Lecture 17: Dynamic Programming

Agenda:

- Matrix-chain multiplication

Reading:

- Textbook pages 323 – 324, 331 – 339

Matrix-chain multiplication:

- Input: matrices A_1, A_2, ..., A_n with dimensions $d_0 \times d_1$, $d_1 \times d_2$, ..., $d_{n-1} \times d_n$, respectively.

- Output: an order in which matrices should be multiplied such that the product $A_1 \times A_2 \times \ldots \times A_n$ is computed using the minimum number of scalar multiplications.

- Fact: suppose A_1 is a $d_1 \times d_2$ matrix, A_2 is a $d_2 \times d_3$ matrix.

  Then A_1 and A_2 is multipliable, and $B = A_1 \times A_2$ can be computed using $d_1 \times d_2 \times d_3$ scalar multiplications.

- Example: $n = 4$ and $(d_0, d_1, \ldots, d_n) = (5, 2, 6, 4, 3)$

  Possible orders with different number of scalar multiplications:

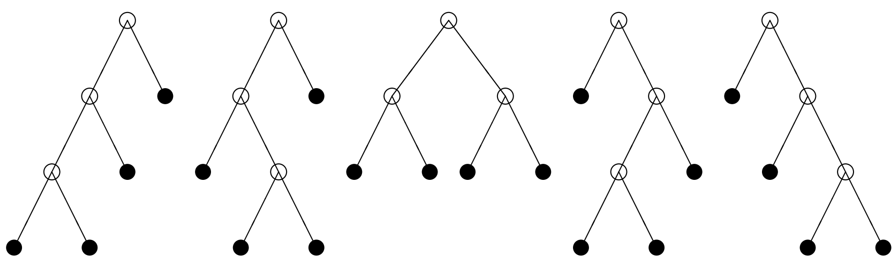
$((A_1 \times A_2) \times A_3) \times A_4$	$5 \times 2 \times 6 + 5 \times 6 \times 4 + 5 \times 4 \times 3 = 240$
$(A_1 \times (A_2 \times A_3)) \times A_4$	$5 \times 2 \times 4 + 2 \times 6 \times 4 + 5 \times 4 \times 3 = 148$
$(A_1 \times A_2) \times (A_3 \times A_4)$	$5 \times 2 \times 6 + 5 \times 6 \times 3 + 6 \times 4 \times 3 = 222$
$A_1 \times ((A_2 \times A_3) \times A_4)$	$5 \times 2 \times 3 + 2 \times 6 \times 4 + 2 \times 4 \times 3 = 102$
$A_1 \times (A_2 \times (A_3 \times A_4))$	$5 \times 2 \times 3 + 2 \times 6 \times 3 + 6 \times 4 \times 3 = 138$

1st Matrix-chain multiplication — brute force:

- a.k.a. *exhaustive enumeration* ...
- Let M_n be the number of multiplication orders
  How big is M_n ???

n	1	2	3	4	5	6	...
M_n	1	1	2	5	14	42	...

- Let C_n be the number of binary trees each with
  - $(n+1)$ leaves, n non-leaves
  - each non-leaf has two children (*full binary tree*)
  - for example $n = 3$:



n	0	1	2	3	4	5	...
C_n	1	1	2	5	14	42	...

C_n:

- These binary trees can be constructed recursively:

root:		1 non-leaf
left subtree:	$j+1$ leaves	j non-leaves
right subtree:	$n-j$ leaves	$n-j-1$ non-leaves

  $j = 0, 1, 2, \ldots, (n-1)$

- C_n — Catalan numbers (1983)

- $C_n = \begin{cases} 1, & \text{when } n = 0, 1 \\ \sum_{j=0}^{n-1} C_j \times C_{n-j-1}, & \text{when } n \geq 2 \end{cases}$

- $M_{n+1} = C_n = \frac{\binom{2n}{n}}{n+1} \approx \frac{4^n}{n\sqrt{\pi n}}$

- Therefore, the brute force approach running time $\in \Omega((4-\epsilon)^n)$!!!

