Find Common Subsequence

$i \leftarrow n, j \leftarrow m, S \leftarrow ()$
while $i > 0$ and $j > 0$ **do**
    if $\pi[i, j] =$ "↖" **then**
        add $A[i]$ to beginning of S, $i \leftarrow i - 1, j \leftarrow j - 1$
    else if $\pi[i, j] =$ "↑" **then**
        $i \leftarrow i - 1$
    else
        $j \leftarrow j - 1$
return S

Variants of Problem

Edit Distance with Insertions and Deletions

Input: a string A and a string B

each time we can delete a letter from A or insert a letter to A

Output: minimum number of operations (insertions or deletions) we need to change A to B?

Variants of Problem

Edit Distance with Insertions and Deletions

Input: a string A and a string B

each time we can delete a letter from A or insert a letter to A

Output: minimum number of operations (insertions or deletions) we need to change A to B?

Example:

- $A =$ ocurrance, $B =$ occurrence
- 3 operations: insert 'c', remove 'a' and insert 'e'

Variants of Problem

Edit Distance with Insertions and Deletions

Input: a string A and a string B

each time we can delete a letter from A or insert a letter to A

Output: minimum number of operations (insertions or deletions) we need to change A to B?

Example:

- $A =$ ocurrance, $B =$ occurrence
- 3 operations: insert 'c', remove 'a' and insert 'e'

Obs. $\#\text{OPs} = \text{length}(A) + \text{length}(B) - 2 \cdot \text{length}(\text{LCS}(A, B))$

Variants of Problem

Edit Distance with Insertions, Deletions and Replacing

Input: a string A and a string B

each time we can delete a letter from A, insert a letter to A or change a letter

Output: how many operations do we need to change A to B?

Variants of Problem

Edit Distance with Insertions, Deletions and Replacing

Input: a string A and a string B

each time we can delete a letter from A, insert a letter to A or change a letter

Output: how many operations do we need to change A to B?

Example:

- $A =$ ocurrance, $B =$ occurrence.
- 2 operations: insert 'c', change 'a' to 'e'

Variants of Problem

Edit Distance with Insertions, Deletions and Replacing

Input: a string A and a string B

each time we can delete a letter from A, insert a letter to A or change a letter

Output: how many operations do we need to change A to B?

Example:

- $A =$ ocurrance, $B =$ occurrence.
- 2 operations: insert 'c', change 'a' to 'e'

- Not related to LCS any more

Edit Distance with Replacing: Reduction to a Variant of LCS

- Need to match letters in A and B, every letter is matched at most once and there should be no crosses.
- However, we can match two different letters: Matching a same letter gives score 2, matching two different letters gives score 1.
- Need to maximize the score.
- DP recursion for the case $i > 0$ and $j > 0$:

$$opt[i,j] = \begin{cases} opt[i-1,j-1] + 2 & \text{if } A[i] = B[j] \\ \max\begin{cases} opt[i-1,j] \\ opt[i,j-1] \\ opt[i-1,j-1]+1 \end{cases} & \text{if } A[i] \neq B[j] \end{cases}$$

- Relation : #OPs = length(A) + length(B) - max_score

Edit Distance (with Replacing): using DP directly

- $opt[i,j], 0 \leq i \leq n, 0 \leq j \leq m$: edit distance between $A[1 .. i]$ and $B[1 .. j]$.

Edit Distance (with Replacing): using DP directly

- $opt[i,j], 0 \le i \le n, 0 \le j \le m$: edit distance between $A[1 .. i]$ and $B[1 .. j]$.
- if $i = 0$ then $opt[i,j] = j$; if $j = 0$ then $opt[i,j] = i$.

