Dynamic Programming

COMP 6651 – Algorithm Design Techniques

Denis Pankratov

Last video…

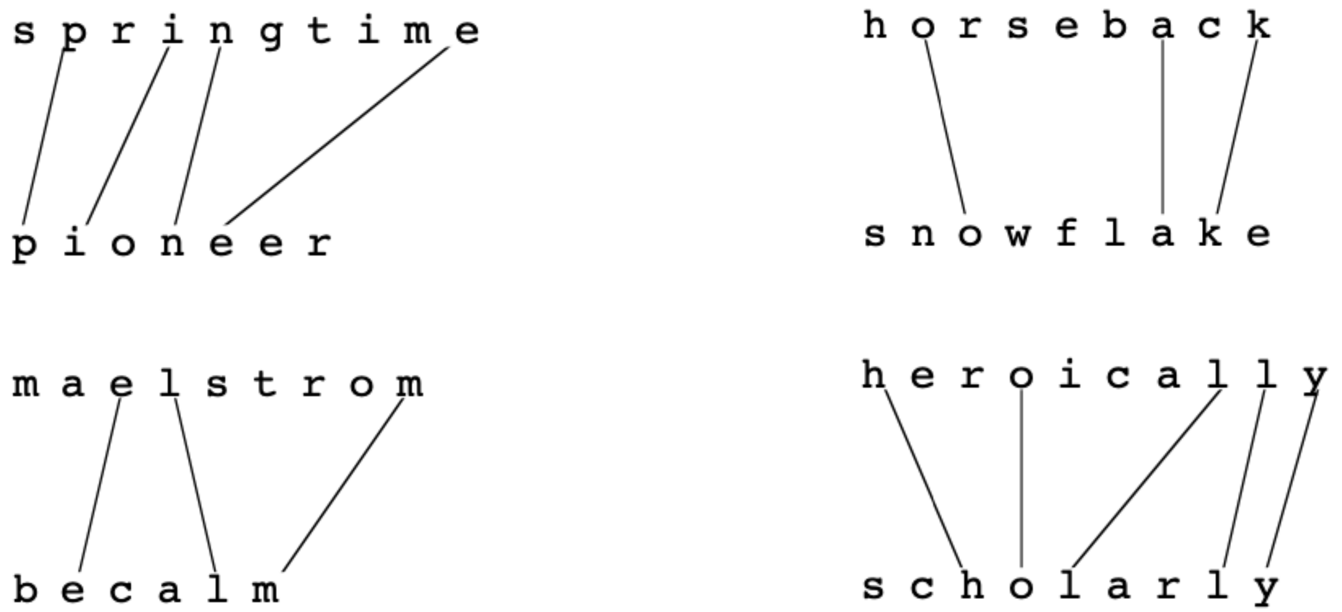
- Dynamic programming paradigm
- Optimal substructure property; proof often based on “cut-and-paste”
- Overlapping subproblems property
- Semantic array
- Computational array
- Correctness argument: semantic array = computational array
- Computing optimal value vs. finding optimal solution
- Examples: Fibonacci sequence, LIS, Weighted Interval Scheduling, Integral Knapsack

Longest Common Subsequence - LCS (CLRS 15.4)

Input: $X[1..m], Y[1..n]$ - two sequences of characters

Output: longest subsequence common to both X and Y

Examples



Sub-problems

We can reduce the problem size by reducing length of X

We can reduce the problem size by reducing length of Y

We may have to do both!

Sub-problem: LCS between $X[1..i]$ and $Y[1..j]$

Optimal Substructure Property

Consider OPT – LCS between $X[1..i]$ and $Y[1..j]$

OPT is one of the following:

- $X[i] = Y[j]$ is the last character of OPT + LCS between $X[1..i-1]$ and $Y[1..j-1]$
- OPT doesn't contain $X[i]$ as the last character, so $OPT =$ LCS between $X[1..i-1]$ and $Y[1..j]$
- OPT doesn't contain $Y[j]$ as the last character, so $OPT =$ LCS between $X[1..i]$ and $Y[1..j-1]$

Proof: "cut and paste" argument.

Computing optimal **value**

Semantic array

$$D[i, j] = \text{length of the LCS between } X[1..i] \text{ and } Y[1..j]$$

Solution to the whole problem is $D[m, n]$

Computational array

$$D[i, j] = \max \begin{cases} D[i-1, j] \\ D[i, j-1] \\ 1 + D[i-1, j-1], & if\ X[i] = Y[j] \end{cases}$$

$D[0, j] = D[i, 0] = 0$ for $i \in \{0, 1, \dots, m\}, j \in \{0, 1, \dots, n\}$

Pseudocode

$LCS(X[1..m], Y[1..n])$

$\quad$ instantiate array $D[0..m, 0..n]$

$\quad \boldsymbol{for}\ i = 0\ \boldsymbol{to}\ m \quad D[i, 0] \leftarrow 0$

$\quad \boldsymbol{for}\ j = 0\ \boldsymbol{to}\ n \quad D[0, j] \leftarrow 0$

$\quad \boldsymbol{for}\ i = 1\ \boldsymbol{to}\ m$

$\quad\quad \boldsymbol{for}\ j = 1\ \boldsymbol{to}\ n$

$\quad\quad\quad D[i, j] \leftarrow \max(D[i-1, j], D[i, j-1])$

$\quad\quad\quad \boldsymbol{if}\ X[i] = Y[j]$

$\quad\quad\quad\quad D[i, j] \leftarrow \max(D[i, j], 1 + D[i-1, j-1])$

$\quad \boldsymbol{return}\ D[m, n]$

$O(n)$

$O(m)$

Overall running time is $O(n \cdot m)$

Example

$X[1..7] =$ | C | O | U | N | T | E | R |

$Y[1..8] =$ | C | O | M | P | U | T | E | R |

$$D[i,j] = \max\begin{cases} D[i-1,j] \\ D[i,j-1] \\ 1 + D[i-1,j-1], \qquad if\ X[i] = Y[j] \end{cases}$$

$D[0,j] = D[i,0] = 0$ for $i \in \{0,1,\dots,m\}, j \in \{0,1,\dots,n\}$

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									
5									
6									
7									

Example $X[1..7] =$ C O U N T E R

$Y[1..8] =$ C O M P U T E R

$$D[i,j] = \max\begin{cases} D[i-1,j] \\ D[i,j-1] \\ 1 + D[i-1,j-1], & if\ X[i] = Y[j] \end{cases}$$

$D[0,j] = D[i,0] = 0$ for $i \in \{0,1,\dots,m\}, j \in \{0,1,\dots,n\}$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								
5	0								
6	0								
7	0								

Example

$X[1..7] =$ C O U N T E R

$Y[1..8] =$ C O M P U T E R

$$D[i,j] = \max \begin{cases} D[i-1,j] \\ D[i,j-1] \\ \mathbf{1 + D[i-1, j-1],} \qquad \boldsymbol{if\ X[i] = Y[j]} \end{cases}$$

$D[0,j] = D[i,0] = 0$ for $i \in \{0, 1, \dots, m\}, j \in \{0, 1, \dots, n\}$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1							
2	0								
3	0								
4	0								
5	0								
6	0								
7	0								

Example

$X[1..7] =$ C O U N T E R

$Y[1..8] =$ C O M P U T E R

$$D[i,j] = \max \begin{cases} D[i-1,j] \\ \boldsymbol{D[i,j-1]} \\ 1 + D[i-1,j-1], & if\ X[i] = Y[j] \end{cases}$$

$D[0,j] = D[i,0] = 0$ for $i \in \{0,1,\ldots,m\}, j \in \{0,1,\ldots,n\}$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	1						
2	0								
3	0								
4	0								
5	0								
6	0								
7	0								

Example

$X[1..7] =$ C O U N T E R

$Y[1..8] =$ C O M P U T E R

$$D[i,j] = \max \begin{cases} D[i-1,j] \\ \boldsymbol{D[i,j-1]} \\ 1 + D[i-1,j-1], \qquad if\ X[i] = Y[j] \end{cases}$$

$D[0,j] = D[i,0] = 0$ for $i \in \{0,1,\ldots,m\}, j \in \{0,1,\ldots,n\}$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1
2	0								
3	0								
4	0								
5	0								
6	0								
7	0								

Example $X[1..7] =$

C	O	U	N	T	E	R

$Y[1..8] =$

C	O	M	P	U	T	E	R

$$D[i,j] = \max \begin{cases} \boldsymbol{D[i-1,j]} \\ D[i,j-1] \\ 1 + D[i-1,j-1], \qquad if\ X[i] = Y[j] \end{cases}$$

$D[0,j] = D[i,0] = 0$ for $i \in \{0,1,\dots,m\}, j \in \{0,1,\dots,n\}$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1
2	0	1							
3	0								
4	0								
5	0								
6	0								
7	0								

Example $X[1..7] =$

C	O	U	N	T	E	R

$Y[1..8] =$

C	O	M	P	U	T	E	R

$$D[i,j] = \max\begin{cases} D[i-1,j] \\ D[i,j-1] \\ \mathbf{1 + D[i-1,j-1],} \qquad \boldsymbol{if\ X[i] = Y[j]} \end{cases}$$

$D[0,j] = D[i,0] = 0$ for $i \in \{0,1,\dots,m\}, j \in \{0,1,\dots,n\}$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1
2	0	1	2						
3	0								
4	0								
5	0								
6	0								
7	0								

