# CS 170
# Efficient Algorithms and Intractable Problems

## Lecture 13
### Dynamic Programming III

Nika Haghtalab and John Wright

EECS, UC Berkeley

# Announcements

Interested in meeting 1-1 with TAs?
→ Fill out a form on Ed
→ General advice for course, midterm performance, and etc.

# 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

# 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).

# Knapsack

# Knapsack

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.

Two variants:

1. With repetition (aka unbounded supply, aka with replacement)

→ For each item $i$, we can take as many copies of it as we want

2. Without repetition (0-1 knapsack, aka without replacement)

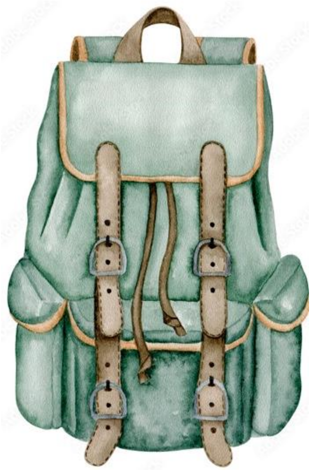
→ For each item, either we take 1 copy or 0 copy of it.

# Knapsack

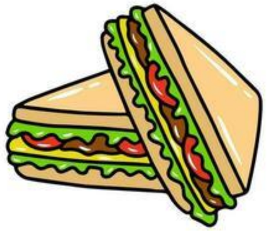
All integers!

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 |

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

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

# Step 1: Subproblems of Knapsack (with repetition)

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

Output: Most valuable combination of items (with repetition), whose total weight is ≤W.

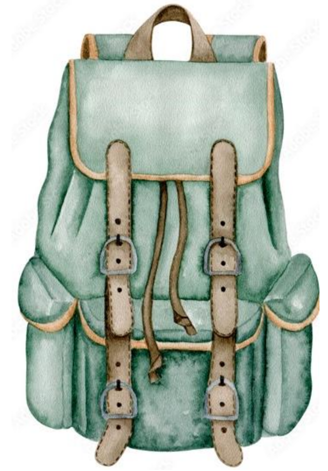
What makes for good subproblems?

- 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).

**Subproblems:** For all $c \leq W$, $K(c) =$ best value achievable for knapsack of capacity $c$.

First solve the problem for small knapsacks

Then larger knapsacks



# Step 2: Recurrence in Knapsack (with repetition)

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

Output: Most valuable combination of items (with repetition), whose total weight is ≤W.

**Step 1:** Subproblems $K(c)$ = best value achievable for knapsack of capacity $c$, for c $\leq W$.

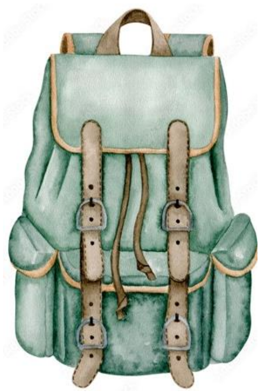
**Step 2:**

Let's say we commit to putting a copy of item $i$ for which $w_i \leq c$ in the knapsack

→ Then only $c - w_i$ capacity remains to be optimally packed.

→ The recurrence relationship

$$K(c) = \max_{i: w_i \leq c} \{v_i + K(c - w_i)\}$$



# Step 3: Design the Algorithm

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

<u>Output</u>: Most valuable combination of items (<u>with repetition</u>), whose total weight is ≤W.

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

$$K(c) = \max_{i: w_i \le c} \{v_i + K(c - w_i)\}$$

Runtime of this algorithm?

Number of subproblems: $O(W)$

Per subproblem, max over $O(n)$ cases
→ $O(n)$ time per subproblem.

Total runtime: $O(nW)$

Knapsack-with-repetition$(W, (w_1, v_1), \ldots, (w_n, v_n))$

An array $K$ of size $W + 1$.

$K[0] = 0$

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

$K[c] = \max_{i: w_i \le c} \{v_i + K(c - w_i)\}$

**return** $K[W]$

# Polynomial vs Pseudo-Polynomial Time

We quantify runtimes as functions of input size.

→ **Input size**: # bits needed to write the input

What is the input size the of Knapsack

- Weight capacity W → Needs $O(\log(W))$ bits
- $n$ items with weights at most $W$ (remove any larger item) → most $O(\log(W))$ bits
- Total input size of knapsack: $O(n\log(W))$

Does the dynamic programming for knapsack run efficiently?

→ Not polynomial time exactly! **Runtime $\boldsymbol{O(nW)}$** but **input size $\boldsymbol{O(n\,log(W))}$**

→ Called a pseudo-polynomial time algorithm

→ A runtime that's polynomial in the numerical value of the input (like W) but not in the size of the input (like $O(n\,log(W))$).

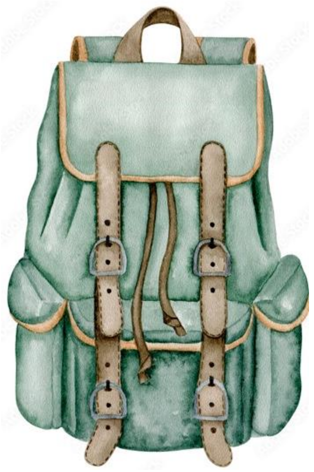
# Knapsack without Repitions

# Knapsack Recap

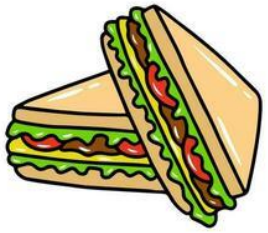
All integers!

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 Variant**

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

**This Variant**

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!

