WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.
Matrix-chain multiplication

**Matrix-chain multiplication**

**Input:** A sequence (chain) of matrices $M_0, \ldots, M_{n-1}$, where $M_i$ has dimension $d_i \times d_{i+1}$

**Output:** A parenthesization that results in the smallest number of multiplications of pairs of values

Possible parenthesizations: $(M_0M_1)M_2$ and $M_0(M_1M_2)$
Approach 3: Divide-and-conquer

What if we knew that the last multiplication in an optimal solution was $(M_0 \ldots M_k)(M_{k+1} \ldots M_{n-1})$?

Then the total cost would be:

- The optimal cost of forming $A$ by multiplying $M_0$ through $M_k$.
- The optimal cost of forming $B$ by multiplying $M_{k+1}$ through $M_{n-1}$.
- The cost of multiplying $A$ and $B$.

Observations:

- The costs of forming $A$ and $B$ have to be the optimal costs, because otherwise there could be a smaller total cost.
- The cost of multiplying $A$ and $B$ will be $d_0 d_{k+1} d_n$, because $A$ is a $d_0 \times d_{k+1}$ matrix and $B$ is a $d_{k+1} \times d_n$ matrix.

Key observation: The best solution to the bigger problem must be formed from the best solutions to the smaller problems.
Divide-and-conquer, continued

Issues to resolve:

1. How do we handle the fact that we don’t know $k$?
2. How do we determine the optimal costs of forming $A$ and $B$?

Ideas:

1. Try all possible values of $k$.
2. Use the same algorithm for each subproblem.

For each $i$ and $j$, $m[i,j]$ is the optimal cost of multiplying $M_i \ldots M_j$.

Problems:

- We end up trying all possibilities.
- We recompute some pieces more than once.
Paradigm: Dynamic programming

Features:
- Define big problems in terms of smaller ones of the same form.
- Solve small problems and store the solutions.
- Build solutions to bigger problems out of stored solutions.

Works when:
- Bigger problems can be defined in terms of smaller problems.
- Optimal solutions to bigger problems are formed from optimal solutions to smaller problems.
- Problems can be ordered so that smaller problems are solved before their solutions are needed to solve bigger problems.

History:
- Popularized by Richard Bellman by work started in 1955
- “Programming” means use of a tabular solution method
Dynamic programming recipe

Recipe for dynamic programming:

1. Characterize an optimal solution.
2. Define a subproblem in terms of smaller subproblems.
3. Determine what information should be stored in each table entry.
4. Determine the base cases.
5. Choose an order of evaluation.
6. Determine the shape of the table or tables needed to store the solutions to the smaller problems.
7. Extract the solution from the table.
Details of order of evaluation

Fill in increasing order of $j-i$ (diagonals)
Within each diagonal, work in increasing order of $j$
First diagonal has $i = j$, costs are all 0.
Solution is at top right corner

Step 6: Determine the shape of the table or tables needed to store the solutions to the smaller problems.
A triangle (or a grid).
Filling in diagonal $j - i = 1$

In each entry, store:

- minimum value, and
- which value $k$ results in that value.

Recall: $m[i, j] = \min_k \{ m[i, k] + m[k + 1, j] + d_i d_{k+1} d_{j+1} \}$

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & \begin{array}{c} 0 \hspace{1cm} 600 \\
\text{k=0} \end{array} & & \\
1 & \begin{array}{c} 0 \\
\text{k=1} \end{array} & 900 & \\
2 & \begin{array}{c} 0 \hspace{1cm} 90 \\
\text{k=2} \end{array} & & \\
3 & & & 0 \\
\end{array}
\]

\[d_0 = 2, \; d_1 = 10, \; d_2 = 30, \; d_3 = 3, \; d_4 = 1\]

\[
m[0, 1] = m[0, 0] + m[1, 1] + d_0 d_1 d_2 = 600
\]

\[
m[1, 2] = m[1, 1] + m[2, 2] + d_1 d_2 d_3 = 900
\]

\[
m[2, 3] = m[2, 2] + m[3, 3] + d_2 d_3 d_4 = 90\]
Filling in diagonal $j - i = 2$

Recall: $m[i, j] = \min_k \{m[i, k] + m[k + 1, j] + d_i d_{k+1} d_{j+1}\}$

$d_0 = 2, d_1 = 10, d_2 = 30, d_3 = 3, d_4 = 1$

Try values $k = 0$ and $k = 1$ for $m[0, 2]$:

$m[0, 2] = \min \begin{cases} m[0, 0] + m[1, 2] + d_0 d_1 d_3 = 960 \\ m[0, 1] + m[2, 2] + d_0 d_2 d_3 = 780 \end{cases}$

Try values $k = 1$ and $k = 2$ for $m[1, 3]$:

$m[1, 3] = \min \begin{cases} m[1, 1] + m[2, 3] + d_1 d_2 d_4 = 390 \\ m[1, 2] + m[3, 3] + d_1 d_3 d_4 = 930 \end{cases}$
Filling in diagonal $j - i = 3$

Try values $k = 0$, $k = 1$, and $k = 2$ for $m[0, 3]$:

$$m[0, 0] + m[1, 3] + d_0 d_1 d_4 = 410$$

$$m[0, 1] + m[2, 3] + d_0 d_2 d_4 = 750$$

$$m[0, 2] + m[3, 3] + d_0 d_3 d_4 = 786$$

$m[0, 3]$ uses $k = 0$ to split $M_0$ and $M_1$ through $M_3$

$m[1, 3]$ uses $k = 1$ to split $M_1$ and $M_2$ through $M_3$

Solution: $M_0(M_1(M_2M_3))$
Dynamic programming analysis

Correctness:

1. Show that a larger optimal solution is formed from smaller optimal solutions.

2. Show that the order of evaluation guarantees that values have been entered in the table before they are needed.

Space: Determine the number of table entries.

Time:

1. Determine the cost of filling entries in the table (typically the product of the number of entries and the cost of filling in each entry)

2. Determine the cost of tracing back through the table to extract the solution.
Expressing the solution in terms of other solutions

Common methods based on types of inputs:

**Set** Determine an order on the elements. Smaller problems are defined using smaller subsets of the elements. Bigger problems are formed by adding elements one at a time.

**Sequence or Grid** Define a problem in terms of the position or positions in the sequence. Smaller problems are at earlier positions. Bigger problems are at later positions.

**Tree** Define a problem on a subtree. Smaller problems are on smaller subtrees. Bigger problems are on bigger subtrees.
Determining what information should be stored in each table entry

Common types of information stored:

**Decision problem**  True or False (solutions to smaller problems)

**Evaluation problem**  Values (solutions to smaller problems)

**Search or constructive problem**  Information indicating which smaller problems led to the optimal solution

Because the cost of the algorithm will depend on the amount of information stored in each entry (cost of calculation as well as cost of reading time), often full solutions are not stored.

Instead, they can be reconstructed in asymptotically the same amount of time required to fill in the table.
Common table shapes

1D  Typically used when each problem has only one variable.

2 1D  Typically used when each problem has two variables, but bigger problems depend only on one-smaller values of one variable. Examples: the input naturally has two variables (e.g. grids or graphs) or there are two inputs.

2D  Typically used when each problem has two variables, and bigger problems depend on more than just one-smaller values of one variable. Sometimes a grid is used, and sometimes a triangle if, for example, $M[i,j] = M[j,i]$ or one not defined.

2 2D  Typically used in situations like for 2 1D tables, but this time for problems with three variables.

3D or higher  Typically used when problems have three or more variables.

For a tree, often nodes are ordered from leaves to root.
Longest common subsequence

A **subsequence** of a string $x_0x_1\ldots x_n$ is any string $x_{i_1}x_{i_2}\ldots x_{i_j}$ such that $0 \leq i_1 \leq i_2 \leq \cdots \leq i_j \leq n$.

Example: The string ”abcde” is a subsequence of the string ”000a0b0000cd0000e”.

**Longest common subsequence**

**Input:** A string $X$ of length $m$ and a string $Y$ of length $n$

**Output:** A string $Z$ that is a subsequence of both $X$ and $Y$ and of maximum length
Step 1: Characterize an optimal solution.

Suppose $Z$ is an optimal solution. Where in $X$ and $Y$ can the last symbol in $Z$ be found?

- Matching the last symbols in both $X$ and $Y$.

- Matching a non-last symbol in $X$.

- Matching a non-last symbol in $Y$. 
Defining subproblems
Step 2: Define a subproblem in terms of smaller subproblems

We’ll use Python slice notation to represent smaller strings, such as

- $Z[: k]$ is the first $k$ symbols (positions 0 to $k – 1$)
- $Z[k :]$ is all but the first $k$ symbols (positions $k$ and on)
- $Z[j : k]$ is symbols $j$ to $k – 1$

Suppose $Z[: k]$ is an LCS of $X[: i]$ and $Y[: j]$.

Claim

Case 1 $x_{i–1} = y_{j–1} \Rightarrow Z[: k – 1]$ is an LCS of $X[: i – 1]$ and $Y[: j – 1]$.

