CMPSC 465
Data Structures and Algorithms
Spring 2022

Instructor: Chunhao Wang

Dynamic Programming

Dynamic Programming

Edit Distance (Textbook Section 6.3)

Running example

$x = ACGTA$ and $y = ATCTG$

Running example

$x = ACGTA$ and $y = ATCTG$

	A	T	C	T	G
A					
C					
G					
T					
A					

Running example

$x = ACGTA$ and $y = ATCTG$

$$
\begin{array}{cccccccccccc}
 & & & A & & T & & C & & T & & G \\
 & 0 & & 1 & & 2 & & 3 & & 4 & & 5 \\
 & & \searrow & & & & & & & & & \\
A & 1 & & 0 & \rightarrow & 1 & & 2 & & 3 & & 4 \\
 & & & & \searrow & & \searrow & & & & & \\
C & 2 & & 1 & & 1 & & 1 & & 2 & & 3 \\
 & & & & & & \searrow & \downarrow & & & & \\
G & 3 & & 2 & & 2 & & 2 & & 2 & & 2 \\
 & & & & & & & & \searrow & & & \\
T & 4 & & 3 & & 2 & & 3 & & 2 & & 3 \\
 & & & & & & & & & & \searrow & \\
A & 5 & & 4 & & 3 & & 3 & & 3 & & 3
\end{array}
$$

Pseudocode

def $\textsc{Edit\_Distance}(x, y)$**:**

Pseudocode

```
def EDIT_DISTANCE(x, y):
    for i = 0, ..., m:
```

Pseudocode

```
def EDIT_DISTANCE(x, y):
    for i = 0, ..., m:
        E(i, 0) = i;
```

Pseudocode

```
def EDIT_DISTANCE(x, y):
    for i = 0, ..., m:
        E(i, 0) = i;
    for j = 0, ..., n:
```

Pseudocode

```
def EDIT_DISTANCE(x, y):
    for i = 0, ..., m:
        E(i, 0) = i;
    for j = 0, ..., n:
        E(0, j) = j;
```

Pseudocode

```
def EDIT_DISTANCE(x, y):
    for i = 0, ..., m:
        E(i, 0) = i;
    for j = 0, ..., n:
        E(0, j) = j;
    for i = 1, ..., m:
```

Pseudocode

```
def EDIT_DISTANCE(x, y):
    for i = 0, ..., m:
        E(i, 0) = i;
    for j = 0, ..., n:
        E(0, j) = j;
    for i = 1, ..., m:
        for j = 1, ..., n:
```

Pseudocode

```
def EDIT_DISTANCE(x, y):
    for i = 0, ..., m:
        E(i, 0) = i;
    for j = 0, ..., n:
        E(0, j) = j;
    for i = 1, ..., m:
        for j = 1, ..., n:
            E(i, j) =
              min{1+E(i-1, j), 1+E(i, j-1), diff(i, j)+E(i-1, j-1)};
```

Pseudocode

```
def EDIT_DISTANCE(x, y):
    for i = 0, ..., m:
        E(i, 0) = i;
    for j = 0, ..., n:
        E(0, j) = j;
    for i = 1, ..., m:
        for j = 1, ..., n:
            E(i, j) =
             min{1 + E(i − 1, j), 1 + E(i, j − 1), diff(i, j) + E(i − 1, j − 1)};
    return E(m, n);
```

Pseudocode

def $\textsc{Edit\_Distance}(x, y)$:

```
for i = 0, ..., m:
    E(i, 0) = i;
for j = 0, ..., n:
    E(0, j) = j;
for i = 1, ..., m:
    for j = 1, ..., n:
        E(i, j) =
          min{1 + E(i−1, j), 1 + E(i, j−1), diff(i, j) + E(i−1, j−1)};
return E(m, n);
```

Running time: $O(mn)$

Finding the alignment

We use an extra table prev to record where each entry of $E(i, j)$ was coming from:

