CS 170
Efficient Algorithms and Intractable Problems

Lecture 14
Dynamic Programming IV (updated)

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

Nika's OH after class today:
→ Meet at the podium of the entrance and walk to nearby benches.
→ Submit request for 1-1 TA. Meeting by today
→ We will finish midterm regrades later this week
→ HW 7 due on Saturday

Next few weeks:
→ John Wright will be lecturing
→ I will be back for some fun lectures towards the end of the semester!

Remember him?!

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

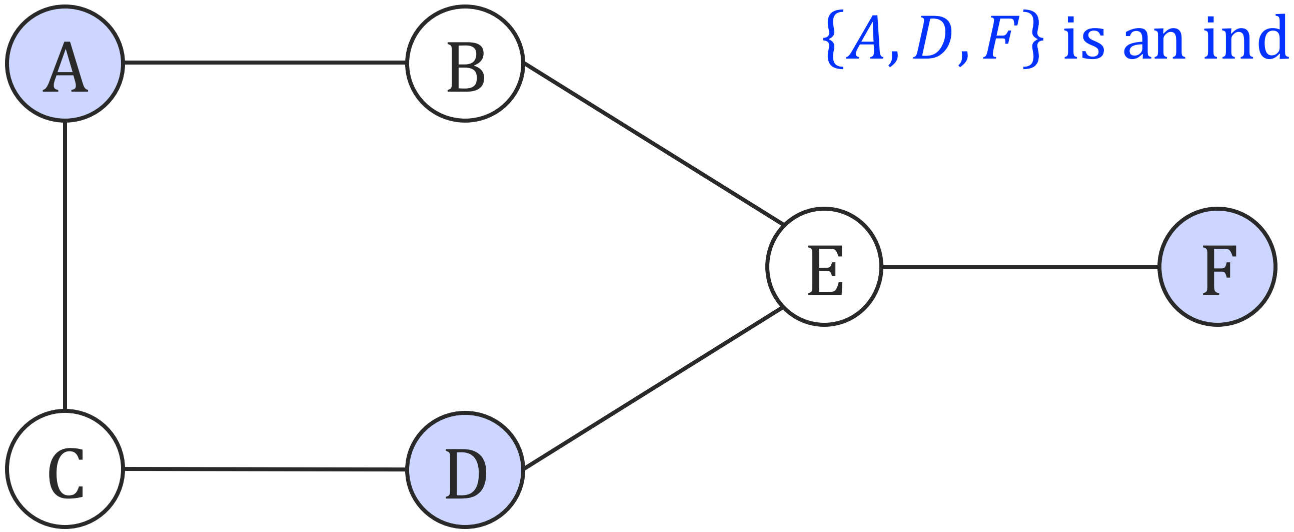
Best way to learn dynamic programming is by doing a lot of examples!

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 $u, v \in S$.



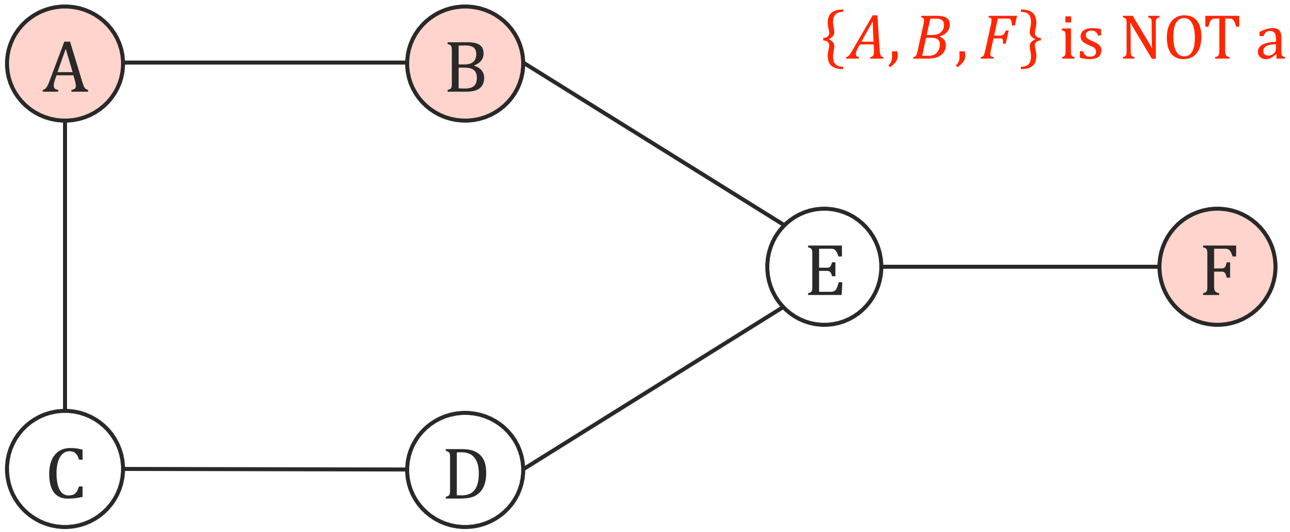
$\{A, D, F\}$ is an independent set.

