

CS 170

Efficient Algorithms and Intractable Problems

Lecture 14

Dynamic Programming IV

Nika Haghtalab and John Wright

EECS, UC Berkeley

HELP! WE'RE LOST!



HELP "CAR 54"...AND WIN CASH
54...\$1,000 PRIZES
ONE...\$10,000 GRAND PRIZE



Help Toody and Muldoon find the shortest round trip route to visit all 33 locations shown on the map.

All you do is draw connecting straight lines from location to location to show the shortest round trip route.

HERE'S THE CORRECT START...

Begin at Chicago, Illinois. From there, lines show correct route as far as Erie, Pennsylvania. Next, do you go to Carlisle, Pennsylvania or Wana, West Virginia? Check the easy instructions on back of this entry blank for details.



OFFICIAL RULES ON REVERSE SIDE

© PROCTER & GAMBLE 1962

One of the most famous Math/CS problems.

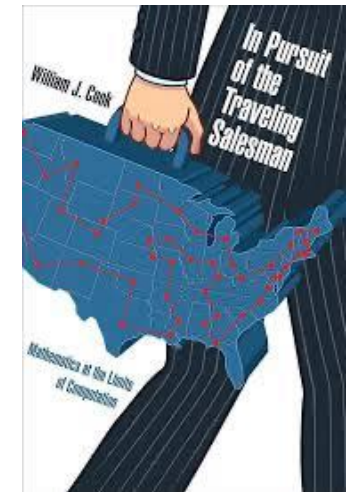
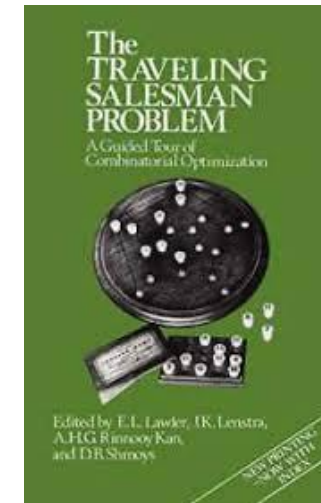
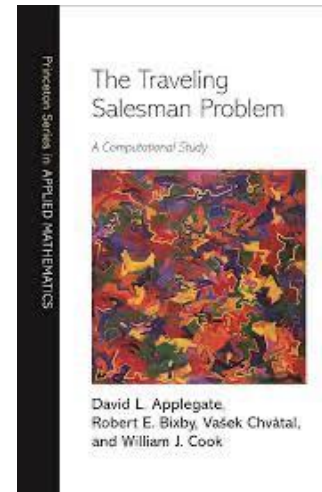
Notoriously difficult.

Recap

The DP algorithm is a substantial improvement over brute force. Take $n = 25$

□ $O(n!) \approx 10^{25}$

□ $O(n^2 2^n) \approx 10^{10}$



Step 1: Subproblems of TSP

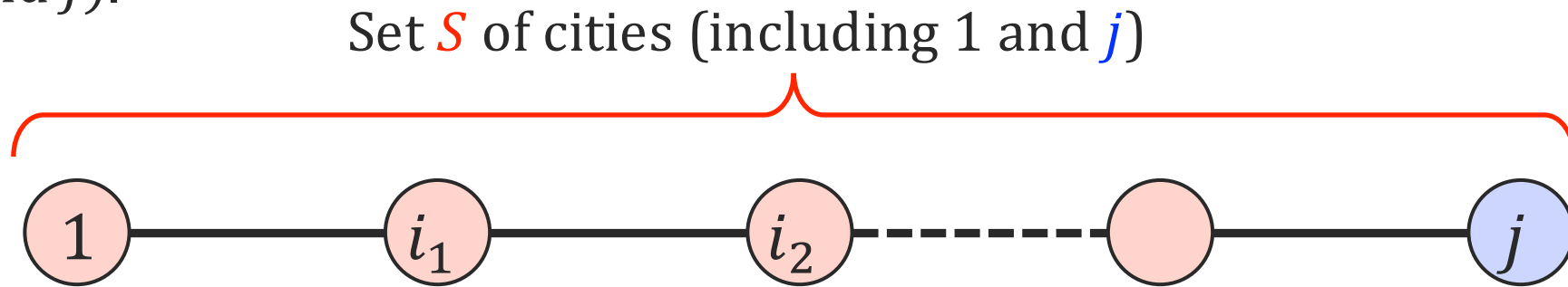
Recap

Input: cities $1 \dots n$ and pairwise distances d_{ij} between cities i and j .

Output: A “tour” of minimum total distance.

Think of subproblems as partial tour!

→ It starts from city 1, ends in city j , and passing through all cities in a set S (which includes 1 and j).



Subproblems: For all $j \leq n$ and $S \subseteq \{1, \dots, n\}$, s.t. S includes 1 and j .

$T[S, j]$ = length of the shortest path visiting all cities in S exactly once, starting from 1 and ending at j .

Step 2: Recurrence Relation for TSP

Recap

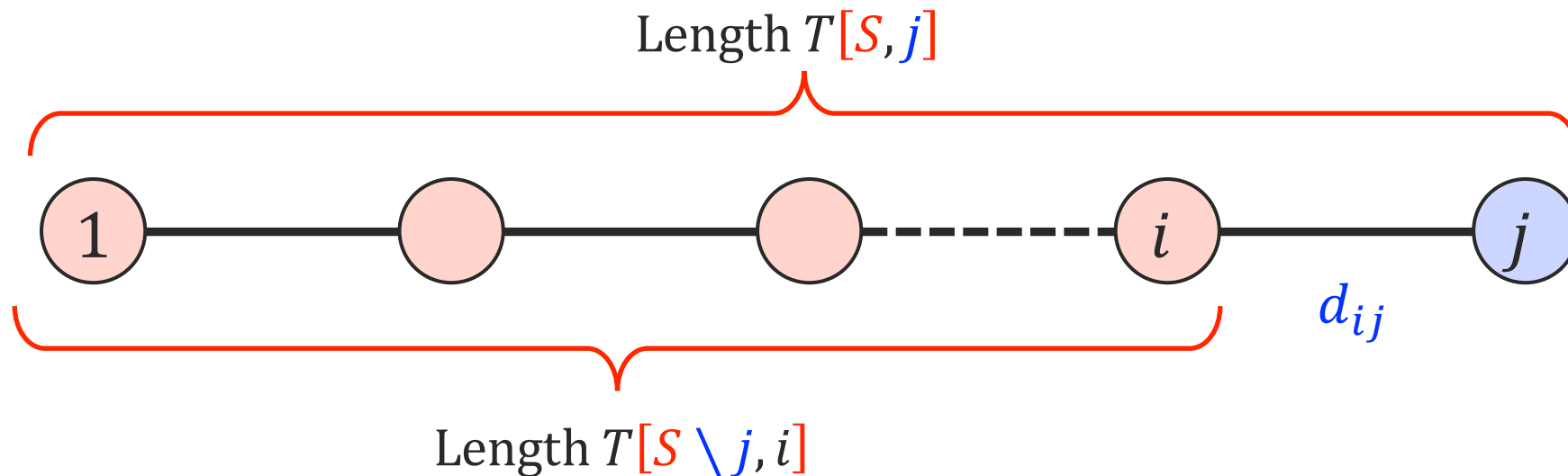
Input: cities $1 \dots n$ and pairwise distances d_{ij} between cities i and j .

Output: A “tour” of minimum total distance.

Subproblems: For all $j \leq n$ and $S \subseteq \{1, \dots, n\}$, s.t. S includes 1 and j .

$T[S, j]$ = length of the shortest path visiting **all cities in S exactly once**, starting from 1 and **ending at j** .

Step 2: Compute $T[S, j]$ using smaller subproblems.