Example

$X[1..7] =$ C O U N T E R

$Y[1..8] =$ C O M P U T E R

$$D[i,j] = \max\begin{cases} D[i-1,j] \\ \boldsymbol{D[i,j-1]} \\ 1 + D[i-1,j-1], & if\ X[i] = Y[j] \end{cases}$$

$D[0,j] = D[i,0] = 0$ for $i \in \{0,1,\ldots,m\}, j \in \{0,1,\ldots,n\}$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2	2
3	0								
4	0								
5	0								
6	0								
7	0								

Example $X[1..7] =$ C O U N T E R

$Y[1..8] =$ C O M P U T E R

$$D[i,j] = \max \begin{cases} D[i-1,j] \\ D[i,j-1] \\ 1 + D[i-1,j-1], \qquad if\ X[i] = Y[j] \end{cases}$$

$D[0,j] = D[i,0] = 0$ for $i \in \{0,1,\dots,m\}, j \in \{0,1,\dots,n\}$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2	2
3	0	1	2	2	2	3	3	3	3
4	0	1	2	2	2	3	3	3	3
5	0	1	2	2	2	3	4	4	4
6	0	1	2	2	2	3	4	5	5
7	0	1	2	2	2	3	4	5	6

Constructing an actual subsequence

As usual, record which choice in max results in value of $D[i, j]$

Store the result in $Prev[i, j]$:

Modify the innermost for loop as follows:

$D[i, j] \leftarrow 0$
$\boldsymbol{if}\ D[i, j-1] > D[i-1, j]$
$\quad D[i, j] \leftarrow D[i, j-1]$
$\quad Prev[i, j] \leftarrow (i, j-1)$
$\boldsymbol{else}$
$\quad D[i, j] \leftarrow D[i-1, j]$
$\quad Prev[i, j] \leftarrow (i-1, j)$
…

…
$\boldsymbol{if}\ X[i] = Y[j]\ \boldsymbol{and}\ 1 + D[i-1, j-1] > D[i, j]$
$\quad D[i, j] \leftarrow 1 + D[i-1, j-1]$
$\quad Prev[i, j] \leftarrow (i-1, j-1)$

Constructing an actual subsequence

$result \leftarrow \emptyset$
$i \leftarrow m$
$j \leftarrow n$
$while\ i > 0\ and\ j > 0$
  $(new_i, new_j) \leftarrow Prev[i, j]$
  $if\ new_i = i - 1\ and\ new_j = j - 1$
    $result.insert\_to\_front(X[i])$
  $i \leftarrow new_i$
  $j \leftarrow new_j$

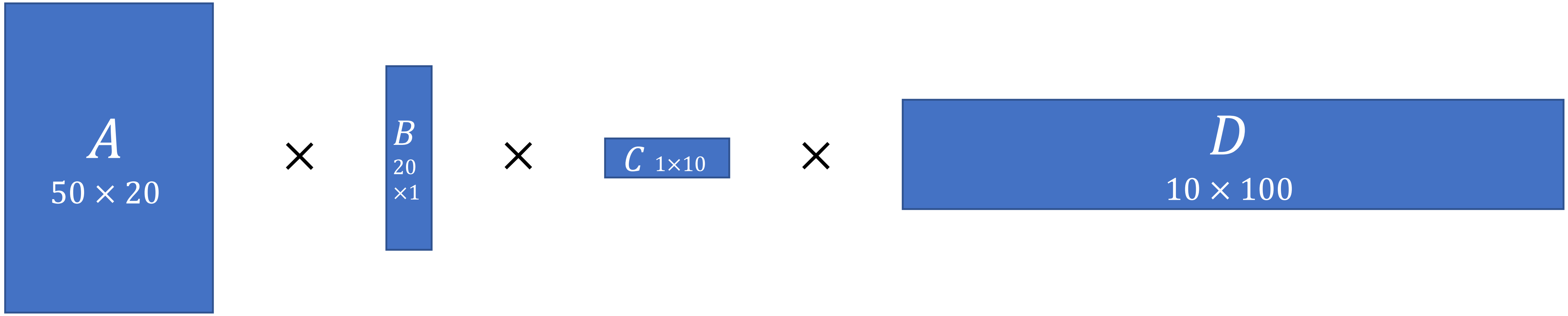
Chain Matrix Multiplication (CLRS 15.2)

Consider multiplying 4 matrices

$$A \times B \times C \times D$$

$size(A) = 50 \times 20 \quad size(B) = 20 \times 1$

$size(C) = 1 \times 10 \quad size(D) = 10 \times 100$



Matrix multiplication is ***not commutative***

in general $A \times B \neq B \times A$

Matrix multiplication is ***associative***

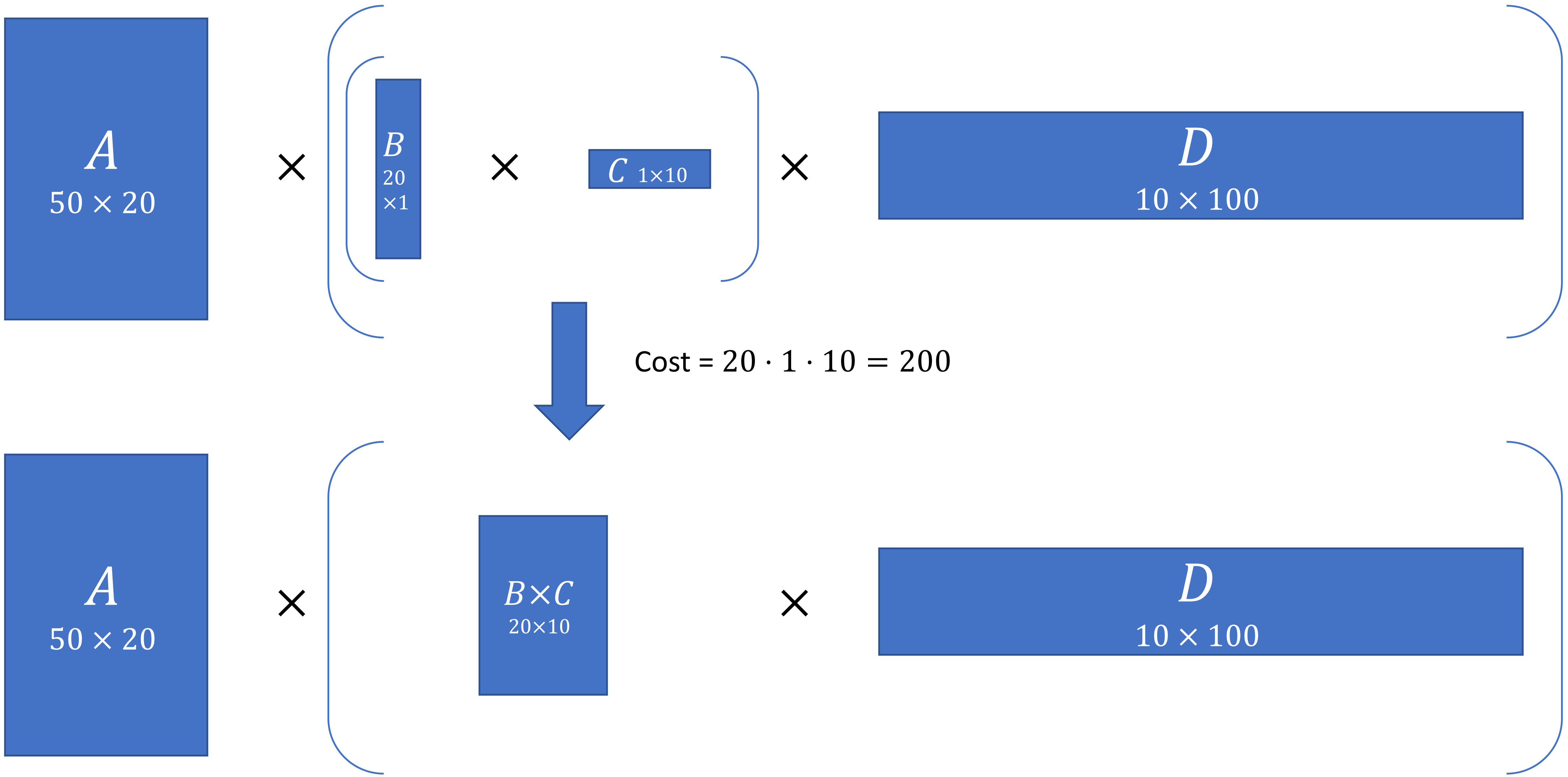
$$(A \times B) \times C = A \times (B \times C)$$

