CS 170
Efficient Algorithms and Intractable Problems

Lecture 14
Dynamic Programming III

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

Homeworks:
→HW7 was released a bit late, so it will be due on Tuesday instead.
→ BUT, no HW Party or additional OH on Tuesdays. Start early and go to Friday/Monday HWP and earlier OHs.

Midterm grades coming soon.
→ We will make an announcement on Ed.

Recap of the last 2 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

→ Fibonacci
→ Shortest Paths (in DAGs, Bellman-Ford, and All-Pair)
→ Longest increasing subsequence
→ Edit distance
→ Knapsack (with repetition)

This lecture

Even more examples!

→ Knapsack (without repetition)

→ Traveling Salesman Problem

→ Independent Sets on Trees

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

By doing more examples today, we will also develop intuition about how to choose subproblems (Recipe's step 1).

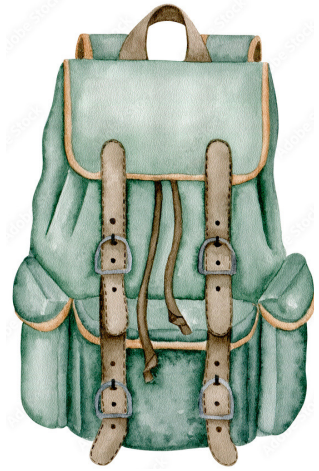
Recap of Knapsack

Knapsack Recap

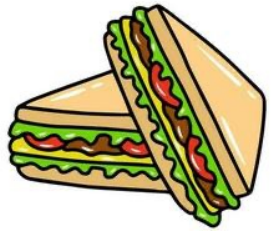
All integers!

Input: A weight capacity W, and n items with (weights, values), $(w_1, v_1), \cdots, (w_n, v_n)$.

Output: Most valuable combination of items, whose total weight is at most W.



W = 10

Item				
Weight:	6	3	4	2
Value:	30	14	16	9

Last lecture

With repetition:
1 tent + 2 sandwiches = **48 value**
Weight =10

This lecture

Without repetition:
1 tent + 1 stove = **46 value**
Weight =10

Step 1: Knapsack Subproblems

Can we still use the same subproblems

$K(c)$ = best value achievable for knapsack of capacity c, for c $\leq W$?

Challenge: We are only allowed one copy of an item, so the subproblem needs to "know" what items we have used and what we haven't.

We need a different way of tracking subproblems!

Idea: Solve knapsack for

- smaller sets of items and smaller capacities!

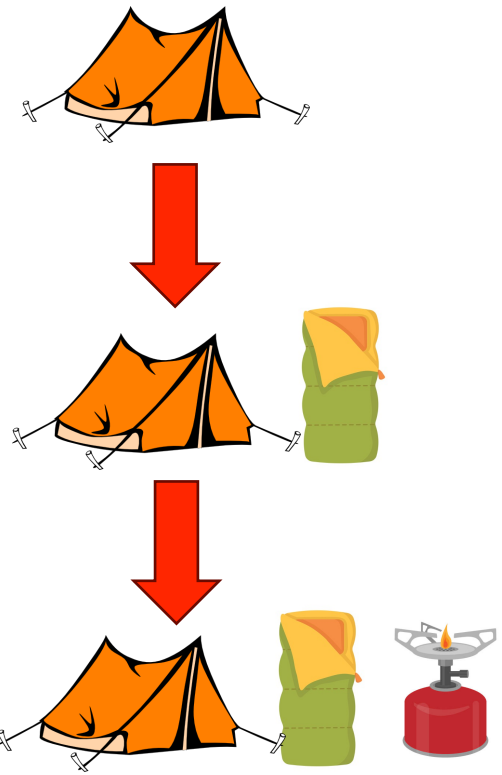
Step 1: Knapsack Subproblems (without repetition)

Input: A weight capacity W, and n items $(w_1, v_1), \cdots, (w_n, v_n)$. All integers.

Output: Most valuable subset of items, whose total weight is ≤W.

First solve the problem for small knapsacks and small sets of items

Then larger knapsacks



And larger item sets

Step 2: Knapsack Recurrence (without repetition)

Input: A weight capacity W, and n items $(w_1, v_1), \cdots, (w_n, v_n)$. All integers.

Output: Most valuable subset of items, whose total weight is $\leq$W.

Step1: Subproblems: For all $c \leq W$ and all $j \leq n$

$K(j, c)$ = best value achievable for knapsack of capacity c using only items $1, \ldots, j$

Discuss

Step 2: Compute $K(j, c)$ using smaller subproblems.

Case 1

Optimal solution using items $1, \ldots, j$ doesn't actually use item j.

$K(j,c) = \max\{ K(j-1, c)$

Case 2

Optimal solution using items $1, \ldots, j$ uses item j.

$K(j-1, c-w_j) + v_j \}$

Hint: keep track of value, leftover capacity, and item set.

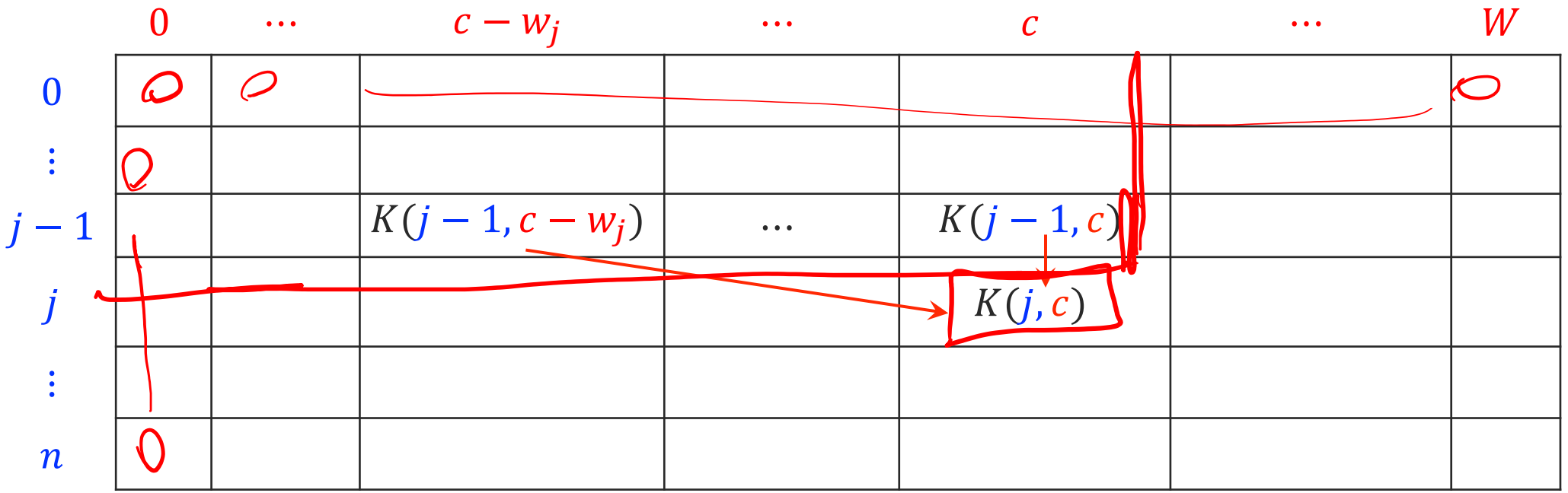
Step 3: Design the Algorithm

Input: A weight capacity W, and n items $(w_1, v_1), \cdots, (w_n, v_n)$. All integers.

Output: Most valuable subset of items, whose total weight is ≤W.