Edit Distance (with Replacing): using DP directly

- $opt[i,j], 0 \le i \le n, 0 \le j \le m$: edit distance between $A[1 .. i]$ and $B[1 .. j]$.
- if $i = 0$ then $opt[i,j] = j$; if $j = 0$ then $opt[i,j] = i$.
- if $i > 0, j > 0$, then

$$opt[i,j] = \begin{cases} & \text{if } A[i] = B[j] \\ & \text{if } A[i] \neq B[j] \end{cases}$$

Edit Distance (with Replacing): using DP directly

- $opt[i, j], 0 \le i \le n, 0 \le j \le m$: edit distance between $A[1 .. i]$ and $B[1 .. j]$.
- if $i = 0$ then $opt[i, j] = j$; if $j = 0$ then $opt[i, j] = i$.
- if $i > 0, j > 0$, then

$$opt[i, j] = \begin{cases} opt[i-1, j-1] & \text{if } A[i] = B[j] \\ & \text{if } A[i] \neq B[j] \end{cases}$$

Edit Distance (with Replacing): using DP directly

- $opt[i,j], 0 \le i \le n, 0 \le j \le m$: edit distance between $A[1 .. i]$ and $B[1 .. j]$.
- if $i = 0$ then $opt[i,j] = j$; if $j = 0$ then $opt[i,j] = i$.
- if $i > 0, j > 0$, then

$$opt[i,j] = \begin{cases} opt[i-1, j-1] & \text{if } A[i] = B[j] \\ \min \begin{cases} opt[i-1, j] + 1 \\ opt[i, j-1] + 1 \\ opt[i-1, j-1] + 1 \end{cases} & \text{if } A[i] \ne B[j] \end{cases}$$

Outline

Computing the Length of LCS

for $j \leftarrow 0$ to m **do**
    $opt[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    $opt[i, 0] \leftarrow 0$
    for $j \leftarrow 1$ to m **do**
        if $A[i] = B[j]$ **then**
            $opt[i, j] \leftarrow opt[i-1, j-1] + 1$
        else if $opt[i, j-1] \geq opt[i-1, j]$ **then**
            $opt[i, j] \leftarrow opt[i, j-1]$
        else
            $opt[i, j] \leftarrow opt[i-1, j]$

Obs. The i-th row of table only depends on $(i-1)$-th row.

Reducing Space to $O(n + m)$

Obs. The i-th row of table only depends on $(i - 1)$-th row.

Q: How to use this observation to reduce space?

Reducing Space to $O(n + m)$

Obs. The i-th row of table only depends on $(i - 1)$-th row.

Q: How to use this observation to reduce space?

A: We only keep two rows: the $(i - 1)$-th row and the i-th row.

Linear Space Algorithm to Compute Length of LCS

```
for j ← 0 to m do
    opt[0, j] ← 0
for i ← 1 to n do
    opt[i mod 2, 0] ← 0
    for j ← 1 to m do
        if A[i] = B[j] then
            opt[i mod 2, j] ← opt[i − 1 mod 2, j − 1] + 1
        else if opt[i mod 2, j − 1] ≥ opt[i − 1 mod 2, j] then
            opt[i mod 2, j] ← opt[i mod 2, j − 1]
        else
            opt[i mod 2, j] ← opt[i − 1 mod 2, j]
return opt[n mod 2, m]
```

How to Recover LCS Using Linear Space?

- Only keep the last two rows: only know how to match $A[n]$

How to Recover LCS Using Linear Space?

- Only keep the last two rows: only know how to match $A[n]$
- Can recover the LCS using n rounds: time $= O(n^2m)$

How to Recover LCS Using Linear Space?

- Only keep the last two rows: only know how to match $A[n]$
- Can recover the LCS using n rounds: time $= O(n^2m)$
- Using Divide and Conquer + Dynamic Programming:

How to Recover LCS Using Linear Space?

- Only keep the last two rows: only know how to match $A[n]$
- Can recover the LCS using n rounds: time $= O(n^2m)$
- Using Divide and Conquer + Dynamic Programming:
  - Space: $O(m + n)$

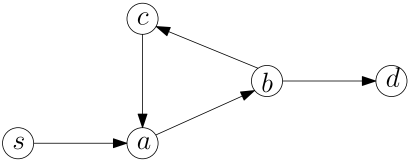
How to Recover LCS Using Linear Space?