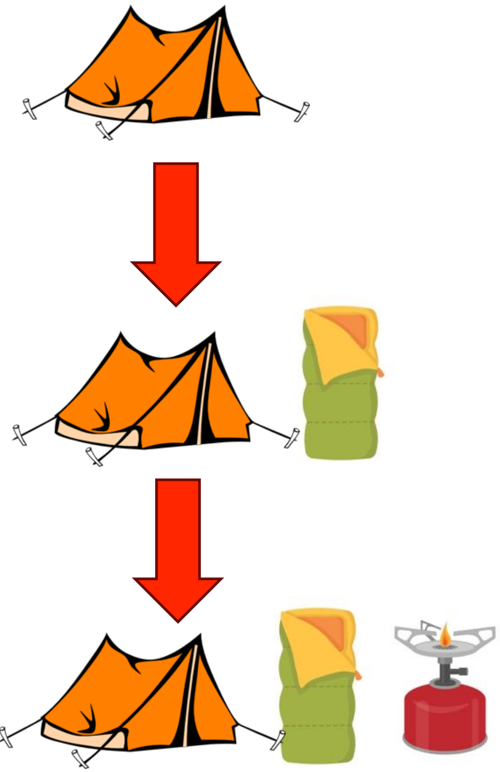
# Step 1: Knapsack Subproblems (without repetition)

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 ≤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 $j$.

**Case 2**

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

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?

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

| | $0$ | $\cdots$ | $c - w_j$ | $\cdots$ | $c$ | $\cdots$ | $W$ |
|---|---|---|---|---|---|---|---|
| $0$ | | | | | | | |
| $\vdots$ | | | | | | | |
| $j-1$ | | | $K(j-1, c-w_j)$ | $\cdots$ | $K(j-1, c)$ | | |
| $j$ | | | | | $K(j, c)$ | | |
| $\vdots$ | | | | | | | |
| $n$ | | | | | | | |

# 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: $O(nW)$.

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

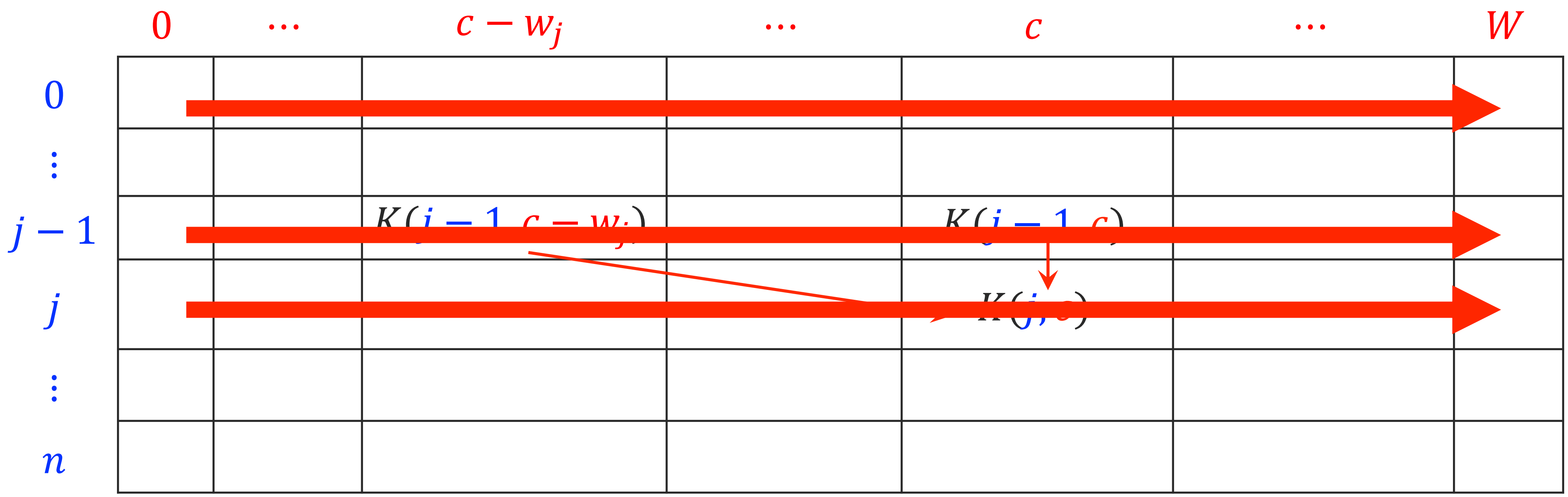
**For** $c = 1, \dots, W$,

$$K[j, c] = \max_{j : w_j < c} \{ K(j-1, c), v_j + K(j-1, c-w_j) \}$$

**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

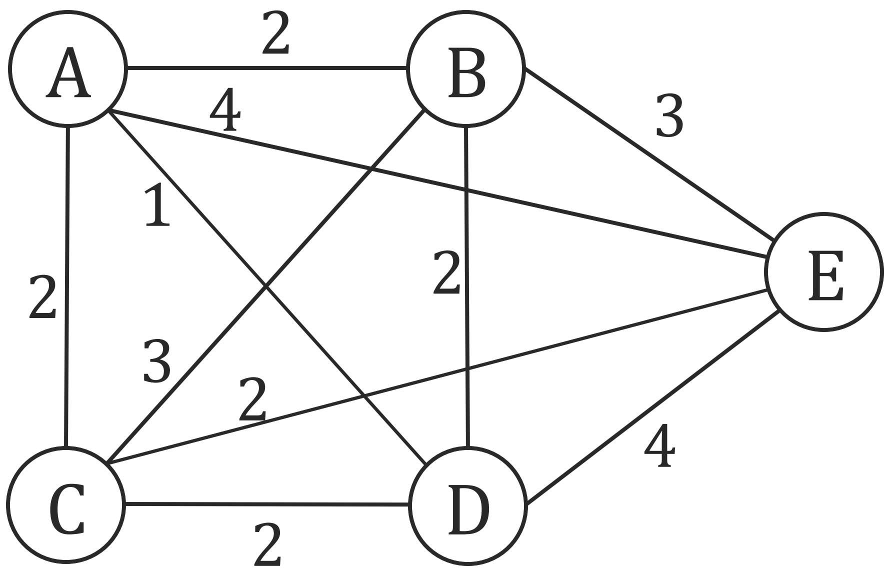
# Traveling Salesperson Problem (TSP)