Finding the alignment

We use an extra table prev to record where each entry of $E(i,j)$ was coming from:

$$\mathrm{prev}(i,j) = \begin{cases} (i-1,j) & \text{if } E(i,j) = 1 + E(i-1,j) \\ (i,j-1) & \text{if } E(i,j) = 1 + E(i,j-1) \\ (i-1,j-1) & \text{if } E(i,j) = \mathrm{diff}(i,j) + E(i-1,j-1) \end{cases}$$

Finding the alignment

We use an extra table prev to record where each entry of $E(i, j)$ was coming from:

$$\mathrm{prev}(i,j) = \begin{cases} (i-1, j) & \text{if } E(i,j) = 1 + E(i-1,j) \\ (i, j-1) & \text{if } E(i,j) = 1 + E(i,j-1) \\ (i-1, j-1) & \text{if } E(i,j) = \mathrm{diff}(i,j) + E(i-1,j-1) \end{cases}$$

def $\textsc{Pring\_Alignment}(x, y, \mathrm{prev})$**:**

Finding the alignment

We use an extra table prev to record where each entry of $E(i, j)$ was coming from:

$$\mathrm{prev}(i, j) = \begin{cases} (i-1, j) & \text{if } E(i, j) = 1 + E(i-1, j) \\ (i, j-1) & \text{if } E(i, j) = 1 + E(i, j-1) \\ (i-1, j-1) & \text{if } E(i, j) = \mathrm{diff}(i, j) + E(i-1, j-1) \end{cases}$$

def $\textsc{Pring\_Alignment}(x, y, \mathrm{prev})$**:**

  Set $i = m, j = n$;

Finding the alignment

We use an extra table prev to record where each entry of $E(i,j)$ was coming from:

$$\mathrm{prev}(i,j) = \begin{cases} (i-1,j) & \text{if } E(i,j) = 1 + E(i-1,j) \\ (i,j-1) & \text{if } E(i,j) = 1 + E(i,j-1) \\ (i-1,j-1) & \text{if } E(i,j) = \mathrm{diff}(i,j) + E(i-1,j-1) \end{cases}$$

def $\textsc{Pring\_Alignment}(x, y, \mathrm{prev})$**:**

 Set $i = m, j = n$;

 while $i \geq 1$ *and* $j \geq 1$**:**

  if $\mathrm{prev}(i,j) = (i-1, j-1)$**:**

Finding the alignment

We use an extra table prev to record where each entry of $E(i,j)$ was coming from:

$$\mathrm{prev}(i,j) = \begin{cases} (i-1,j) & \text{if } E(i,j) = 1 + E(i-1,j) \\ (i,j-1) & \text{if } E(i,j) = 1 + E(i,j-1) \\ (i-1,j-1) & \text{if } E(i,j) = \mathrm{diff}(i,j) + E(i-1,j-1) \end{cases}$$

def $\textsc{Pring\_Alignment}(x, y, \mathrm{prev})$**:**

- Set $i = m, j = n$;
- **while** $i \geq 1$ *and* $j \geq 1$**:**
  - **if** $\mathrm{prev}(i,j) = (i-1, j-1)$**:**
    - $\mathrm{print\_back}\binom{y_i}{x_i}$;
    - $i = i-1$, $j = j-1$;

Finding the alignment

We use an extra table prev to record where each entry of $E(i,j)$ was coming from:

$$\mathrm{prev}(i,j) = \begin{cases} (i-1,j) & \text{if } E(i,j) = 1 + E(i-1,j) \\ (i,j-1) & \text{if } E(i,j) = 1 + E(i,j-1) \\ (i-1,j-1) & \text{if } E(i,j) = \mathrm{diff}(i,j) + E(i-1,j-1) \end{cases}$$

def $\textsc{Pring\_Alignment}(x, y, \mathrm{prev})$**:**

  Set $i = m, j = n$;

  while $i \geq 1$ *and* $j \geq 1$**:**

    if $\mathrm{prev}(i,j) = (i-1, j-1)$**:**

      $\mathrm{print\_back}\binom{y_i}{x_i}$;