- Only keep the last two rows: only know how to match $A[n]$
- Can recover the LCS using n rounds: time $= O(n^2m)$
- Using Divide and Conquer + Dynamic Programming:
  - Space: $O(m + n)$
  - Time: $O(nm)$

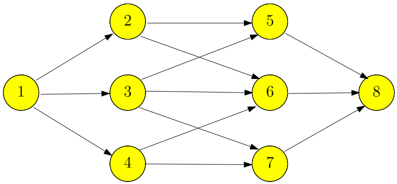
Outline

Directed Acyclic Graphs

Def. A directed acyclic graph (DAG) is a directed graph without (directed) cycles.



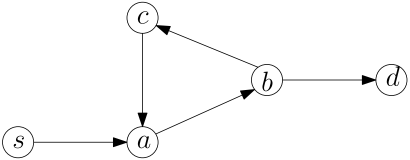
not a DAG



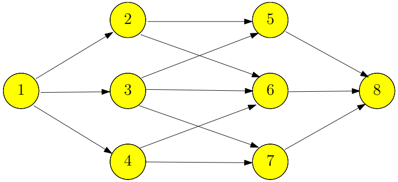
a DAG

Directed Acyclic Graphs

Def. A directed acyclic graph (DAG) is a directed graph without (directed) cycles.



not a DAG



a DAG

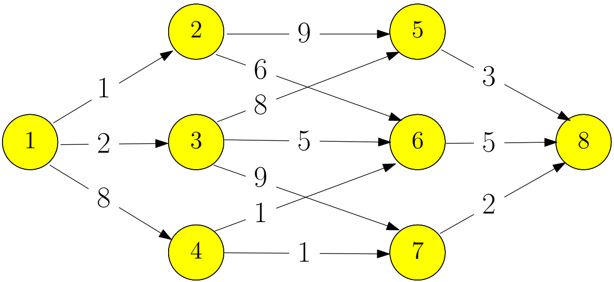
Lemma A directed graph is a DAG if and only its vertices can be topologically sorted.

Shortest Paths in DAG

Input: directed acyclic graph $G = (V, E)$ and $w : E \to \mathbb{R}$.

Assume $V = \{1, 2, 3 \cdots, n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

Output: the shortest path from 1 to i, for every $i \in V$

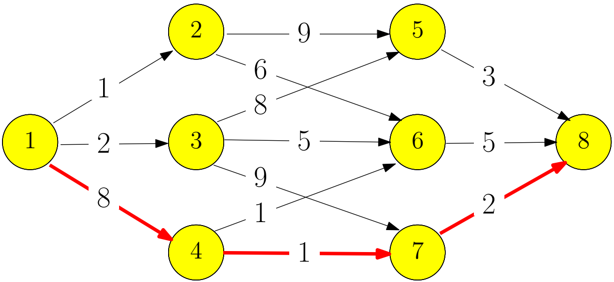


Shortest Paths in DAG

Input: directed acyclic graph $G = (V, E)$ and $w : E \to \mathbb{R}$.

Assume $V = \{1, 2, 3 \cdots, n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

Output: the shortest path from 1 to i, for every $i \in V$



Shortest Paths in DAG

- $f[i]$: length of the shortest path from 1 to i

$$f[i] = \begin{cases} & i = 1 \\ & i = 2, 3, \cdots, n \end{cases}$$

Shortest Paths in DAG

- $f[i]$: length of the shortest path from 1 to i

$$f[i] = \begin{cases} 0 & i = 1 \\ & i = 2, 3, \cdots, n \end{cases}$$

Shortest Paths in DAG

- $f[i]$: length of the shortest path from 1 to i

$$f[i] = \begin{cases} 0 & i = 1 \\ \min_{j:(j,i)\in E} \{f(j) + w(j,i)\} & i = 2, 3, \cdots, n \end{cases}$$

Shortest Paths in DAG

- Use an adjacency list for incoming edges of each vertex i

Shortest Paths in DAG

1: $f[1] \leftarrow 0$
2: **for** $i \leftarrow 2$ to n **do**
3:     $f[i] \leftarrow \infty$
4:     **for** each incoming edge (j, i) of i **do**
5:         **if** $f[j] + w(j, i) < f[i]$ **then**
6:             $f[i] \leftarrow f[j] + w(j, i)$