Step 2: Recurrence Relation for TSP

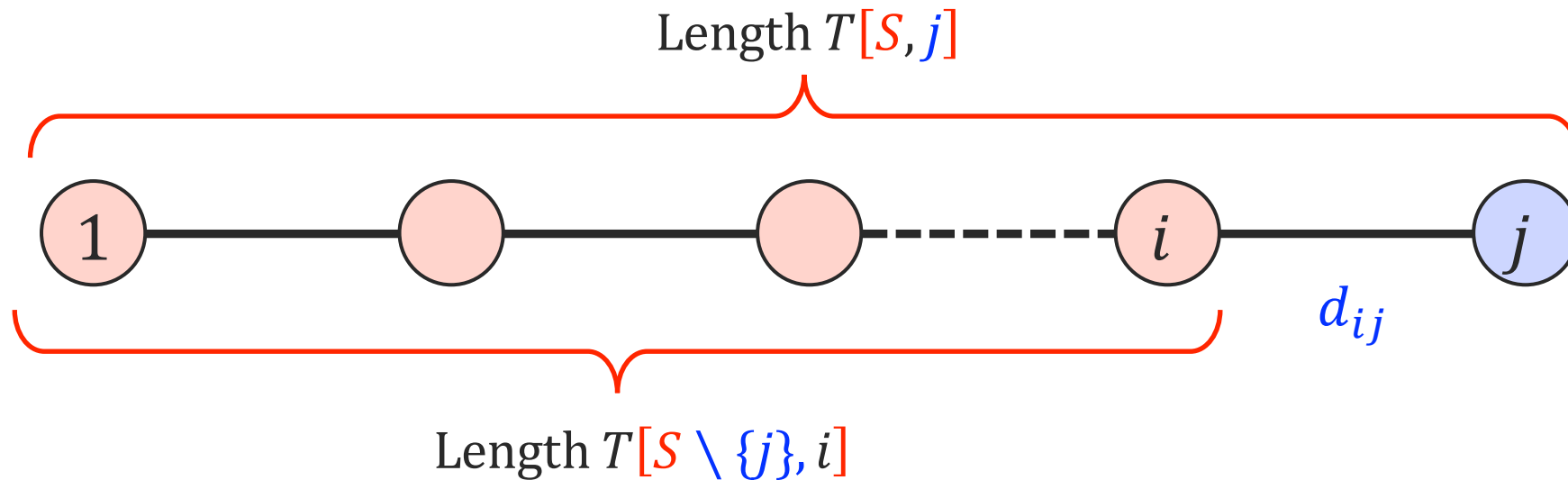
Recap

Input: cities $1 \dots n$ and pairwise distances d_{ij} between cities i and j .

Output: A “tour” of minimum total distance.

Recurrence relation: **We don't know which city i is the 2nd to last.**

→ Take the minimum over all $i \in S$ such that $i \neq j$.



$$\rightarrow T[S, j] = \min\{T[S \setminus \{j\}, i] + d_{ij} \mid i \in S \text{ and } i \neq j\}$$

Step 2: Base Cases and the Final Solution

Recap

Input: cities $1 \dots n$ and pairwise distances d_{ij} between cities i and j .

Output: A “tour” of minimum total distance.

Recurrence relation: $T[S, j] = \min\{T[S \setminus \{j\}, i] + d_{ij} \mid i \in S \text{ and } i \neq j\}$

Base cases: $T[\{1\}, 1] = 0$ and for all other S of size ≥ 2 , $T[S, 1] = \infty$.

No partial path allowed to start and ends at 1.

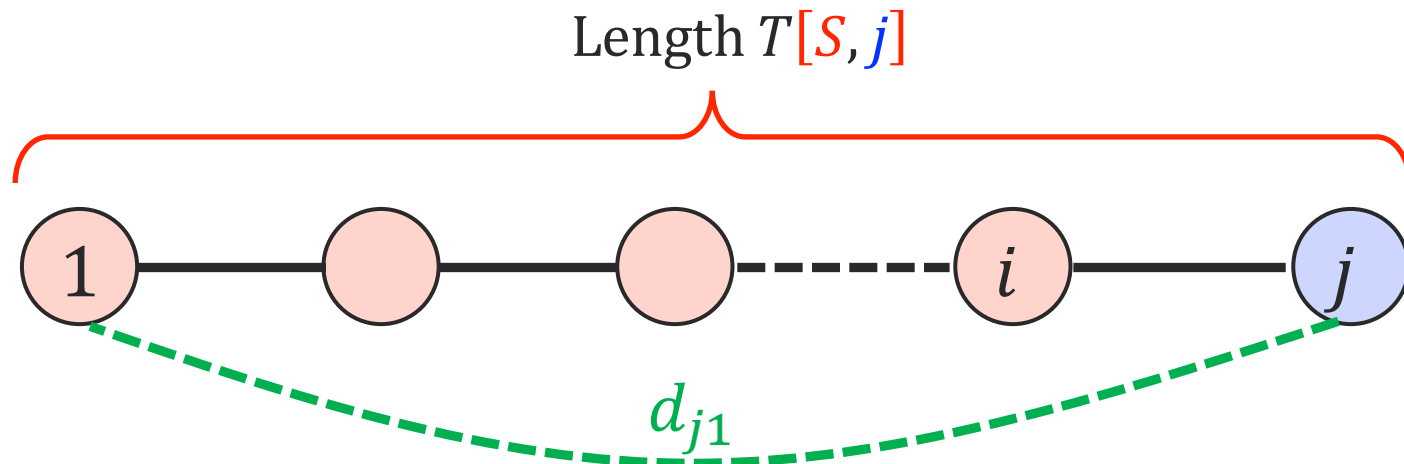
Final solution:

→ Add the final $(j, 1)$ edge cost:

$$T[\{1, \dots, n\}, j] + d_{j1}$$

→ Find the best j :

$$\min_{j \neq 1} T[\{1, \dots, n\}, j] + d_{j1}$$



Step 3: Design the algorithm

Input: cities $1 \dots n$ and pairwise distances d_{ij} between cities i and j .

Output: A “tour” of minimum total distance.

New

$O(2^n \times n)$ number of subproblems.

For each subproblem, we take min of $\leq n$ values:

□ Work per subproblem $O(n)$

Total runtime: $O(n^2 2^n)$.

TSP($d_{ij}: i, j \in [n]$)

An array T of size $2^n \times n$.

$T[\{1\}, 1] = 0$, $T[S, 1] = \infty$ for all sets S

For set size $s = 2, \dots, n$

For sets S , s.t. $|S| = s$, $1 \in S$

For $j \in S$

$$T[S, j] = \min_{i \in S: i \neq j} \{T[S \setminus \{j\}, i] + d_{ij}\}$$

return $\min_{j \neq 1} T[\{1, \dots, n\}, j] + d_{j1}$

Recap of the last 3 lectures

Dynamic Programming!

The recipe!

Step 1. Identify subproblems (aka optimal substructure)

Step 2. Find a recursive formulation for the subproblems

Step 3. Design the Dynamic Programming Algorithm

- Memo-ize computation starting from smallest subproblems and building up.

We saw a lot of examples already

→ Shortest Paths (in DAGs, Bellman-Ford, and All-Pair), Longest increasing subsequence, Edit distance, Knapsack, Traveling Salesman Problem, ...

This lecture

Last lecture on Dynamic Programming

→ Independent Sets on Trees

→ Chain Matrix Multiplication

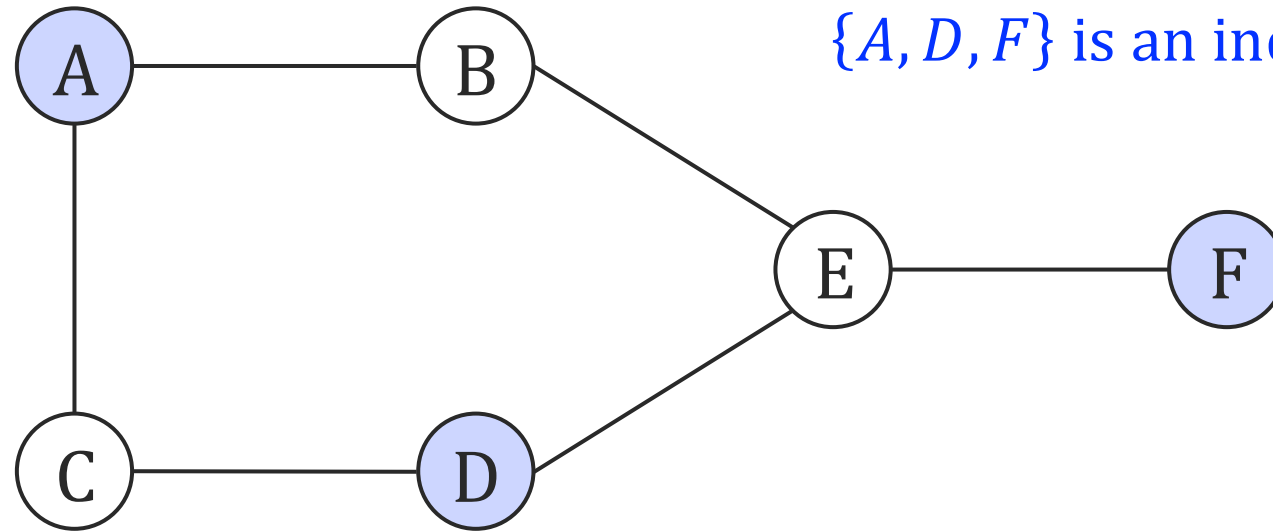
Best way to learn dynamic programming is by doing a lot of examples!

