DAA
Unit- II
Greedy and Dynamic Programming

By Mrs. B.A. Khivsara
Asst. Professor
Department of Computer Engineering
SNJB’s KBJ COE, Chandwad
Greedy Method
Greedy Method

- **Greedy Principal:** are typically used to solve optimization problem. Most of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a **feasible solution**. We are required to find a feasible solution that either minimizes or maximizes a given **objective function**. In the most common situation we have:
Greedy Method

- **Subset Paradigm**: Devise an algorithm that works in stages
- Consider the inputs in an order based on some selection procedure
- Use some optimization measure for selection procedure
- At every stage, examine an input to see whether it leads to an optimal solution
- If the inclusion of input into partial solution yields an infeasible solution, discard the input; otherwise, add it to the partial solution
Control Abstraction for Greedy

function greedy(C: set): set;
begin
    S := Ø;
    while (not solution(S) and C ≠ Ø ) do
        begin
            x := select(C);
            C := C - {x};
            if feasible(S ∪ {x})
                then S := S ∪ {x};
            end;
        if solution(S) then return(S) else return(Ø);
    end;
end;
Control Abstraction Explanation

- **C**: A set of objects;
- **S**: The set of objects that have already been used;
- **feasible()**: A function that checks if a set is a feasible solution;
- **solution()**: A function that checks if a set provides a solution;
- **select()**: A function for choosing most next object.
- An **objective function** that we are trying to optimize.
## Greedy Vs Divide and Conquer

<table>
<thead>
<tr>
<th><strong>Greedy</strong></th>
<th><strong>Divide and Conquer</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Used when need to find optimal solution</td>
<td>No optimal solution, used when problem have only one solution</td>
</tr>
<tr>
<td>Does not work parallel</td>
<td>work parallel by dividing big problem in smaller sub problem and running then parallel</td>
</tr>
<tr>
<td>Example: Knapsack, Activity selection</td>
<td>Example: Sorting, Searching</td>
</tr>
</tbody>
</table>
Knapsack problem using Greedy Method
The Knapsack Problems

- The **knapsack problem** is a problem of optimization: Given a set of items \( n \), each with a weight \( w \) and profit \( p \), determine the number of each item to include in a knapsack such that the total weight is less than or equal to a given knapsack limit \( M \) and the total Profit is maximum.
The Knapsack Problems

- **The Integer Knapsack Problem**
  
  Maximize \( \sum_{i=1}^{n} p_i x_i \)

  Subject to \( \sum_{i=1}^{n} w_i x_i \leq M \)

- **The 0-1 Knapsack Problem**: same as integer knapsack except that the values of \( x_i \)'s are restricted to 0 or 1.

- **The Fractional Knapsack Problem**: same as integer knapsack except that the values of \( x_i \)'s are between 0 and 1.
The knapsack algorithm

- **The greedy algorithm:**
  Step 1: Sort $p_i/w_i$ into nonincreasing order.
  Step 2: Put the objects into the knapsack according to the sorted sequence as possible as we can.

- e. g.  $n = 3$, $M = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$
  $(w_1, w_2, w_3) = (18, 15, 10)$
  Sol: $p_1/w_1 = 25/18 = 1.39$
  $p_2/w_2 = 24/15 = 1.6$
  $p_3/w_3 = 15/10 = 1.5$
  Optimal solution: $x_1 = 0$, $x_2 = 1$, $x_3 = 1/2$
  total profit = $24 + 7.5 = 31.5$
Job sequencing using Greedy Method
The problem is stated as below.

- There are $n$ jobs to be processed on a machine.
- Each job $i$ has a deadline $d_i \geq 0$ and profit $p_i \geq 0$.
- $P_i$ is earned iff the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.
A feasible solution is a subset of jobs J such that each job is completed by its deadline.

An optimal solution is a feasible solution with maximum profit value.

**Example**: Let \( n = 4 \),
\[
(p_1,p_2,p_3,p_4) = (100,10,15,27),
(d_1,d_2,d_3,d_4) = (2,1,2,1)
\]
JOB SEQUENCING WITH DEADLINES (Contd..)