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

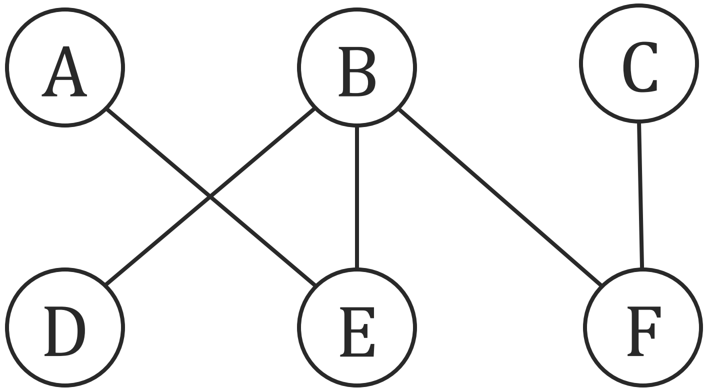
<u>Input</u>: Undirected Graph $G = (V, E)$ and G is a tree.

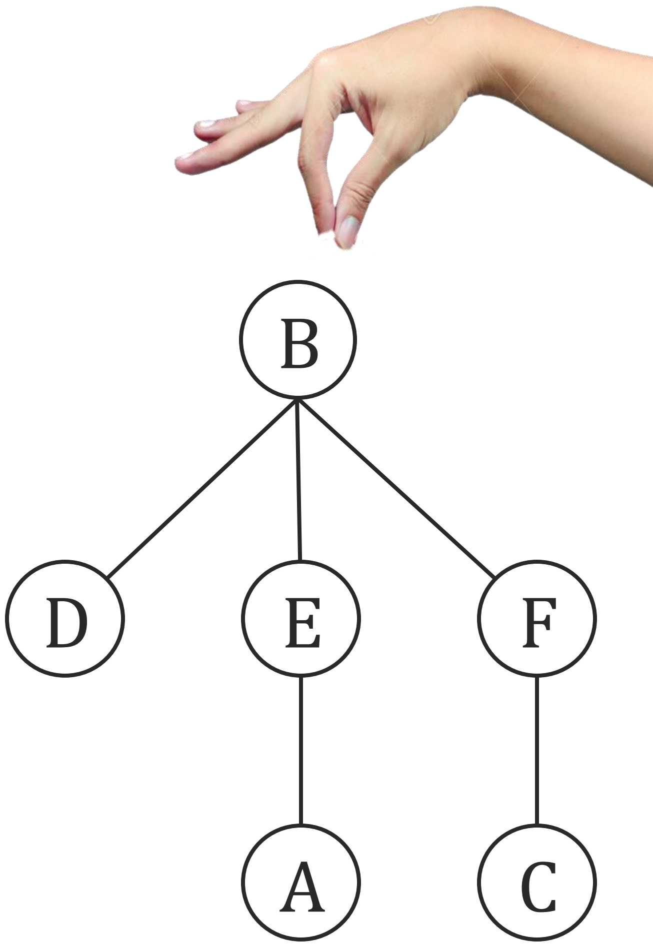
<u>Output</u>: Largest "independent set" of G.

Recall, trees don't have cycles!

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

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

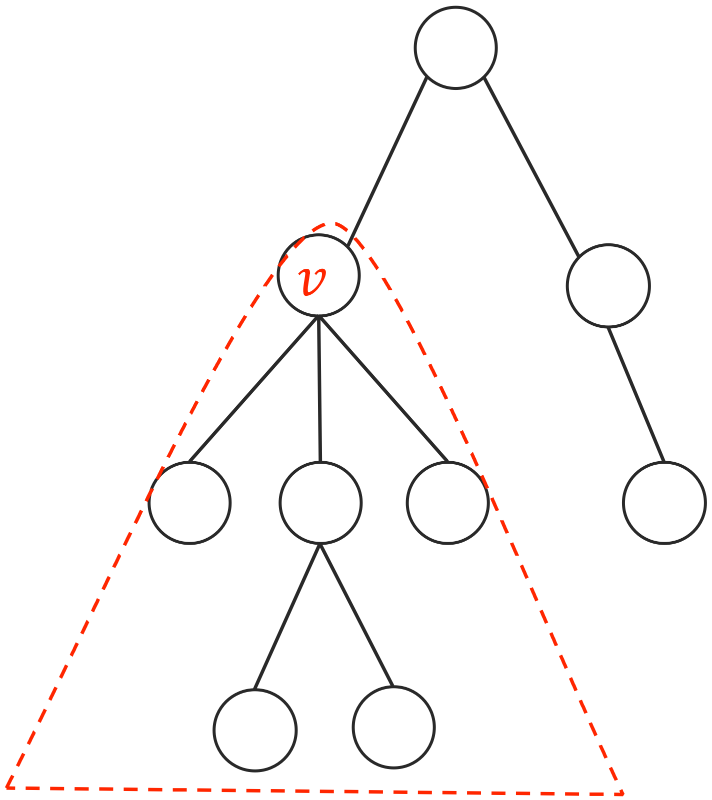




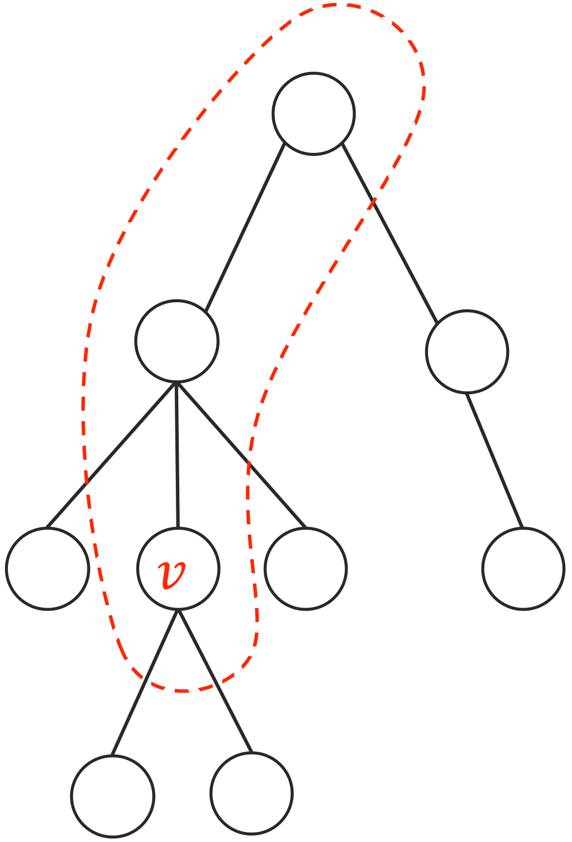
Which choice of subproblem is more appropriate?

