

# Lecture 18: Dynamic Programming

## Agenda:

- Matrix-chain multiplication
- Dynamic programming: more characteristics
- Longest common subsequence

## Reading:

- Textbook pages 323 – 324, 331 – 350

## Lecture 18: Dynamic Programming

### Matrix-chain multiplication:

- Input: matrices  $A_1, A_2, \dots, A_n$  with dimensions  $d_0 \times d_1, d_1 \times d_2, \dots, d_{n-1} \times d_n$ , respectively.
- Output: an order in which matrices should be multiplied such that the product  $A_1 \times A_2 \times \dots \times A_n$  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  $(d_0, d_1, \dots, d_n) = (5, 2, 6, 4, 3)$

Possible orders with different number of scalar multiplications:

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

- An equivalent question: How we put the parentheses?

## Lecture 18: Dynamic Programming

2<sup>nd</sup> implementation — recursion:

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

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

2<sup>nd</sup> implementation — recursion (cont'd):

- Solving the recurrence:

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

- So, recursion running time  $T(n) \in \Theta(3^n)$
- Again, lots of repeated function calls ...
- Try **memoization** — 3<sup>rd</sup> approach  
An exercise !!!

## Lecture 18: Dynamic Programming

4<sup>th</sup> implementation — dynamic programming:

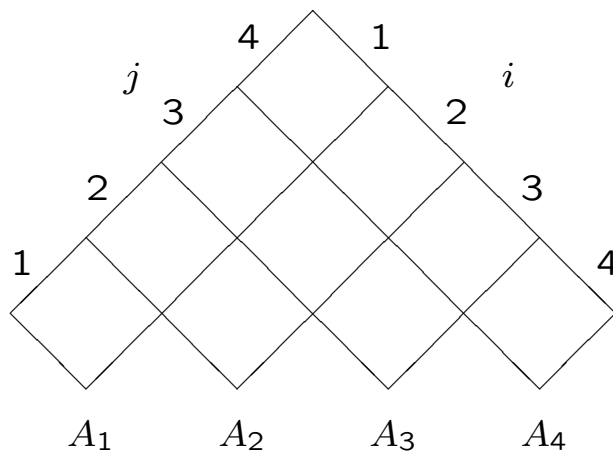
- Pseudocode:

```

procedure dpM(1, n)

for  $i \leftarrow 1$  to  $n$  do
   $M(i, i) \leftarrow 0$ 
for  $shift \leftarrow 1$  to  $n$  do
  for  $i \leftarrow 1$  to  $n - shift$  do
     $j \leftarrow i + shift$ 
     $cost \leftarrow \infty$ 
    for  $t \leftarrow i$  to  $j - 1$  do
       $new \leftarrow M(i, t) + M(t + 1, j) + d_{i-1} \times d_t \times d_j$ 
      if  $new < cost$  then
         $cost \leftarrow new$ 
     $M(i, j) \leftarrow cost$ 
return  $M(1, n)$ 
  
```

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



## Lecture 18: Dynamic Programming

4<sup>th</sup> implementation — dynamic programming:

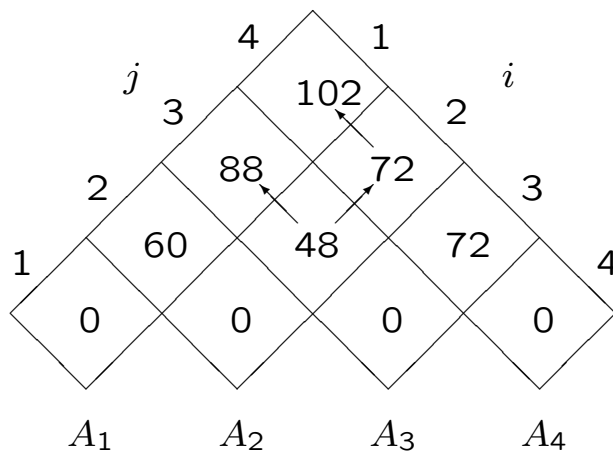
- Pseudocode:

```

procedure dpM(1, n)

for  $i \leftarrow 1$  to  $n$  do
   $M(i, i) \leftarrow 0$ 
for  $shift \leftarrow 1$  to  $n$  do
  for  $i \leftarrow 1$  to  $n - shift$  do
     $j \leftarrow i + shift$ 
     $cost \leftarrow \infty$ 
    for  $t \leftarrow i$  to  $j - 1$  do
       $new \leftarrow M(i, t) + M(t + 1, j) + d_{i-1} \times d_t \times d_j$ 
      if  $new < cost$  then
         $cost \leftarrow new$ 
     $M(i, j) \leftarrow cost$ 
return  $M(1, n)$ 
  