Independent Sets (in Trees)

Input: Undirected Graph $G = (V, E)$

Output: Largest “independent set” of G .

Definition: $S \subseteq V$ is an **independent set** of G if there are no edges between any $u, v \in S$.

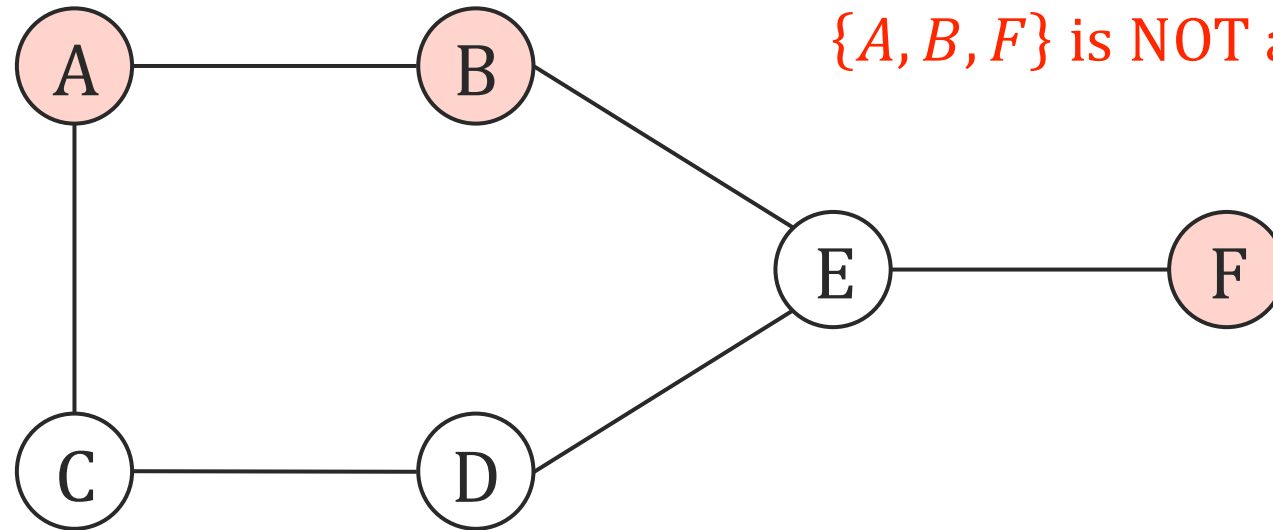


Independent Sets (in Trees)

Input: Undirected Graph $G = (V, E)$

Output: Largest “independent set” of G .

Definition: $S \subseteq V$ is an **independent set** of G if there are no edges between any $u, v \in S$.



$\{A, B, F\}$ is NOT an independent set.

Finding largest independent set **can't be done in polynomial** time in **general graphs**.
For **trees**, dynamic programming gives $O(|V|)$ algorithm!

Independent Sets in Trees

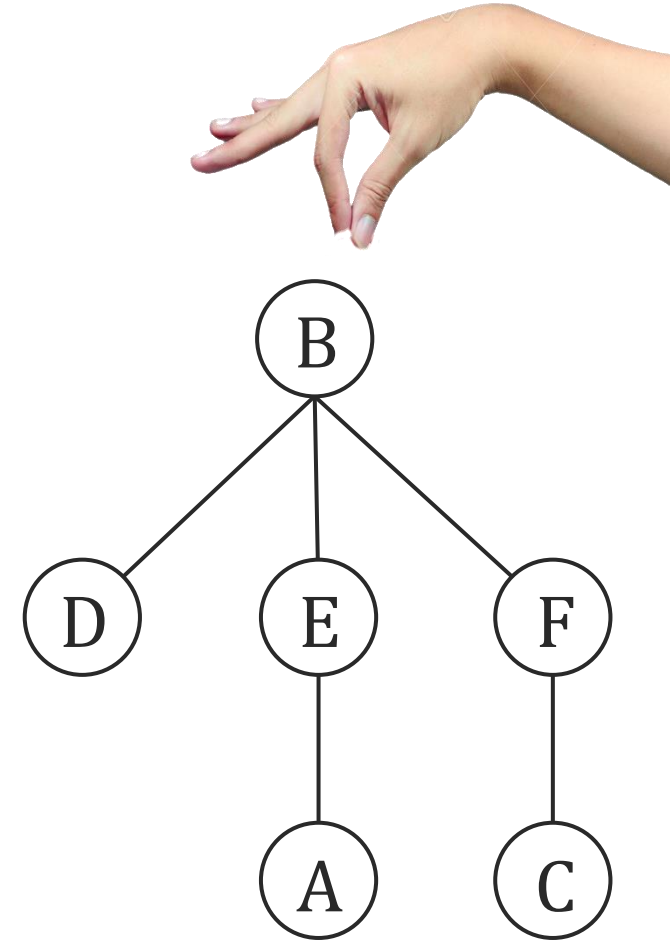
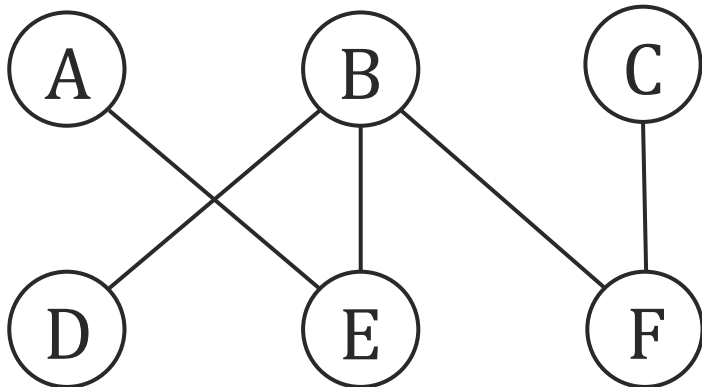
Input: Undirected Graph $G = (V, E)$ and G is a tree.

Output: Largest “independent set” of G .

Recall, trees don't have cycles!

→ We can pick a node of a tree and say that it's the **root**

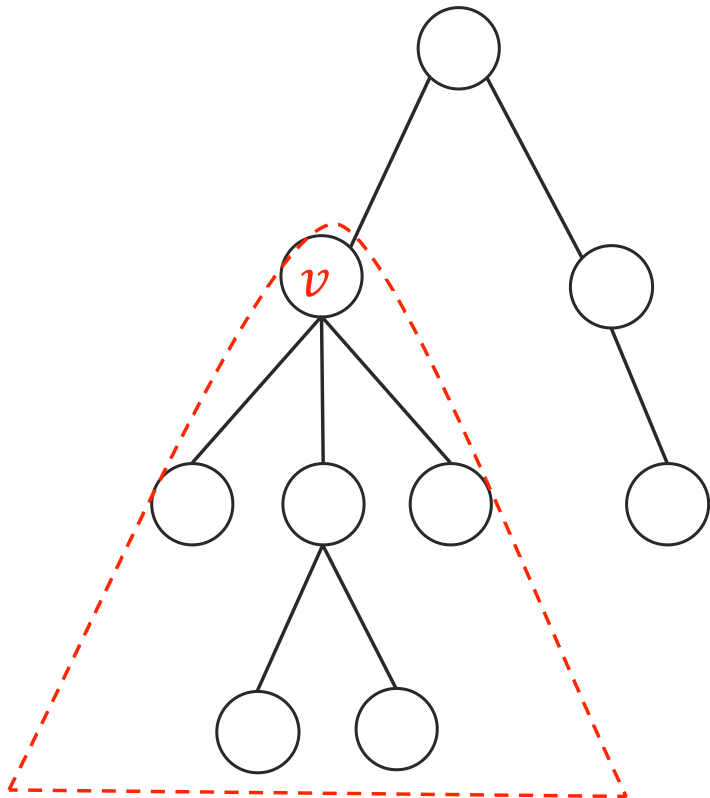
→ Rooted trees create a natural order between nodes, parent to children.