Discuss

Max IS in the subtree rooted at a node



Max IS in the ancestors of a node



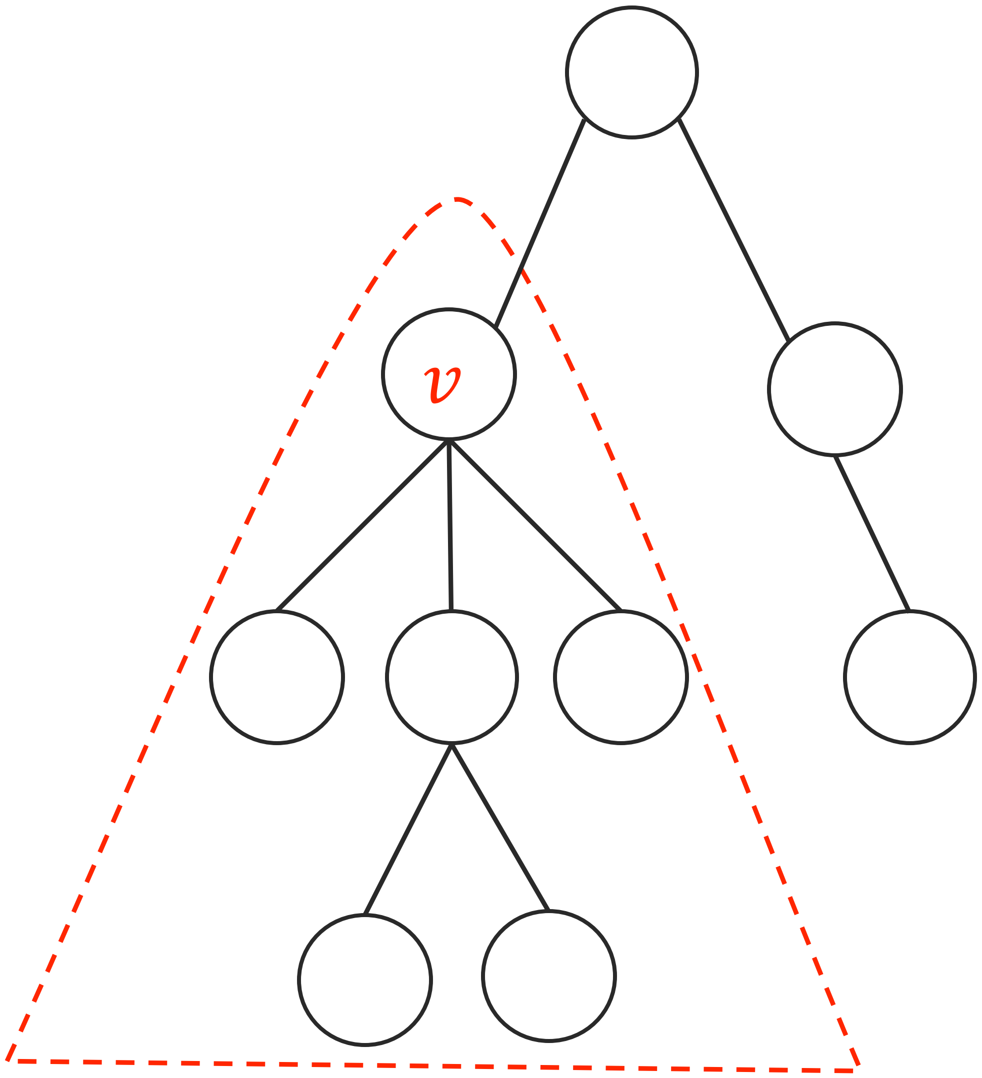
Step 1: Subproblems for Independent Sets