Shortest Paths in DAG

- Use an adjacency list for incoming edges of each vertex i

Shortest Paths in DAG

$f[1] \leftarrow 0$
for $i \leftarrow 2$ to n **do**
  $f[i] \leftarrow \infty$
  for each incoming edge (j, i) of i **do**
    if $f[j] + w(j, i) < f[i]$ **then**
      $f[i] \leftarrow f[j] + w(j, i)$
      $\pi(i) \leftarrow j$

Shortest Paths in DAG

- Use an adjacency list for incoming edges of each vertex i

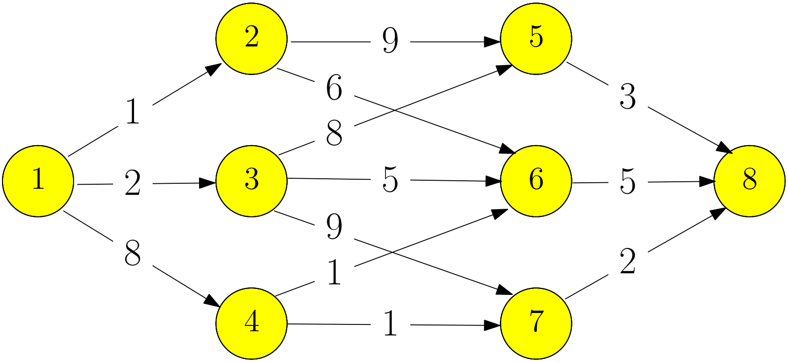
Shortest Paths in DAG

$f[1] \leftarrow 0$
for $i \leftarrow 2$ to n **do**
  $f[i] \leftarrow \infty$
  for each incoming edge (j, i) of i **do**
    if $f[j] + w(j, i) < f[i]$ **then**
      $f[i] \leftarrow f[j] + w(j, i)$
      $\pi(i) \leftarrow j$

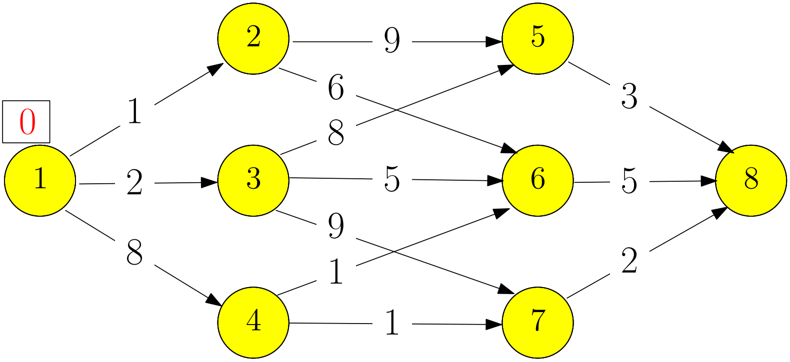
print-path(t)

if $t = 1$ **then**
  print(1)
  return
print-path($\pi(t)$)
print(",", t)