      $i = i-1$, $j = j-1$;

    if $\mathrm{prev}(i,j) = (i-1, j)$**:**

Finding the alignment

We use an extra table prev to record where each entry of $E(i,j)$ was coming from:

$$\mathrm{prev}(i,j) = \begin{cases} (i-1,j) & \text{if } E(i,j) = 1 + E(i-1,j) \\ (i,j-1) & \text{if } E(i,j) = 1 + E(i,j-1) \\ (i-1,j-1) & \text{if } E(i,j) = \mathrm{diff}(i,j) + E(i-1,j-1) \end{cases}$$

```
def PRING_ALIGNMENT(x, y, prev):
    Set i = m, j = n;
    while i ≥ 1 and j ≥ 1:
        if prev(i, j) = (i − 1, j − 1):
            print_back(y_i / x_i);
            i = i − 1, j = j − 1;
        if prev(i, j) = (i − 1, j):
            print_back(- / x_i);
            i = i − 1;
```

Finding the alignment

We use an extra table prev to record where each entry of $E(i,j)$ was coming from:

$$\mathrm{prev}(i,j) = \begin{cases} (i-1,j) & \text{if } E(i,j) = 1 + E(i-1,j) \\ (i,j-1) & \text{if } E(i,j) = 1 + E(i,j-1) \\ (i-1,j-1) & \text{if } E(i,j) = \mathrm{diff}(i,j) + E(i-1,j-1) \end{cases}$$

def $\textsc{Pring\_Alignment}(x, y, \mathrm{prev})$**:**

  Set $i = m, j = n$;

  while $i \geq 1$ *and* $j \geq 1$**:**

    if $\mathrm{prev}(i,j) = (i-1, j-1)$**:**

      $\mathrm{print\_back}\binom{y_i}{x_i}$;

      $i = i-1$, $j = j-1$;

    if $\mathrm{prev}(i,j) = (i-1, j)$**:**

      $\mathrm{print\_back}\binom{-}{x_i}$;

      $i = i-1$;

    if $\mathrm{prev}(i,j) = (i, j-)$**:**

Finding the alignment

We use an extra table prev to record where each entry of $E(i,j)$ was coming from:

$$\mathrm{prev}(i,j) = \begin{cases} (i-1,j) & \text{if } E(i,j) = 1 + E(i-1,j) \\ (i,j-1) & \text{if } E(i,j) = 1 + E(i,j-1) \\ (i-1,j-1) & \text{if } E(i,j) = \mathrm{diff}(i,j) + E(i-1,j-1) \end{cases}$$

```
def PRING_ALIGNMENT(x, y, prev):
    Set i = m, j = n;
    while i ≥ 1 and j ≥ 1:
        if prev(i, j) = (i − 1, j − 1):
            print_back(y_i over x_i);
            i = i − 1, j = j − 1;
        if prev(i, j) = (i − 1, j):
            print_back(- over x_i);
            i = i − 1;
        if prev(i, j) = (i, j−):
            print_back(y_j over -);
            j = j − 1;
```

Dynamic Programming

0-1 Knapsack (Textbook Section 6.4)

0-1 Knapsack

0-1 Knapsack Problem

A Thief has a backpack with certain capacity. There is a set of items with certain weight and value. **Goal:** pack the backpack with the largest value

0-1 Knapsack

0-1 Knapsack Problem

A Thief has a backpack with certain capacity. There is a set of items with certain weight and value. **Goal:** pack the backpack with the largest value

- Doesn't have the greedy choice property

0-1 Knapsack

0-1 Knapsack Problem

A Thief has a backpack with certain capacity. There is a set of items with certain weight and value. **Goal:** pack the backpack with the largest value