How do we memo-ize the subproblems in this recurrence relation?

$K(j, c) = \max\{ K(j-1, c) , v_j + K(j-1, c-w_j)\}$, base cases: $K(0, c) = 0$ and $K(j, 0) = 0$



Runtime of this algorithm

Input: A weight capacity W, and n items $(w_1, v_1), \cdots, (w_n, v_n)$. All integers.

Output: Most valuable subset of items, whose total weight is ≤W.

$O(nW)$ number of subproblems.

For each subproblem, we take max of 2 values:
→ Work per subproblem $O(1)$

Total runtime: $O(nW)$. → pseudo-poly time alg.

Space complexity: $O(nW)$

Knapsack-no-rep$(W, (w_1, v_1), \dots, (w_n, v_n))$

An array K of size $(n+1)\times(W+1)$.

For $c = 0, \dots, W$: $K[0, c] = 0$

For $j = 0, \dots, n$: $K[j, 0] = 0$

For $j = 1, \dots, n$:

For $c = 1, \dots, W$,

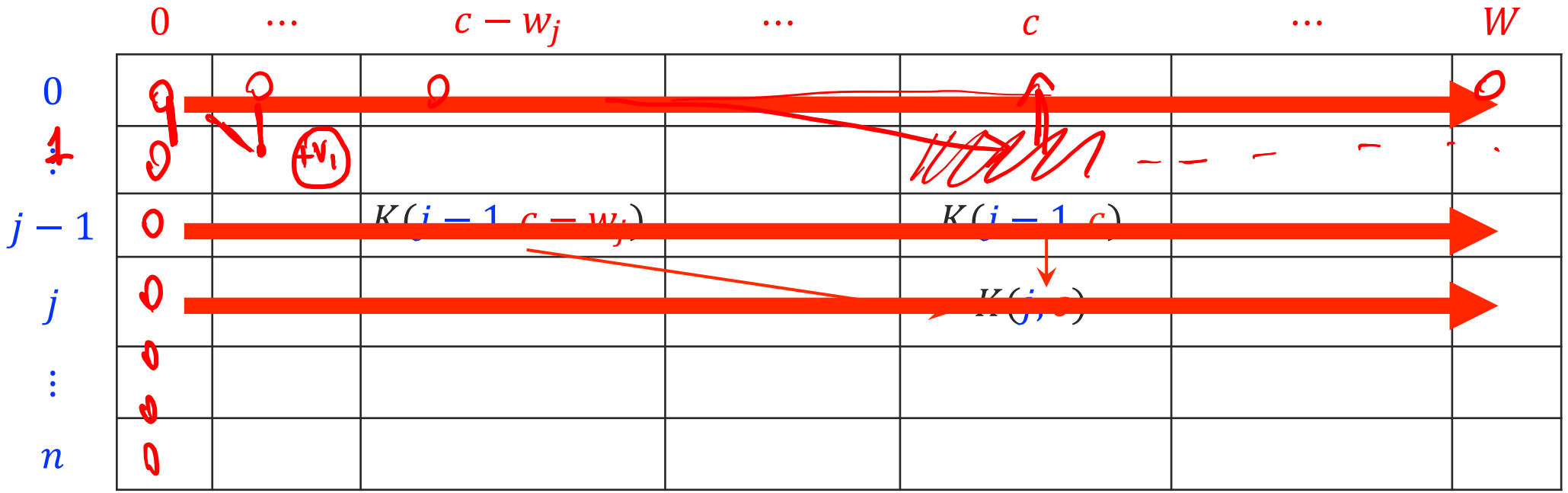
$$K[j, c] = \max\{ K[j-1, c] , v_j + K[j-1, c-w_j] \}$$

$j: w_j < c$

return $K[n, W]$

Runtime of this algorithm

Fill in the table one row at a time and keep only the last row.



Total runtime: $O(nW)$.

Space complexity: ~~$O(nW)$~~ $O(W)$

For $c = 1, \ldots, W$,

$$K[j, c] = \max\{ K[j-1, c], v_j + K[j-1, c-w_j] \}$$

return $K[n, W]$

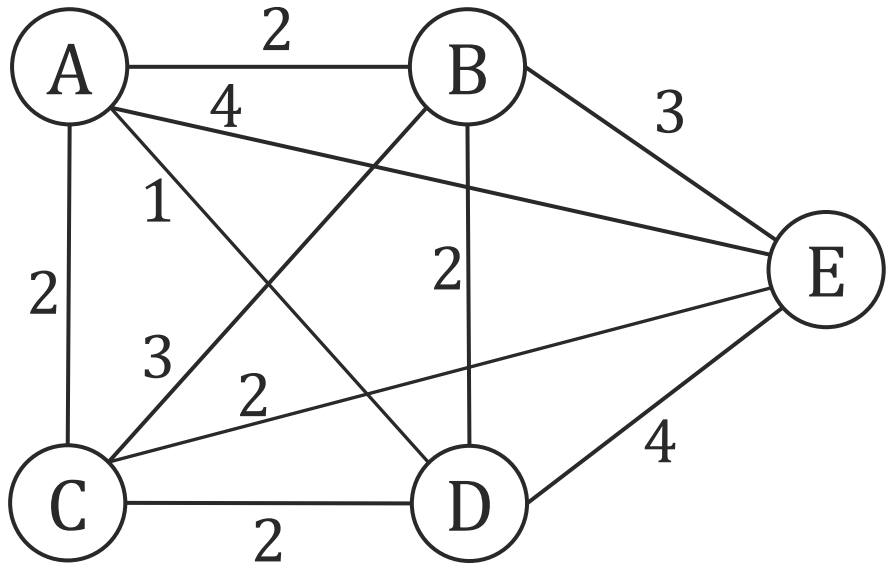
Traveling Salesperson Problem

Traveling Salesperson Problem (TSP)

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

Output: A "tour" of minimum total distance.

Definition: A **tour** is a path through the cities, that
1) Starts from city 1
2) Visits every city, exactly once
3) Returns to city 1



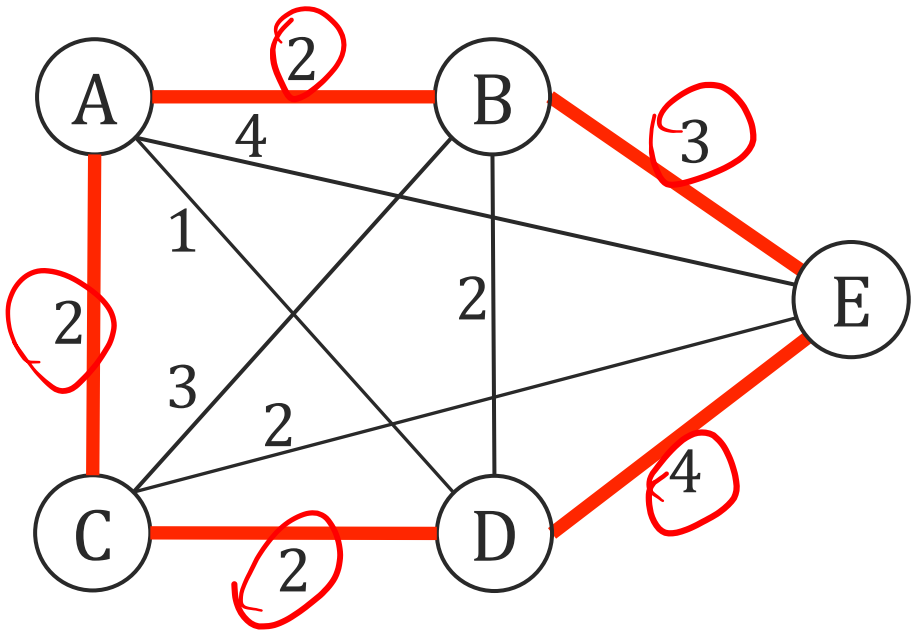
Traveling Salesperson Problem (TSP)

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

Output: A "tour" of minimum total distance.