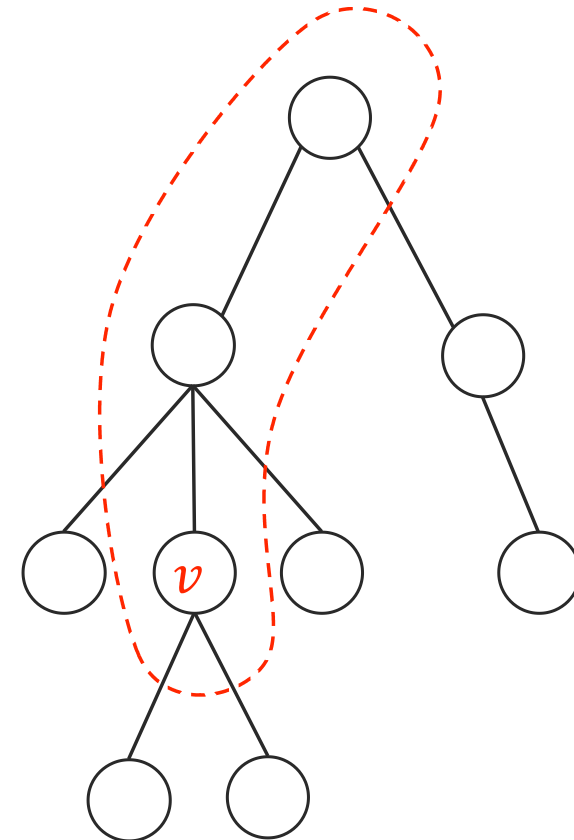
Which choice of subproblem is more appropriate?

Discuss

Max IS in the subtree
rooted at a node



Max IS in
the ancestors of a node



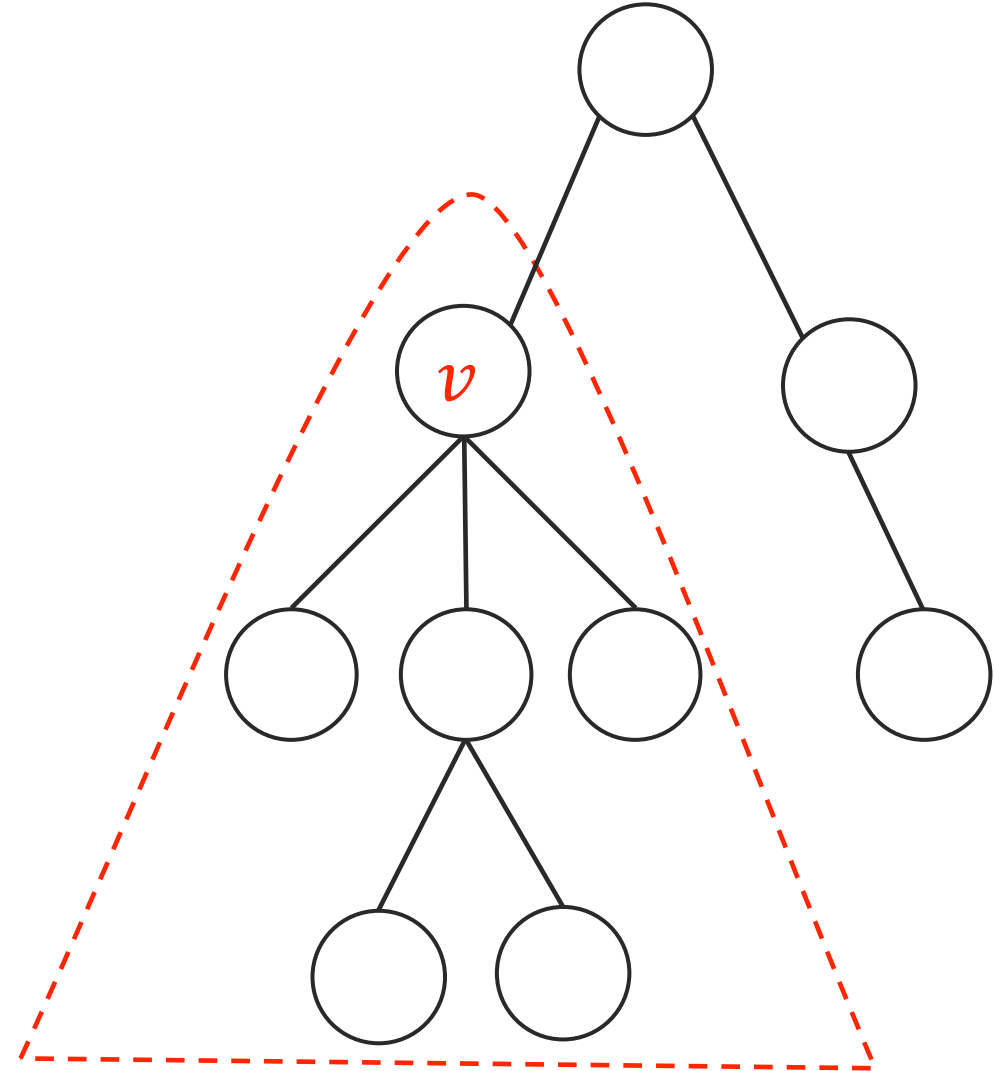
Step 1: Subproblems for Independent Sets

Input: Undirected Graph $G = (V, E)$ and G is a tree.

Output: Largest “independent set” of G .

Subproblems: For each $v \in V$

$I(v)$ = Size of max independent set in
subtree rooted at v .



Step 2: Recurrence for Independent Sets

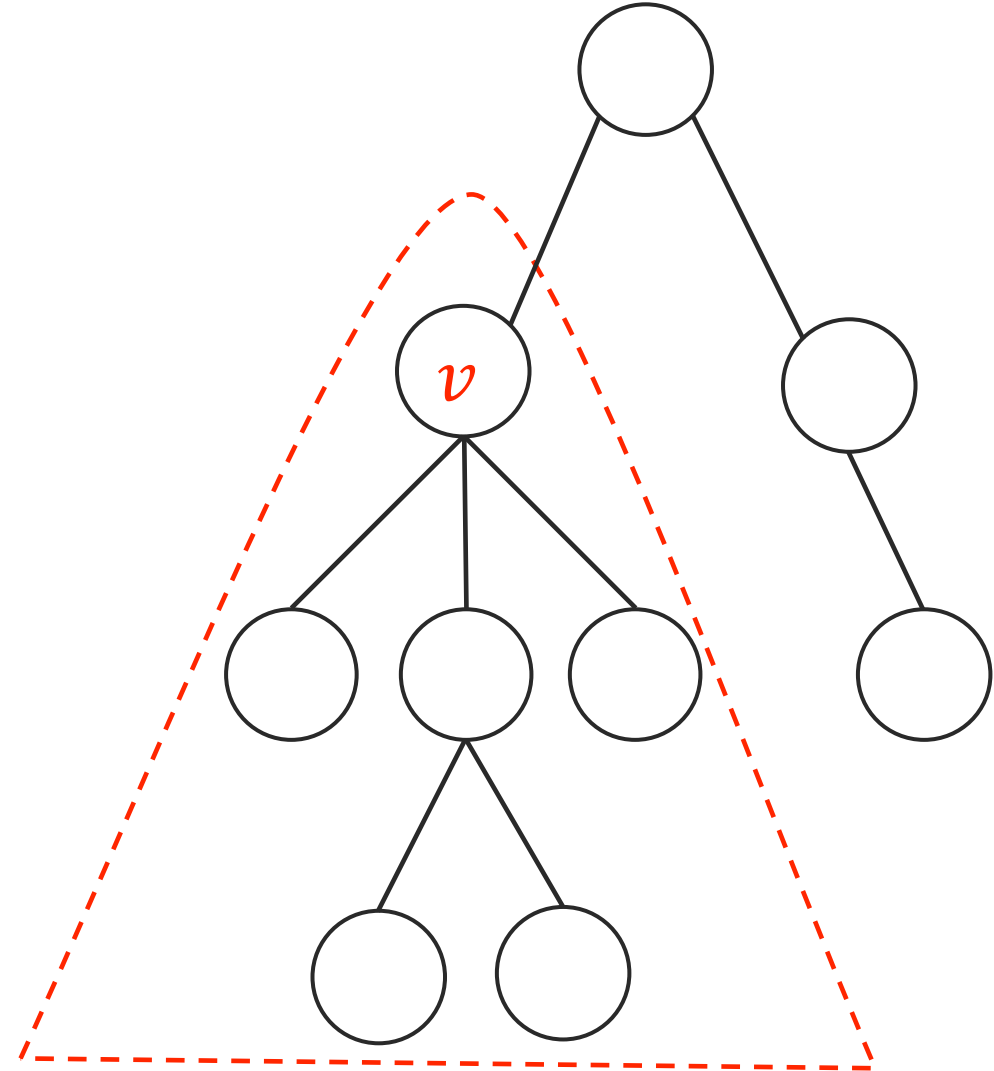
Input: Undirected Graph $G = (V, E)$ and G is a tree.