<table>
<thead>
<tr>
<th>Sr.No.</th>
<th>Feasible Solution</th>
<th>Processing Sequence</th>
<th>Profit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>(1,2)</td>
<td>(2,1)</td>
<td>110</td>
</tr>
<tr>
<td>(ii)</td>
<td>(1,3)</td>
<td>(1,3) or (3,1)</td>
<td>115</td>
</tr>
<tr>
<td>(iii)</td>
<td>(1,4)</td>
<td>(4,1)</td>
<td>127 is the optimal one</td>
</tr>
<tr>
<td>(iv)</td>
<td>(2,3)</td>
<td>(2,3)</td>
<td>25</td>
</tr>
<tr>
<td>(v)</td>
<td>(3,4)</td>
<td>(4,3)</td>
<td>42</td>
</tr>
<tr>
<td>(vi)</td>
<td>(1)</td>
<td>(1)</td>
<td>100</td>
</tr>
<tr>
<td>(vii)</td>
<td>(2)</td>
<td>(2)</td>
<td>10</td>
</tr>
<tr>
<td>(viii)</td>
<td>(3)</td>
<td>(3)</td>
<td>15</td>
</tr>
<tr>
<td>(ix)</td>
<td>(4)</td>
<td>(4)</td>
<td>27</td>
</tr>
</tbody>
</table>
GREEDY ALGORITHM TO OBTAIN AN OPTIMAL SOLUTION for Job Scheduling

- Consider the jobs in the decreasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.

- In the example considered before, the decreasing profit vector is

\[
\begin{pmatrix}
100 & 27 & 15 & 10
\end{pmatrix}
\begin{pmatrix}
2 & 1 & 2 & 1
\end{pmatrix}
\]

\[
p_1 \quad p_4 \quad p_3 \quad p_2 \quad d_1 \quad d_4 \quad d_3 \quad d_2
\]
GREEDY ALGORITHM TO OBTAIN AN OPTIMAL SOLUTION (Contd..)

J = \{ 1\} is a feasible one
J = \{ 1, 4\} is a feasible one with processing sequence (4,1)
J = \{ 1, 3, 4\} is not feasible
J = \{ 1, 2, 4\} is not feasible
J = \{ 1, 4\} is optimal
Activity Selection Using Greedy Method
The activity selection problem

Problem: n activities, $S = \{1, 2, \ldots, n\}$, each activity $i$ has a \textbf{start time} $s_i$ and a \textbf{finish time} $f_i$, $s_i \leq f_i$.

Activity $i$ occupies time interval $[s_i, f_i]$.

$i$ and $j$ are \textbf{compatible} if $s_i \geq f_j$ or $s_j \geq f_i$.

The problem is to select a maximum-size set of \textbf{mutually compatible} activities.
Example:

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_i</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>f_i</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

The solution set = \{1, 4, 8, 11\}

Algorithm:

**Step 1:** Sort $f_i$ into nondecreasing order. After sorting, $f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_n$.

**Step 2:** Add the next activity $i$ to the solution set if $i$ is compatible with each in the solution set.

**Step 3:** Stop if all activities are examined. Otherwise, go to step 2.

Time complexity: $O(n \log n)$
Solution of the example:

<table>
<thead>
<tr>
<th>i</th>
<th>$s_i$</th>
<th>$f_i$</th>
<th>accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>8</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>10</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>11</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>12</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>13</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>14</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Solution = \{1, 4, 8, 11\}
Dynamic programming
Dynamic Programming

- **Principal**: Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.

- Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

1) Overlapping Subproblems
2) Optimal Substructure
Dynamic Programming

1) Overlapping Subproblems: Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don’t have to recomputed.

2) Optimal Substructure: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
The principle of optimality

- Dynamic programming is a technique for finding an \textit{optimal} solution

- The \textit{principle of optimality} applies if the optimal solution to a problem always contains optimal solutions to all subproblems
Differences between Greedy, D&C and Dynamic

- **Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.