2^{nd} implementation — recursion:

- Cannot afford exhaustive enumeration ...
- Try recursion?
  - $M(i,j)$ — the minimum number of scalar multiplications needed to compute product $A_i \times A_{i+1} \times \ldots \times A_j$ $(i \leq j)$
  - $M(i,j) = \begin{cases} 0, & \text{if } i = j \\ \min_{i \leq k < j}\{M(i,k) + M(k+1,j) + d_{i-1}d_kd_j\}, & \text{if } i < j \end{cases}$
  - for example,

$$M(1,4) = \min \left\{ \begin{array}{l} M(1,1) + M(2,4) + d_0 \times d_1 \times d_4 \\ M(1,2) + M(3,4) + d_0 \times d_2 \times d_4 \\ M(1,3) + M(4,4) + d_0 \times d_3 \times d_4 \end{array} \right\}$$

  - pseudocode:

```
procedure M(i, j)

if i = j then
    return 0
else
    cost ← ∞
    for t ← i to j − 1 do
        new ← M(i, t) + M(t + 1, j) + d_{i−1} × d_t × d_j
        if new < cost then
            cost ← new
return cost
```

  - running time: $n = |j - i|$

$$T(n) = \begin{cases} c_1, & \text{when } n = 0 \\ c_2 + \sum_{j=0}^{n-1} \left(T(j) + T(n-j-1)\right), & \text{when } n \geq 1 \end{cases}$$

2^{nd} implementation — recursion (cont'd):

- Solving the recurrence:

$$
\begin{aligned}
T(n) &= c_2 + \sum_{j=0}^{n-1} (T(j) + T(n-j-1)) \\
&= c_2 + 2\sum_{j=0}^{n-1} T(j) \\
&= \left(c_2 + 2\sum_{j=0}^{n-2} T(j)\right) + 2T(n-1) \\
&= T(n-1) + 2T(n-1) \\
&= 3T(n-1) \\
&= 3^2 T(n-2) \\
&= \ldots \\
&= 3^n T(0) \\
&= c_1 3^n
\end{aligned}
$$

- So, recursion running time $T(n) \in \Theta(3^n)$

- Again, lots of repeated function calls ...

- Try **memoization** — 3^{rd} approach
  An exercise !!!

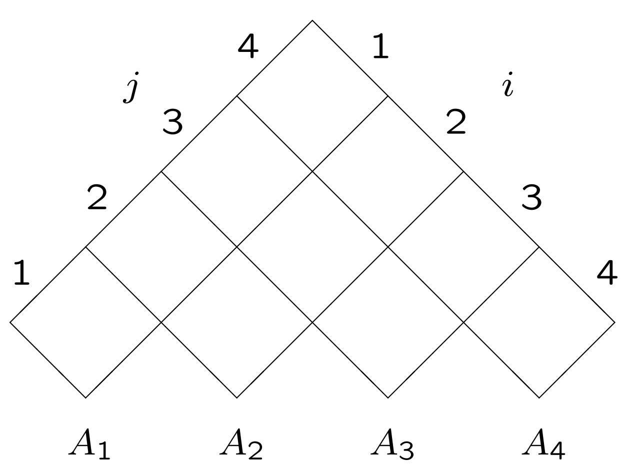
4^{th} implementation — dynamic programming:

- Pseudocode:

```
procedure dpM(1, n)

for i ← 1 to n do
    M(i, i) ← 0
for shift ← 1 to n do
    for i ← 1 to n − shift do
        j ← i + shift
        cost ← ∞
        for t ← i to j − 1 do
            new ← M(i, t) + M(t + 1, j) + d_{i−1} × d_t × d_j
            if new < cost then
                cost ← new
        M(i, j) ← cost
return M(1, n)
```

- Trace the example $n = 4$ and $(d_0, d_1, \ldots, d_n) = (5, 2, 6, 4, 3)$:



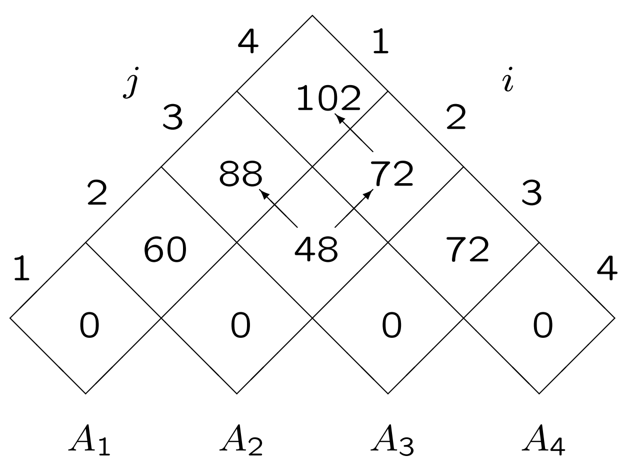
4^{th} implementation — dynamic programming:

- Pseudocode:

```
procedure dpM(1, n)

for i ← 1 to n do
    M(i,i) ← 0
for shift ← 1 to n do
    for i ← 1 to n − shift do
        j ← i + shift
        cost ← ∞
        for t ← i to j − 1 do
            new ← M(i,t) + M(t+1,j) + d_{i−1} × d_t × d_j
            if new < cost then
                cost ← new
        M(i,j) ← cost
return M(1, n)
```

- Trace the example $n = 4$ and $(d_0, d_1, \ldots, d_n) = (5, 2, 6, 4, 3)$:



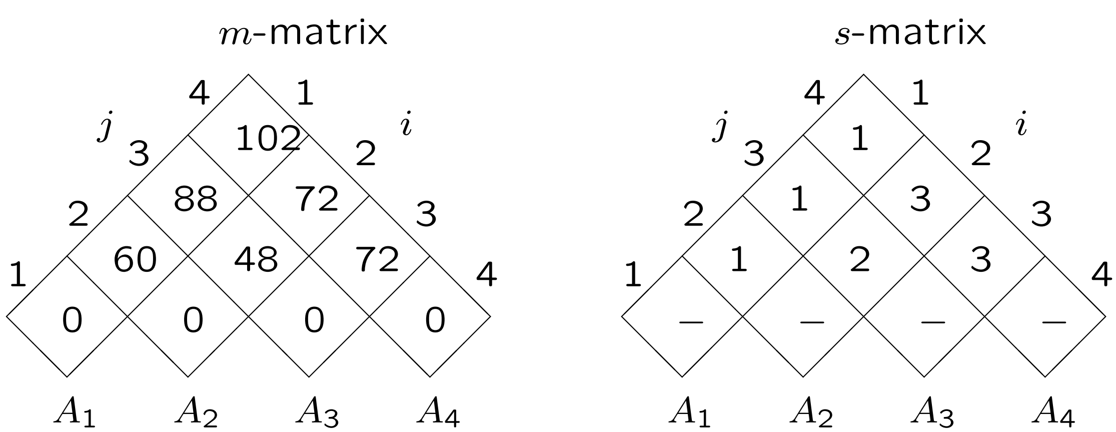
4^{th} implementation — dynamic programming:

- Pseudocode:

```
procedure dpM(1, n)

for i ← 1 to n do
    M(i, i) ← 0
for shift ← 1 to n do
    for i ← 1 to n − shift do
        j ← i + shift
        cost ← ∞
        for t ← i to j − 1 do
            new ← M(i, t) + M(t + 1, j) + d_{i−1} × d_t × d_j
            if new < cost then
                cost ← new
        M(i, j) ← cost
return M(1, n)
```

- Trace the example $n = 4$ and $(d_0, d_1, \ldots, d_n) = (5, 2, 6, 4, 3)$:



- The innermost `for` loopbody takes constant time ...
  So `dpM`(n) worst case running time $\in \Theta(n^3)$.

Have you understood the lecture contents?

well	ok	not-at-all	topic
☐	☐	☐	matrix-chain multiplication
☐	☐	☐	deriving recurrence
☐	☐	☐	avoiding re-computation
☐	☐	☐	memoization
☐	☐	☐	bottom-up —– dynamic programming

Dynamic programming key characteristics:

- Recurrence relation exists
- Recursive calls overlap
- Small number of subproblems
- Huge number of calls
- Avoid re-computation
- Bottom-up computation
- Top-down trace

Other problems suited to Dynamic programming:

- String matching: Longest Common Subsequence (next lecture)
- Optimal binary search tree construction (textbook page 356)
- All pair shortest paths in (di)graphs (CMPUT 304)
- Optimal layout in VLSI (could be a thesis topic :-))

Some more observations on Matrix-chain multiplication:

- Suppose we have computed the order of multiplications
- Suppose the last matrix multiplication is between $(A_1 \times \ldots \times A_j)$ and $(A_{j+1} \times \ldots \times A_n)$
- Then the suborders obtained from the original order are optimal orders for the subproblems, respectively (why ???)
- We call this ... *optimal substructures*
- Equivalently, we need to
  - compute optimal orders for
    - multiplying matrices $A_1, A_2, \ldots, A_j$
    - multiplying $A_{j+1}, A_{j+2}, \ldots, A_n$,
    - for every index $j = 1, 2, \ldots, (n-1)$
  - combine them into an order to multiplying $A_1, A_2, \ldots, A_n$
  - choose the best order out of the $(n-1)$ possibilities

Longest common subsequence (LCS) problem:

Definitions:
- Sequence/string:
  `dynamicprogramming` is a sequence over the English alphabet
- Base/letter/character
- Subsequence:
  the given sequence with zero or more bases left out
  e.g., `dog` is a subsequence of `dynamicprogramming`
  WARNing: bases appear in the same order, but not necessarily consecutive
- Common subsequence
- LCS problem: given two sequences $X = x_1 x_2 \ldots x_n$ and $Y = y_1 y_2 \ldots y_m$, find a maximum-length common subsequence of them.

- The LCS problem has the "optimal substructure" ...
  - if x_n is NOT in the LCS (to be computed), then we only need to compute an LCS of $x_1 x_2 \ldots x_{n-1}$ and $y_1 y_2 \ldots y_m$ ...
  - similarly, if y_m is NOT in the LCS (to be computed), then we only need to compute an LCS of $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_{m-1}$ ...
  - if x_n and y_m are both in the LCS (to be computed), then $x_n = y_m$ and we need to compute an LCS of $x_1 x_2 \ldots x_{n-1}$ and $y_1 y_2 \ldots y_{m-1}$;
    and then adding x_n to the end to form an LCS for the original problem

Longest common subsequence (LCS) problem (cont'd):

- Therefore,

  Letting $DP[n,m]$ to denote the length of an LCS of X and Y,

  $$DP[n,m] = \max \begin{cases} DP(x_1x_2\ldots x_{n-1}, y_1y_2\ldots y_m), \\ DP(x_1x_2\ldots x_n, y_1y_2\ldots y_{m-1}), \\ DP(x_1x_2\ldots x_{n-1}, y_1y_2\ldots y_{m-1}) + 1, \quad \text{if } x_n = y_m \end{cases}$$

- Correctness

- In general, let $DP[i,j]$ denote the length of an LCS of $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$.

- Recurrence:

  $$DP[i,j] = \max \begin{cases} DP[i-1,j], \\ DP[i,j-1], \\ DP[i-1,j-1] + 1, \quad \text{if } x_i = y_j \end{cases}$$

- Base cases ???

Longest common subsequence (LCS) problem (cont'd)

— solving the recurrence:

- Divide-and-Conquer running time: $\Omega(3^{\min\{n,m\}})$
- Memoization: $\Theta(n \times m)$
- Dynamic programming:

Order of computations ???

```
procedure dpLCS(X, Y)

n ← length[X]
m ← length[Y]
for i ← 1 to m do
    DP(i, 0) ← 0
for j ← 0 to n do
    DP(0, j) ← 0
for i ← 1 to m do
    for j ← 1 to n do
        if x_i = y_j then
            DP[i, j] ← DP[i − 1, j − 1] + 1
        else if DP[i − 1, j] ≥ DP[i, j − 1] then
            DP[i, j] ← DP[i − 1, j]
        else
            DP[i, j] ← DP[i, j − 1]
return DP[n, m]
```

Longest common subsequence (LCS) problem (cont'd):

- Correctness
- Can return an associated LCS ... trace back
- Running time: $\Theta(n \times m)$

  There are $n \times m$ entries each takes constant time to compute.

  Can be reduced to $\Theta(n \times \frac{m}{\log m})$ (CMPUT 606)
- Space requirement ... $\Theta(n \times m)$

  Can be reduced to $\Theta(\min\{n, m\})$ (CMPUT 606)
- Applications:
  - Human (and other species) Genome Project
  - Detecting cheating :-)