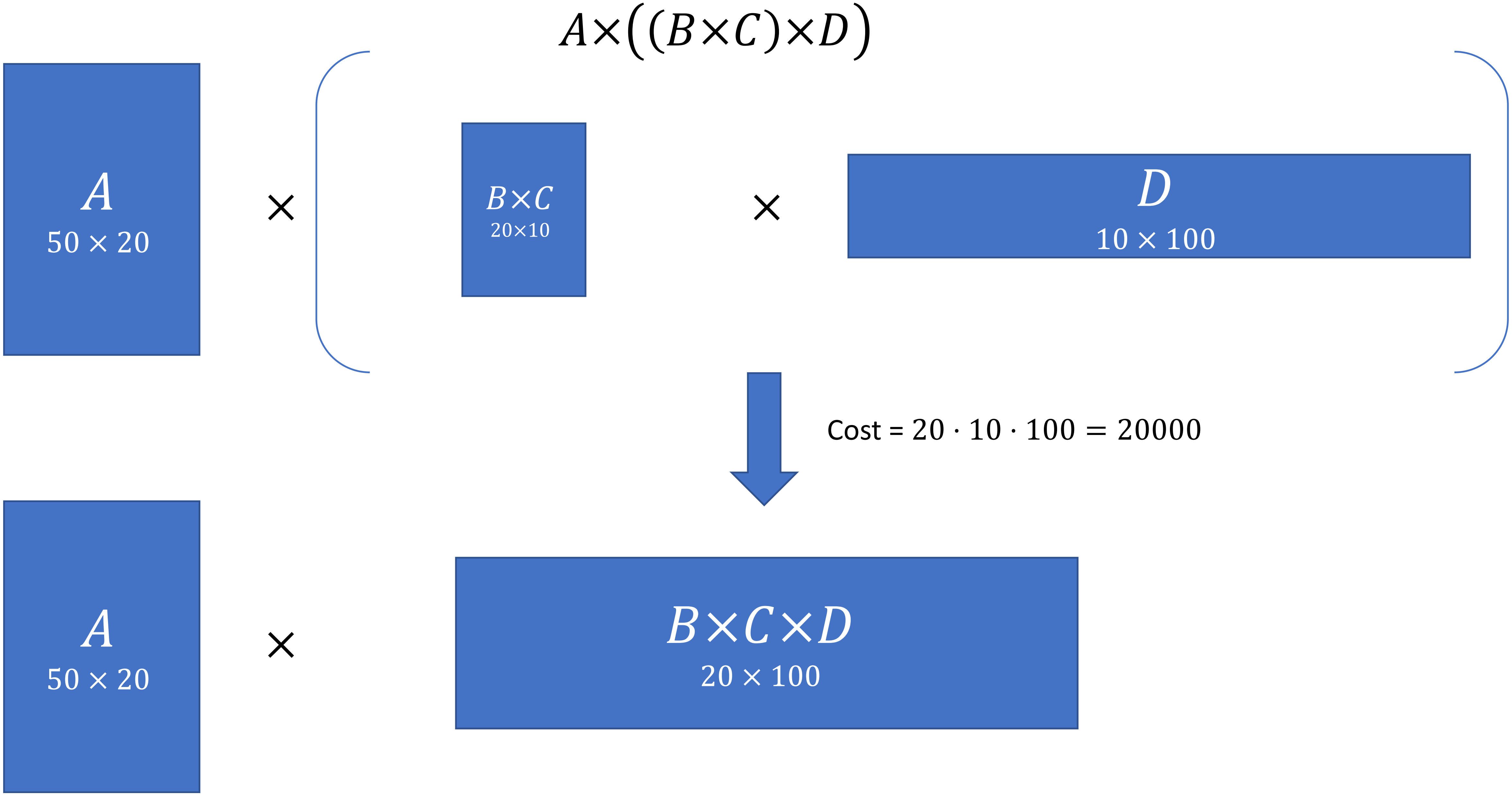
Parenthesis affect the order in which matrices are multiplied, but doesn't affect the result

Different orders can have vastly different costs in terms of multiplications of individual elements

Assume that $m \times n$ matrix with $n \times p$ matrix takes mnp multiplications

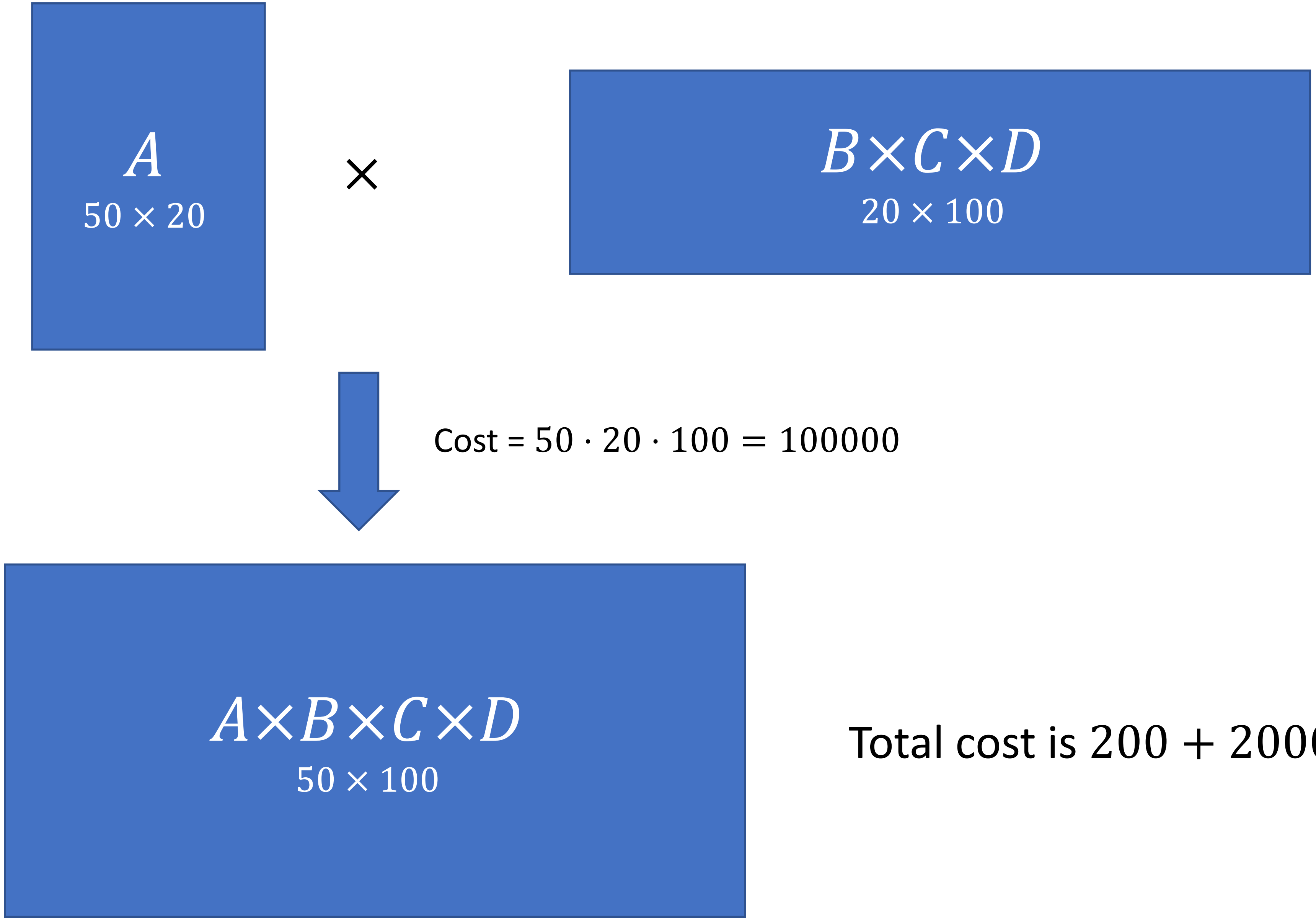
$$A\times((B\times C)\times D)$$





$A \times ((B \times C) \times D)$
A
50×20
$\times$
$B \times C$
20×10
$\times$
D
10×100
Cost = $20 \cdot 10 \cdot 100 = 20000$
A
50×20
$\times$
$B \times C \times D$
20×100

$$A\times\big((B\times C)\times D\big)$$



Total cost is $200 + 20000 + 100000 = 120200$

$$A \times B \times C \times D$$

$$size(A) = 50 \times 20 \quad size(B) = 20 \times 1$$
$$size(C) = 1 \times 10 \quad size(D) = 10 \times 100$$

Consider different orders:

Parenthesization	Cost Computation	Cost
$A\times\big((B\times C)\times D\big)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120,200
$\big(A\times(B\times C)\big)\times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60,200
$(A\times B)\times(C\times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7,000

Chain Matrix Multiplication

More generally, determine an optimal order to multiply matrices

$$A_1 \times A_2 \times \cdots \times A_n$$

with respective dimensions

$$p_0{\times}p_1, p_1{\times}p_2, \ldots, p_{n-1}{\times}p_n$$

Formally:

Input: $P[0..n]$ - array of n positive integers, representing dimensions of matrices as above

Output: optimal parenthesization to minimize the total cost of multiplying

Sub-problems

last multiplication

Consider an optimal parenthesization

$$\underbrace{\left(A_1 \times (A_2 \times \cdots \times A_i)\right)} \times \underbrace{\left(((A_{i+1} \times (A_{i+2} \times A_{i+3})) \times \cdots \times A_n\right)}$$

Some multiplication is going to be performed last according to this parenthesization

This naturally partitions the original problem into two sub-problems

sub-problems

Sub-problems

More generally, sub-problems are defined by two indices i and j

Find minimum cost parenthesization of

$$A_i \times A_{i+1} \times \cdots \times A_j$$

where $1 \le i \le j \le n$

Optimal substructure property

Let $OPT[i,j]$ denote the minimum cost of parenthesization of

$$A_i \times A_{i+1} \times \cdots \times A_j$$

Then $OPT[i,j]$ consists of performing kth multiplication last for some $k \in \{i, i+1, \ldots, j-1\}$ (assuming $j > i$) and optimally parenthesizing

$$A_i \times A_{i+1} \times \cdots \times A_k \text{ and } A_{k+1} \times \cdots \times A_j$$

Therefore: $OPT[i,j] = OPT[i,k] + OPT[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$

Proof: “cut and paste” argument

Computing optimal **value**

Semantic array:

$$D[i,j] = \text{minimum cost of multiplication of } A_i \times \cdots \times A_j$$