Definition: A **tour** is a path through the cities, that
1) Starts from city 1
2) Visits every city, exactly once
3) Returns to city 1

Tour of distance: 13



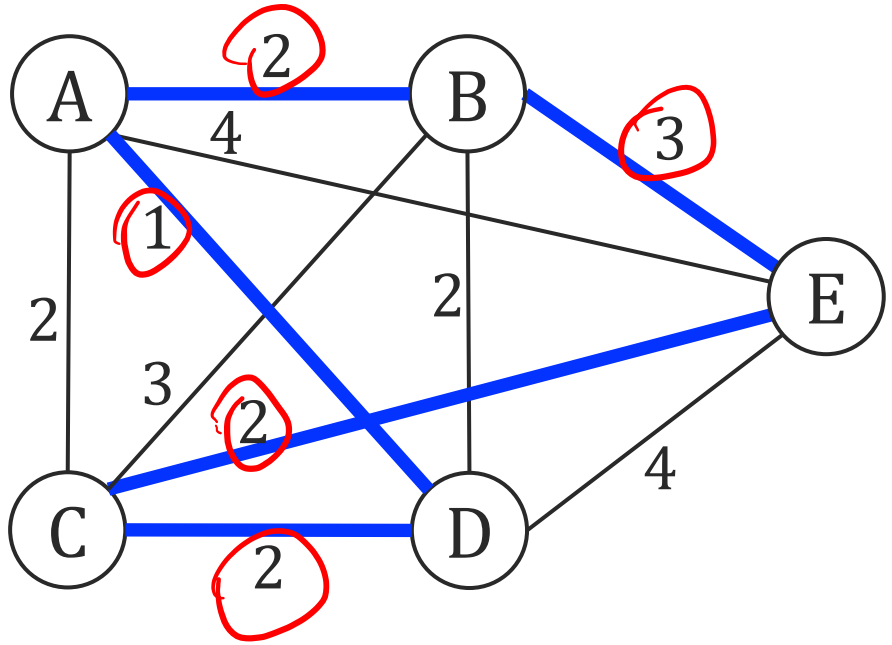
Traveling Salesperson Problem (TSP)

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

Output: A "tour" of minimum total distance.

Definition: A **tour** is a path through the cities, that
1) Starts from city 1
2) Visits every city, exactly once
3) Returns to city 1

Tour of distance: 10



Traveling Salesperson Problem (TSP)

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

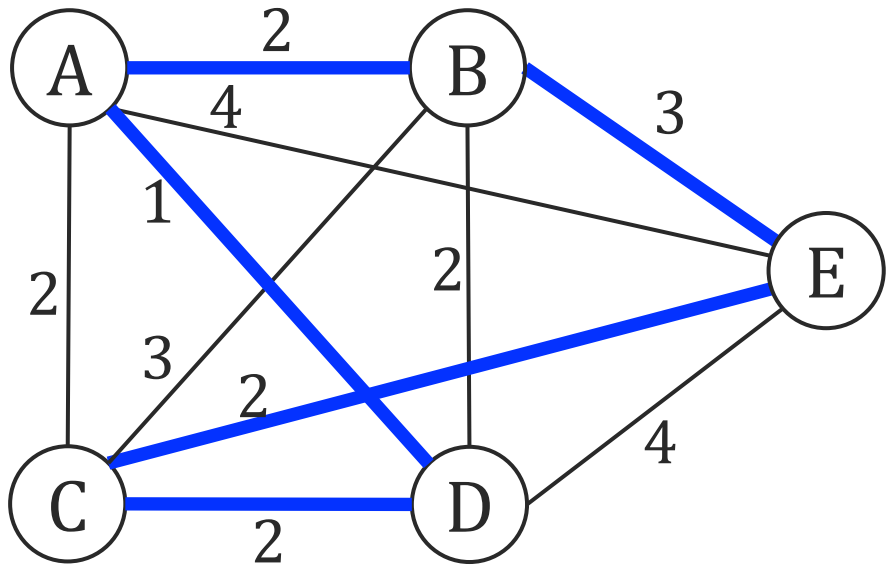
Output: A "tour" of minimum total distance.

Definition: A **tour** is a path through the cities, that
1) Starts from city 1
2) Visits every city, exactly once
3) Returns to city 1

Naïve brute force algorithm:
→ $(n-1)!$ Tours
→ Each $O(n)$ to compute distance.
→ $O(n!)$ runtime

Dynamic programming gives us $O(n^2 2^n)$

Tour of distance: 10



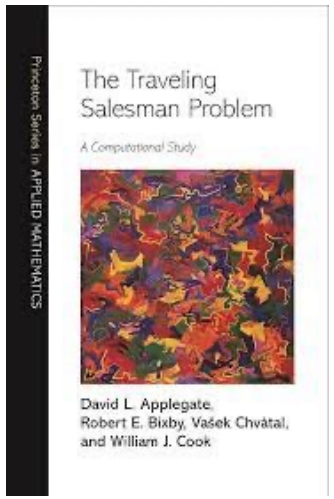
One of the most famous Math/CS problems.

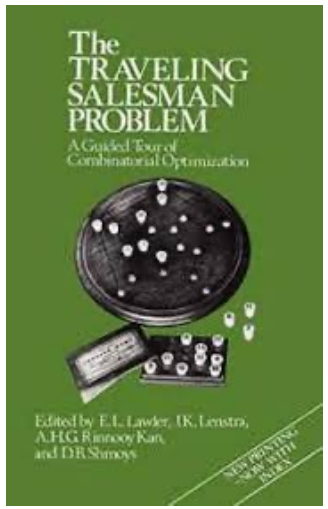
Notoriously difficult.

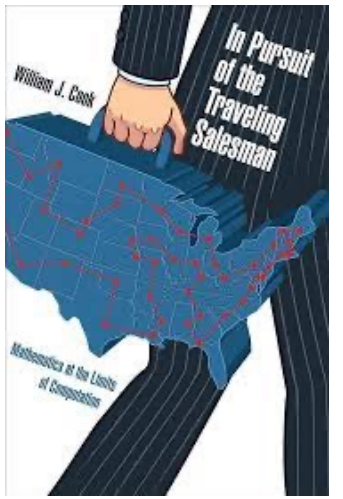
The DP algorithm is a substantial improvement over brute force. Take $n = 25$