Input: cities $1 \dots n$ 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

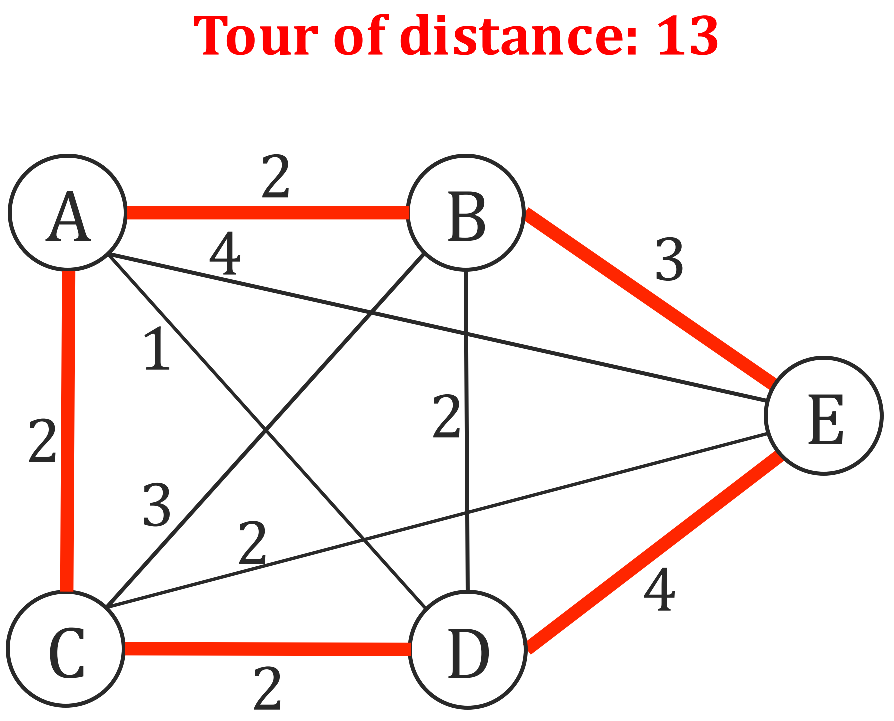


# Traveling Salesperson Problem (TSP)

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



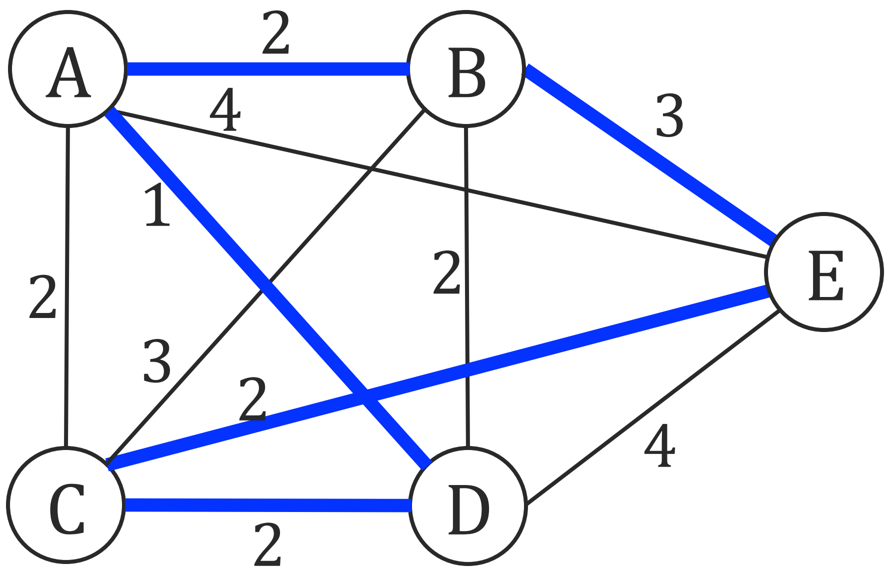
# Traveling Salesperson Problem (TSP)

<u>Input</u>: cities 1 … $n$ and pairwise distances $d_{ij}$ between cities $i$ and $j$.

<u>Output</u>: 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)

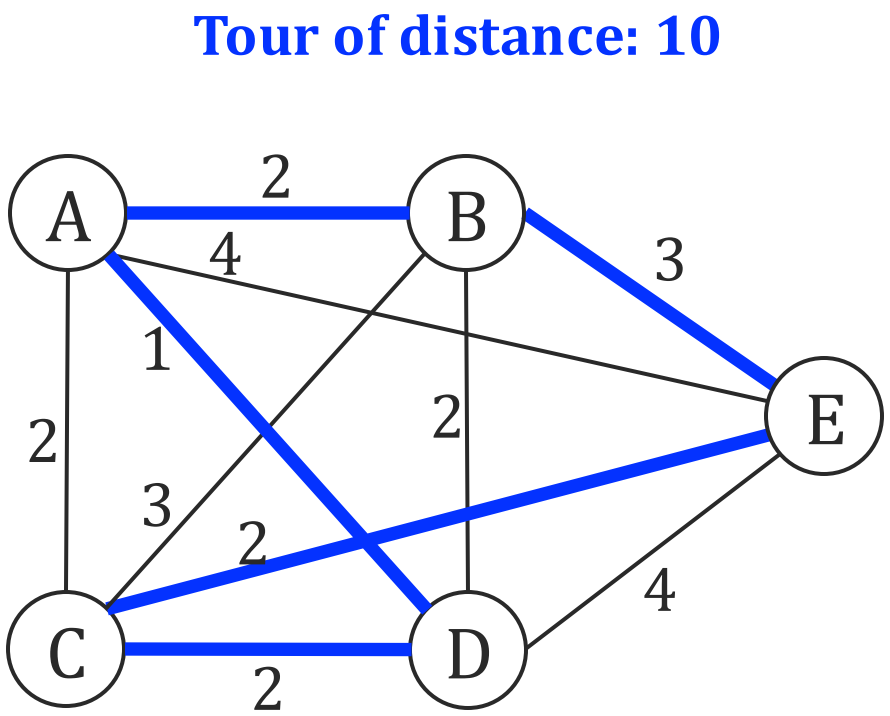
Input: cities 1 … $n$ and pairwise distances $d_{ij}$ between cities $i$ and $j$.

