Lecture 18: Dynamic Programming

Agenda:

- Matrix-chain multiplication
- Longest common subsequence

Reading:

- Textbook pages 323 – 324, 331 – 350

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.

- Let $T(n)$ be the number of multiplication orders for n matrices.

- There are $n-1$ possiblities for the highest level paranthesis: $(A_1 \ldots A_i)(A_{i+1} \ldots A_n)$ i can be anyware between 1 to $n-1$.

- The number of ways to put parentheses for each of $(A_1 \ldots A_i)$ and $(A_{i+1} \ldots A_n)$ is $T(i)$ and $T(n-i)$, respectively.

- Therefore:

$$T(n) = \begin{cases} 1, & \text{when } n = 0, 1 \\ \sum_{i=1}^{n-1} T(i) \times T(n-i), & \text{when } n \geq 2 \end{cases}$$

- It can be shown that $T(n) \in \Omega(\frac{4^n}{n\sqrt{\pi n}})$.

Use dynamic programming:

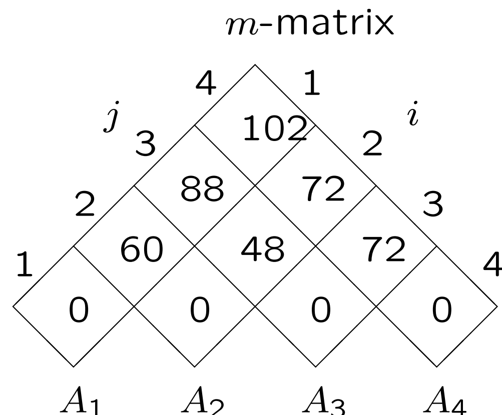
- Step 1: Define $M[i,j]$ $(1 \le i \le j)$: the minimum number of scalar multiplications needed to compute product $A_i \times A_{i+1} \times \ldots \times A_j$ $(i \le j)$
- Step 2: The recurrence to fill in the entries of the array:

$$M[i,j] = \begin{cases} 0, & \text{if } i = j \\ \min_{i \le k < j}\{M[i,k] + M[k+1,j] + d_{i-1}d_k d_j\}, & \text{if } i < j \end{cases}$$

- Step 3:

```
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
        M[i, j] ← ∞
        for k ← i to j − 1 do
            new ← M[i, k] + M[k + 1, j] + d_{i−1} × d_k × d_j
            if new < M[i, j] then
                M[i, j] ← new
return M[1, n]
```

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



Dynamic programming:

- To obtain the actual ordering:

```
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
        M[i, j] ← ∞
        for k ← i to j − 1 do
            new ← M[i, k] + M[k + 1, j] + d_{i−1} × d_k × d_j
            if new < M[i, j] then
                M[i, j] ← new
                S[i, j] ← k
return M[1, n]
```

- We call Print-Opt-Order $(S, 1, n)$:

```
procedure Print-Opt-Order(S, i, j)

If i = j then
    Print ("A_i")
Else
    Print ``(``
        Print-Opt-Order (S, i, S[i, j])
        Print-Opt-Order (S, S[i, j] + 1, j)
        Print ``)''
```

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

m-matrix

i \ j	1	2	3	4
1	0	60	88	102
2		0	48	72
3			0	72
4				0

s-matrix

i \ j	1	2	3	4
1	–	1	1	1
2		–	2	3
3			–	3
4				–

A_1 A_2 A_3 A_4

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

- Some final observations:
  - Suppose we have computed the order of multiplications
  - And the last multiplication is between $(A_1 \times \ldots \times A_j)$ and $(A_{j+1} \times \ldots \times A_n)$
  - Then the suborders $(A_1 \times \ldots \times A_j)$ and $(A_{j+1} \times \ldots \times A_n)$ are optimal orders for the subproblems, respectively.
  - 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 $1 \le j \le n - 1$, and combine them into an order to multiplying $A_1, A_2, \ldots, A_n$
  - choose the best order out of the $(n - 1)$ possibilities

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 :-))

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

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

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

- Correctness

- In general, let $DP[i, j]$ denote the length of an LCS of $x_1x_2\ldots x_i$ 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: $\Omega(3^{\min\{n,m\}})$
- 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 n do
    for j ← 1 to m 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 :-)

Have you understood the lecture contents?

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