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

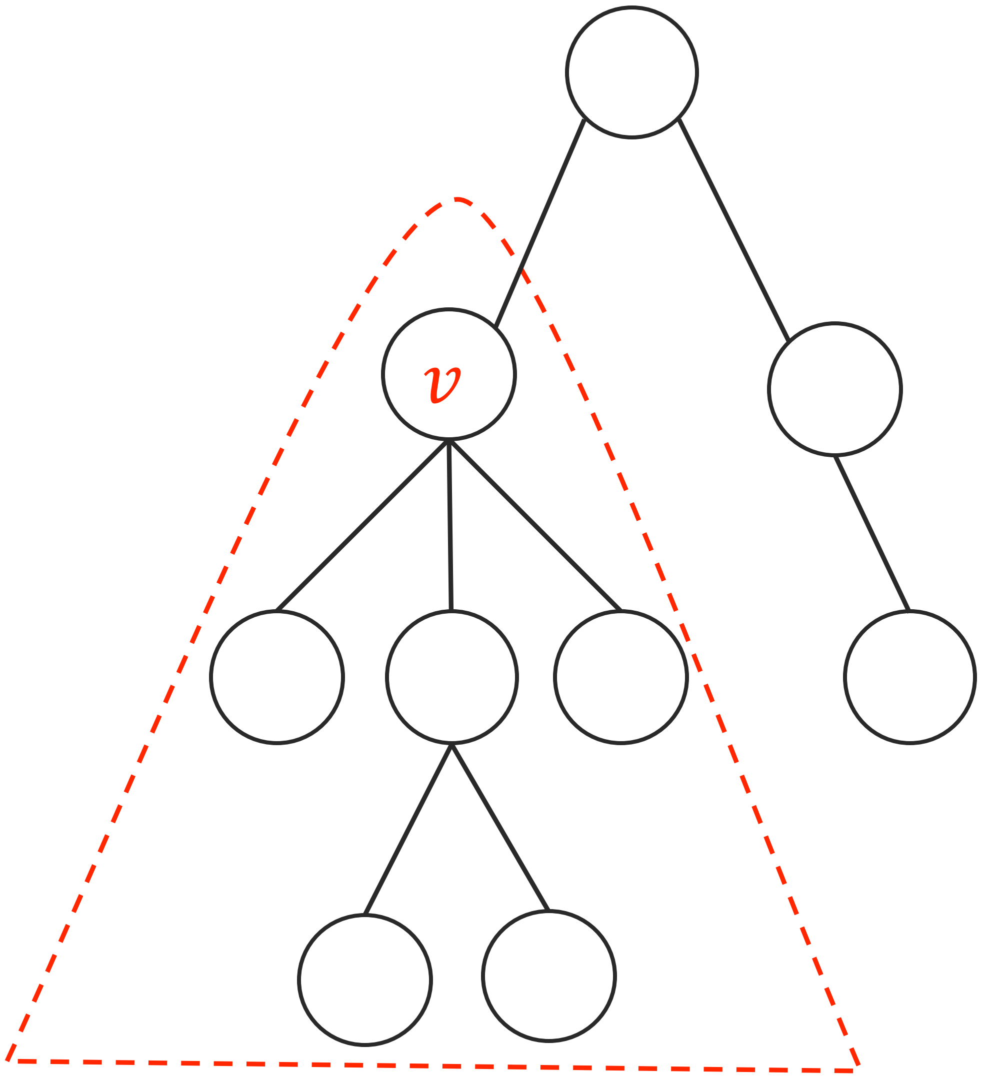
Input: Undirected Graph $G = (V, E)$ and G is a tree.

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:

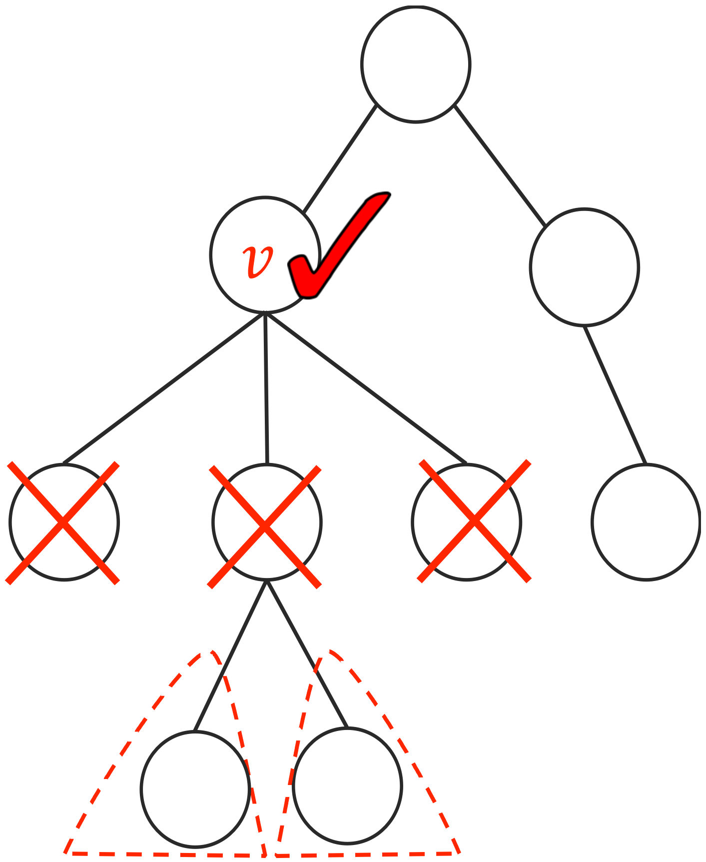
Recurrence: Compute $I[v]$ using smaller subproblems (its descendants)

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:

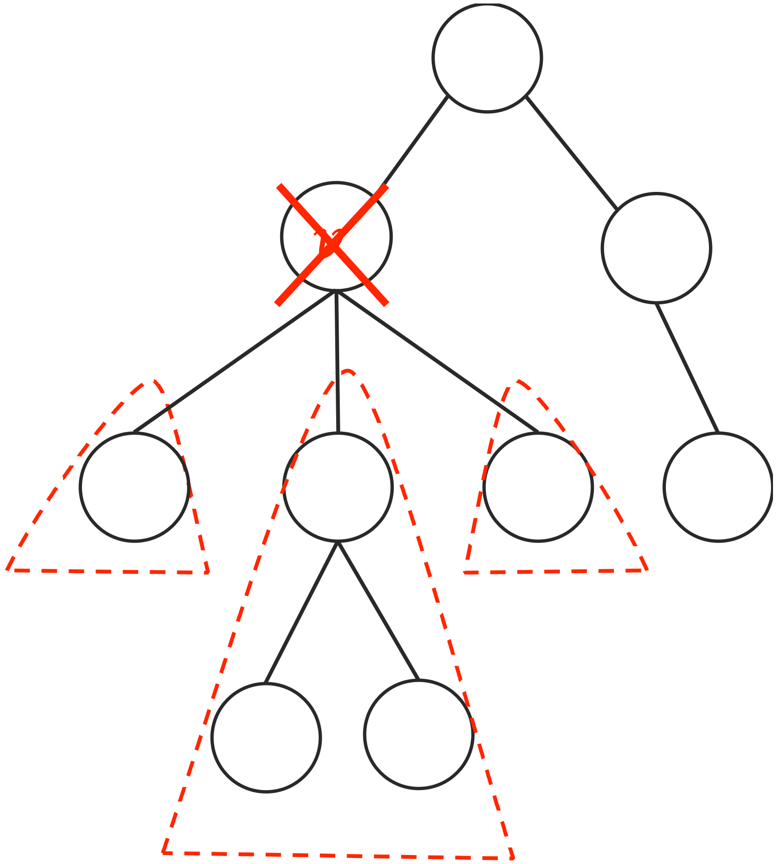
Recurrence: Compute $I[v]$ using smaller subproblems (its descendants)

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.

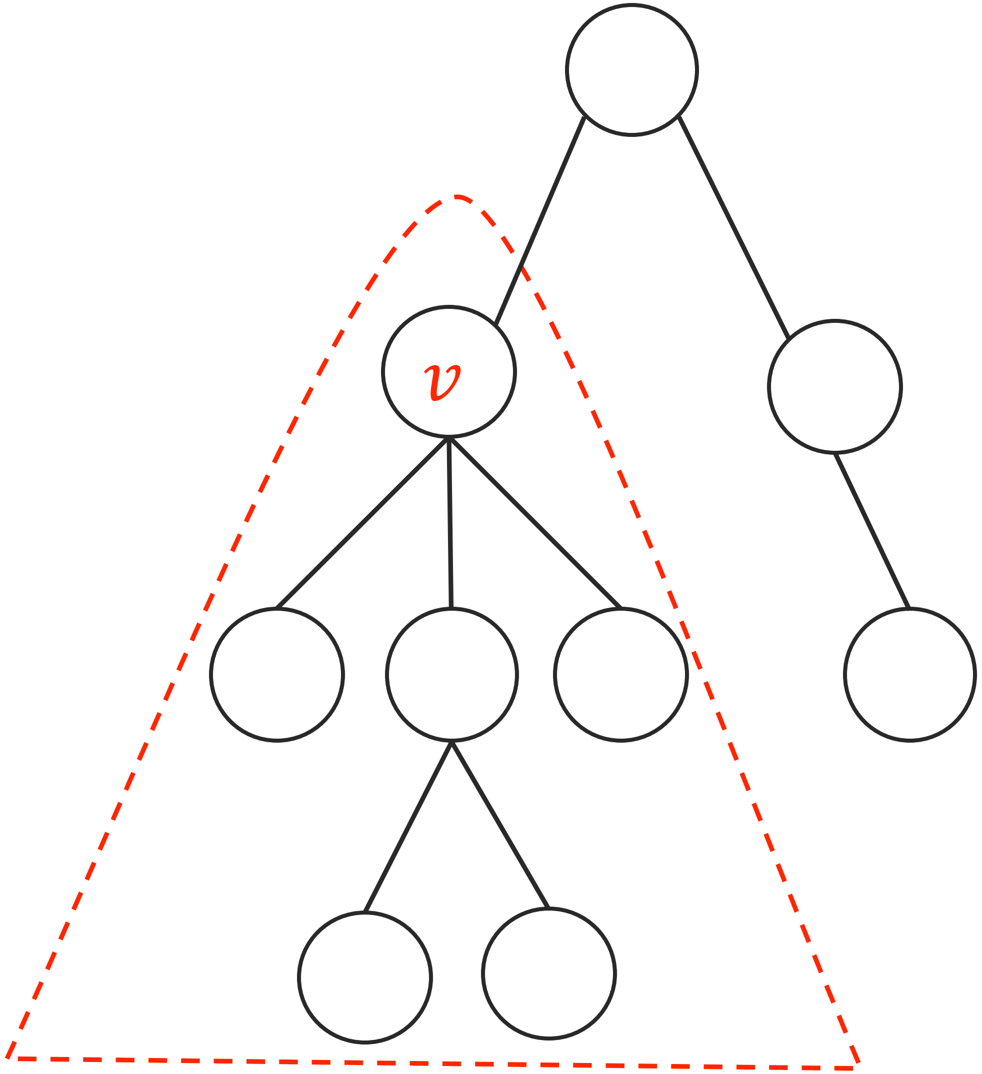
Output: Largest "independent set" of G.