Solution to the whole problem is $D[1,n]$

Computational array:

$$D[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} D[i,k] + D[k+1,j] + p_{i-1} \cdot p_k \cdot p_j & i < j \end{cases}$$

Sub-problem $A_i \times \cdots \times A_j$ has **size** $s = j - i$ number of matrix multiplications

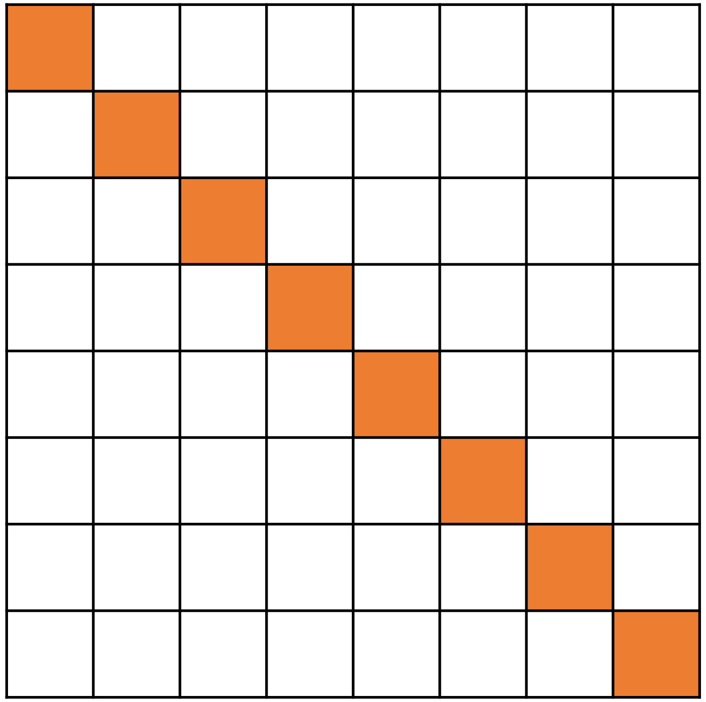
Smallest sub-problem size $s = 0$, i.e., $j = i$, in which case there is nothing to multiply

$$D[i, i] = 0$$

Working on a sub-problem of size s we rely on having solved sub-problems of size $< s$

Order of filling in the array

$s = 0$



Sub-problem $A_i \times \cdots \times A_j$ has **size** $s = j - i$ number of matrix multiplications

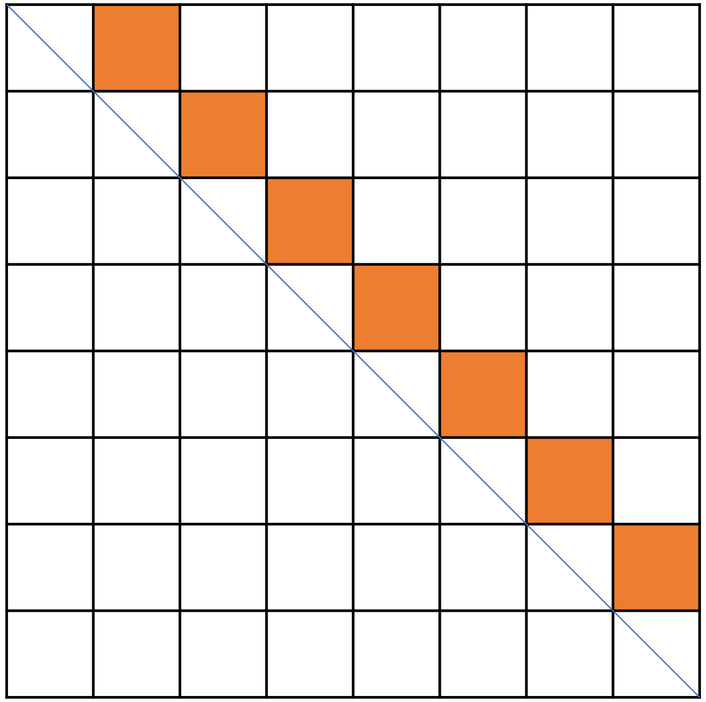
Smallest sub-problem size $s = 0$, i.e., $j = i$, in which case there is nothing to multiply

$$D[i, i] = 0$$

Working on a sub-problem of size s we rely on having solved sub-problems of size $< s$

Order of filling in the array

$s = 1$



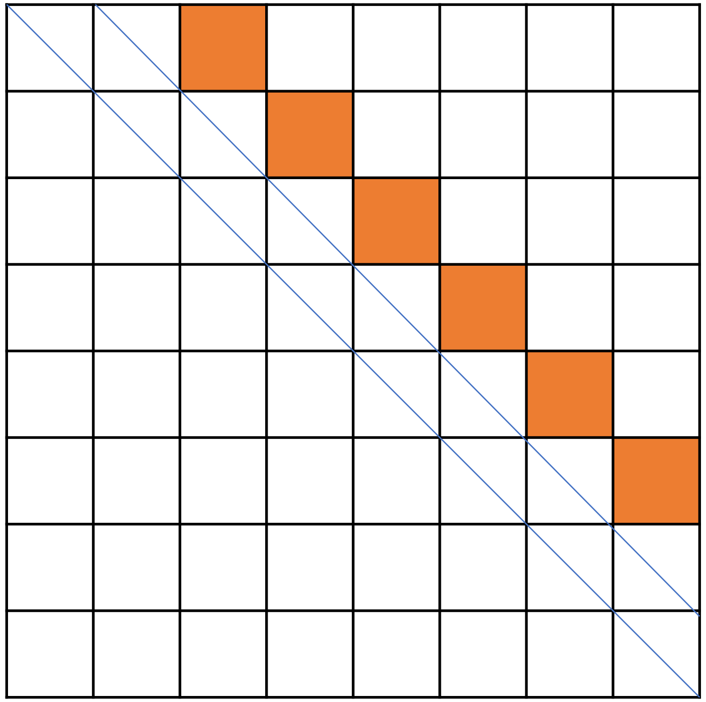
Sub-problem $A_i \times \cdots \times A_j$ has **size** $s = j - i$ number of matrix multiplications

Smallest sub-problem size $s = 0$, i.e., $j = i$, in which case there is nothing to multiply

$$D[i, i] = 0$$

Working on a sub-problem of size s we rely on having solved sub-problems of size $< s$

Order of filling in the array



$s = 2$

Sub-problem $A_i \times \cdots \times A_j$ has **size** $s = j - i$ number of matrix multiplications

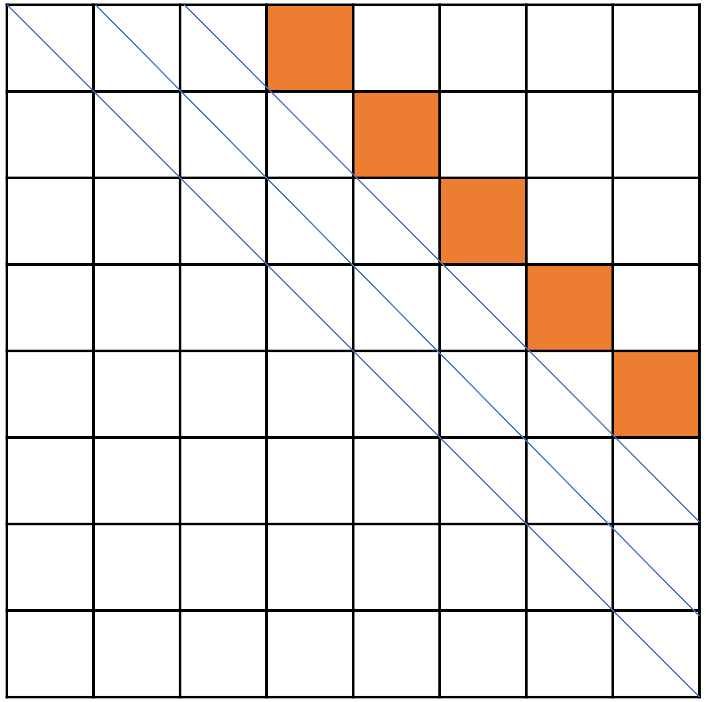
Smallest sub-problem size $s = 0$, i.e., $j = i$, in which case there is nothing to multiply

$$D[i, i] = 0$$

Working on a sub-problem of size s we rely on having solved sub-problems of size $< s$

Order of filling in the array

$s = 3$



Sub-problem $A_i \times \cdots \times A_j$ has **size** $s = j - i$ number of matrix multiplications

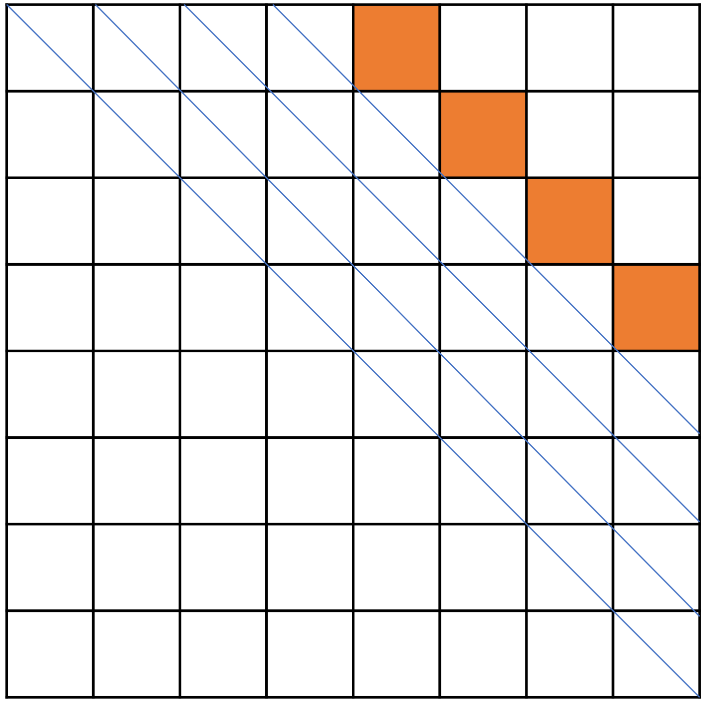
Smallest sub-problem size $s = 0$, i.e., $j = i$, in which case there is nothing to multiply

$$D[i,i] = 0$$

Working on a sub-problem of size s we rely on having solved sub-problems of size $< s$