→$O(n!) \approx 10^{25}$

→$O(n^2 2^n) \approx 10^{10}$





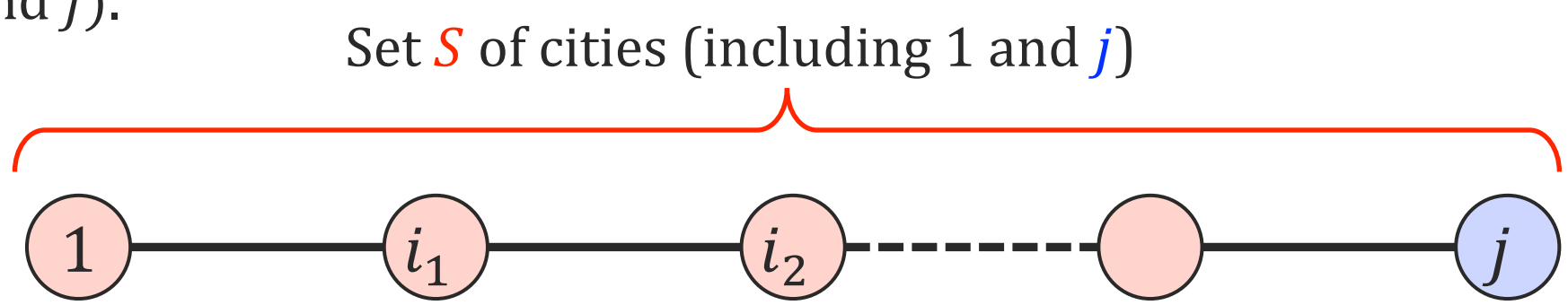


Step 1: Subproblems of TSP

Input: cities 1 … 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

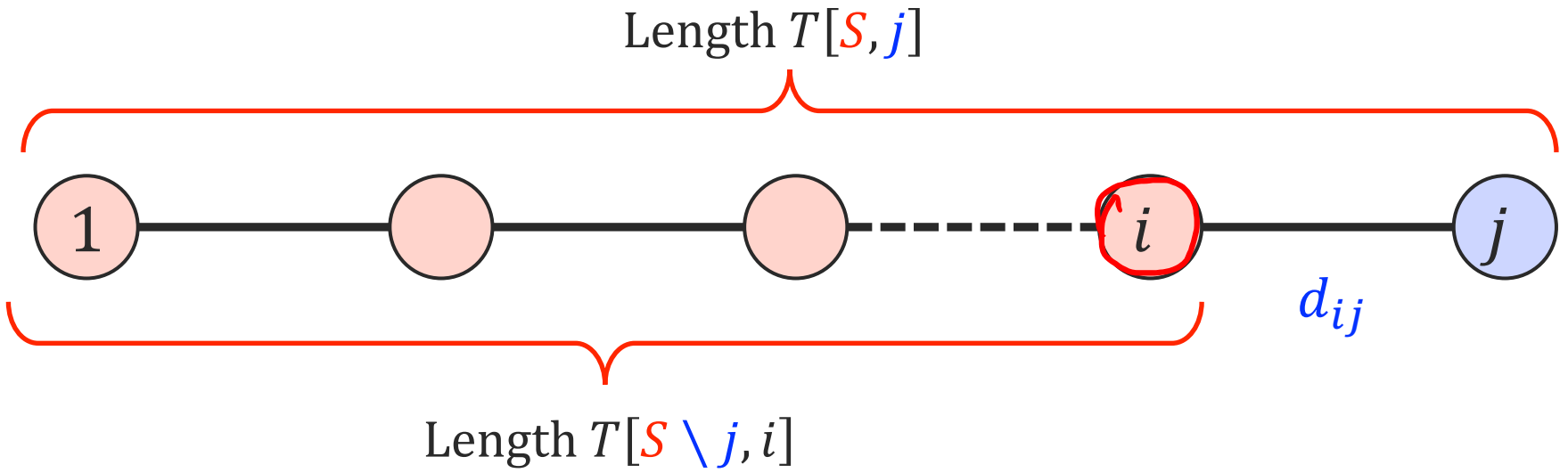
Input: cities $1 \ldots n$ and pairwise distances d_{ij} between cities i and j.

Output: A "tour" of minimum total distance.

Subproblems: For all $j \leq n$ and $S \subseteq \{1, \ldots, n\}$, 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.



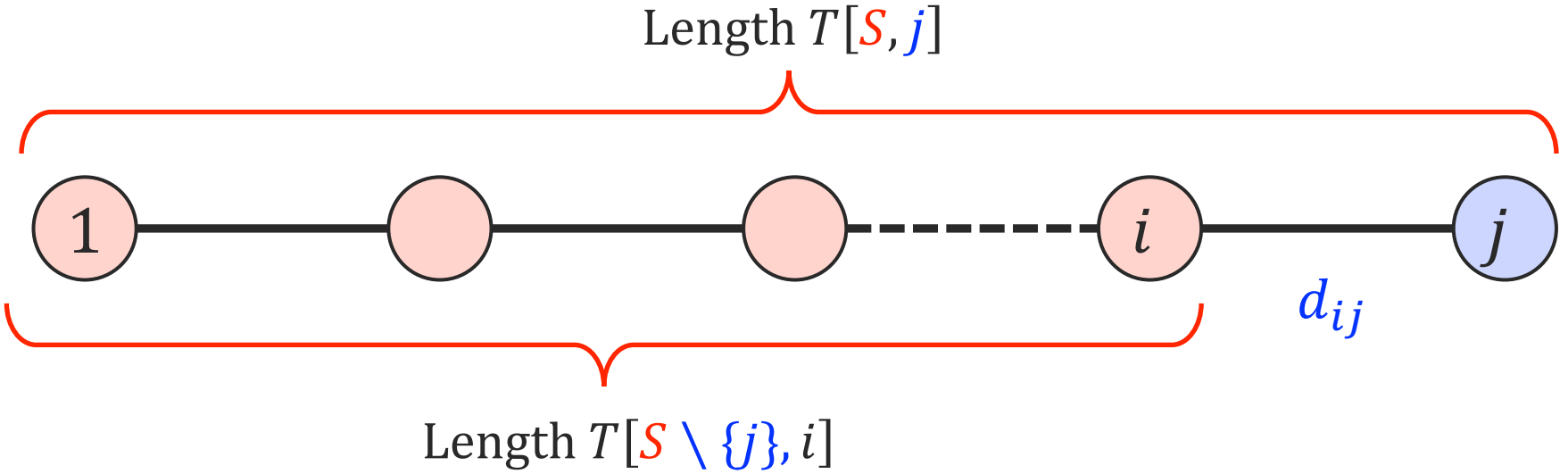
Step 2: Recurrence Relation for TSP

Input: cities 1 … 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 $i \neq j$.



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

$T(S,j)$ = len shortest path, visits every node in S exactly once and starts at 1 ends at j

Step 2: Base Cases and the Final Solution

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

Output: A "tour" of minimum total distance.

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

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

invalid subproblem

No partial path allowed to start and ends at 1.

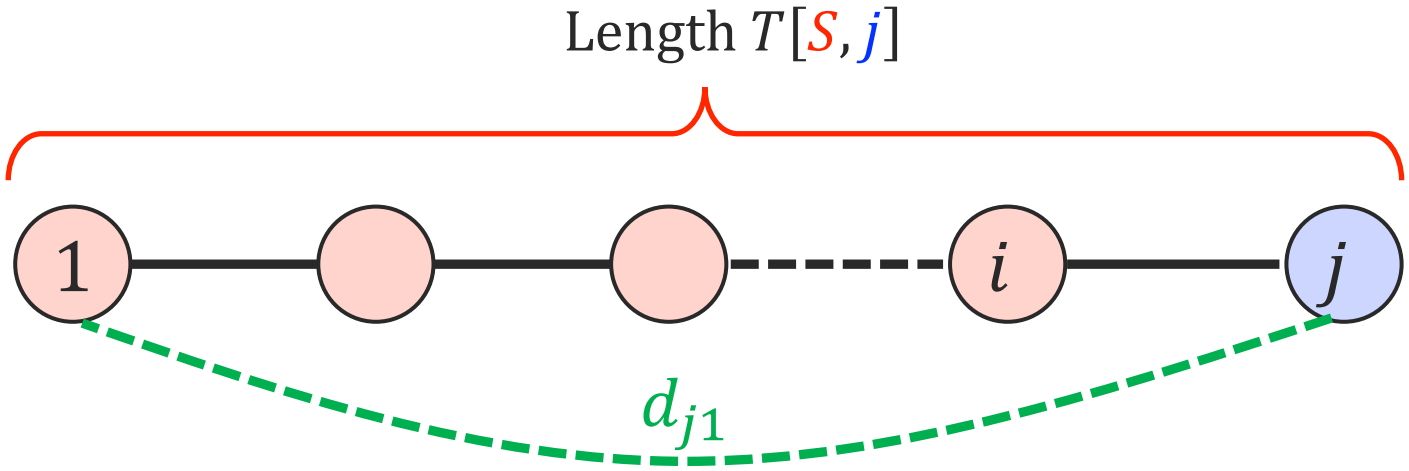
Final solution:

→ 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

$T(S, j)$ = len

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

Output: A "tour" of minimum total distance.