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 $O(n^2 2^n)$



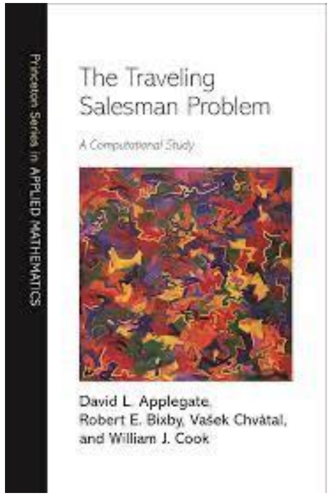

One of the most famous Math/CS problems.

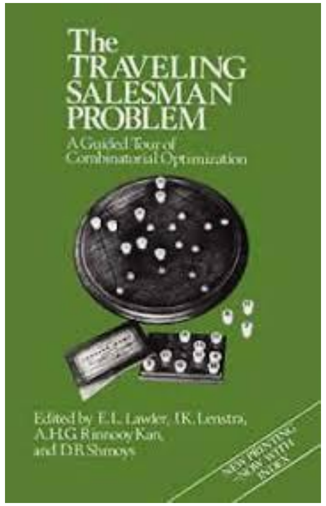
Notoriously difficult.

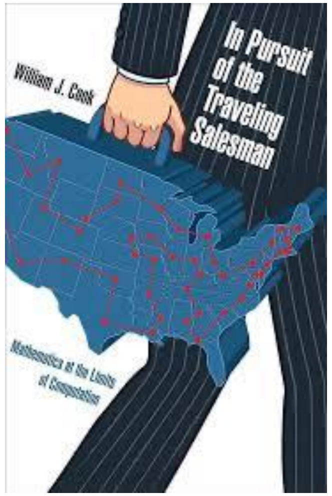
The DP algorithm is a substantial improvement over brute force. Take $n = 25$

→ $O(n!) \approx 10^{25}$

→ $O(n^2 2^n) \approx 10^{10}$



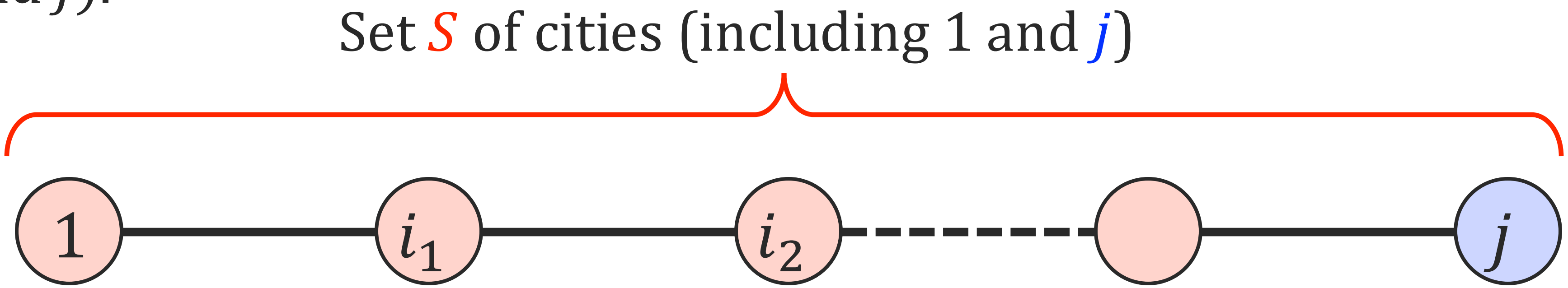



# Step 1: Subproblems of TSP

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 $S \subseteq \{1, \ldots, n\}$, 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: Recurrence Relation for TSP

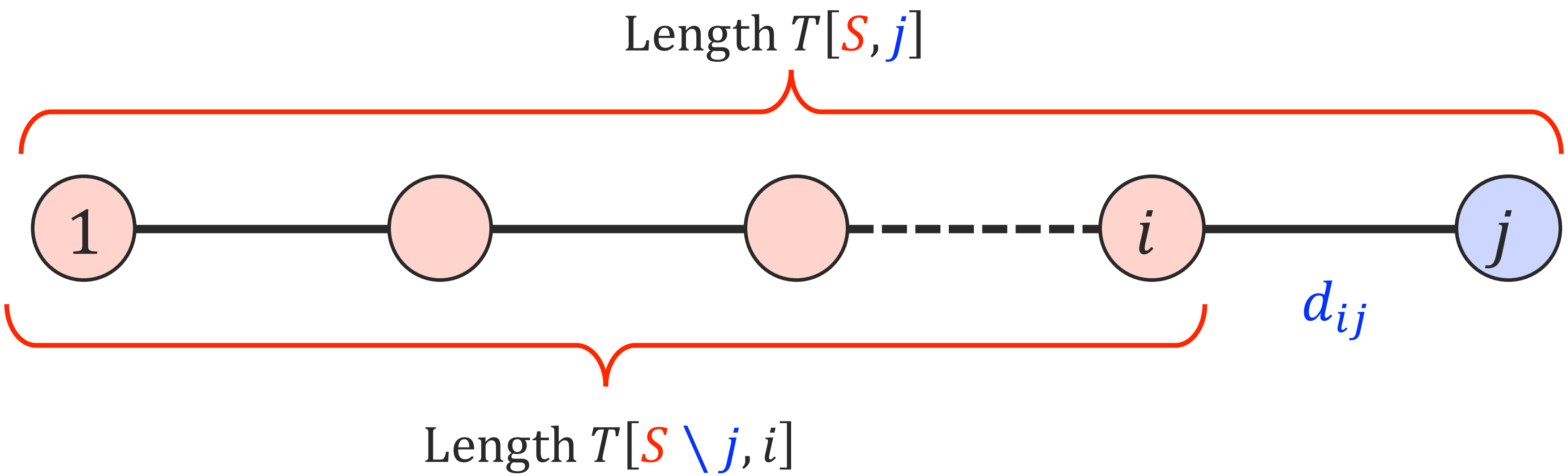
Input: cities $1 \ldots n$ and pairwise distances $d_{ij}$ between cities $i$ and $j$.

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.