```

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



## Lecture 18: Dynamic Programming

### 4<sup>th</sup> implementation — dynamic programming:

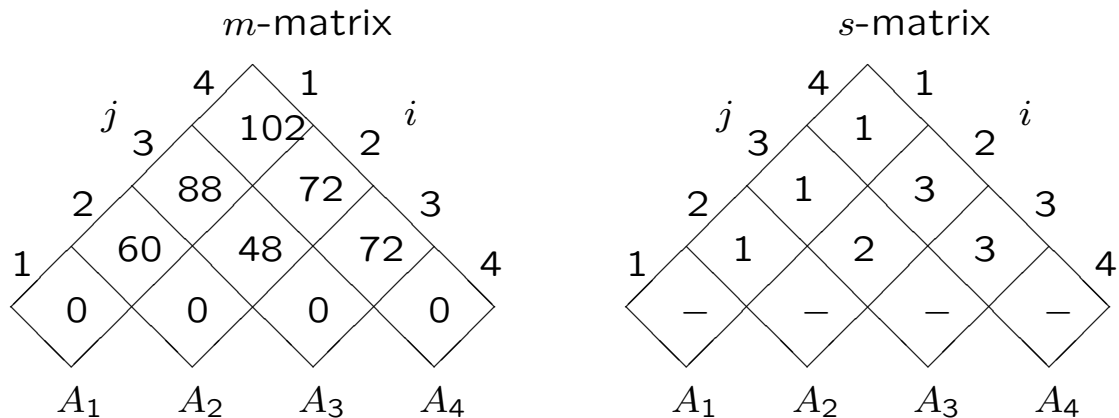
- Pseudocode:

```

procedure dpM(1, n)

for  $i \leftarrow 1$  to  $n$  do
   $M(i, i) \leftarrow 0$ 
for  $shift \leftarrow 1$  to  $n$  do
  for  $i \leftarrow 1$  to  $n - shift$  do
     $j \leftarrow i + shift$ 
     $cost \leftarrow \infty$ 
    for  $t \leftarrow i$  to  $j - 1$  do
       $new \leftarrow M(i, t) + M(t + 1, j) + d_{i-1} \times d_t \times d_j$ 
      if  $new < cost$  then
         $cost \leftarrow new$ 
     $M(i, j) \leftarrow cost$ 
return  $M(1, n)$ 
  
```

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



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

## Lecture 18: 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 :-))



## Lecture 19: Dynamic Programming

### Some more observations on Matrix-chain multiplication:

- Suppose we have computed the order of multiplications
- Suppose the last matrix multiplication is between  $(A_1 \times \dots \times A_j)$  and  $(A_{j+1} \times \dots \times A_n)$
- 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  $A_1, A_2, \dots, A_j$
    - \* multiplying  $A_{j+1}, A_{j+2}, \dots, A_n$ ,
    - \* for every index  $j = 1, 2, \dots, (n - 1)$
  - combine them into an order to multiplying  $A_1, A_2, \dots, A_n$
  - choose the best order out of the  $(n - 1)$  possibilities

## Lecture 19: Dynamic Programming

### 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  $X = x_1x_2\dots x_n$  and  $Y = y_1y_2\dots y_m$ , 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  $x_1x_2\dots x_{n-1}$  and  $y_1y_2\dots y_m$  ...
    - similarly, if  $y_m$  is NOT in the LCS (to be computed), then we only need to compute an LCS of  $x_1x_2\dots x_n$  and  $y_1y_2\dots y_{m-1}$  ...
    - 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  $x_1x_2\dots x_{n-1}$  and  $y_1y_2\dots y_{m-1}$ ;  
and then adding  $x_n$  to the end to form an LCS for the original problem

## Lecture 19: Dynamic Programming

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

- Therefore,

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

$$\text{max length of } \begin{cases} LCS(x_1x_2 \dots x_{n-1}, y_1y_2 \dots y_m), \\ LCS(x_1x_2 \dots x_n, y_1y_2 \dots y_{m-1}), \\ LCS(x_1x_2 \dots x_{n-1}, y_1y_2 \dots y_{m-1}) + 'x'_n, \quad \text{if } x_n = y_m \end{cases}$$

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

## Lecture 19: Dynamic Programming

### 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 xi = yj 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]
```

## Lecture 19: Dynamic Programming

### 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 :-)

## Lecture 19: Dynamic Programming

Have you understood the lecture contents?

well	ok	not-at-all	topic
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	matrix-chain multiplication
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	deriving recurrence
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	avoiding re-computation
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	top-down — memoization
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	bottom-up — dynamic programming
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	optimal substructure
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	LCS computation