Start at 1, pass through every city exactly once, return to 1

$O(2^n \times n)$ 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

will be helpful to keep track of the actual path.

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

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

$T(S, 1) = \infty$

For $j \in S$

Additional bookkeeping here

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

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

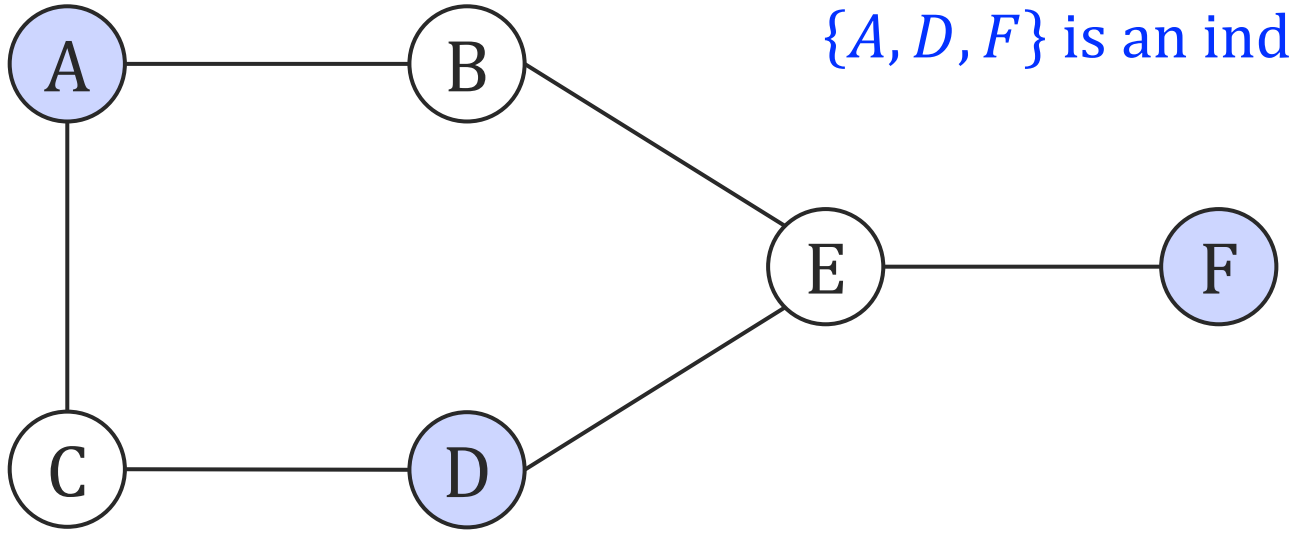
Next up: Independent Sets in Trees
(3 min break and
please close the auditorium doors)

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 $u, v \in S$.



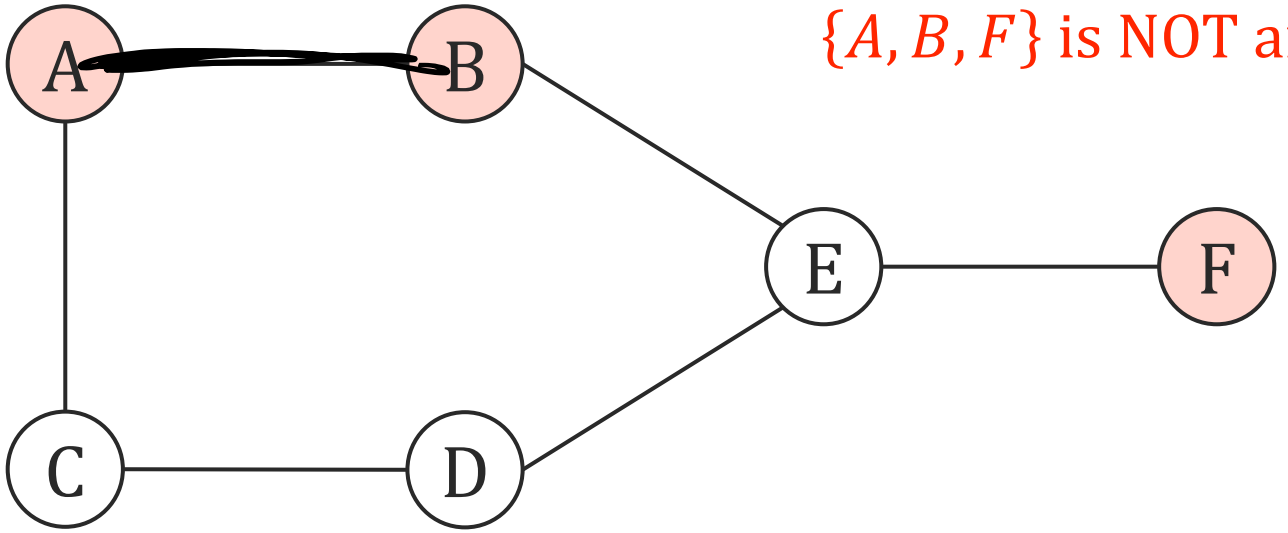
$\{A, D, F\}$ is an independent set.

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

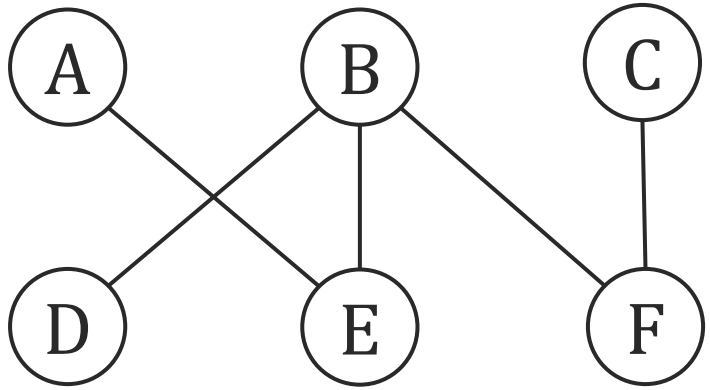
Input: Undirected Graph $G = (V, E)$ and G is a tree.

Output: Largest "independent set" of G.