Order of filling in the array

$s = 4$



Sub-problem $A_i \times \cdots \times A_j$ has **size** $s = j - i$ number of matrix multiplications

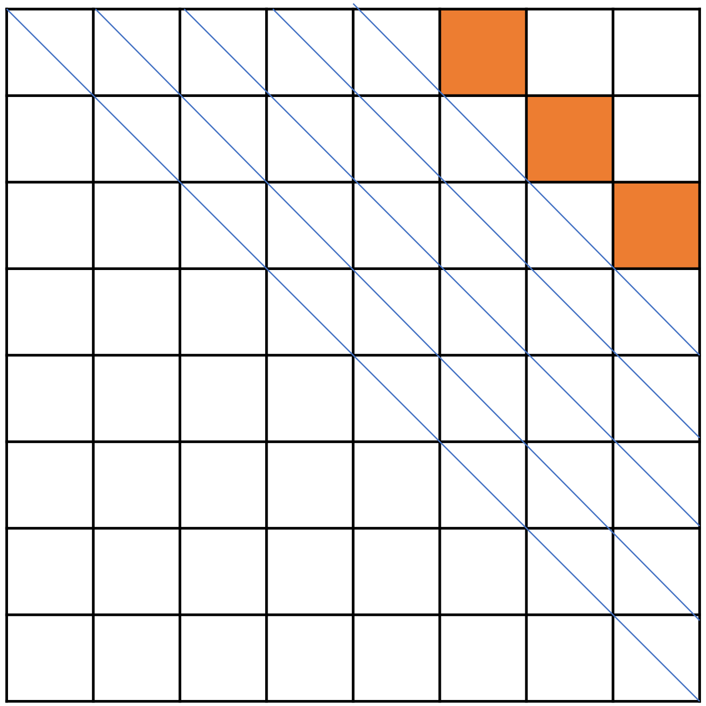
Smallest sub-problem size $s = 0$, i.e., $j = i$, in which case there is nothing to multiply

$$D[i, i] = 0$$

Working on a sub-problem of size s we rely on having solved sub-problems of size $< s$

Order of filling in the array

$s = 5$



Sub-problem $A_i \times \cdots \times A_j$ has **size** $s = j - i$ number of matrix multiplications

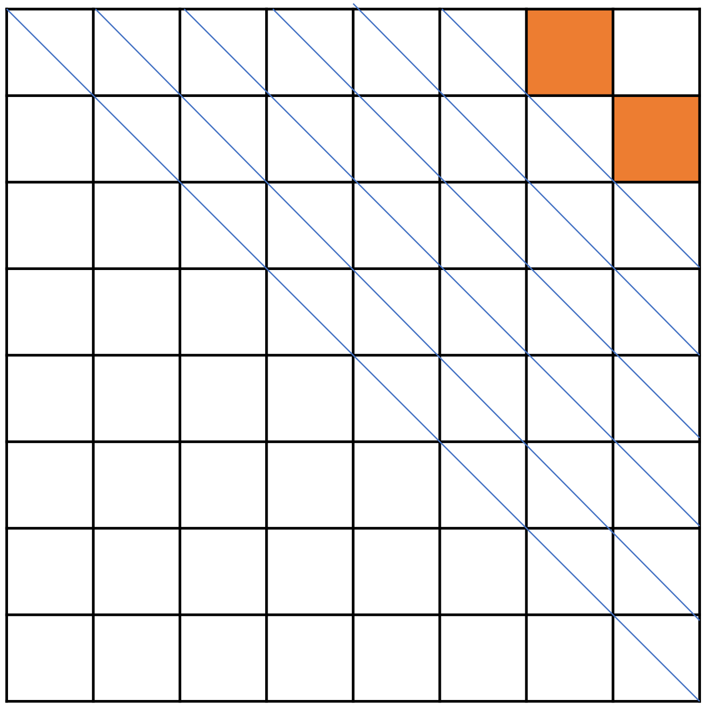
Smallest sub-problem size $s = 0$, i.e., $j = i$, in which case there is nothing to multiply

$$D[i, i] = 0$$

Working on a sub-problem of size s we rely on having solved sub-problems of size $< s$

Order of filling in the array

$s = 6$



Sub-problem $A_i \times \cdots \times A_j$ has **size** $s = j - i$ number of matrix multiplications

Smallest sub-problem size $s = 0$, i.e., $j = i$, in which case there is nothing to multiply

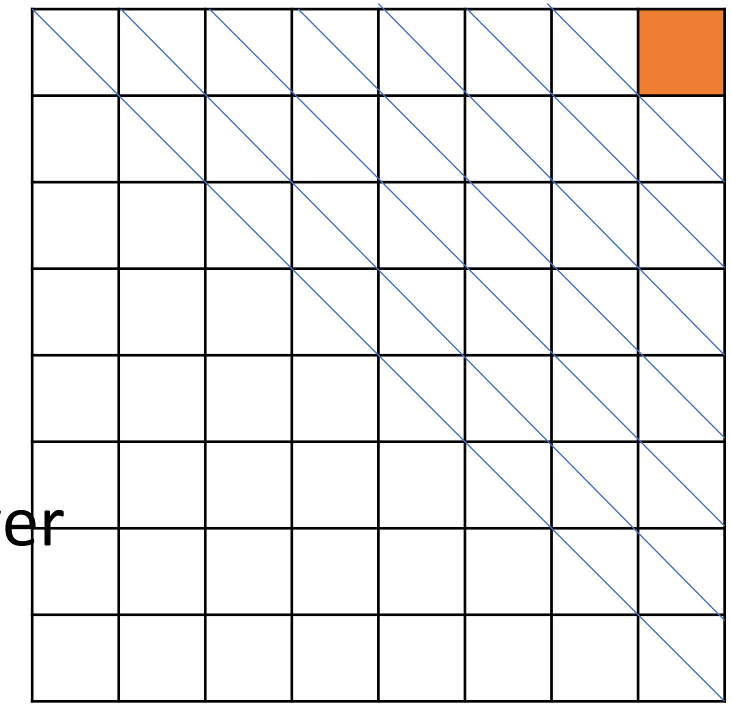
$$D[i, i] = 0$$

Working on a sub-problem of size s we rely on having solved sub-problems of size $< s$

Order of filling in the array

$s = 7$

In this example this is the final answer



Pseudocode

Overall running time is $O(n^3)$
Can show that it is also $\Omega(n^3)$

$MatrixChainMult(P[0..n])$
  initialize array $D[1..n, 1..n]$
  $\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ n \qquad D[i,i] \leftarrow 0$
  $\boldsymbol{for}\ s = 1\ \boldsymbol{to}\ n-1$
    $\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ n-s$
      $j \leftarrow i + s$
      $D[i,j] \leftarrow \infty$
      $\boldsymbol{for}\ k = i\ \boldsymbol{to}\ j-1$
        $D[i,j] \leftarrow \min(D[i,j], D[i,k] + D[k+1,j] + P[i]P[k]P[j])$
$\boldsymbol{return}\ D[1,n]$

$O(n)$

$O(n)$

$O(n)$

Computing actual parenthesization

As usual, to compute an actual parenthesization remember the choice of k resulting in the min value of $D[i,j]$

Record these values in array $Prev[i,j]$, change the inner-most for-loop as follows:

$$
\begin{aligned}
& for\ k = i\ to\ j-1 \\
& \quad if\ D[i,k] + D[k+1,j] + P[i]P[k]P[j] < D[i,j] \\
& \qquad D[i,j] \leftarrow D[i,k] + D[k+1,j] + P[i]P[k]P[j] \\
& \qquad Prev[i,j] \leftarrow k
\end{aligned}
$$

Using $Prev$ to print actual parenthesization

The following recursive function prints the actual parenthesization

$PrintParenthesization(Prev[1..n, 1..n], i, j)$

$\quad if\ i = j$

$\qquad print$ “A_i”

$\quad else$

$\qquad print$ “(“

$\qquad PrintParenthesization(Prev, i, Prev[i, j])$

$\qquad PrintParenthesization(Prev, Prev[i, j] + 1, j)$

$\qquad print$ “)”

Initial call is to $PrintParenthesization(Prev, 1, n)$

Edit distance

Notion of closeness between strings

Extent to which two strings can be aligned

Example: SNOWY vs. SUNNY

S	—	N	O	W	Y
S	U	N	N	—	Y

Cost: 3

—	S	N	O	W	—	Y
S	U	N	—	—	N	Y

Cost: 5

Edit distance

Minimum number of edits needed to transform one string into another

Edits:

- Character insertions
- Character deletions
- Character substitutions

S	—	N	O	W	Y
S	U	N	N	—	Y

Cost: 3

SNOWY → SUNNY:
Insert U
Substitute O → N
Delete W

Edit distance

Input: $x[1..m], y[1..n]$ - two strings

Output: edit distance between x and y

Dynamic programming approach

Sub-problems defined by decreasing length of x and y, i.e. *semantic array* is:

$$E[i,j] = \text{edit distance between } x[1..i] \text{ and } y[1..j]$$

Example of a sub-problem

$E[7,5]$

$\boxed{\text{E X P O N E N}}$ T I A L

$\boxed{\text{P O L Y N}}$ O M I A L

Optimal substructure

Consider right-most column in subproblem $E[i,j]$

It can be one of three things:

$$\begin{matrix} x[i] \\ - \end{matrix} \quad \textbf{or} \quad \begin{matrix} - \\ y[j] \end{matrix} \quad \textbf{or} \quad \begin{matrix} x[i] \\ y[j] \end{matrix}$$

1st case: delete $x[i]$ at cost 1 and align $x[1..i-1]$ with $y[1..j]$

2nd case: insert $y[j]$ at cost 1 and align $x[1..i]$ with $y[1..j-1]$

3rd case: if $x[i] = y[j]$ then align $x[1..i-1]$ with $y[1..j-1]$

otherwise substitute $x[i] \to y[j]$ then align $x[1..i-1]$ with $y[1..j-1]$

OPT picks the best of these three options

Computational array

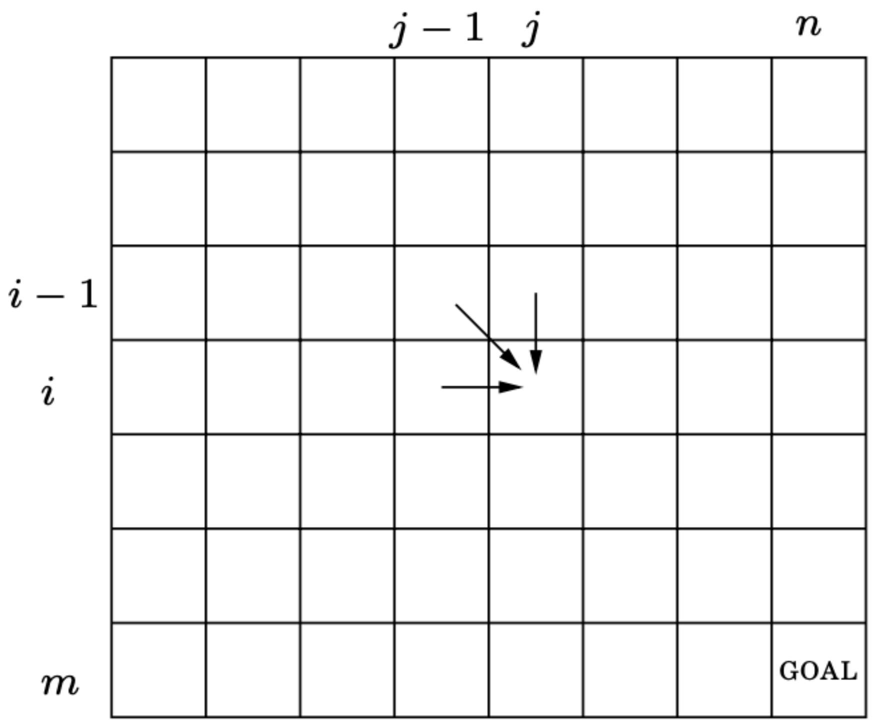
$$E[i,j] = \min(1 + E[i-1,j], 1 + E[i,j-1], \mathbb{I}(x[i] \neq y[j]) + E[i-1,j-1])$$

$\mathbb{I}(x[i] \neq y[j])$ is the indicator function which is 1 if $x[i] \neq y[j]$ and 0 otherwise

Base cases:

$E[0,j] = j$ and $E[i,0] = i$

Solution to the whole problem is $E[m,n]$



		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

```
E X P O N E N — T I A L
— — P O L Y N O M I A L
```

Pseudocode

$EditDistance(x[1..m], y[1..n])$

initialize array $E[0..m, 0..n]$

$\boldsymbol{for}\ i = 0\ \boldsymbol{to}\ m \quad E[i,0] \leftarrow i$

$\boldsymbol{for}\ j = 0\ \boldsymbol{to}\ n \quad E[0,j] \leftarrow j$

$\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ m$

$\quad \boldsymbol{for}\ j = 1\ \boldsymbol{to}\ n$

$$E[i,j] \leftarrow \min\begin{pmatrix} 1 + E[i-1,j], 1 + E[i,j-1], \\ \mathbb{I}(x[i] \neq y[j]) + E[i-1,j-1] \end{pmatrix}$$

$\boldsymbol{return}\ E[m,n]$

Overall running time $O(mn)$

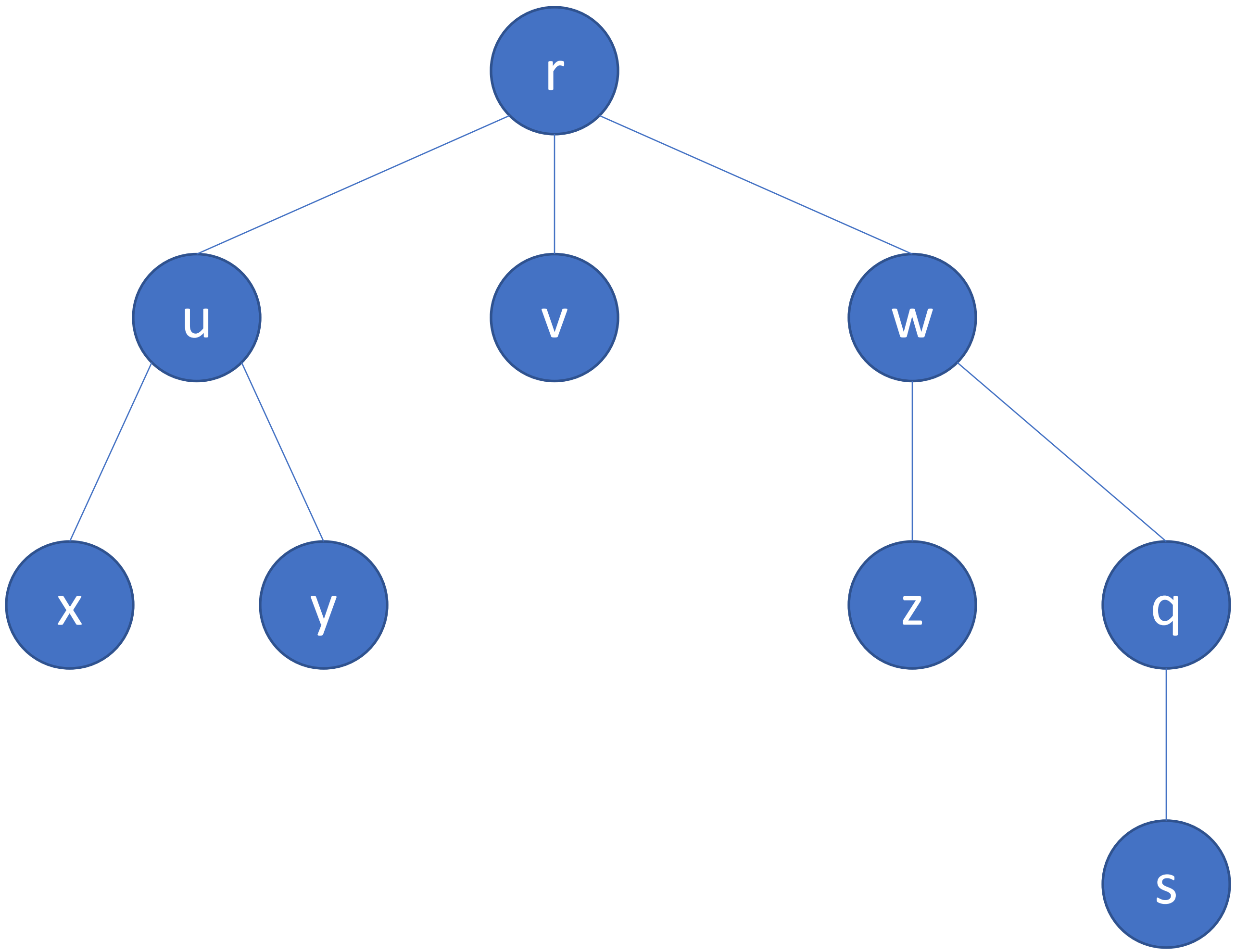
$O(n)$

$O(m)$

Exercise:

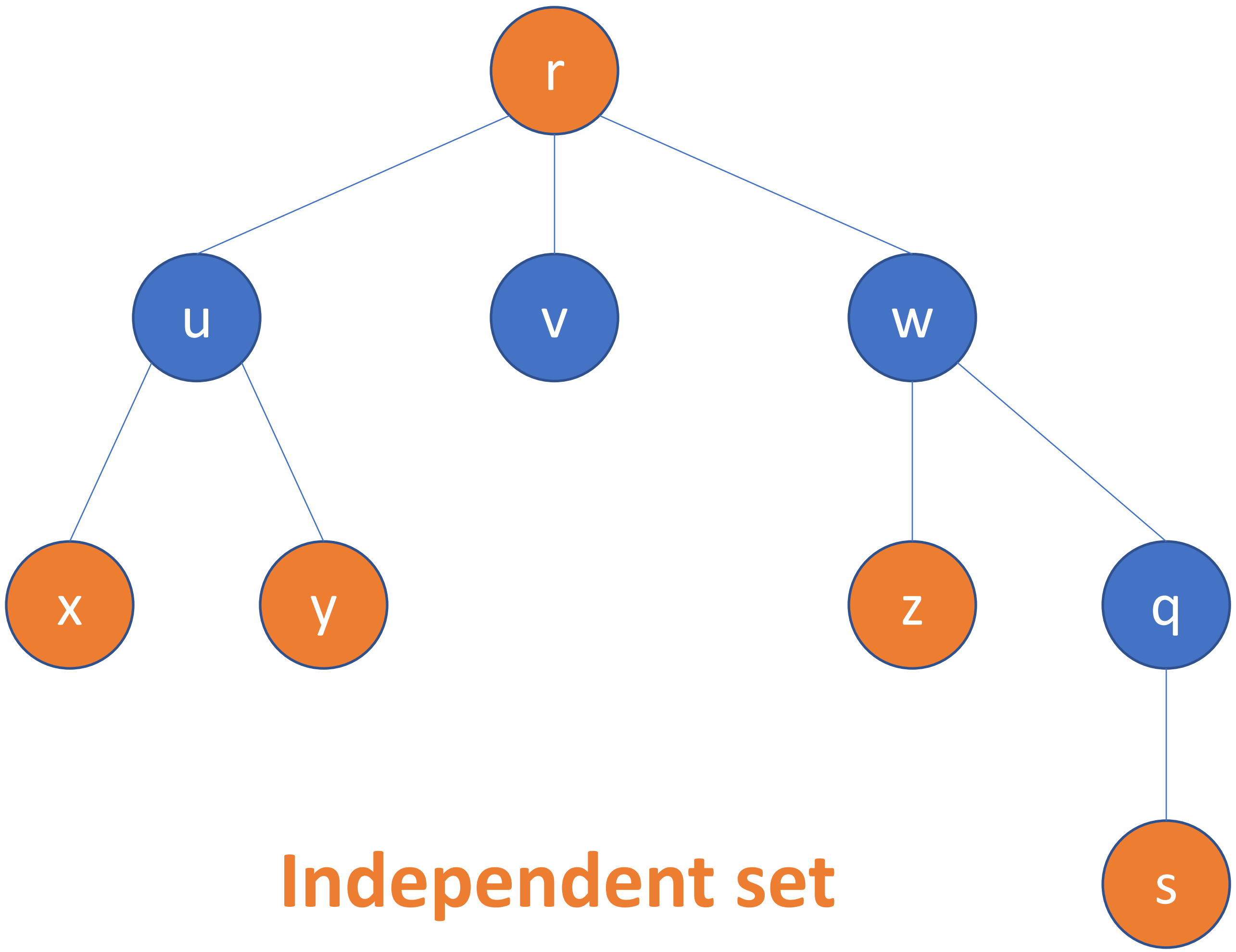
- Extend the algorithm to return a sequence of edit operations that result in minimum edit distance

Independent Sets in Trees



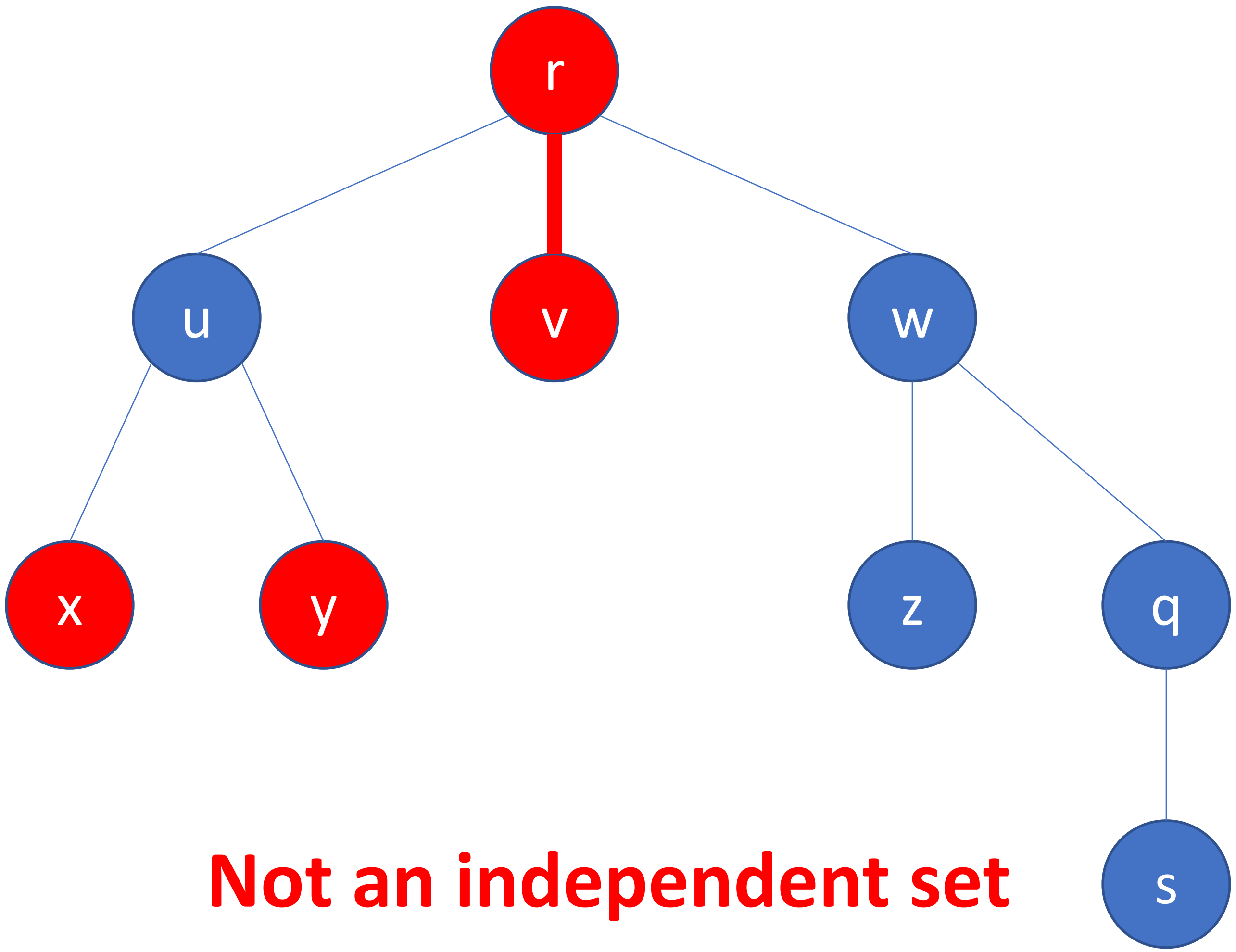
Independent set: set of nodes that are not connected by edges

Independent Sets in Trees



Independent set: set of nodes that are not connected by edges

Independent Sets in Trees



Not an independent set

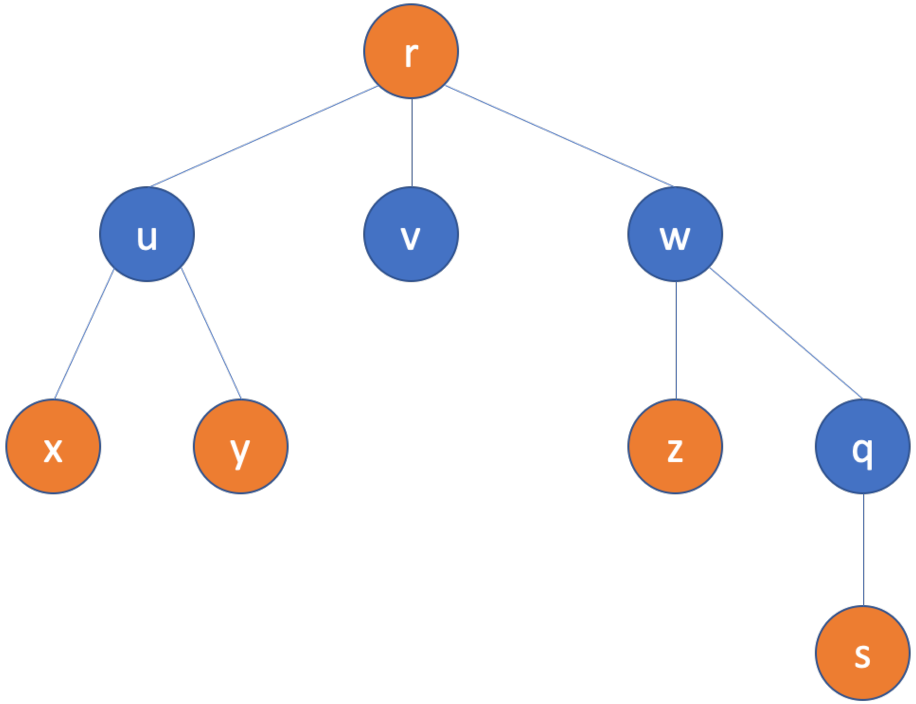
Independent set: set of nodes that are not connected by edges