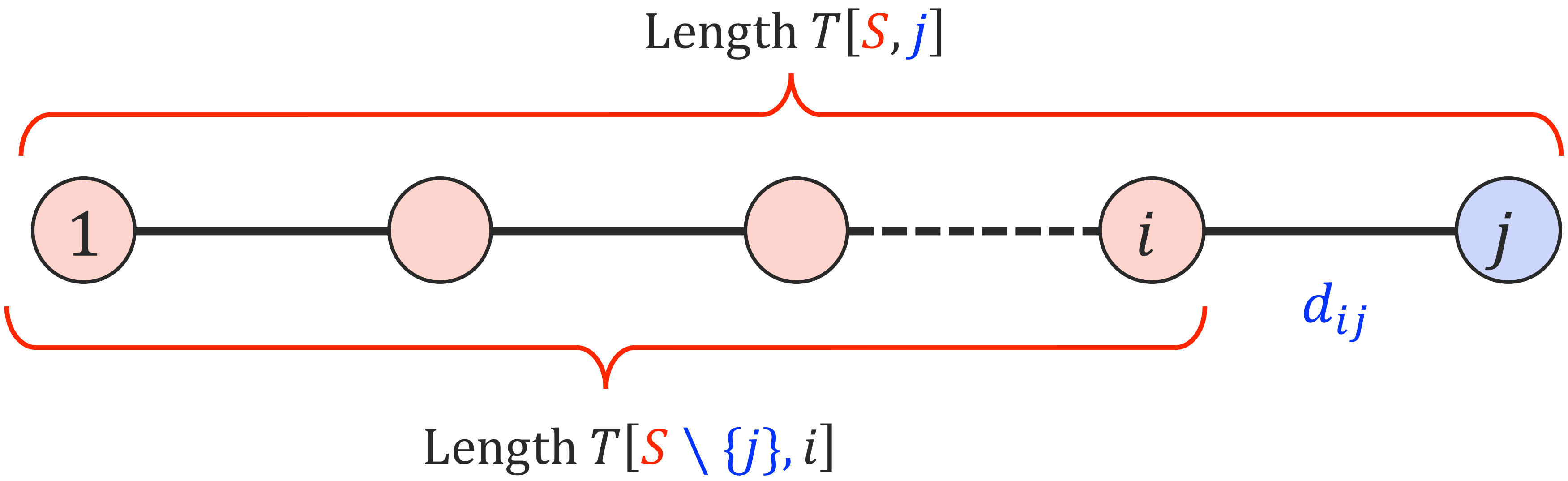


# Step 2: Recurrence Relation for TSP

Input: cities $1 \ldots n$ 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 $i \neq j$.




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

# Step 2: Base Cases and the Final Solution

Input: cities $1 \ldots n$ 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 $\geq 2$, $T[S, 1] = \infty$.

No partial path allowed to start and ends at 1.

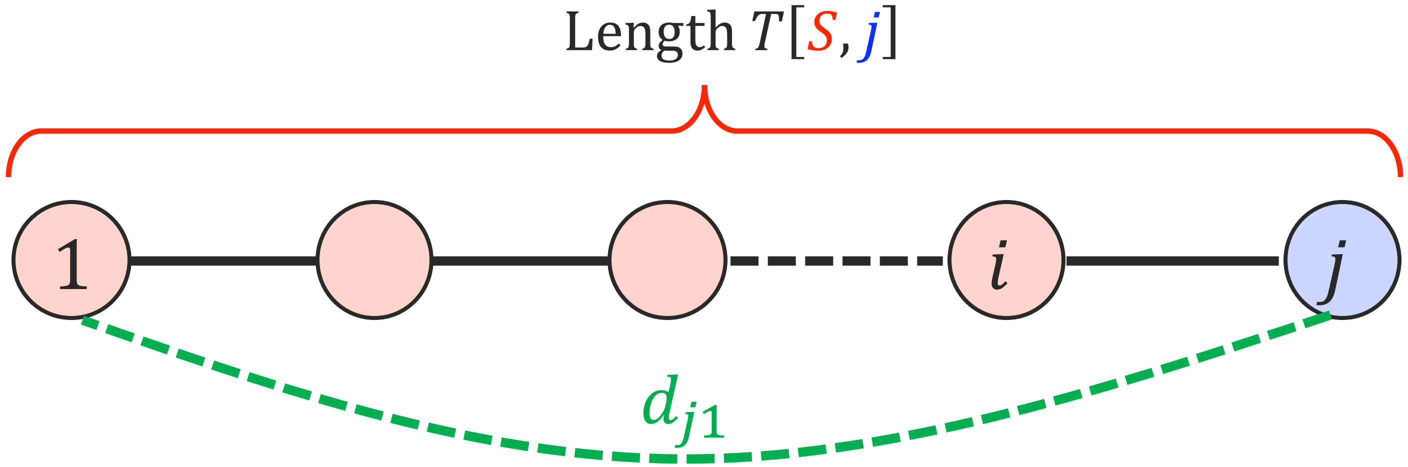
Final solution:

→ Add the final $(j, 1)$ edge cost:

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

→ Find the best $j$:

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



# 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.

$O(2^n \times n)$ 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** $j \in S$