- Doesn't have the greedy choice property
- But it has the optimal substructure property:

0-1 Knapsack

0-1 Knapsack Problem

A Thief has a backpack with certain capacity. There is a set of items with certain weight and value. **Goal:** pack the backpack with the largest value

- Doesn't have the greedy choice property
- But it has the optimal substructure property:
  Suppose the optimal packing has weight $\leq W$. If we remove item j from it, the remaining packing must be the optimal packing for capacity $W - w_j$ with items excluding j

Subproblem

- **Subproblem**: $K(w, j)$ — the maximum value achievable using a backpack of capacity w and items $1, \ldots, j$

Subproblem

- **Subproblem**: $K(w, j)$ — the maximum value achievable using a backpack of capacity w and items $1, \ldots, j$
- **Optimal solution:** $K(W, n)$

Subproblem

- **Subproblem**: $K(w, j)$ — the maximum value achievable using a backpack of capacity w and items $1, \ldots, j$
- **Optimal solution:** $K(W, n)$
- **Recurrence:**

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

Subproblem

- **Subproblem**: $K(w, j)$ — the maximum value achievable using a backpack of capacity w and items $1, \ldots, j$
- **Optimal solution:** $K(W, n)$
- **Recurrence:**

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

- **Base case:** $K(0, j) = 0$ for all j and $K(w, 0) = 0$ for all w

Pseudocode

def KNAPSACK(W, w, v):

Pseudocode

def $\textsc{Knapsack}(W, w, v)$**:**

Set $K(0, j) = 0, K(w, 0) = 0$ for all j, w;

Pseudocode

def $\textsc{Knapsack}(W, w, v)$**:**

Set $K(0, j) = 0, K(w, 0) = 0$ for all j, w;

for $j = 1, \ldots, n$**:**

Pseudocode

def $\textsc{Knapsack}(W, w, v)$**:**

Set $K(0, j) = 0, K(w, 0) = 0$ for all j, w;

for $j = 1, \ldots, n$**:**

for $w = 1, \ldots, W$**:**

Pseudocode

```
def KNAPSACK(W, w, v):
    Set K(0, j) = 0, K(w, 0) = 0 for all j, w;
    for j = 1, ..., n:
        for w = 1, ..., W:
            if w_j > w:
```

Pseudocode

```
def KNAPSACK(W, w, v):
    Set K(0, j) = 0, K(w, 0) = 0 for all j, w;
    for j = 1, ..., n:
        for w = 1, ..., W:
            if w_j > w:
                K(w, j) = K(w, j − 1);
```

Pseudocode

```
def KNAPSACK(W, w, v):
    Set K(0, j) = 0, K(w, 0) = 0 for all j, w;
    for j = 1, ..., n:
        for w = 1, ..., W:
            if w_j > w:
                K(w, j) = K(w, j − 1);
            else:
```

Pseudocode

def $\textsc{Knapsack}(W, w, v)$**:**

Set $K(0, j) = 0, K(w, 0) = 0$ for all j, w;

for $j = 1, \ldots, n$**:**

for $w = 1, \ldots, W$**:**

if $w_j > w$**:**

$K(w, j) = K(w, j - 1)$;

else:

$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$;

Pseudocode

```
def KNAPSACK(W, w, v):
    Set K(0, j) = 0, K(w, 0) = 0 for all j, w;
    for j = 1, ..., n:
        for w = 1, ..., W:
            if w_j > w:
                K(w, j) = K(w, j − 1);
            else:
                K(w, j) = max{K(w − w_j, j − 1) + v_j, K(w, j − 1)};
    return K(W, n);
```

Pseudocode

```
def KNAPSACK(W, w, v):
    Set K(0, j) = 0, K(w, 0) = 0 for all j, w;
    for j = 1, ..., n:
        for w = 1, ..., W:
            if w_j > w:
                K(w, j) = K(w, j - 1);
            else:
                K(w, j) = max{K(w - w_j, j - 1) + v_j, K(w, j - 1)};
    return K(W, n);
```