- **Divide-and-conquer.** Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

- **Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
Divide and conquer Vs Dynamic Programming

- Divide-and-Conquer: a top-down approach.
- Many smaller instances are computed more than once.

- Dynamic programming: a bottom-up approach.
- Solutions for smaller instances are stored in a table for later use.
0/1 knapsack using Dynamic Programming
0 - 1 Knapsack Problem

Knapsack problem.

- Given $n$ objects, Item $i$ weighs $w_i > 0$ and has Profit $p_i > 0$. Knapsack has capacity of $M$.
- $X_i = 1$ if object is placed in the knapsack otherwise $X_i = 0$

Maximize

$$\sum_{i=1}^{n} p_i x_i$$

Subject to

$$\sum_{i=1}^{n} w_i x_i \leq M$$
0 - 1 Knapsack Problem

- $S^i = \text{pair}(p,w)$ i.e. profit and weight
- $S^i_1 = \{(P,W)|(P+pi+1 ,W+wi+1)\}$
### 0/1 Knapsack - Example

<table>
<thead>
<tr>
<th>Item</th>
<th>Profit</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

- No of object $n=3$
- Capacity of Knapsack $M = 6$
0/1 Knapsack – Example Solution

- $S^0 = \{(0,0)\}$
- $S^0_1$ will be obtained by adding Profit Weight of First object to $S^0$
  - $S^0_1 = \{(1,2)\}$
- $S^1$ will be obtained by merging $S^0$ and $S^0_1$
  - $S^1 = \{(0,0),(1,2)\}$
  - $S^1_1$ will be obtained by adding Profit Weight of second object to $S^1$
  - $S^1_1 = \{(2,3),(3,5)\}$
- $S^2$ will be obtained by merging $S^1$ and $S^1_1$
  - $S^1 = \{(0,0),(1,2),(2,3),(3,5)\}$
  - $S^2_1$ will be obtained by adding Profit Weight of third object to $S^2$
  - $S^2_1 = \{(5,4),(6,6),(7,7),(8,9)\}$
- $S^3$ will be obtained by merging $S^2$ and $S^2_1$
  - $S^3 = \{(0,0),(1,2),(2,3),(3,5),(5,4),(6,6),(7,7),(8,9)\}$
Pair(7,7) and (8,9) will be deleted as it exceed weight of Knapsack(M=6)

Pair(3,5) will be deleted by dominance Rule

So : $S^3 = \{(0,0),(1,2),(2,3),(5,4),(6,6)\}$

Last pair is (6,6) which is generated by $S^3$, so $X_3 = 1$

Now subtract profit and weight of third object from (6,6)

(6-5,6-4)=(1,2)

(1,2) is generated by $S^1$, so $X_1 = 1$

Now subtract profit and weight of third object from (1,2)

(1-1,2-2)=(0,0)

As nothing is generated by $S^2$ so $X_2 = 0$

**Answer is : Total Profit is 6**

object 1 and 3 is selected to put into knapsack.
Binomial Coefficient using Dynamic Programming
The Binomial Coefficient

\[ \binom{n}{k} = \frac{n!}{k!(n-k)!} \text{ for } 0 < k < n \]

\[ \binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases} \]
The recursive algorithm

binomialCoef(n, k)

1. if $k = 0$ or $k = n$
2. then return 1
3. else return (binomialCoef(n - 1, k - 1) + binomialCoef(n - 1, k))
Dynamic Solution for Binomial Coefficient

• Use a matrix $C$ of $n+1$ rows, $k+1$ columns where

$$c[n,k] = \binom{n}{k}$$

• Establish a recursive property. Rewrite in terms of matrix $B$:

$$C[i, j] = \begin{cases} 
C[i-1, j-1] + C[i-1, j] & 0 < j < i \\
1 & j = 0 \text{ or } j = i 
\end{cases}$$