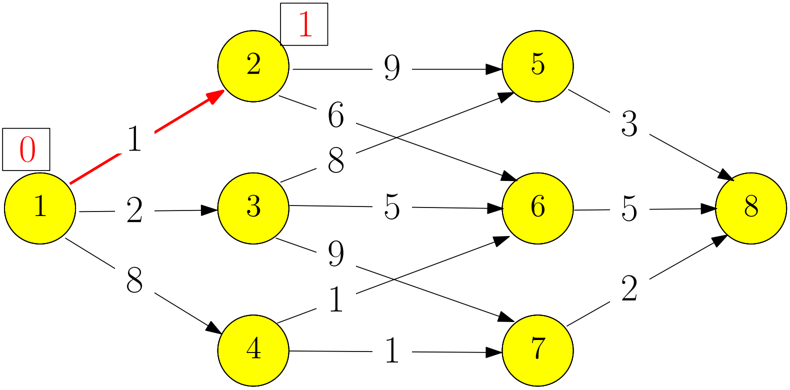
Example



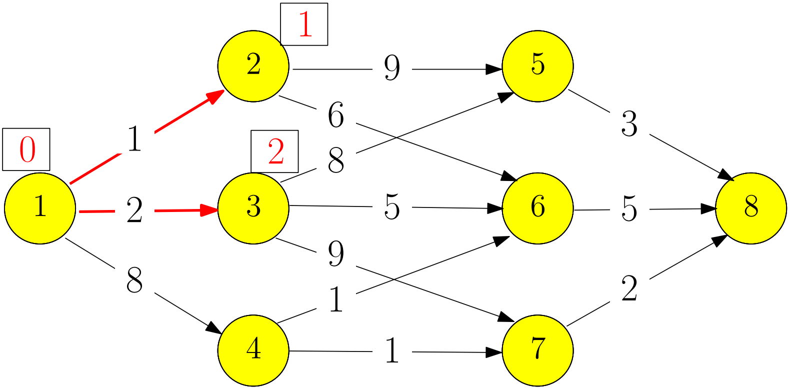
Example



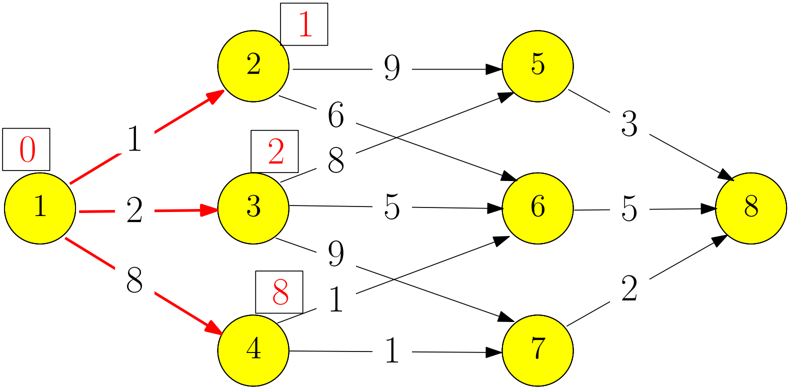
Example



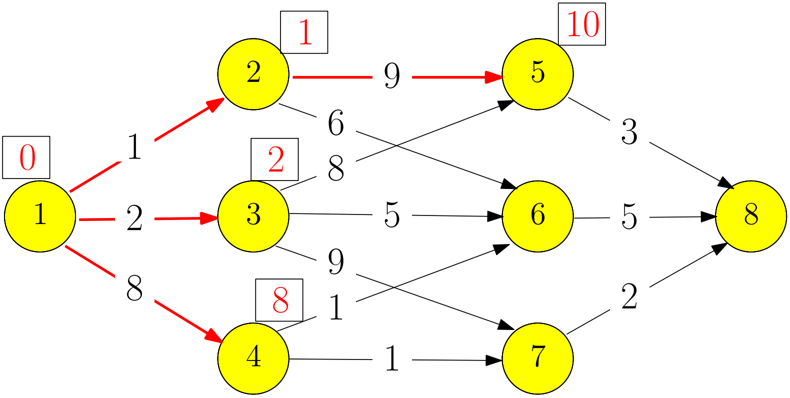
Example



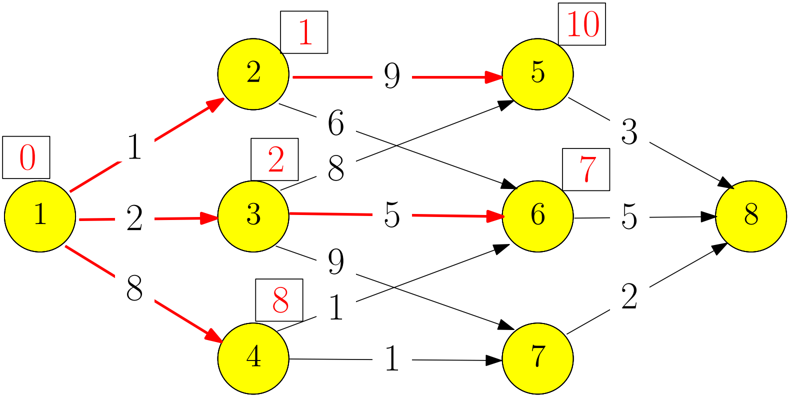
Example



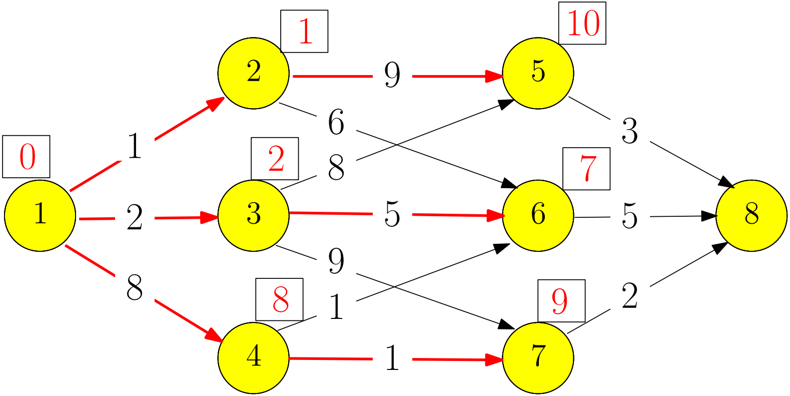
Example



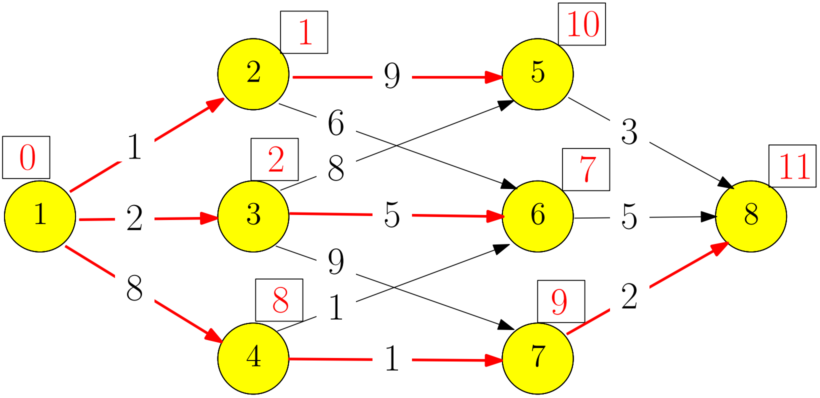
Example



Example



Example



Variant: Heaviest Path in a Directed Acyclic Graph

Heaviest Path in a Directed Acyclic Graph

Input: directed acyclic graph $G = (V, E)$ and $w : E \to \mathbb{R}$.

Assume $V = \{1, 2, 3 \cdots, n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

Output: the path with the largest weight (the heaviest path) from 1 to n.

- $f[i]$: weight of the heaviest path from 1 to i

$$f[i] = \begin{cases} & i = 1 \\ & i = 2, 3, \cdots, n \end{cases}$$

Variant: Heaviest Path in a Directed Acyclic Graph

Heaviest Path in a Directed Acyclic Graph

Input: directed acyclic graph $G = (V, E)$ and $w : E \to \mathbb{R}$.
Assume $V = \{1, 2, 3 \cdots, n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

Output: the path with the largest weight (the heaviest path) from 1 to n.

- $f[i]$: weight of the heaviest path from 1 to i

$$f[i] = \begin{cases} 0 & i = 1 \\ & i = 2, 3, \cdots, n \end{cases}$$

Variant: Heavist Path in a Directed Acyclic Graph

Heaviest Path in a Directed Acyclic Graph

Input: directed acyclic graph $G = (V, E)$ and $w : E \to \mathbb{R}$.

Assume $V = \{1, 2, 3 \cdots, n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

Output: the path with the largest weight (the heaviest path) from 1 to n.

