute Fibonacci(n)
function fib(integer n): integer
  if n == 0 or n == 1:
    return n
  else-if f[n] != -1:
    return f[n]
  else
    f[n] = fib(n - 1) + fib(n - 2)
    return f[n]
  fi
end
```



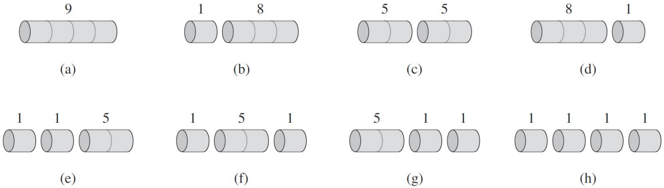
Fibonacci - version 3

```
// fib -- compute Fibonacci(n)
function fib(integer n): integer
  if n == 0 or n == 1:
    return n
  fi
  let u := 0
  let v := 1
  for i := 2 to n:
    let t := u + v
    u := v
    v := t
  repeat
  return v
end
```

The Rod-Cutting Problem

- Let p_i denote the price of a rod with length i inches.
- Rod lengths are always an integral number of inches.
- 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.
- 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
price p_i	1	5	8	9



The Rod-Cutting Problem

- We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end, for $i = 1, 2, \ldots, n$.
- Let us denote a decomposition into pieces using ordinary 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 $1 \leq k \leq n$, then an optimal decomposition

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

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

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

The Rod-Cutting Problem

- We frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

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

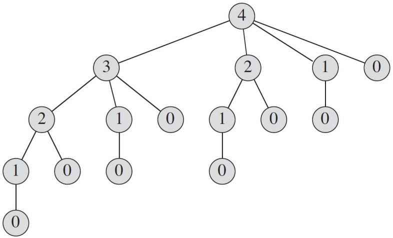
- Rod-cutting problem exhibits *optimal substructure*: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.
- A recursive formulation:

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

The Rod-Cutting Problem

Recursive Top Down Algorithm

```
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
```



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

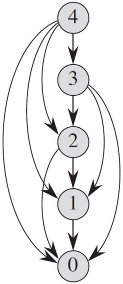
Note: Try with $n = 40$; it would take enough time.

The Rod-Cutting Problem

Top-down Algorithm With Memoization

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

```
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
```



The Rod-Cutting Problem

Bottom-up Algorithm With Memoization

```
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]
```

The Rod-Cutting Problem

Bottom-up Algorithm With Memoization: Reconstructing a solution

```
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
```

Matrix-chain Multiplication

Algorithm for 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
```

Matrix-chain Multiplication

A Problem of Chain $\langle A_1, A_2, A_3 \rangle$

- Sizes of A_1, A_2, and A_3 are 10×100, 100×5, and 5×50, respectively.
- $(A_1 A_2) A_3$ requires $10 \times 100 \times 5 + 10 \times 5 \times 50 = 5000 + 2500 = 7500$ scalar multiplications.
- $A_1 (A_2 A_3)$ requires $100 \times 5 \times 50 + 10 \times 100 \times 50 = 25000 + 50000 = 75000$ scalar multiplications.

Matrix-chain Multiplication

Problem Statement

- Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Counting the Number of Parenthesizations

- The number of alternative parenthesizations of a sequence of n matrices by $P(n)$ given as follows:

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

Matrix-chain Multiplication

Applying Dynamic Programming

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution.
- Construct an optimal solution from computed information.

Matrix-chain Multiplication

The Structure of an Optimal Parenthesization

- Suppose, we need to evaluate $A_{i..j} = A_i A_{i+1} \cdots A_j$, $i \leq j$.
- If $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$.
- Then compute $A_{i..k} A_{k+1..j}$.
- The way we parenthesize the "prefix" subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$.
  - Why? If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then we could substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce another way to parenthesize $A_i A_{i+1} \cdots A_j$ whose cost was lower than the optimum: a contradiction.

Matrix-chain Multiplication

The Structure of an Optimal Parenthesization

- We can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems, finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions.