• Solve all “smaller instances of the problem” in a bottom-up fashion by computing the rows in $B$ in sequence starting with the first row.
The C Matrix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ B[i-1,j-1] \quad B[i-1,j] \quad B[i,j] \]
Example: Compute $C[4,2]=\binom{4}{2}$

- Row 0: $C[0,0] = 1$
- Row 1: $C[1,0] = 1$
  $C[1,1] = 1$
- Row 2: $C[2,0] = 1$
  $C[2,1] = C[1,0] + C[1,1] = 2$
  $C[2,2] = 1$
- Row 3: $C[3,0] = 1$
- Row 4: $C[4,0] = 1$
  $C[4,1] = C[3,0] + C[3,1] = 4$
Algorithm For Binomial coefficient

- Algo bin(n,k )
  1. for i = 0 to n // every row
  2. for j = 0 to minimum( i, k )
  3. if j = 0 or j = i // column 0 or diagonal
  4. then B[ i , j ] = 1
  5. else B[ i , j ] = B[i -1, j -1] + B[i -1, j ]
  6. return B[ n, k ]
Number of iterations

$$\sum_{i=0}^{n} \sum_{j=0}^{\min(i,k)} 1 = \sum_{i=0}^{k} \sum_{j=0}^{i} 1 + \sum_{i=k+1}^{n} \sum_{j=0}^{k} 1 =$$

$$\sum_{i=0}^{k} (i + 1) + \sum_{i=k+1}^{n} (k + 1) =$$

$$\frac{(k + 2)(k + 1)}{2} + (n - k)(k + 1) =$$

$$\frac{(2n - k + 2)(k + 1)}{2}$$
Optimal Binary Search Tree (OBST) using Dynamic Programming
Optimal binary search trees

- Example: binary search trees for 3, 7, 9, 12;
• A full binary tree may not be an optimal binary search tree if the identifiers are searched for with different frequency.

• Consider these two search trees. If we search for each identifier with equal probability:
  – In first tree, the average number of comparisons for successful search is 2.4. \[ \frac{1+2+2+3+4}{5} = 2.4 \]
  – Comparisons for second tree is 2.2.

• The second tree has
  – a better worst case search time than the first tree.
  – a better average behavior. \[ \frac{1+2+2+3+3}{5} = 2.2 \]
Optimal binary search trees

- In evaluating binary search trees, it is useful to add a special square node at every place there is a null link.
  - We call these nodes *external nodes*.
  - We also refer to the external nodes as *failure nodes*.
  - The remaining nodes are *internal nodes*.
  - A binary tree with external nodes added is an *extended binary tree*.
Optimal binary search trees

- **External / internal path length**
  - The sum of all external / internal nodes’ levels.
- For example
  - Internal path length, $I$, is:
    $$I = 0 + 1 + 1 + 2 + 3 = 7$$
  - External path length, $E$, is:
    $$E = 2 + 2 + 4 + 4 + 3 + 2 = 17$$
- A binary tree with $n$ internal nodes are related by the formula
  $$E = I + 2n$$
Optimal binary search trees

• n identifiers are given.

$P_i, 1 \leq i \leq n$ : Successful Probability

$Q_i, 0 \leq i \leq n$ : Unsuccessful Probability

Where,

$$\sum_{i=1}^{n} P_i + \sum_{i=0}^{n} Q_i = 1$$
• Identifiers : 4, 5, 8, 10, 11, 12, 14
• Internal node : successful search, $P_i$
• External node : unsuccessful search, $Q_i$

- The expected cost of a binary tree:
  $$\sum_{i=1}^{n} P_i \times \text{level}(a_i) + \sum_{i=0}^{n} Q_i \times (\text{level}(E_i) - 1)$$

- The level of the root : 1
The dynamic programming approach for Optimal binary search trees

To solve OBST, requires to find answer for Weight (w), Cost (c) and Root (r) by:

\[ W(i,j) = p(j) + q(j) + w(i,j-1) \]

\[ C(i,j) = \min \{ c(i,k-1) + c(k,j) \} + w(i,j) \quad \ldots \quad i < k \leq j \]

\[ r = k \text{ (for which value of } c(i,j) \text{ is small)} \]
Optimal binary search trees

• Example
  – Let \( n = 4 \), \((a_1, a_2, a_3, a_4) = (\text{do, for, void, while})\).