Running time: $O(nW)$

Pseudocode

```
def Knapsack(W, w, v):
    Set K(0, j) = 0, K(w, 0) = 0 for all j, w;
    for j = 1, ..., n:
        for w = 1, ..., W:
            if w_j > w:
                K(w, j) = K(w, j − 1);
            else:
                K(w, j) = max{K(w − w_j, j − 1) + v_j, K(w, j − 1)};
    return K(W, n);
```

Running time: $O(nW)$

Question: is this a polynomial-time algorithm?

Pseudocode

```
def KNAPSACK(W, w, v):
    Set K(0, j) = 0, K(w, 0) = 0 for all j, w;
    for j = 1, ..., n:
        for w = 1, ..., W:
            if w_j > w:
                K(w, j) = K(w, j − 1);
            else:
                K(w, j) = max{K(w − w_j, j − 1) + v_j, K(w, j − 1)};
    return K(W, n);
```

Running time: $O(nW)$

Question: is this a polynomial-time algorithm? No!

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w \backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Running example

Example: $W = 10$

item	1	2	3	4
w_j	6	3	4	2
v_j	30	14	16	9

The K table:

$w\backslash j$	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	9
3	0	0	14	14	14
4	0	0	14	16	16
5	0	0	14	16	23
6	0	30	30	30	30
7	0	30	30	30	30
8	0	30	30	30	39
9	0	30	44	44	44
10	0	30	44	46	46

Dynamic Programming

Chain matrix multiplication (Textbook Section 6.5)

Chain matrix multiplication

We have n matrices $M_1, M_2, \ldots, M_n$

Chain matrix multiplication

We have n matrices $M_1, M_2, \ldots, M_n$
Need to compute

$$M_1 \cdot M_2 \cdots M_n$$

Chain matrix multiplication

We have n matrices $M_1, M_2, \ldots, M_n$

Need to compute

$$M_1 \cdot M_2 \cdots M_n$$

The dimensions of these matrices are:

$$M_1 \in \mathbb{R}^{m_0 \times m_1}, M_2 \in \mathbb{R}^{m_1 \times m_2}, \ldots, M_n \in \mathbb{R}^{m_{n-1} \times m_n}$$

Chain matrix multiplication

We have n matrices $M_1, M_2, \ldots, M_n$
Need to compute

$$M_1 \cdot M_2 \cdots M_n$$

The dimensions of these matrices are:

$$M_1 \in \mathbb{R}^{m_0 \times m_1}, M_2 \in \mathbb{R}^{m_1 \times m_2}, \ldots, M_n \in \mathbb{R}^{m_{n-1} \times m_n}$$

Recall if $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ then the cost for computing $A \cdot B$ is $m \cdot n \cdot p$

Chain matrix multiplication

We have n matrices $M_1, M_2, \ldots, M_n$
Need to compute

$$M_1 \cdot M_2 \cdots M_n$$

The dimensions of these matrices are:

$$M_1 \in \mathbb{R}^{m_0 \times m_1}, M_2 \in \mathbb{R}^{m_1 \times m_2}, \ldots, M_n \in \mathbb{R}^{m_{n-1} \times m_n}$$

Recall if $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ then the cost for computing $A \cdot B$ is $m \cdot n \cdot p$

Also, matrix multiplication is associative:
$A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$

Chain matrix multiplication

We have n matrices $M_1, M_2, \ldots, M_n$

Need to compute

$$M_1 \cdot M_2 \cdots M_n$$

The dimensions of these matrices are:

$$M_1 \in \mathbb{R}^{m_0 \times m_1}, M_2 \in \mathbb{R}^{m_1 \times m_2}, \ldots, M_n \in \mathbb{R}^{m_{n-1} \times m_n}$$

Recall if $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ then the cost for computing $A \cdot B$ is $m \cdot n \cdot p$

Also, matrix multiplication is associative:
$A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$

Question: what's the best way for computing $M_1 \cdot M_2 \cdots M_n$? i.e., where to put the parentheses?