Output: Largest “independent set” of G .

Subproblems: For each $v \in V$

$I(v)$ = Size of max independent set in
subtree rooted at v .

Recurrence: Compute $I[v]$ using smaller
subproblems (its descendants)



Two Cases:

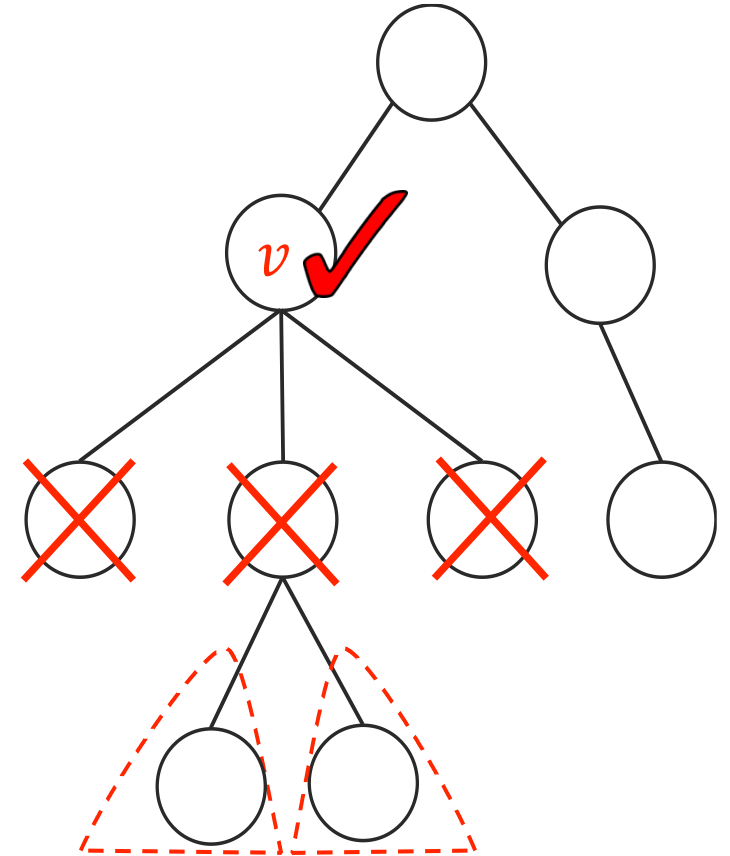
Recurrence: Compute $I[v]$ using smaller subproblems (its descendants)

Case 1: The optimal solution for $I[v]$ uses v .

None of the **children of v** can be in the independent set.

Recurse to the grandchildren levels:

$$I[v] = 1 + \sum_{u:\text{grandchild of } v} I[u]$$



Two Cases:

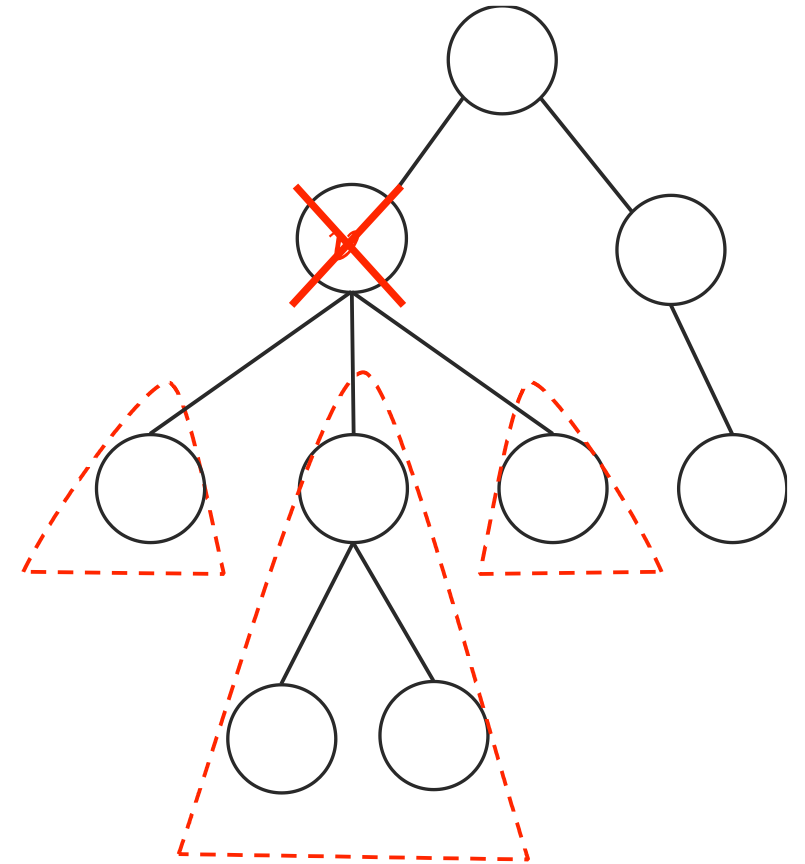
Recurrence: Compute $I[v]$ using smaller subproblems (its descendants)

Case 2: The optimal solution for $I[v]$ does NOT use v .

This doesn't restrict the optimal solution in the children of v .

Recurse to the children levels:

$$I[v] = \sum_{u: \text{child of } v} I[u]$$



Step 2: Recurrence for Independent Sets

Input: Undirected Graph $G = (V, E)$ and G is a tree.