Matrix-chain Multiplication

A Recursive Solution

- Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$. For the full problem, the lowestcost way to compute $A_{1..n}$ is $m[1,n]$.
- Then,

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$

This recursive equation assumes that we know the value of k, which we do not. There are only $j-i$ possible values for k; $k = i, i+1, \ldots, j-1$. Therefore,

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \,, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \,. \end{cases}$$

Matrix-chain Multiplication

Step 2: A Recursive Solution

- The $m[i,j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution.
- We define $s[i,j]$ to be a value of k at which we split the product $A_iA_{i+1}\cdots A_j$ in an optimal parenthesization.
- That is, $s[i,j]$ equals a value k such that

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$

Matrix-chain Multiplication

Algorithm for Matrix Chain Multiplication

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new table
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                      // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Matrix-chain Multiplication

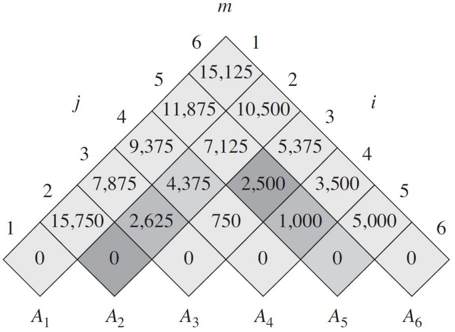
Algorithm for Matrix Chain Multiplication: Explanation

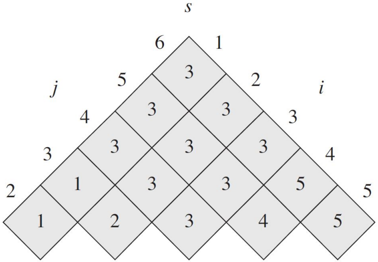
- The algorithm first computes $m[i,i] = 0$ for $i = 1, 2, \ldots, n$ (the minimum costs for chains of length 1) in lines 3–4.
- It then uses recurrence to compute $m[i, i+1]$ for $i = 1, 2, \ldots, n-1$ (the minimum costs for chains of length $l = 2$) during the first execution of the `for` loop in lines 5–13.
- The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ (the minimum costs for chains of length $l = 3$), and so forth.
- At each step, the $m[i,j]$ cost computed in lines 10–13 depends only on table entries $m[i,k]$ and $m[k+1,j]$ already computed.

Matrix-chain Multiplication

Algorithm for Matrix Chain Multiplication: An Example

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000 \,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$

Matrix-chain Multiplication

Constructing an Optimal Solution

- Each entry s records a value of k such that an optimal parenthesization of $A_i A_{i+1} \ldots A_j$ splits the product between A_k and A_{k+1}.
- The final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]} A_{s[1,n]+1..n}$.
- We can determine the earlier matrix multiplications recursively, since $s[1, s[1,n]]$ determines the last matrix multiplication when computing $A_{1..s[1,n]}$ and $s[s[1,n]+1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1..n}$.

Matrix-chain Multiplication

Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

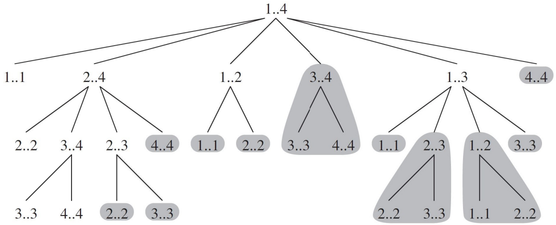
When to Use Dynamic Programming

- The optimal solution should contain optimal substructure.
  - A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.
- A recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems.
  - When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems.

Matrix-chain Multiplication

Recursive Algotithm

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
        + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
        + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```



Matrix-chain Multiplication

Recursive Algotithm: Complexity is $\Omega(2^n)$

$$
\begin{aligned}
T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \\
&\geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\
&= 2 \sum_{i=0}^{n-2} 2^{i} + n \\
&= 2(2^{n-1} - 1) + n \\
&= 2^n - 2 + n \\
&\geq 2^{n-1}
\end{aligned}
$$

Matrix-chain Multiplication

Recursive Algotithm: Top Down With Memoization

```
MEMOIZED-MATRIX-CHAIN(p)
n = p.length − 1
let m[1..n, 1..n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
            + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

Longest Common Subsequence

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, x_k \rangle$ is a subsequence of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$.
  - For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.
- Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.
  - For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y. $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are the longest common subsequences.
- In the longest-common-subsequence problem, we are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum length common subsequence of X and Y.

White Board

White Board