Case 2 $x_{i–1} \neq y_{j–1}$ and $x_{i–1} \neq z_{k–1} \Rightarrow Z[: k]$ is an LCS of $X[: i – 1]$ and $Y[: j]$.

Case 3 $x_{i–1} \neq y_{j–1}$ and $y_{j–1} \neq z_{k–1} \Rightarrow Z[: k]$ is an LCS of $X[: i]$ and $Y[: j – 1]$.

For $C[i, j]$ the length of the LCS of $X[: i]$ and $Y[: j]$:

$C[i, j] = C[i – 1, j – 1] + 1$ if $x_{i–1} = y_{j–1}$

$C[i, j] = \max\{C[i – 1, j], C[i, j – 1]\}$ otherwise
Steps 3 through 5

Step 3: Determine what information should be stored in each table entry.

Store length of subsequence as well as whether the best value of $C[i,j]$ was found using $C[i-1,j-1]$, $C[i-1,j]$, or $C[i,j-1]$.

Step 4: Determine the base cases.

Use value 0 to denote empty string, so $C[0,0] = C[0,j] = C[i,0] = 0$

Step 5: Choose an order of evaluation.

$C[i,j] = C[i-1,j-1] + 1$ if $x_{i-1} = y_{j-1}$

$C[i,j] = \max\{C[i-1,j], C[i,j-1]\}$ if $x_{i-1} = y_{j-1}$

We need to know $C[i-1,j-1]$, $C[i-1,j]$, and $C[i,j-1]$ before $C[i,j]$.

Use increasing values of $i$ and then increasing values of $j$ (hence row by row and then column by column).
Completion and analysis

Step 6: Determine the shape of the table or tables needed to store the solutions to the smaller problems.

A grid of dimensions $(m + 1) \times (n + 1)$

Step 7: Extract the solution from the table.

$C[m,n]$ contains the length of the longest common subsequence of the inputs.

To extract the sequence, store information on which of the three values led to the best answer.

Analysis:

Space: $O(mn)$

Time: $O(mn)$
Implementation details

Order of evaluation by diagonal:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Issues to consider:

- How many diagonals are there?
- Given a diagonal and a column, how can you figure out the row?
- For a given diagonal, what is the first column to evaluate?
- For a given diagonal, what is the last column to evaluate?

See sample code on website for calculating just the value and also calculating the value and extracting the subsequence.
All-pairs cheapest paths

**Input:** A graph $G$ with non-negative edge weights

**Output:** The least-cost paths between each pair of vertices in $G$

Note: Other methods needed when there can be negative cycles.

Naïve method: Use Dijkstra $n$ times.

Floyd-Warshall dynamic programming algorithm
Knapsack (or 0-1 knapsack)

Note: In this version, each entire object is either included in or excluded from the knapsack.

**Knapsack**

**Input:** A set of $n$ types of objects, where object $i$ has positive integer weight $w_i$ and positive integer value $v_i$, and an integer weight bound $W$

**Output:** A subset of objects with total weight at most $W$ and the maximum total value
Options for dynamic programming on graphs

Idea:

- Removing a set of one or more vertices to break a graph into smaller graphs
- Solve problems on smaller graphs that can be combined to solve the original problem on a larger graph
- Solutions to additional problems on smaller graphs may also be needed

Challenges in splitting up a graph:

- Using a large set of vertices to break up the graph may make it expensive to account for solutions that involve vertices in the set
- Breaking a graph into a large number of smaller graphs may make it expensive to combine the information in the smaller graphs
Comparing paradigms

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Exhaustive</th>
<th>Greedy</th>
<th>D-and-C</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subproblems</td>
<td>All</td>
<td>Some</td>
<td>Some</td>
<td>Some</td>
</tr>
<tr>
<td>Applicability</td>
<td>Wide</td>
<td>Narrow</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Speed</td>
<td>Slow</td>
<td>Fast</td>
<td>Medium</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Divide-and-conquer:
- assumes you know which problems are needed
- works top down
- may repeat subproblems

Dynamic programming:
- does not assume you know which subproblems are needed
- works bottom up (top down version **memoization** exists)
- does not repeat subproblems
Module summary

Topics covered:

- Matrix-chain multiplication
- Paradigm: Dynamic programming
- Dynamic programming recipe
- Solving matrix-chain multiplication
- Dynamic programming analysis
- General approaches to dynamic programming
- Longest common subsequence
- All-pairs cheapest paths
- Knapsack
- Finding the height of a tree
- Comparing paradigms