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)

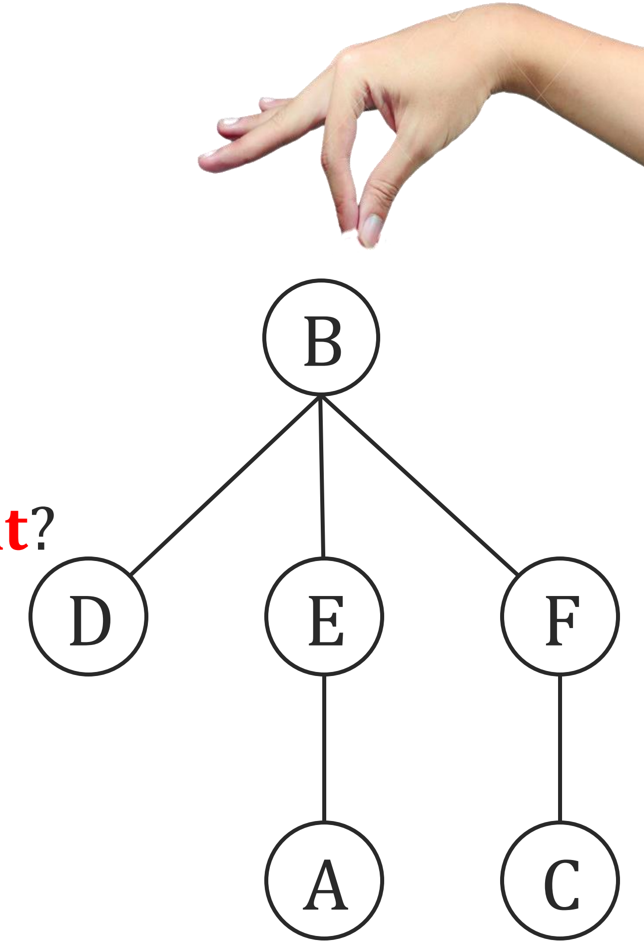
$$I[v] = \max\left\{1 + \sum_{u:\text{grandchild of } v} I[u],\ \sum_{u:\text{ child of } v} I[u]\right\}$$



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**?

Lecture 5-6 material!

Recall, post numbers in DFS(G):

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

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 $\leq$ 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 $i = 1, \dots, n$