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

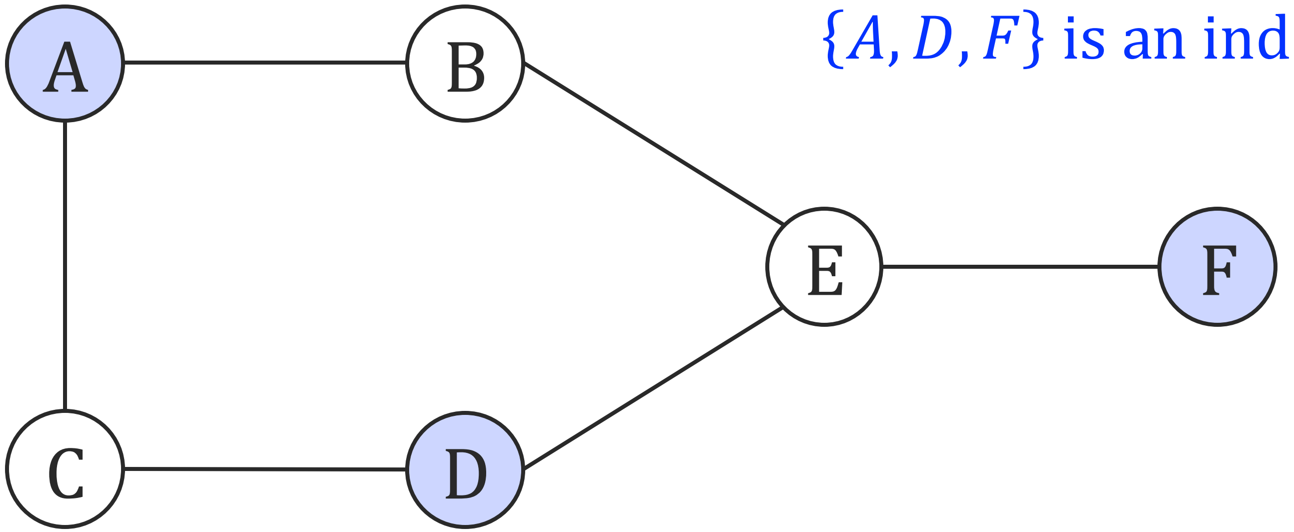
**return** $\min_{j \neq 1} T[\{1, \dots, n\}, j] + d_{j1}$

# 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 $u, v \in S$.



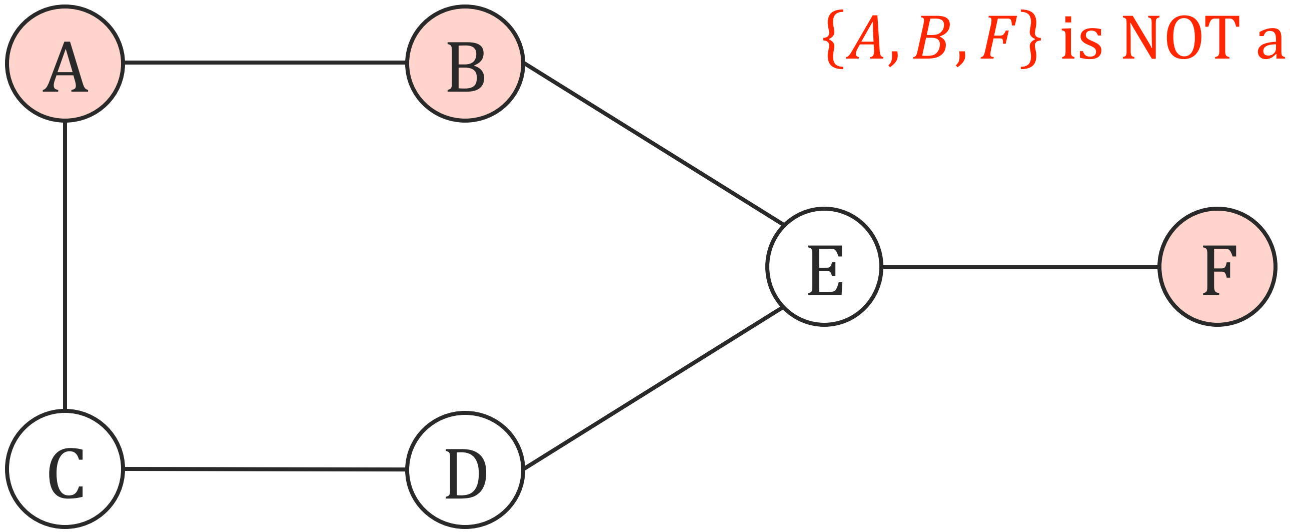

$\{A, D, F\}$ is an independent set.

# 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

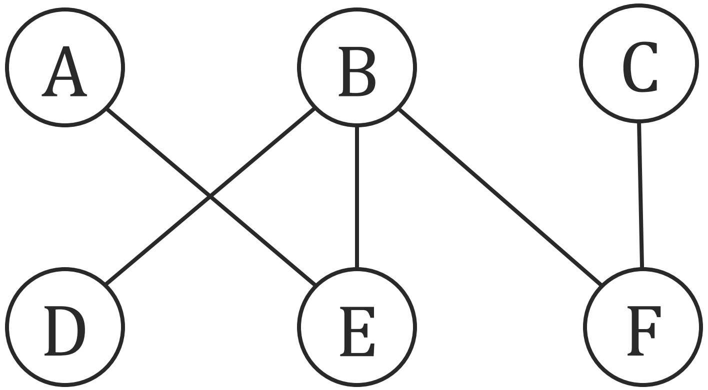
Input: Undirected Graph $G = (V, E)$ and G is a tree.

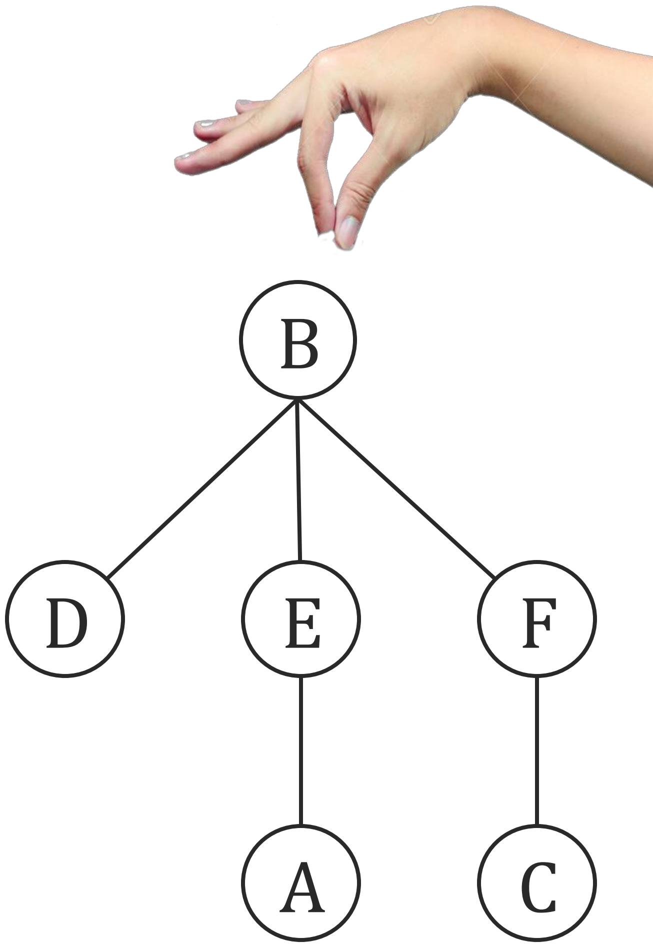
Output: Largest "independent set" of $G$.