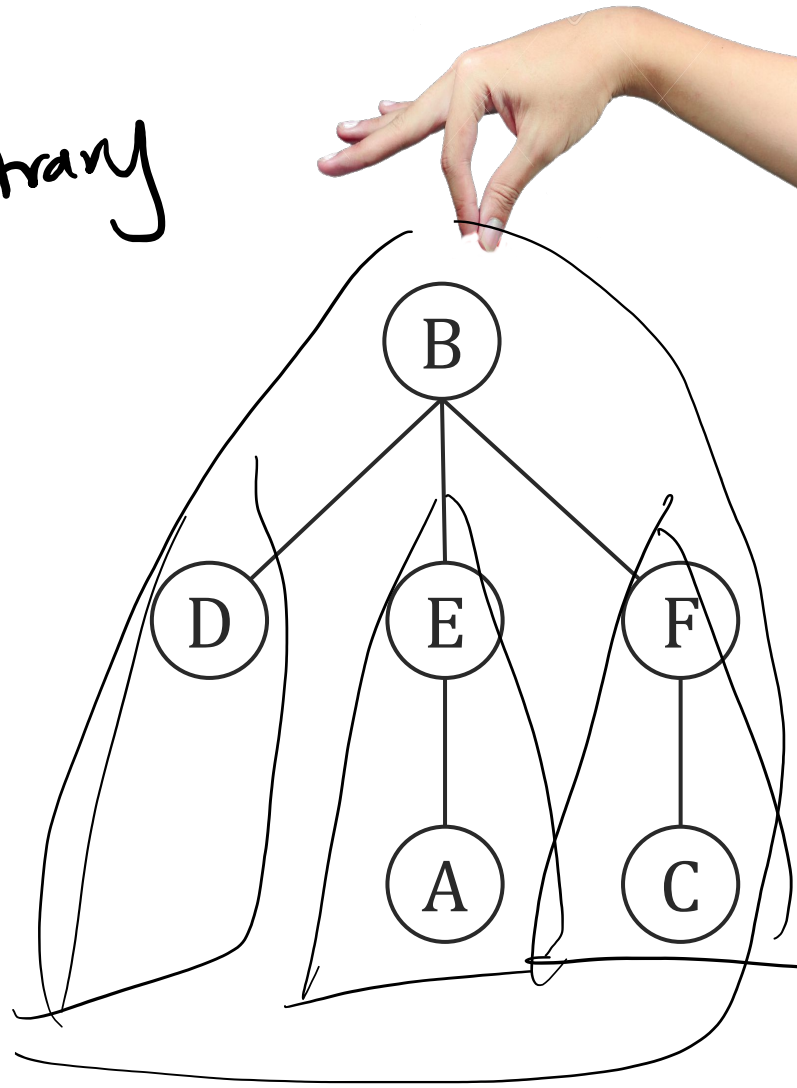
Recall, trees don't have cycles!

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

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



Can be arbitrary



Step 1: Subproblems for Independent Sets

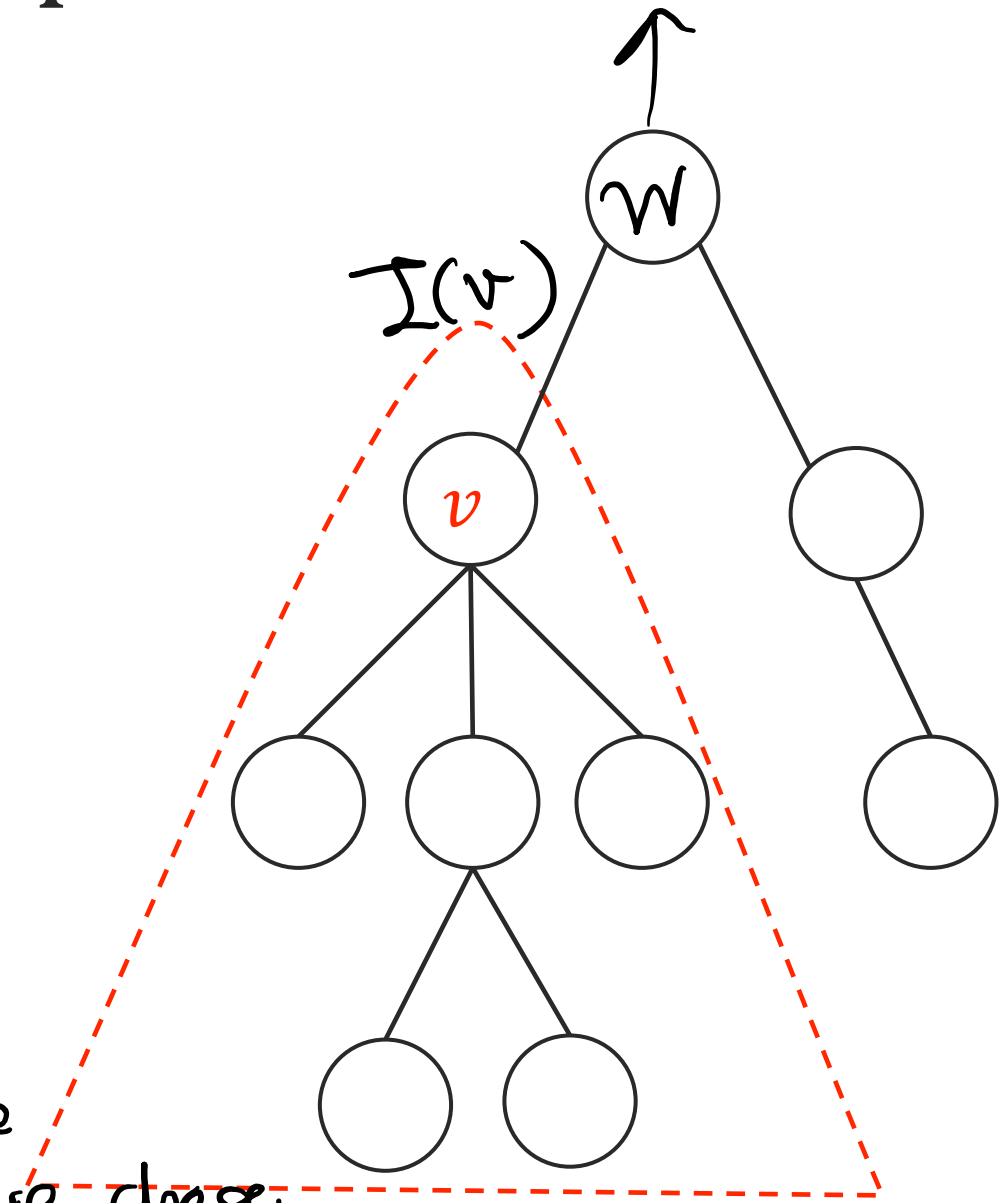
Input: Undirected Graph $G = (V, E)$ and G is a tree.

Output: Largest "independent set" of G.

Subproblems: For each $v \in V$

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

this and what's the subtree rooted at v of course depends on the choice of root w. But the final outcome doesn't care what w we chose.