    Let \((p_1, p_2, p_3, p_4) = (3, 3, 1, 1)\)
    and \((q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)\).

  – Initially \( w_{ii} = q_i \), \( c_{ii} = 0 \), and \( r_{ii} = 0 \), \( 0 \leq i \leq 4 \)

    \[
    \begin{align*}
    w_{01} &= p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 8 \\
    c_{01} &= w_{01} + \min\{c_{00} + c_{11}\} = 8, \ r_{01} = 1 \\
    w_{12} &= p_2 + w_{11} + w_{22} = p_2 + q_2 + w_{11} = 7 \\
    c_{12} &= w_{12} + \min\{c_{11} + c_{22}\} = 7, \ r_{12} = 2 \\
    w_{23} &= p_3 + w_{22} + w_{33} = p_3 + q_3 + w_{22} = 3 \\
    c_{23} &= w_{23} + \min\{c_{22} + c_{33}\} = 3, \ r_{23} = 3 \\
    w_{34} &= p_4 + w_{33} + w_{44} = p_4 + q_4 + w_{33} = 3 \\
    c_{34} &= w_{34} + \min\{c_{33} + c_{44}\} = 3, \ r_{34} = 4
    \end{align*}
    \]
Optimal binary search trees

- $w_{ii} = q_i$
- $w_{ij} = p_k + w_{i,k-1} + w_{kj}$
- $c_{ij} = w_{ij} + \text{min} 1, l_i l_j l_{i,j}
- c_{ii} = 0$
- $r_{ii} = 0$
- $r_{ij} = l$

$(a_1, a_2, a_3, a_4) = (\text{do, for, void, while})$  
$(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$  
$(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$

Computation is carried out row-wise from row 0 to row 4

The optimal search tree as the result
Chain Matrix Multiplication using Dynamic Programming
Matrix-Chain multiplication

• We are given a sequence

\[ \langle A_1, A_2, \ldots, A_n \rangle \]

• And we wish to compute

\[ A_1 A_2 \ldots A_n \]
Matrix-Chain multiplication (cont.)

- Matrix multiplication is associative, and so all parenthesizations yield the same product.
- For example, if the chain of matrices is \( A_1 A_2 \ldots A_4 \) then the product \( A_1 A_2 A_3 A_4 \) can be fully paranthesized in five distinct way:

\[
\begin{align*}
&(A_1 (A_2 (A_3 A_4))) \\
&(A_1 ((A_2 A_3) A_4)) \\
&((A_1 A_2 )(A_3 A_4)) \\
&((A_1 (A_2 A_3)) A_4) \\
&(((A_1 A_2 ) A_3) A_4) \\
\end{align*}
\]
Matrix-Chain multiplication

MATRIX-MULTIPLY (A, B)

if columns [A] ≠ rows [B]
   then error “incompatible dimensions”
else for i ← 1 to rows [A]
   do for j ← 1 to columns [B]
      do C[i, j] ← 0
      do for k ← 1 to columns [A]
         do C[i, j] ← C[i, j] + A[i, k]*B[k, j]
return C
Matrix-Chain multiplication (cont.)

Cost of the matrix multiplication:

An example: \( \langle A_1 A_2 A_3 \rangle \)

\( A_1 : \ 10 \times 100 \)

\( A_2 : \ 100 \times 5 \)

\( A_3 : \ 5 \times 50 \)
Matrix-Chain multiplication (cont.)

If we multiply \(((A_1A_2)A_3)\) we perform \(10 \cdot 100 \cdot 5 = 5000\) scalar multiplications to compute the \(10 \times 5\) matrix product \(A_1A_2\), plus another \(10 \cdot 5 \cdot 50 = 2500\) scalar multiplications to multiply this matrix by \(A_3\), for a total of \(7500\) scalar multiplications.