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

return $I[v_n]$

3 Min Break and Attendance

Password: subtrees

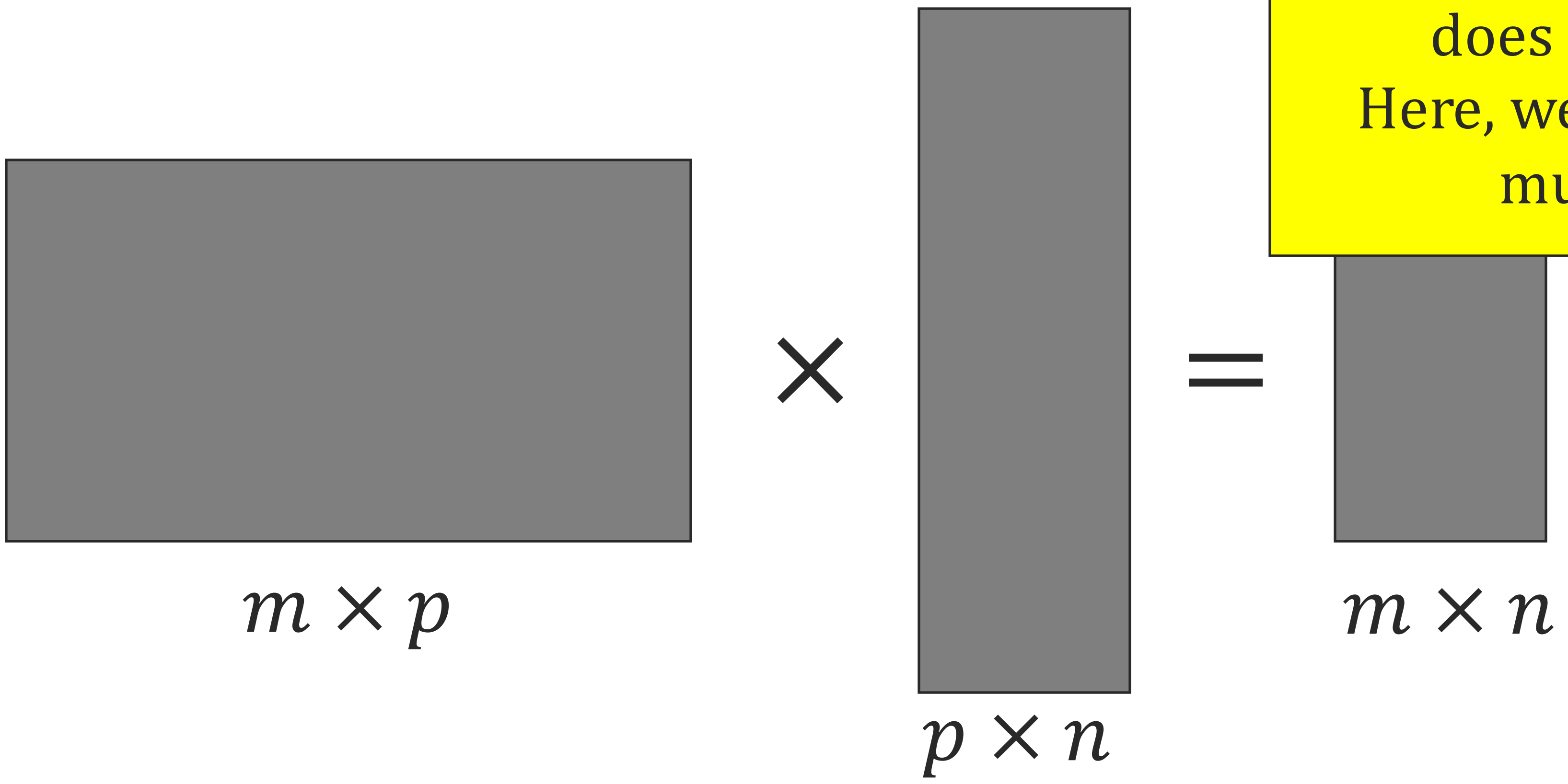
Sign in using @berkeley.edu

https://forms.gle/W4zaMWqNzJmA3wMw6

Chain Matrix Multiplication

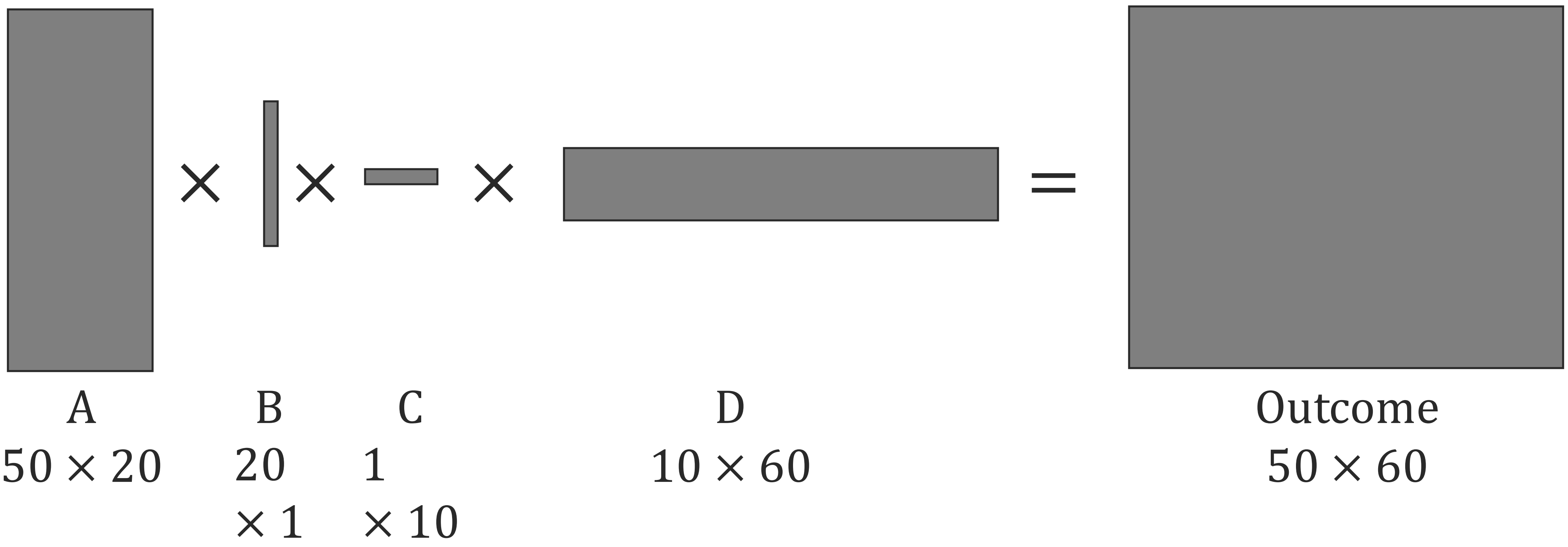
Matrix Multiplication

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



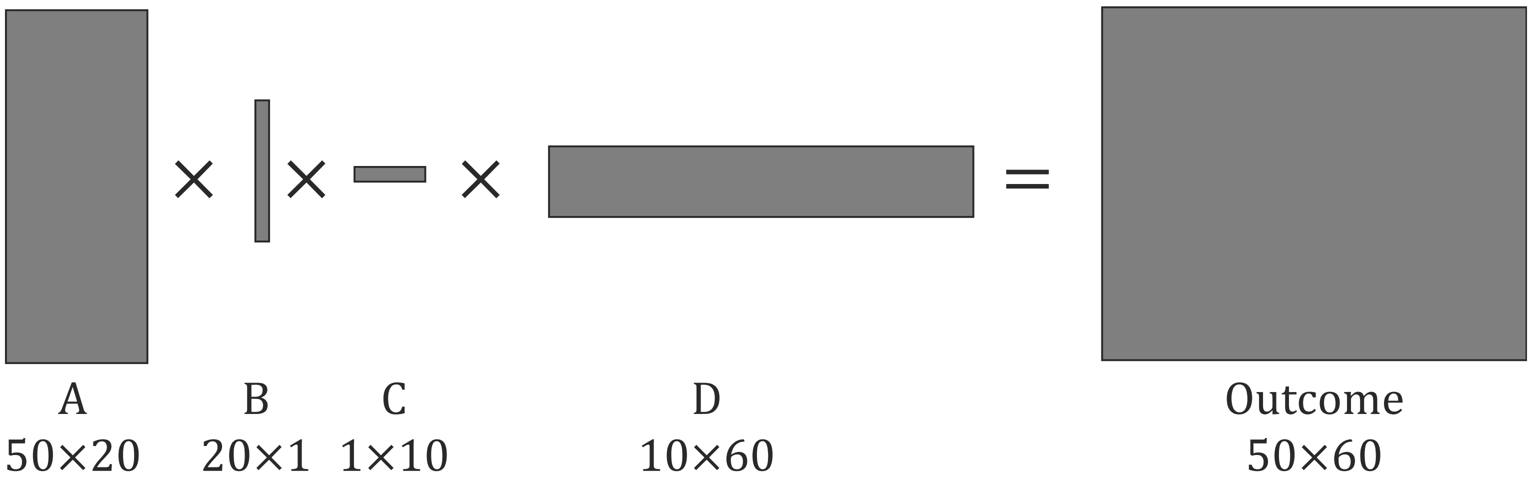
Number of operations:
→ Outcome matrix of size $m \times n$
→ Each cell is a dot product of two vectors of length p, so O(p)
→ Total: $O(mnp)$