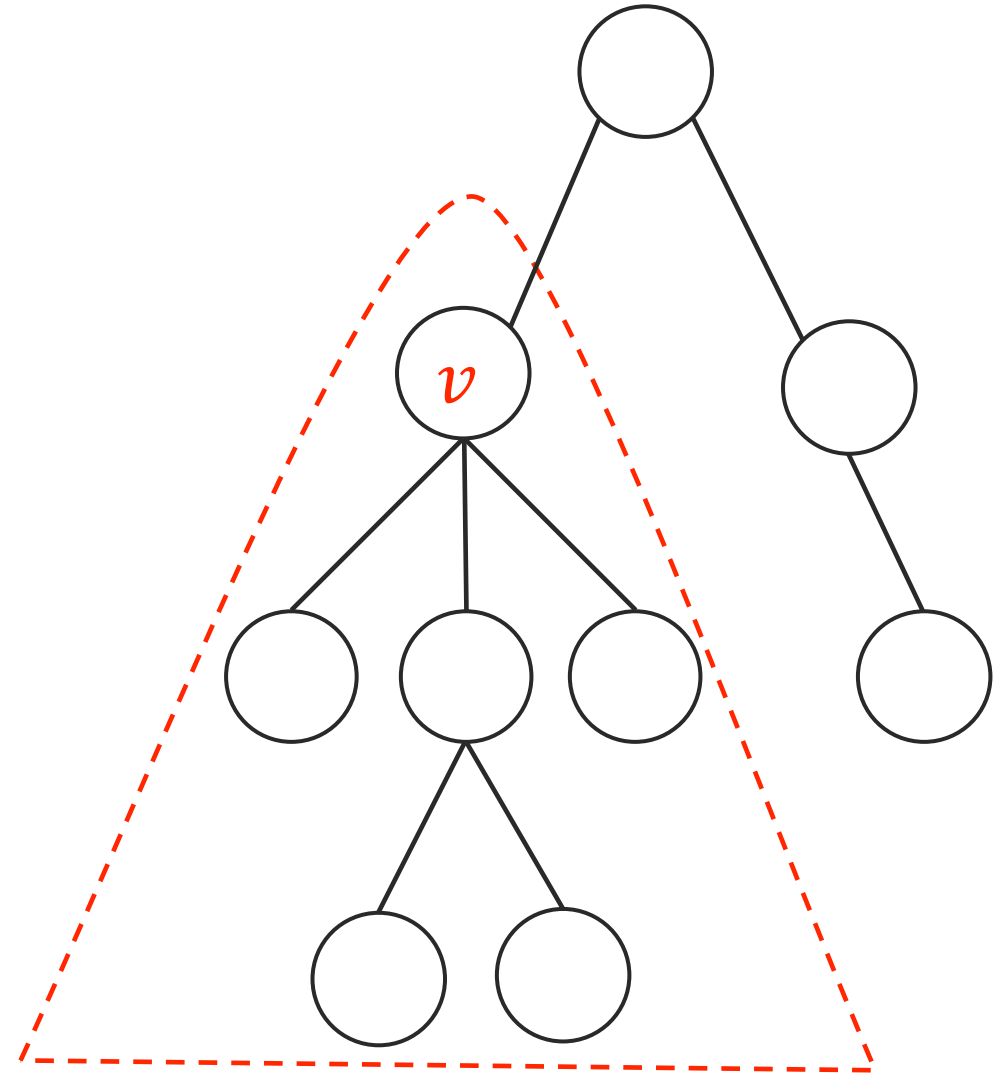
Output: Largest “independent set” of G .

Subproblems: For each $v \in V$

$I(v)$ = Size of max independent set in
subtree rooted at v .

Recurrence: Compute $I[v]$ using smaller
subproblems (its descendants)

$$I[v] = \max \left\{ 1 + \sum_{u: \text{grandchild of } v} I[u], \sum_{u: \text{child of } v} I[u] \right\}$$



Step 3: Design the Algorithm

Input: Undirected Graph $G = (V, E)$ and G is a tree.

Output: Largest “independent set” of G .

We need a data structure to store the tree easily.

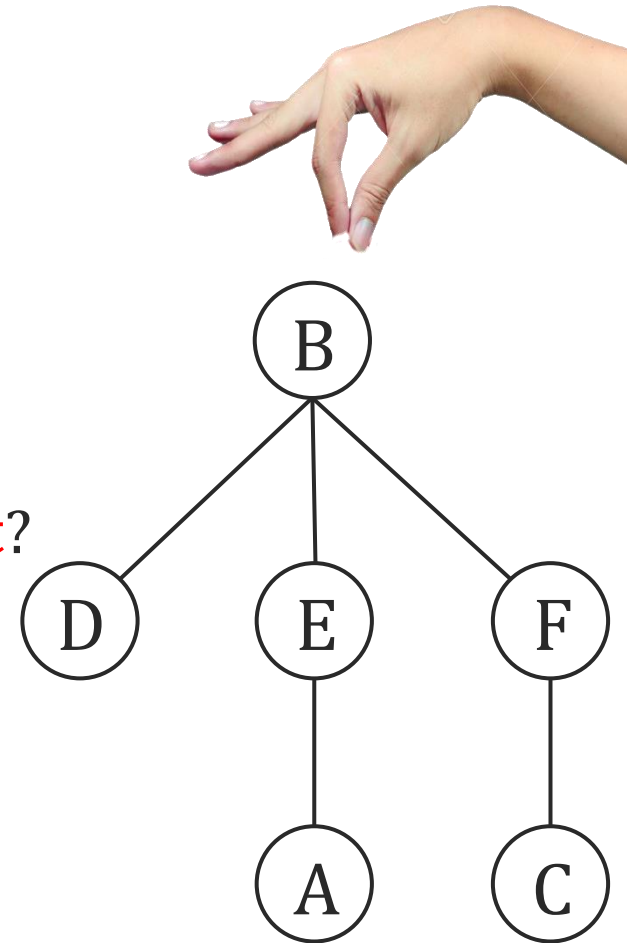
→ How to ensure that **every child is processed before the parent**?

Recall, **post** numbers in DFS(G):

- If u is a descendent of v : $post(u) < post(v)$.

Lecture 5-6
material!

Bottom-up: memo-ize in **increasing order** of **post** numbers, in any DFS traversal.



Step 3: Design the Algorithm

Input: Undirected Graph $G = (V, E)$ and G is a tree.

Output: Largest “independent set” of G .

1. In trees: $|E| = |V| - 1$.
 2. DFS Runtime = $O(|V|)$
 3. Each edge is looked at ≤ 2 times.
→ Once for its parent's subproblem.
□ Once for its grandparent's subproblem.
- Total work for all subproblems = $O(|E|) = O(|V|)$.

Total runtime: $O(|V|)$.

Independent-Set-Tree($G = (V, E)$)

An array I of size n .

sort $v_1 \dots v_n$ in increasing **post** order of DFS(G)

For $i = 1, \dots, n$

$$I[v_i] = \max \left\{ \begin{array}{l} 1 + \sum_{u: \text{grandchild of } v_i} I[u], \\ \sum_{u: \text{child of } v_i} I[u] \end{array} \right\}$$