Example of chain matrix multiplication

Consider $M_1 \in \mathbb{R}^{50\times 20}, M_2 \in \mathbb{R}^{20\times 1}, M_3 \in \mathbb{R}^{1\times 10}, M_4 = \mathbb{R}^{10\times 100}$

Example of chain matrix multiplication

Consider $M_1 \in \mathbb{R}^{50\times 20}, M_2 \in \mathbb{R}^{20\times 1}, M_3 \in \mathbb{R}^{1\times 10}, M_4 = \mathbb{R}^{10\times 100}$

There are many ways to do multiplication

Example of chain matrix multiplication

Consider $M_1 \in \mathbb{R}^{50\times 20}, M_2 \in \mathbb{R}^{20\times 1}, M_3 \in \mathbb{R}^{1\times 10}, M_4 = \mathbb{R}^{10\times 100}$

There are many ways to do multiplication

- $M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$

Example of chain matrix multiplication

Consider $M_1 \in \mathbb{R}^{50\times 20}, M_2 \in \mathbb{R}^{20\times 1}, M_3 \in \mathbb{R}^{1\times 10}, M_4 = \mathbb{R}^{10\times 100}$

There are many ways to do multiplication

- $M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$
  Cost: $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100 = 10200$

Example of chain matrix multiplication

Consider $M_1 \in \mathbb{R}^{50\times20}, M_2 \in \mathbb{R}^{20\times1}, M_3 \in \mathbb{R}^{1\times10}, M_4 = \mathbb{R}^{10\times100}$

There are many ways to do multiplication

- $M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$
  Cost: $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100 = 10200$
- $(M_1 \cdot ((M_2 \cdot M_3)) \cdot M_4$

Example of chain matrix multiplication

Consider $M_1 \in \mathbb{R}^{50\times 20}, M_2 \in \mathbb{R}^{20\times 1}, M_3 \in \mathbb{R}^{1\times 10}, M_4 = \mathbb{R}^{10\times 100}$

There are many ways to do multiplication

- $M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$
  Cost: $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100 = 10200$
- $(M_1 \cdot ((M_2 \cdot M_3)) \cdot M_4$
  Cost: $20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100 = 60200$

Example of chain matrix multiplication

Consider $M_1 \in \mathbb{R}^{50\times 20}, M_2 \in \mathbb{R}^{20\times 1}, M_3 \in \mathbb{R}^{1\times 10}, M_4 = \mathbb{R}^{10\times 100}$

There are many ways to do multiplication

- $M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$
  Cost: $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100 = 10200$
- $(M_1 \cdot ((M_2 \cdot M_3)) \cdot M_4$
  Cost: $20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100 = 60200$
- $(M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$

Example of chain matrix multiplication

Consider $M_1 \in \mathbb{R}^{50\times 20}, M_2 \in \mathbb{R}^{20\times 1}, M_3 \in \mathbb{R}^{1\times 10}, M_4 = \mathbb{R}^{10\times 100}$

There are many ways to do multiplication

- $M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$
  Cost: $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100 = 10200$
- $(M_1 \cdot ((M_2 \cdot M_3)) \cdot M_4$
  Cost: $20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100 = 60200$
- $(M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$
  Cost: $50 \cdot 20 \cdot 1 \cdot 10 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100 = 7000$

Example of chain matrix multiplication

Consider $M_1 \in \mathbb{R}^{50\times 20}, M_2 \in \mathbb{R}^{20\times 1}, M_3 \in \mathbb{R}^{1\times 10}, M_4 = \mathbb{R}^{10\times 100}$