Chain Matrix Multiplication



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

Chain Matrix Multiplication

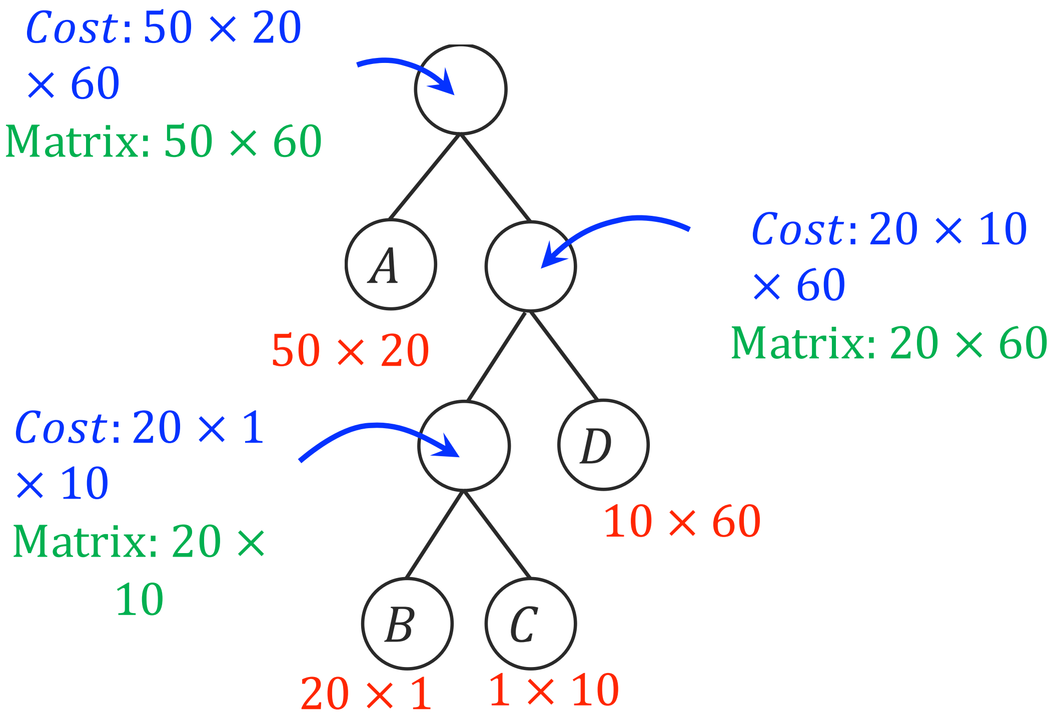


Parenthesization	Cost of Computation
$A \times ((B \times C) \times D)$	
$(A \times (B \times C)) \times D$	
$(A \times B) \times (C \times D)$	

Chain Matrix Multiplication

Input: Matrices $A_1, \ldots, A_n$, where matrix A_i is of dimension $m_{i-1} \times m_i$.
Output: Minimum cost of multiplying $A_1 \times \ \ldots, \times A_n$.

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 $A_1, \ldots, A_n$, where matrix A_i is of dimension $m_{i-1} \times m_i$.
Output: Minimum cost of multiplying $A_1 \times \ldots, \times A_n$.

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

$Cost[i, j]$ =Minimum cost of multiplying $A_i \times A_{i+1} \ldots, \times A_j$ 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 $A_1, \ldots, A_n$, where matrix A_i is of dimension $m_{i-1} \times m_i$.
Output: Minimum cost of multiplying $A_1 \times \ldots, \times A_n$.

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

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

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

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

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

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

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

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

This slide has been updated to fix a typo in the recurrence relation

Order of Computation

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

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

This slide has been updated to fix a typo in the recurrence relation

Step 3: Memo-ization

<u>Input</u>: Matrices $A_1, \dots, A_n$, where matrix A_i is of dimension $m_{i-1} \times m_i$.
<u>Output</u>: 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$(m_0, m_1, \cdots, m_n)$

An array C of size $n \times n$

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

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

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

$j \leftarrow i + s$

$$C[i, j] = \min_{k: i \le k \le j} \begin{Bmatrix} Cost[i, k] + Cost[k+1, j] \\ + m_{i-1} \times m_k \times m_j \end{Bmatrix}$$

Return $C[1, n]$

This slide has been updated to fix a typo in the recurrence relation

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

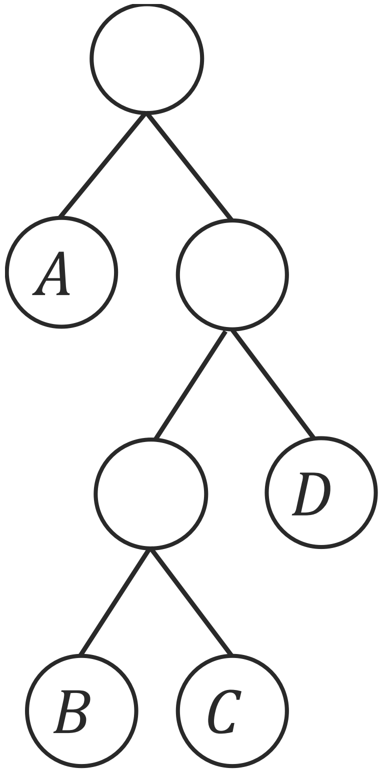
The input is an array $x_1, \ldots, x_n$ and subproblem is $x_1, \ldots, x_i$

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

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

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 $\{x_1, \ldots, x_j\}$ for all j (e.g., Floyd Warshall) or all subsets of $\{x_1, \ldots, x_n\}$ (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