Independent Sets in Trees

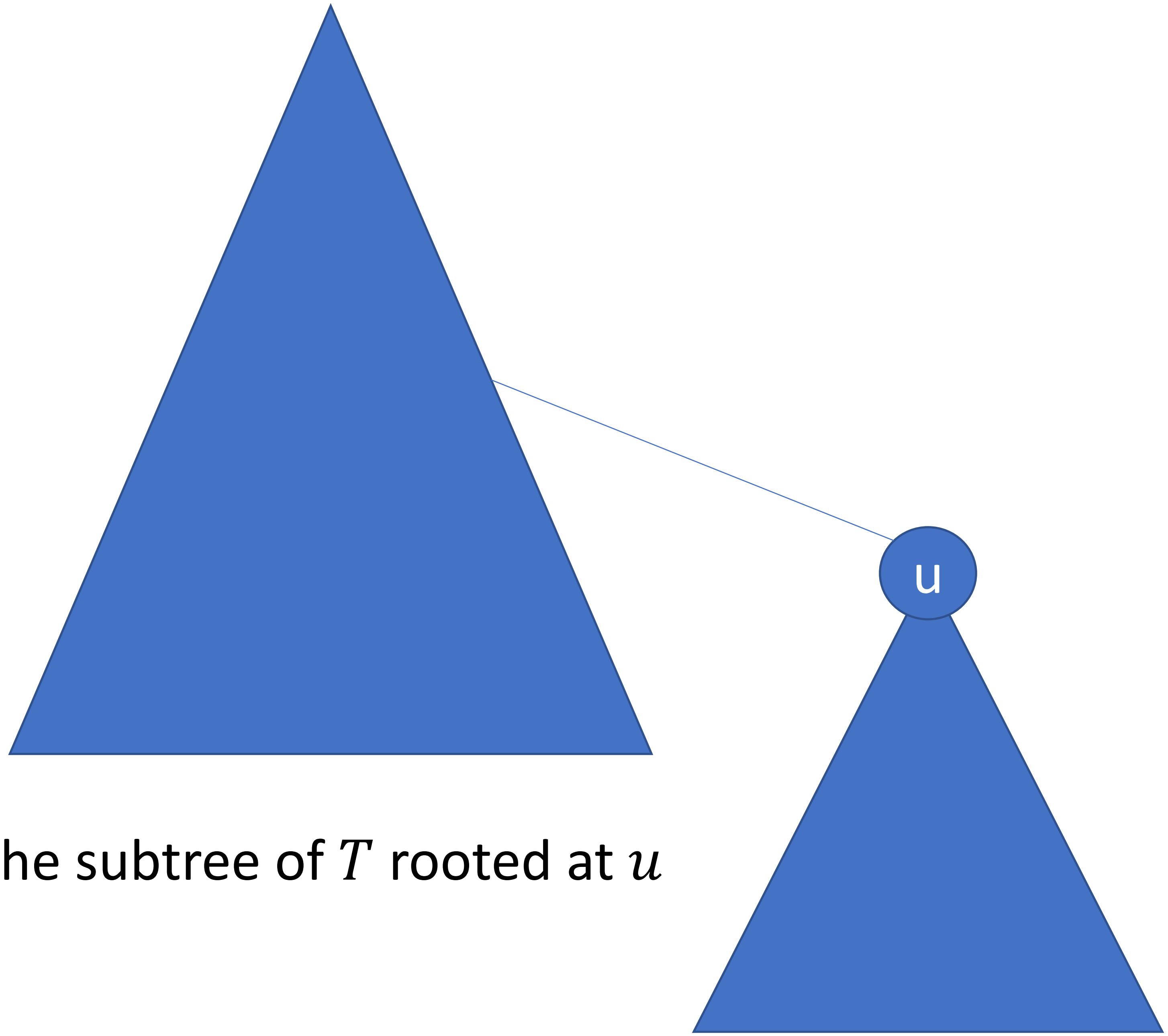
Input: tree T with nodes V

Output: $S \subseteq V$ such that S is independent and $|S|$ is as large as possible

Example:



Sub-problems



For a node u define T_u as the subtree of T rooted at u

Optimal substructure property

Maximum in independent set in T_u

either

contains u, doesn't contain its children, and contains optimal solutions over grandchildren

or

doesn't contain u, contains optimal solutions over its children

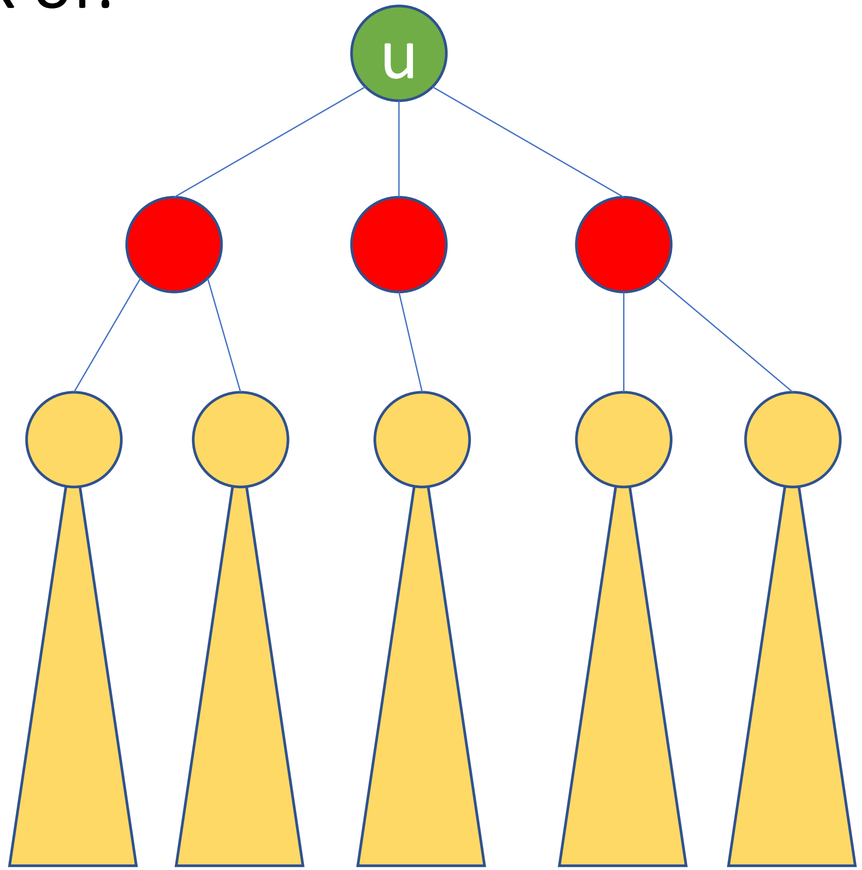
Proof: "cut-and-paste" argument

Optimal substructure property

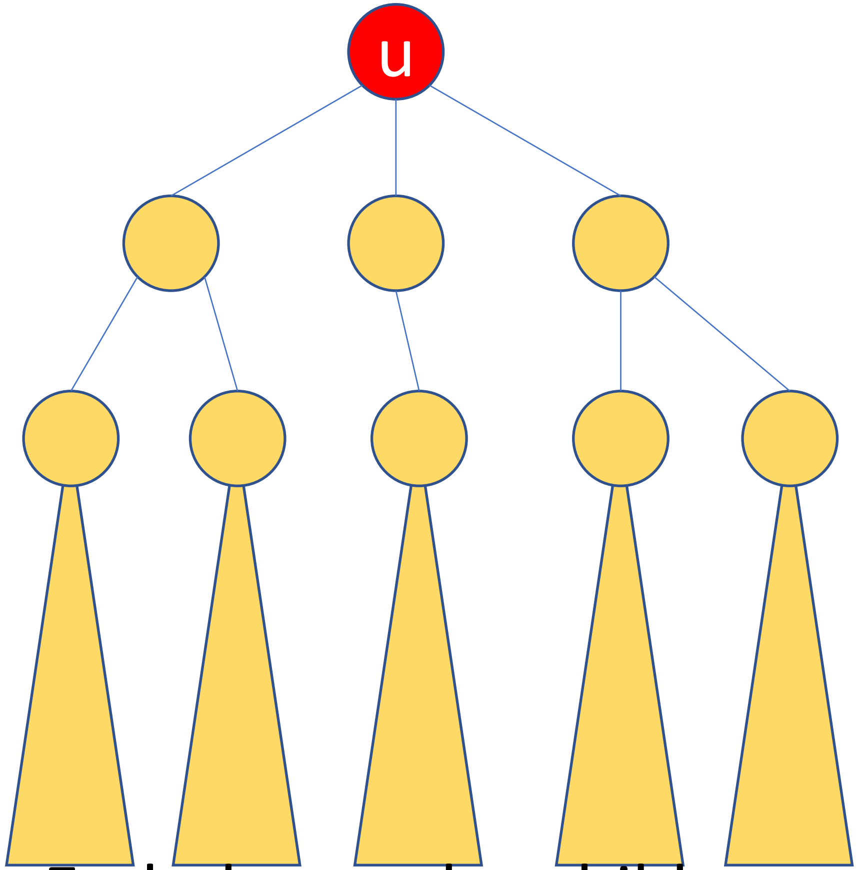
OPT for T_u is max of:

Children of u

Grandchildren of u



Include u, exclude children,
Solve grandchildren optimally



Exclude u, solve children optimally

Computing optimal **value**

Semantic array:

$$D[u] = \text{size of a maximum independent set in } T_u$$

Optimal value for the whole problem is $D[r]$, where r is the root of T

Computational array:

$$D[u] = \max\left(1 + \sum_{grandchildren\ w\ of\ u} D[w], \sum_{children\ w\ of\ u} D[w]\right)$$

Exercises:

- Write down pseudocode for this DP algorithm.

- Analyze its runtime. Your algorithm should run in $O(|V|)$ time.

- Extend the algorithm to return a maximum independent set itself and not just its size