There are many ways to do multiplication

- $M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$
  Cost: $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100 = 10200$
- $(M_1 \cdot ((M_2 \cdot M_3)) \cdot M_4$
  Cost: $20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100 = 60200$
- $(M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$
  Cost: $50 \cdot 20 \cdot 1 \cdot 10 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100 = 7000$

Goal: find a way to do multiplication with the minimum cost

- **Subproblem:**
  $C(i, j)$ — the minimum cost for multiplying $M_i, M_{i+1}, \ldots, M_j$

- **Subproblem:**
  $C(i, j)$ — the minimum cost for multiplying $M_i, M_{i+1}, \ldots, M_j$
- **Recurrence:**

Dynamic programming

- **Subproblem:**
  $C(i, j)$ — the minimum cost for multiplying $M_i, M_{i+1}, \ldots, M_j$
- **Recurrence:**

$$M_i M_{i+1} \cdots M_k \quad M_{k+1} M_{k+2} \cdots M_j$$

Dynamic programming

- **Subproblem:**
  $C(i, j)$ — the minimum cost for multiplying $M_i, M_{i+1}, \ldots, M_j$
- **Recurrence:**

$$\big(M_i M_{i+1} \cdots M_k\big)\big(M_{k+1} M_{k+2} \cdots M_j\big)$$

Dynamic programming

- **Subproblem:**
  $C(i, j)$ — the minimum cost for multiplying $M_i, M_{i+1}, \ldots, M_j$
- **Recurrence:**

$$\Big(\overset{m_{i-1} \times m_i}{M_i} M_{i+1} \cdots \overset{m_{k-1} \times m_k}{M_k}\Big)\Big(M_{k+1} M_{k+2} \cdots M_j\Big)$$

Dynamic programming

- **Subproblem:**
  $C(i, j)$ — the minimum cost for multiplying $M_i, M_{i+1}, \ldots, M_j$
- **Recurrence:**

$$\Big(\underbrace{\overset{m_{i-1} \times m_i}{M_i} M_{i+1} \cdots \overset{m_{k-1} \times m_k}{M_k}}_{m_{i-1} \times m_k}\Big)\Big(M_{k+1} M_{k+2} \cdots M_j\Big)$$

Dynamic programming

- **Subproblem:**
  $C(i, j)$ — the minimum cost for multiplying $M_i, M_{i+1}, \ldots, M_j$
- **Recurrence:**

$$\Big(\underbrace{\overset{m_{i-1}\times m_i}{M_i}\, M_{i+1}\cdots \overset{m_{k-1}\times m_k}{M_k}}_{m_{i-1}\times m_k}\Big)\Big(\underbrace{M_{k+1}M_{k+2}\cdots M_j}_{m_k\times m_j}\Big)$$

Dynamic programming

- **Subproblem:**
  $C(i,j)$ — the minimum cost for multiplying $M_i, M_{i+1}, \ldots, M_j$
- **Recurrence:**

$$\big(\underbrace{\overset{m_{i-1}\times m_i}{M_i} M_{i+1} \cdots \overset{m_{k-1}\times m_k}{M_k}}_{m_{i-1}\times m_k}\big)\big(\underbrace{M_{k+1} M_{k+2} \cdots M_j}_{m_k \times m_j}\big)$$

So, $\quad C(i,j) = \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1}\cdot m_k \cdot m_j\}$

Dynamic programming

- **Subproblem:**
  $C(i, j)$ — the minimum cost for multiplying $M_i, M_{i+1}, \ldots, M_j$
- **Recurrence:**

$$\underbrace{\Big(\overset{m_{i-1}\times m_i}{M_i} M_{i+1} \cdots \overset{m_{k-1}\times m_k}{M_k}\Big)}_{m_{i-1}\times m_k}\underbrace{\Big(M_{k+1} M_{k+2} \cdots M_j\Big)}_{m_k \times m_j}$$