return $I[v_n]$

3 Min Break and Attendance

Password: subtrees

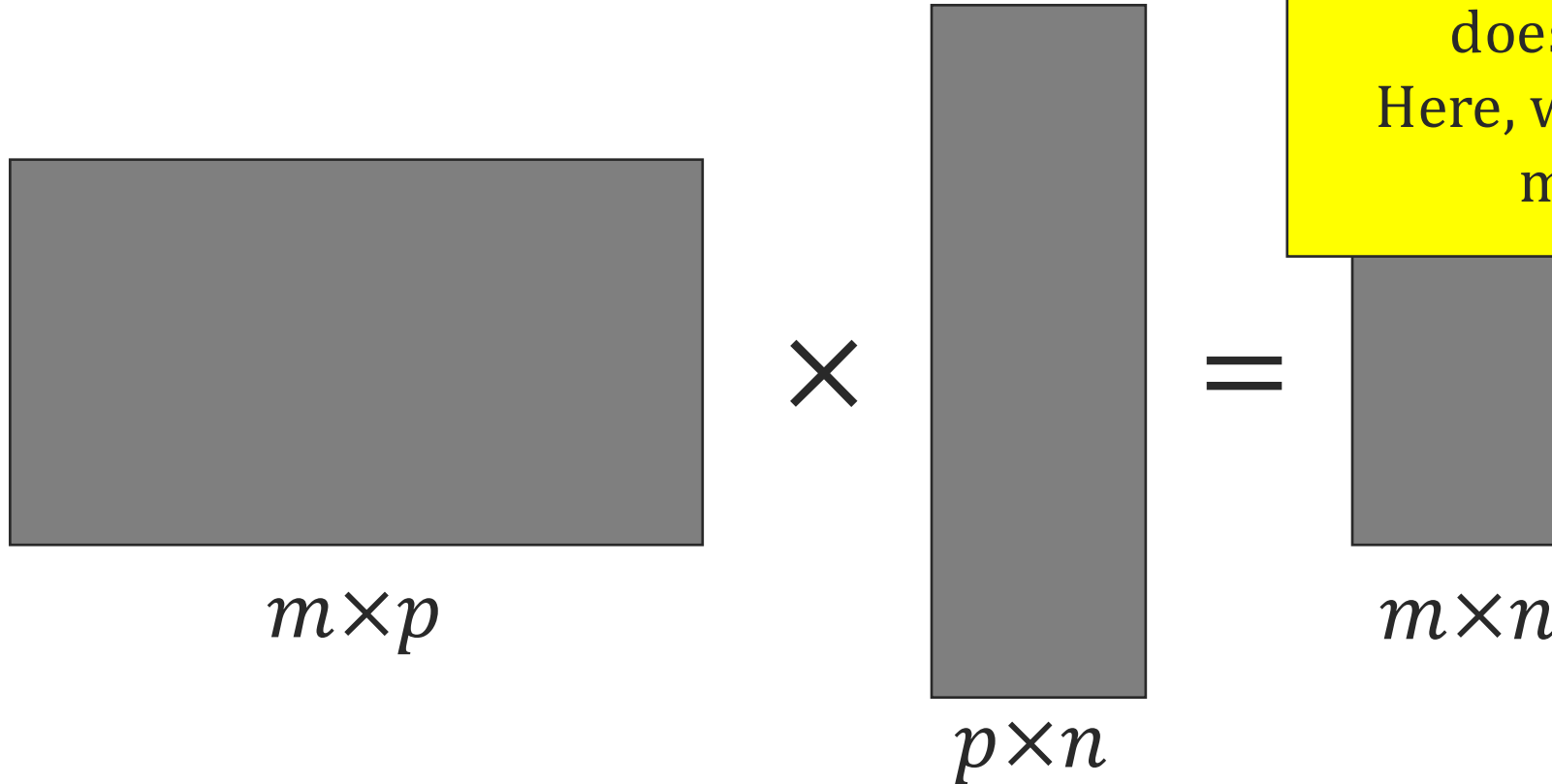


Sign in using @berkeley.edu

<https://forms.gle/W4zaMWqNzJmA3wMw6>

Chain Matrix Multiplication

Matrix Multiplication

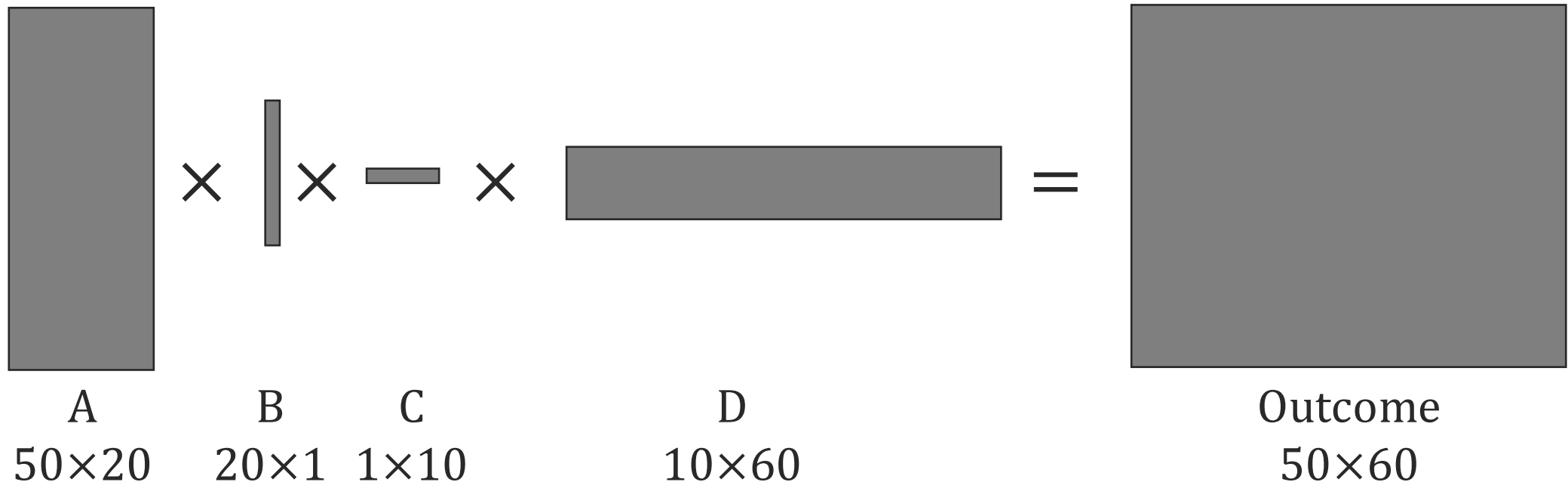


Lecture 2:
Fast matrix multiplication
does slightly better!
Here, we work with naïve
multiplication.

Number of operations:

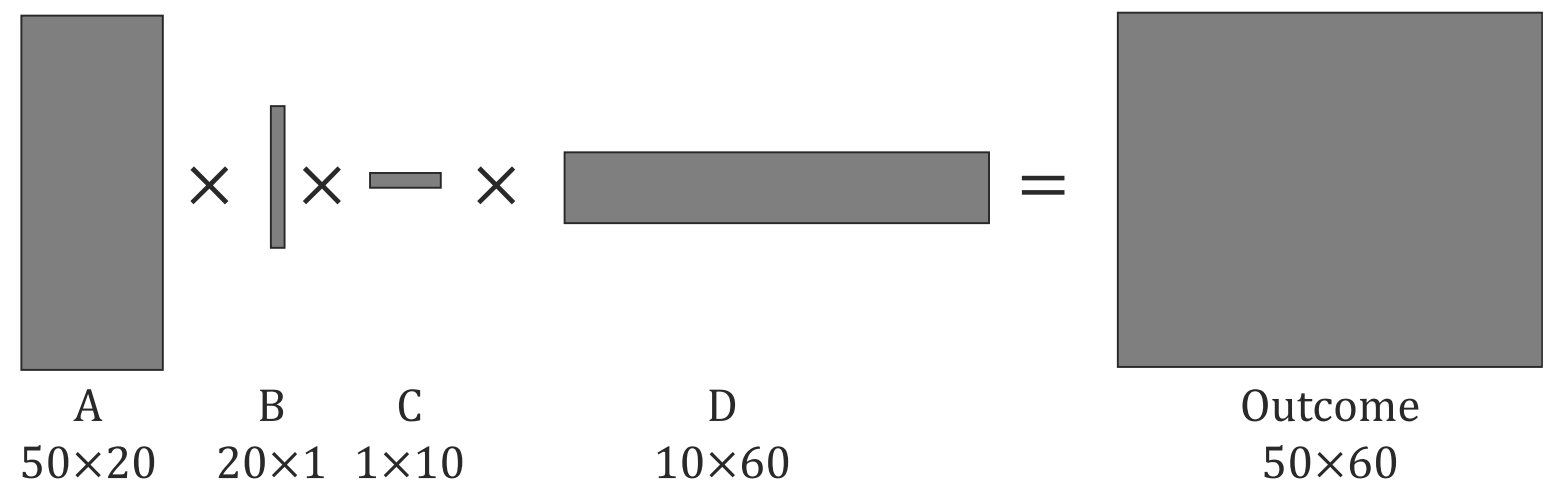
- Outcome matrix of size $m \times n$
- Each cell is a dot product of two vectors of length p , so $O(p)$
- Total: $O(mnp)$

Chain Matrix Multiplication



Matrix multiplication is associative (can put parenthesis anywhere), but not commutative (can't switch left and right order)