If we multiply \((A_1A_2A_3)\) we perform \(100 \cdot 5 \cdot 50 = 25000\) scalar multiplications to compute the \(100 \times 50\) matrix product \(A_2A_3\), plus another \(10 \cdot 100 \cdot 50 = 50000\) scalar multiplications to multiply \(A_1\) by this matrix, for a total of \(75000\) scalar multiplications.
Matrix-Chain multiplication (cont.)

- The problem:
  Given a chain \( \langle A_1, A_2, \ldots, A_n \rangle \) of \( n \) matrices, where matrix \( A_i \) has dimension \( p_{i-1} \times p_i \), fully paranthesize the product \( A_1 A_2 \ldots A_n \) in a way that minimizes the number of scalar multiplications.
Matrix-Chain multiplication (cont.)

Step 1: The structure of an optimal parenthesization (op)

- Find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems.
- Let $A_{i...j}$ denote the matrix product $A_i A_{i+1} ... A_j$.
- Any parenthesization of $A_i A_{i+1} ... A_j$ must split the product between $A_k$ and $A_{k+1}$ for $i \leq k < j$. 
Matrix-Chain multiplication (cont.)

Step 2: A recursive solution:

• Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i...j}$
• Thus, the cost of a cheapest way to compute $A_{1...n}$ would be $m[1,n]$. 
Recursive definition for the minimum cost of paranthesization:

\[
m[i, j] = \begin{cases} 
0 & \text{if } i = j \\
\min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{if } i < j.
\end{cases}
\]
Matrix-Chain multiplication (cont.)

That is $s[i, j]$ equals a value $k$ such that

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

$$s[i, j] = k$$
Matrix-Chain multiplication (cont.)

Step 3: Computing the optimal costs

It is easy to write a recursive algorithm based on recurrence for computing $m[i,j]$.

We compute the optimal cost by using a tabular, bottom-up approach.
Matrix-Chain multiplication (Contd.)

MATRIX-CHAIN-ORDER(\(p\))

\(n \leftarrow \text{length}[p] - 1\)

\(\text{for } i \leftarrow 1 \text{ to } n\)

\(\quad \text{do } m[i,i] \leftarrow 0\)

\(\text{for } l \leftarrow 2 \text{ to } n\)

\(\quad \text{do for } i \leftarrow 1 \text{ to } n-l+1\)

\(\quad \quad \text{do } j \leftarrow i+l-1\)

\(\quad \quad \quad m[i,j] \leftarrow \infty\)

\(\quad \quad \text{for } k \leftarrow i \text{ to } j-1\)

\(\quad \quad \quad \text{do } q \leftarrow m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\)

\(\quad \quad \quad \text{if } q < m[i,j]\)

\(\quad \quad \quad \text{then } m[i,j] \leftarrow q\)

\(\quad \quad \quad s[i,j] \leftarrow k\)

\text{return } m \text{ and } s
Matrix-Chain multiplication (cont.)

An example:

<table>
<thead>
<tr>
<th>matrix</th>
<th>dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>30 x 35</td>
</tr>
<tr>
<td>$A_2$</td>
<td>35 x 15</td>
</tr>
<tr>
<td>$A_3$</td>
<td>15 x 5</td>
</tr>
<tr>
<td>$A_4$</td>
<td>5 x 10</td>
</tr>
<tr>
<td>$A_5$</td>
<td>10 x 20</td>
</tr>
<tr>
<td>$A_6$</td>
<td>20 x 25</td>
</tr>
</tbody>
</table>

$m[2,5] = \min\begin{cases}m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 100 + 35 \cdot 5 \cdot 20 = 7125 \\m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$

$= (7125)$
Matrix-Chain multiplication (cont.)
Step 4: Constructing an optimal solution

An optimal solution can be constructed from the computed information stored in the table $s[1...n, 1...n]$.  

Final Answer : $((A1(A2*A3)) ((A4*A5)A6))$
Matrix-Chain multiplication (Contd.)

RUNNING TIME:

Matrix-chain order yields a running time of $O(n^3)$