So, $$C(i, j) = \min_{i \le k < j} \{C(i, k) + C(k+1, j) + m_{i-1} \cdot m_k \cdot m_j\}$$

- **Base case:** $C(i, i) = 0$

Dynamic programming

- **Subproblem:**
  $C(i,j)$ — the minimum cost for multiplying $M_i, M_{i+1}, \ldots, M_j$
- **Recurrence:**

$$\Big(\underbrace{\overset{m_{i-1}\times m_i}{M_i} M_{i+1} \cdots \overset{m_{k-1}\times m_k}{M_k}}_{m_{i-1}\times m_k}\Big)\Big(\underbrace{M_{k+1} M_{k+2} \cdots M_j}_{m_k \times m_j}\Big)$$

So, $$C(i,j) = \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

- **Base case:** $C(i,i) = 0$
- **Optimal solution:** $C(1,n)$

Pseudocode

def CHAIN_MATRIX(m):

Pseudocode

def CHAIN_MATRIX*(m)*:

  for $i = 1 \ldots n$:

Pseudocode

```
def CHAIN_MATRIX(m):
    for i = 1 ... n:
        C(i, i) = 0;
```

Pseudocode

def $\textsc{Chain\_Matrix}(m)$**:**

  for $i = 1 \ldots n$**:**

    $C(i, i) = 0;$

  for $s = 1 \ldots n - 1$**:**

Pseudocode

```
def CHAIN_MATRIX(m):
    for i = 1 ... n:
        C(i, i) = 0;
    for s = 1 ... n − 1:
        for i = 1 ... n − s:
```

Pseudocode

```
def CHAIN_MATRIX(m):
    for i = 1 ... n:
        C(i, i) = 0;
    for s = 1 ... n − 1:
        for i = 1 ... n − s:
            j = i + s;
```

Pseudocode

```
def CHAIN_MATRIX(m):
    for i = 1 ... n:
        C(i, i) = 0;
    for s = 1 ... n − 1:
        for i = 1 ... n − s:
            j = i + s;
            C(i, j) = min_{i≤k<j}{C(i, k), C(k + 1, j) + m_{i−1} · m_k · m_j};
```

Pseudocode

```
def CHAIN_MATRIX(m):
    for i = 1 ... n:
        C(i, i) = 0;
    for s = 1 ... n − 1:
        for i = 1 ... n − s:
            j = i + s;
            C(i, j) = min_{i≤k<j}{C(i, k), C(k + 1, j) + m_{i−1} · m_k · m_j};
    return C(1, n);
```

Pseudocode

```
def CHAIN_MATRIX(m):
    for i = 1 ... n:
        C(i, i) = 0;
    for s = 1 ... n − 1:
        for i = 1 ... n − s:
            j = i + s;
            C(i, j) = min_{i≤k<j}{C(i, k), C(k + 1, j) + m_{i−1} · m_k · m_j};
    return C(1, n);
```

Running time:

Pseudocode

```
def CHAIN_MATRIX(m):
    for i = 1 ... n:
        C(i, i) = 0;
    for s = 1 ... n − 1:
        for i = 1 ... n − s:
            j = i + s;
            C(i, j) = min_{i ≤ k < j} {C(i, k), C(k + 1, j) + m_{i−1} · m_k · m_j};
    return C(1, n);
```

Running time:

$O(n^2)$ entries to fill; $O(n)$ operations to fill in each entry

Pseudocode

def $\textsc{Chain\_Matrix}(m)$**:**

```
for i = 1 ... n:
    C(i, i) = 0;
for s = 1 ... n − 1:
    for i = 1 ... n − s:
        j = i + s;
        C(i, j) = min_{i≤k<j} {C(i, k), C(k + 1, j) + m_{i−1} · m_k · m_j};
return C(1, n);
```

Running time:

$O(n^2)$ entries to fill; $O(n)$ operations to fill in each entry

Total running time: $O(n^3)$