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.



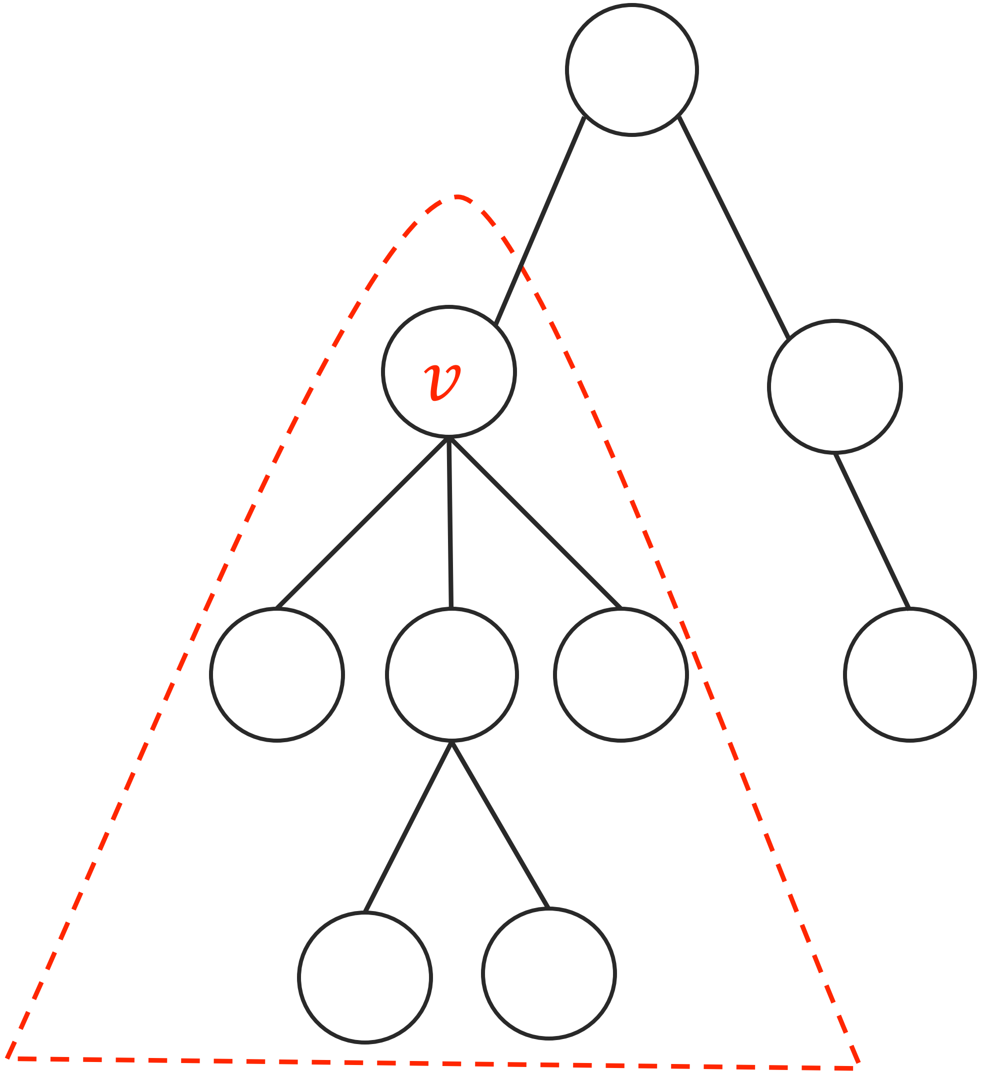
# Step 1: Subproblems for Independent Sets

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

Output: Largest "independent set" of $G$.

**Subproblems:** For each $v \in V$

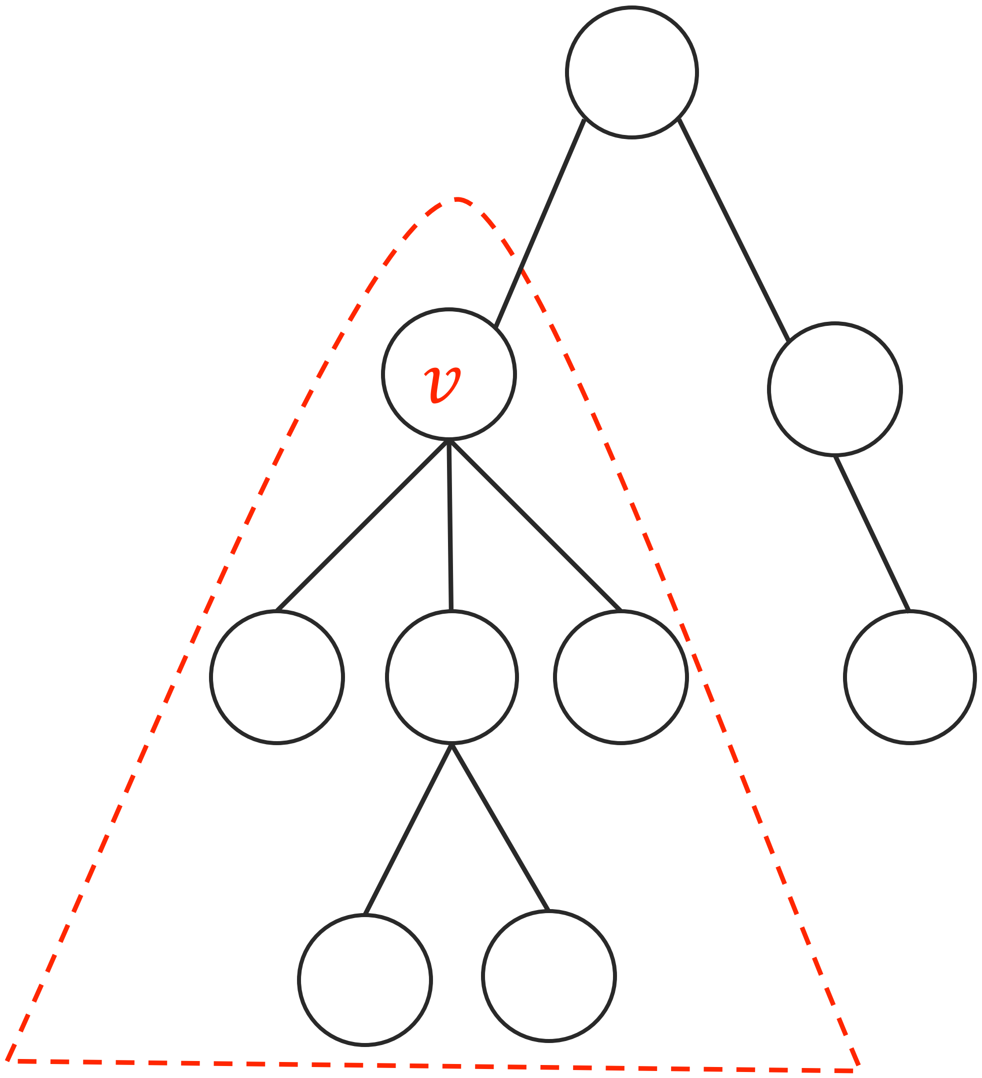
$I(v) =$ Size of max independent set in subtree rooted at $v$.