Step 2: Recurrence for Independent Sets

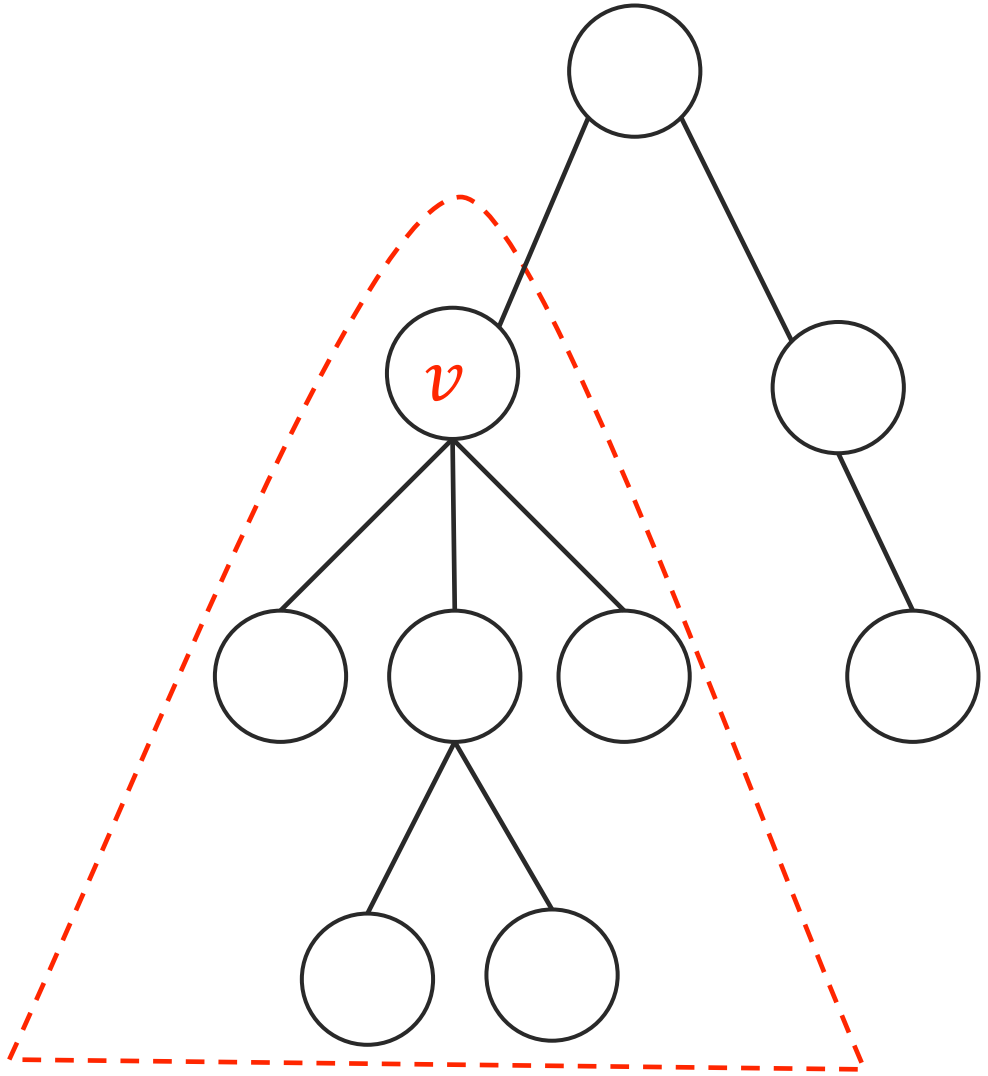
Input: Undirected Graph $G = (V, E)$ and G is a tree.

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)

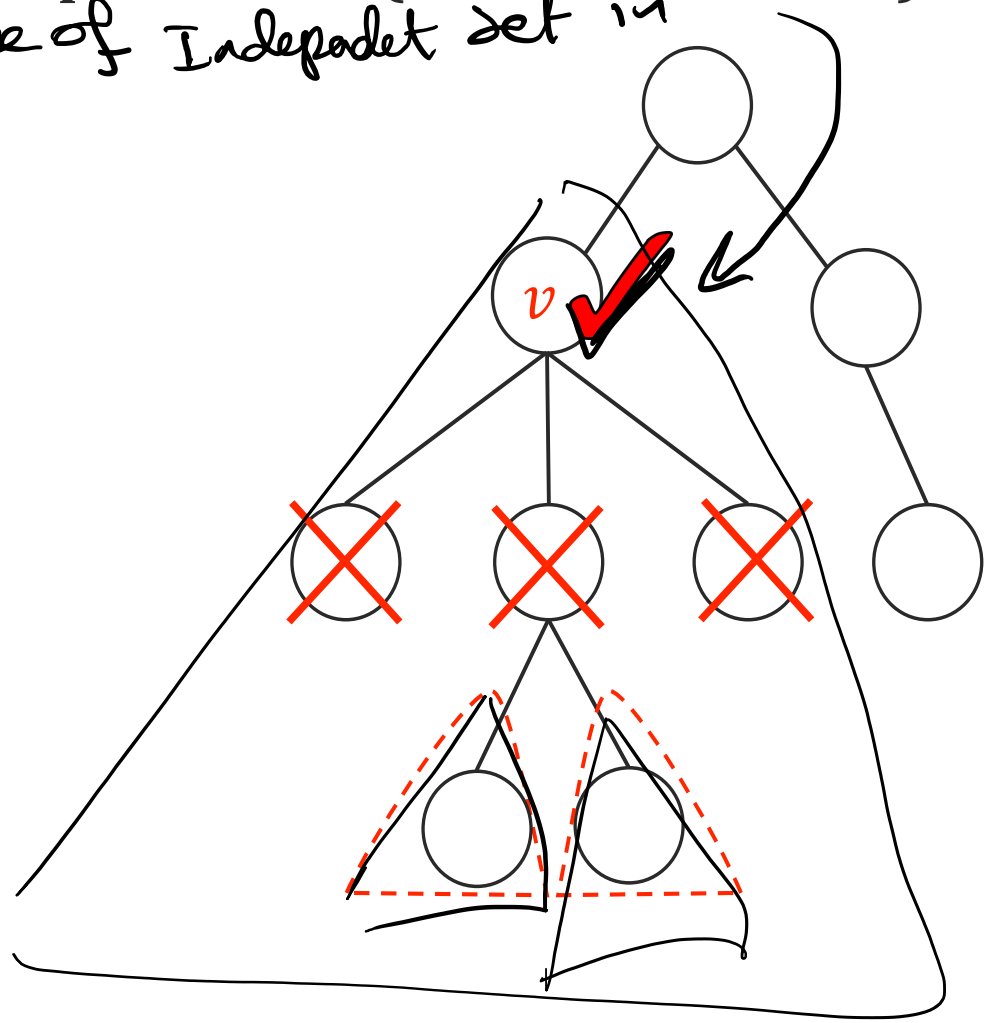


Two Cases:

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

→ optimal size of Independet set in

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:

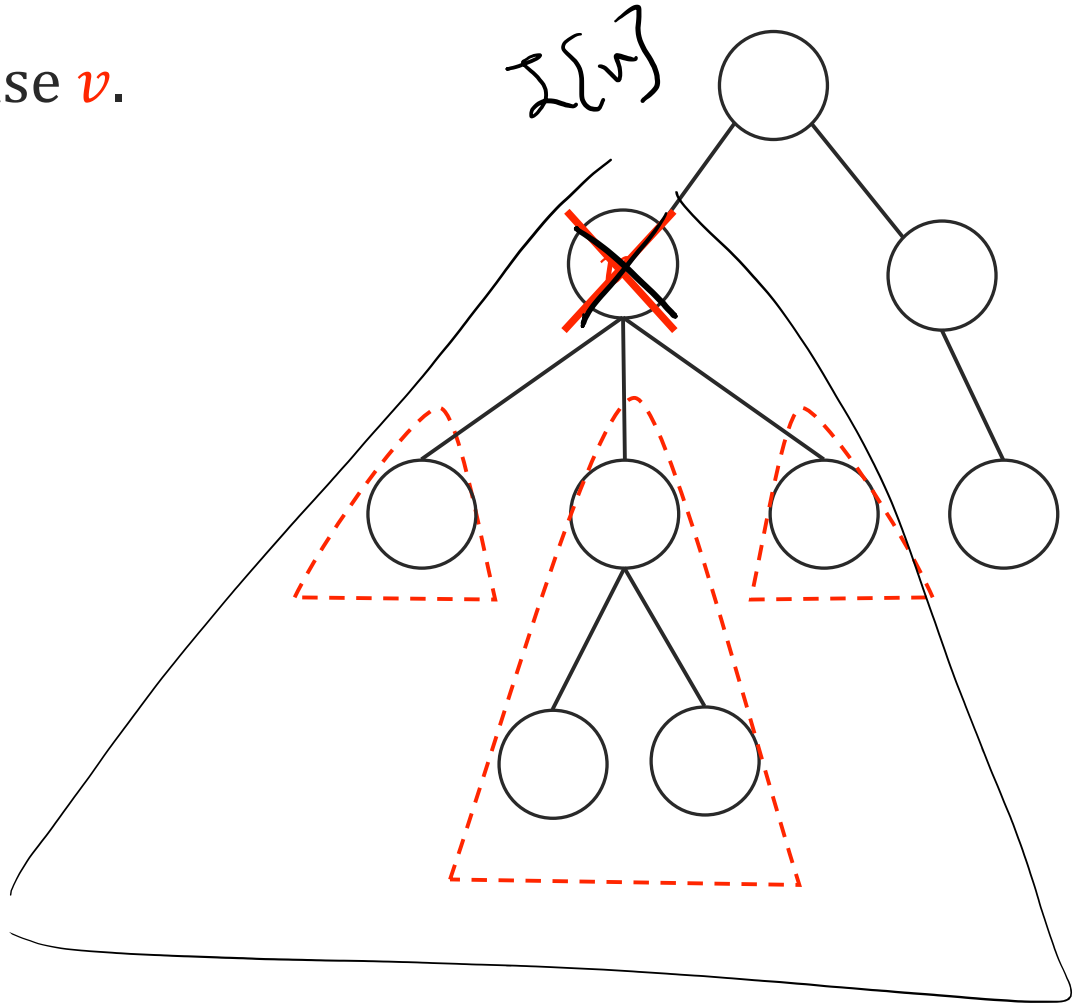
b/c I added v

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

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.

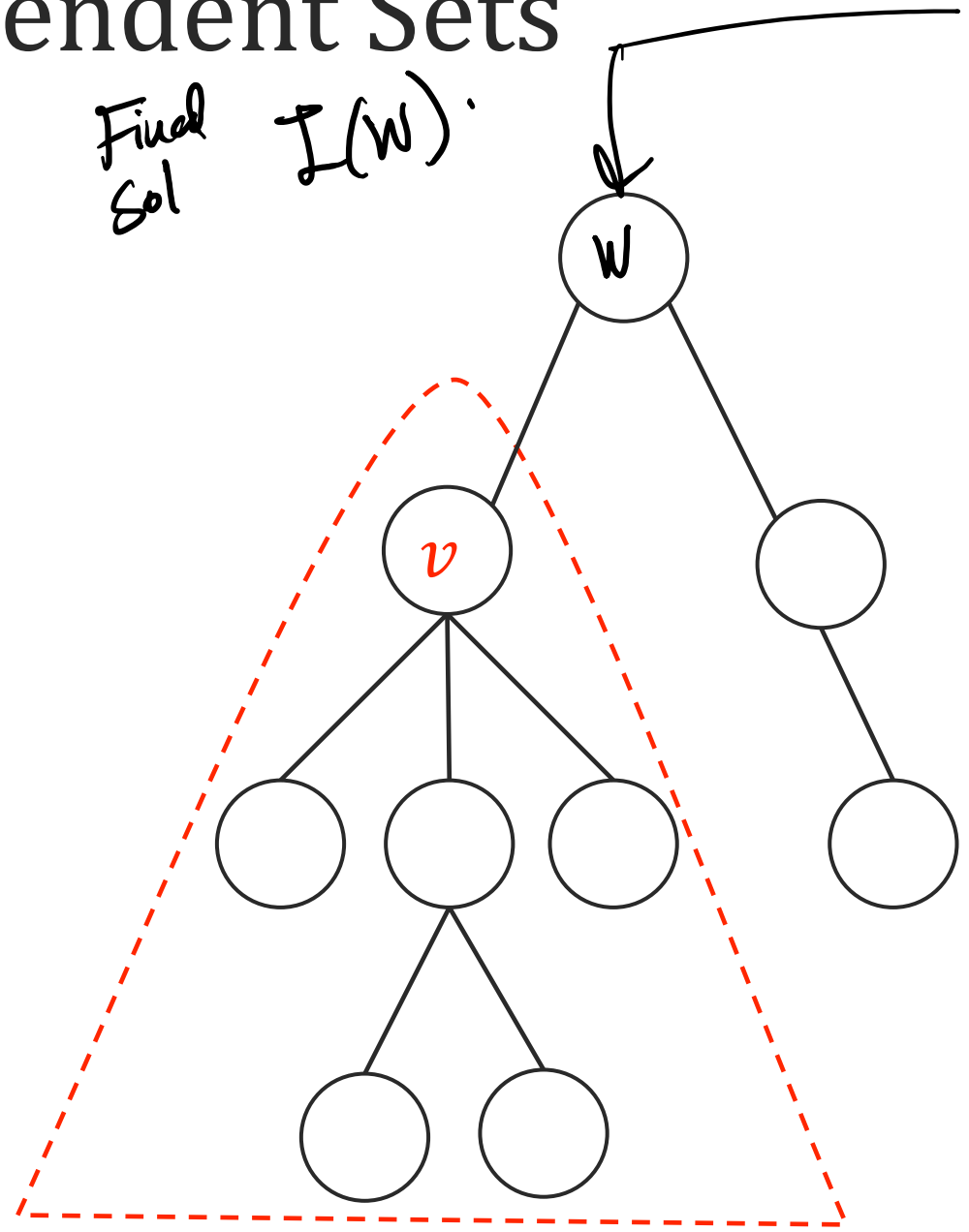
Recurse to the children levels:

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

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.

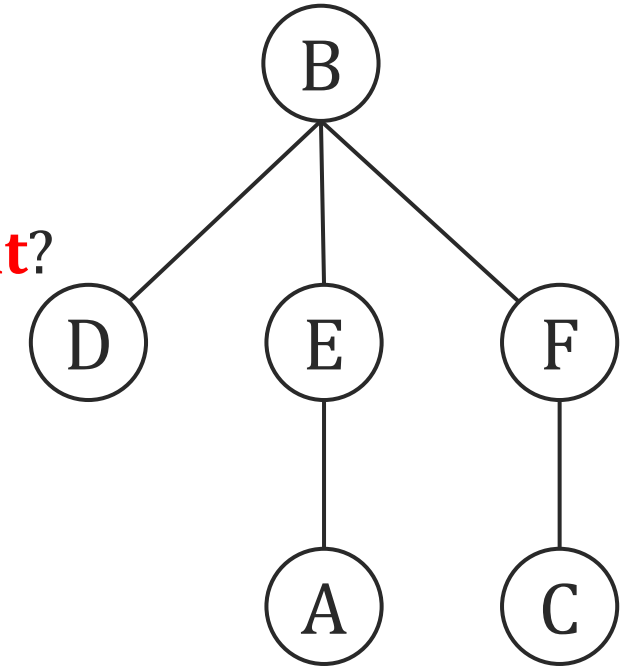
Recurrence: Compute $I[v]$ using smaller subproblems (its descendants)

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

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 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 $i = 1, \dots, n$

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

return $I[v_n]$ root according to DFS

Wrap up

We did lots of dynamic programming!

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

Next time: A different paradigm of algorithm design

→ Linear Programming