Lecture 17: Dynamic Programming

Agenda:

- Knapsack (cont'd)
- Matrix-chain multiplication

Reading:

- Textbook pages 323 – 324, 331 – 339

2nd example problem — Knapsack

- A knapsack with capacity C
- n items with weights $w_1, \ldots, w_n \in \mathbb{N}$ and values $v_1, \ldots, v_n \in \mathbb{N}$.
- For each $S \subseteq \{1, \ldots, n\}$ let $K(S) = \sum_{i \in S} w_i$.
- Find $M = \max_{S \subseteq \{1,\ldots,n\}} \{K(S) | K(S) \leq C\}$.
- Step 1: Define array $A[i, D]$, $0 \leq i \leq n$ and $0 \leq D \leq C$. $A[i, D]$ is the value of best possible knapsack of weight at most D using only items from 1 to i. Final solution value: $A[n, C]$.
- Step 2: If $i = 0$ or $D = 0$ then trivially $A[i, D] = 0$.

  Else, $A[i, D] = \max \left\{ \begin{array}{l} A[i-1, D] \\ \text{(if } D \geq w_i) \; v_i + A[i-1, D-w_i] \end{array} \right.$.
- Step 3: Pseudocode:

```
procedure Knapsack

for i ← 0 to n do
    A[i, 0] ← 0
for D ← 0 to C do
    A[0, D] ← 0
for i ← 1 to n do
    for D ← 1 to C do
        A[i, D] ← A[i − 1, D]
        if D ≥ w_i and A[i, D] < A[i − 1, D − w_i] + v_i then
            A[i, D] ← A[i − 1, D − w_i] + v_i
return A[n, C]
```

2nd example problem — Knapsack

- How to find the set of items of the optimal packing?
- After each update of $A[i, D]$ store in another array $B[i, D]$ wether or not i is included in the solution of $A[i, D]$.
- `procedure Knapsack`
  `** ` $B[i, D]$ `indicates whether` i `is in the solution of` $A[i, D]$

```
for i ← 1 to n do
    A[i, 0] ← 0
for D ← 0 to C do
    A[0, D] ← 0
for i ← 1 to n do
    for D ← 1 to C do
        A[i, D] ← A[i − 1, D]
        B[i, D] ← NO
        if D ≥ w_i and A[i, D] < A[i − 1, D − w_i] + v_i then
            A[i, D] ← A[i − 1, D − w_i] + v_i
            B[i, D] ← YES
return A[n, C]
```

- `procedure Print-Opt-Knapsack` (i, D)

```
If i = 0 or D = 0 then
    return
Else
    If B[i, D] = YES then
        Print (i)
        Print-Opt-Knapsack (i − 1, D − w_i)
    else if B[i, D] = NO then
        Print-Opt-Knapsack (i − 1, D)
```

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 — Recursion:

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

  How big is $T(n)$???

n	1	2	3	4	5	6	...
$T(n)$	1	1	2	5	14	42	...

- Consider the highest level parenthesis: $(A_1 \dots A_i)(A_{i+1} \dots A_n)$.

- There are $n-1$ possiblities: i can be anyware between 1 to $n-1$ (e.g $A_1(A_2 \dots A_n)$ to $(A_1 \dots A_{n-1})A_n$.

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

- Therefore:

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

- Solving this recurrence (not easy) shows: $T(n) = \frac{\binom{2n}{n}}{n+1} \approx \frac{4^n}{n\sqrt{\pi n}}$

- Recursive program will have similar running time: $\Omega(3^n)$.

- Cannot afford this!!

Use dynamic programming:

- Step 1: Define $M[i,j]$ $(1 \leq i \leq j)$: the minimum number of scalar multiplications needed to compute product $A_i \times A_{i+1} \times \ldots \times A_j$ $(i \leq j)$

- Step 2: The recurrence to fill in the entries of the array:

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

- for example,

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

- Step 3: Pseudocode (to obtain the optimal cost):

```
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]
```

4^{th} implementation — dynamic programming:

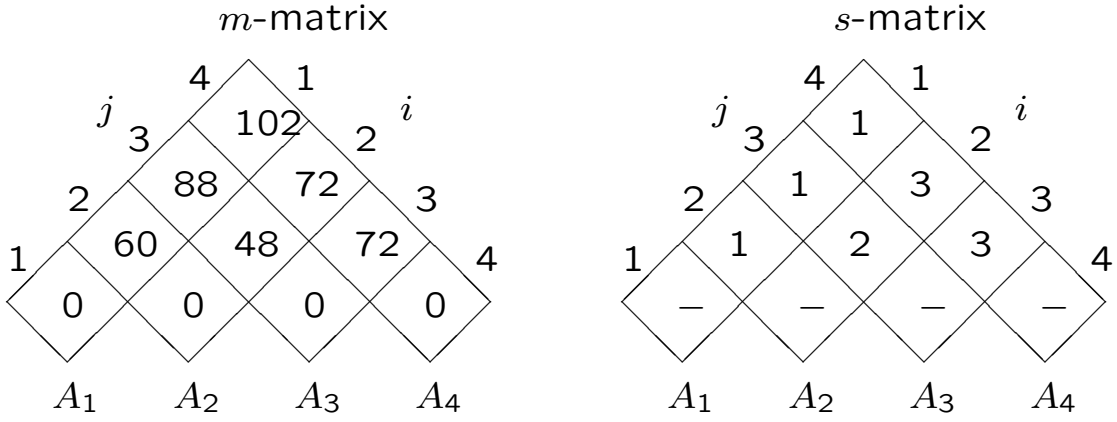
- To obtain the actual ordering:

```
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 $(d_0, d_1, \ldots, d_n) = (5, 2, 6, 4, 3)$:



- 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 $1 \leq j \leq n-1$, 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

Have you understood the lecture contents?

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