- $f[i]$: weight of the heaviest path from 1 to i

$$f[i] = \begin{cases} 0 & i = 1 \\ \max_{j:(j,i)\in E} \{f(j) + w(j,i)\} & i = 2, 3, \cdots, n \end{cases}$$

Outline

Matrix Chain Multiplication

Matrix Chain Multiplication

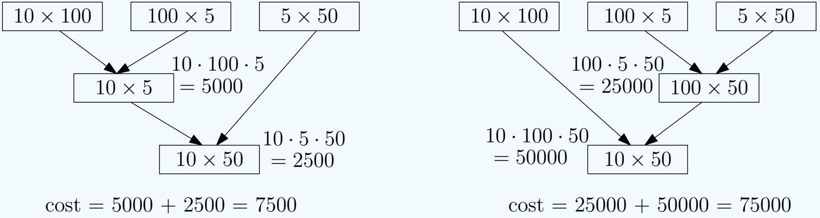
Input: n matrices $A_1, A_2, \cdots, A_n$ of sizes $r_1 \times c_1, r_2 \times c_2, \cdots, r_n \times c_n$, such that $c_i = r_{i+1}$ for every $i = 1, 2, \cdots, n-1$.

Output: the order of computing $A_1 A_2 \cdots A_n$ with the minimum number of multiplications

Fact Multiplying two matrices of size $r \times k$ and $k \times c$ takes $r \times k \times c$ multiplications.

Example:

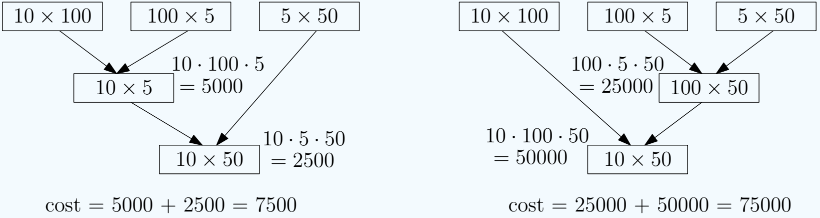
- $A_1 : 10 \times 100, \quad A_2 : 100 \times 5, \quad A_3 : 5 \times 50$



- $(A_1A_2)A_3$: $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$
- $A_1(A_2A_3)$: $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$

Example:

- $A_1 : 10 \times 100, \quad A_2 : 100 \times 5, \quad A_3 : 5 \times 50$



- $(A_1 A_2) A_3$: $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$
- $A_1 (A_2 A_3)$: $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$

Matrix Chain Multiplication: Design DP

Matrix Chain Multiplication: Design DP

- Assume the last step is $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$

Matrix Chain Multiplication: Design DP

- Assume the last step is $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$
- Cost of last step: $r_1 \times c_i \times c_n$

Matrix Chain Multiplication: Design DP

- Assume the last step is $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$
- Cost of last step: $r_1 \times c_i \times c_n$
- Optimality for sub-instances: we need to compute $A_1 A_2 \cdots A_i$ and $A_{i+1} A_{i+2} \cdots A_n$ optimally

Matrix Chain Multiplication: Design DP

- Assume the last step is $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$
- Cost of last step: $r_1 \times c_i \times c_n$
- Optimality for sub-instances: we need to compute $A_1 A_2 \cdots A_i$ and $A_{i+1} A_{i+2} \cdots A_n$ optimally
- $opt[i, j]$: the minimum cost of computing $A_i A_{i+1} \cdots A_j$

Matrix Chain Multiplication: Design DP

- Assume the last step is $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$
- Cost of last step: $r_1 \times c_i \times c_n$
- Optimality for sub-instances: we need to compute $A_1 A_2 \cdots A_i$ and $A_{i+1} A_{i+2} \cdots A_n$ optimally
- $opt[i, j]$: the minimum cost of computing $A_i A_{i+1} \cdots A_j$

$$opt[i,j] = \begin{cases} 0 & i = j \\ \min_{k: i \le k < j} \left(opt[i,k] + opt[k+1,j] + r_i c_k c_j \right) & i < j \end{cases}$$