Chain Matrix Multiplication



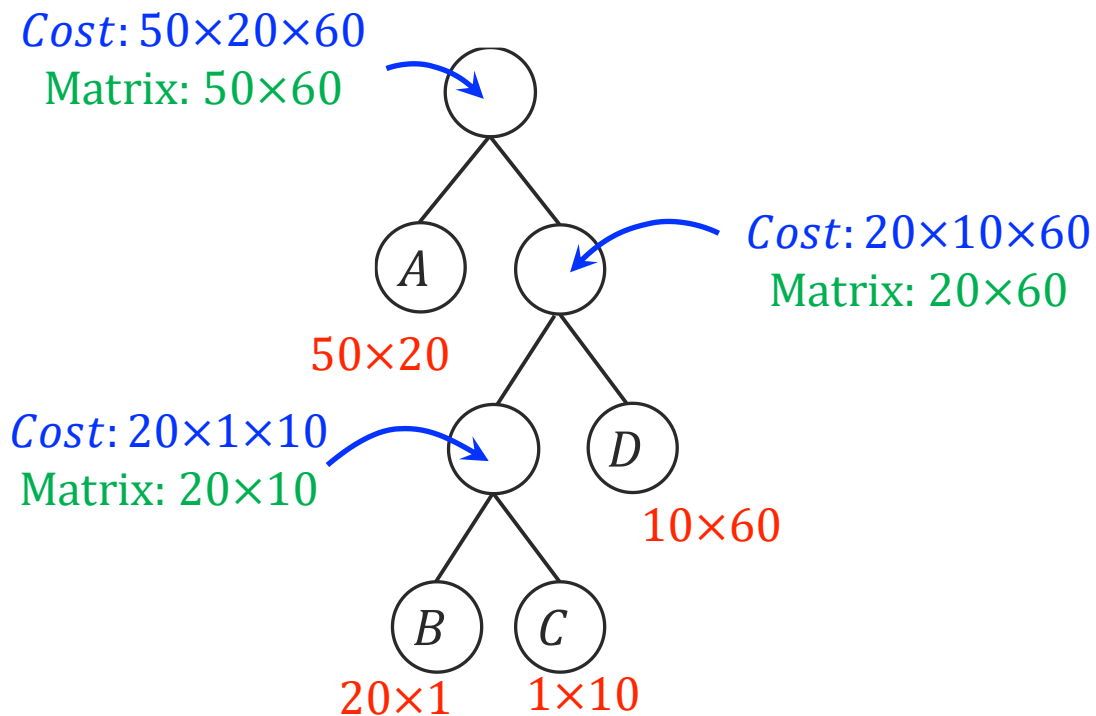
Parenthesization	Cost of Computation
$A \times ((B \times C) \times D)$	
$(A \times (B \times C)) \times D$	
$(A \times B) \times (C \times D)$	

Chain Matrix Multiplication

Input: Matrices $A_1, \dots, A_n$, where matrix A_i is of dimension $m_{i-1} \times m_i$.

Output: Minimum cost of multiplying $A_1 \times \dots \times A_n$.

Parenthesizations correspond to binary Trees



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

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

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

Step 1: Subproblems

Input: Matrices $A_1, \dots, A_n$, where matrix A_i is of dimension $m_{i-1} \times m_i$.

Output: Minimum cost of multiplying $A_1 \times \dots \times A_n$.

Subproblem choice: The cost of multiplying a contiguous subset of the matrices

$Cost[i, j]$ = Minimum cost of multiplying $A_i \times A_{i+1} \dots \times A_j$ for $i \leq j$

Why is this a good choice?

For a tree to be optimal, every subtree also has to be optimal.

Natural subproblem order, start from leaves and consider every subtree.

Step 2: Recurrence Relation

Input: Matrices $A_1, \dots, A_n$, where matrix A_i is of dimension $m_{i-1} \times m_i$.

Output: Minimum cost of multiplying $A_1 \times \dots \times A_n$.

Subproblem choice: The cost of multiplying a contiguous subset of the matrices

$Cost[i, j]$ = Minimum cost of multiplying $A_i \times A_{i+1} \dots \times A_j$ for $i \leq j$

To multiply $A_i \times A_{i+1} \dots \times A_j$, we have to parenthesize it, say by splitting at k :

$$A_i \times A_{i+1} \dots \times A_j = (A_i \times \dots \times A_k) \times (A_{k+1} \times \dots \times A_j):$$

$$\begin{aligned} Cost[i, j] &= Cost[i, k] + Cost[k + 1, j] + \text{Cost of multiplying } m_{i-1} \times m_k \text{ by } m_k \times m_j \text{ matrices} \\ &= Cost[i, k] + Cost[k + 1, j] + m_{i-1} \times m_k \times m_j \end{aligned}$$

For the best parenthesization of the $A_i \times A_{i+1} \dots \times A_j$:

$$Cost[i, j] = \min_{k: i \leq k \leq j} \{ Cost[i, k] + Cost[k + 1, j] + m_{i-1} \times m_k \times m_j \}$$

Order of Computation

$$Cost[i, j] = \min_{k: i \leq k \leq j} \{Cost[i, k] + Cost[k + 1, j] + m_{i-1} \times m_k \times m_j\}$$

Go by the increasing size of $j - i$:

- Base case: $Cost[i, i] = 0$ for all $i = 1, \dots, n$
- Start from $s = j - i$ being $1, 2, \dots, n - 1$
- Fill in diagonally

Step 3: Memo-ization

Input: Matrices $A_1, \dots, A_n$, where matrix A_i is of dimension $m_{i-1} \times m_i$.

Output: Minimum cost of multiplying $A_1 \times \dots \times A_n$.

Number of subproblems is $O(n^2)$

Per subproblem:

- Minimize over $O(n)$ choices for identity of k .
 - Each value takes $O(1)$ to compute
- Total of $O(n)$ cost per subproblem.

Total runtime $O(n^3)$

Chain-Matrix-Mult($m_0, m_1, \dots, m_n$)

An array C of size $n \times n$

For $i = 1, \dots, n$, $C[i, i] = 0$

For $s = 1 \dots, n - 1$

For $i = 1, \dots, n - s$

$j \leftarrow i + s$

$$C[i, j] = \min_{k: i \leq k \leq j} \left\{ \text{Cost}[i, k] + \text{Cost}[k + 1, j] + m_{i-1} \times m_k \times m_j \right\}$$

Return $C[1, n]$

Summary of Subproblem

Remember the Recipe

The recipe!

Step 1. Identify subproblems (aka optimal substructure)

Step 2. Find a recursive formulation for the subproblems

Step 3. Design the Dynamic Programming Algorithm

- Memo-ize computation starting from smallest subproblems and building up.

What makes for good subproblems?

- Not too many of them (the more subproblems the slower the DP algorithm)
- Must have enough information in it to compute subproblems recursively (needed for step 2).

Common Subproblem on Arrays

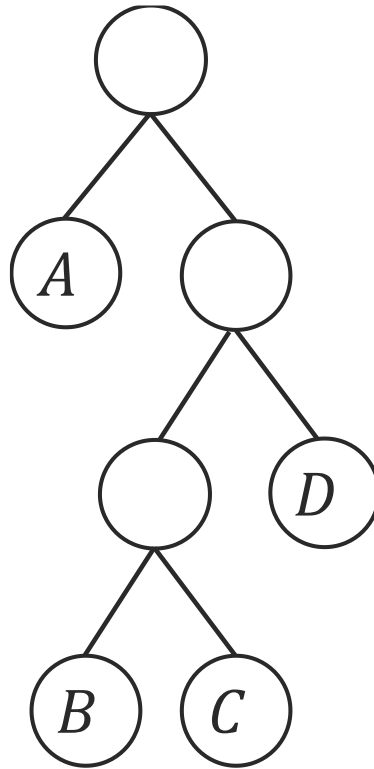
The input is an array $x_1, \dots, x_n$ and subproblem is $x_1, \dots, x_i$

The input is an array $x_1, \dots, x_n$ and subproblem is $x_i, \dots, x_j$

The input is two array $x_1, \dots, x_n$ and $y_1, \dots, y_n$ and subproblems $x_1, \dots, x_i$ and $y_1, \dots, y_j$ or in some cases $x_i, \dots, x_j$ and $y_r, \dots, y_s$.

Common Subproblems on Trees

The input is a tree (or something that can be interpreted as a tree), the subproblems are subtrees



Common Subproblems for Graphs

You might need more creativity!

Problem might be about cycles (like Traveling salesperson), but it's easier to think about subpaths as subproblems:

- It is harder to recurse from a big cycle to a smaller cycles
- It is easier to recurse from a longer path to a shorter path

Problem might be about paths (like All-Pair Shortest Path, or TSP), but it helps to track internal vertices:

- Subproblems may need to take into account sets of vertices
- Sets like $\{x_1, \dots, x_j\}$ for all j (e.g., Floyd Warshall) or all subsets of $\{x_1, \dots, x_n\}$ (e.g., Traveling Saleperson).

Wrap up

We did lots of dynamic programming!

Dynamic programming can be best learned by practice! Do lots more example at home.

Next time:

→ Linear Programming