# 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 $v \in V$

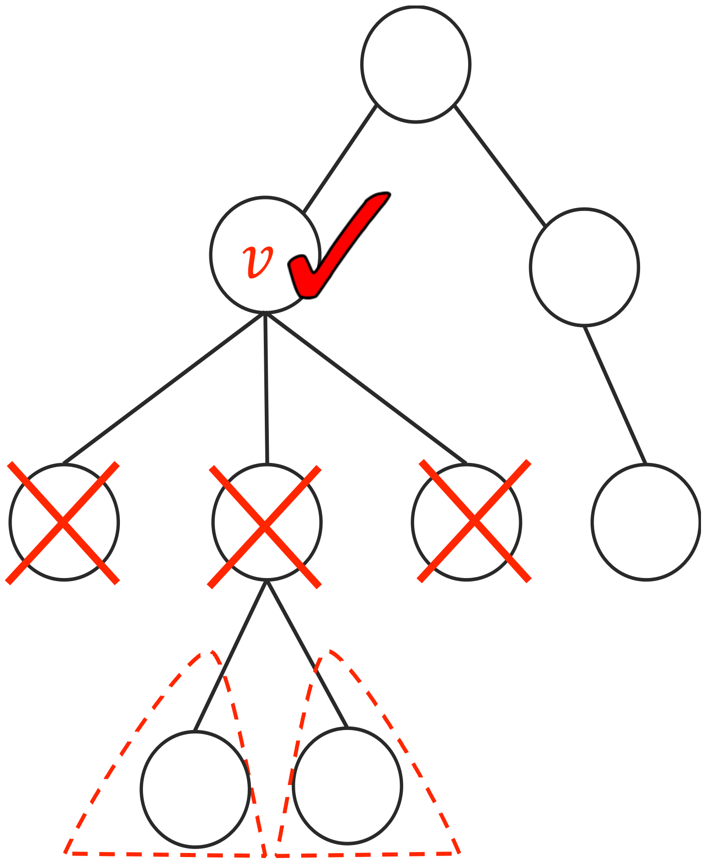
$I(v) =$ Size of max independent set in subtree rooted at $v$.

**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.

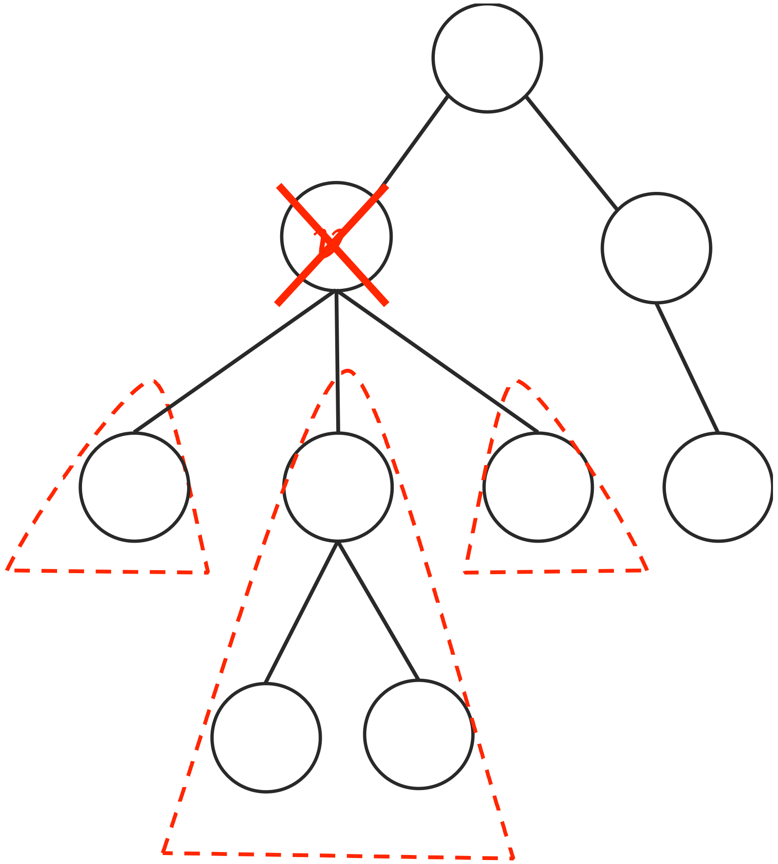
Recurse to the grandchildren levels:

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

# 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.

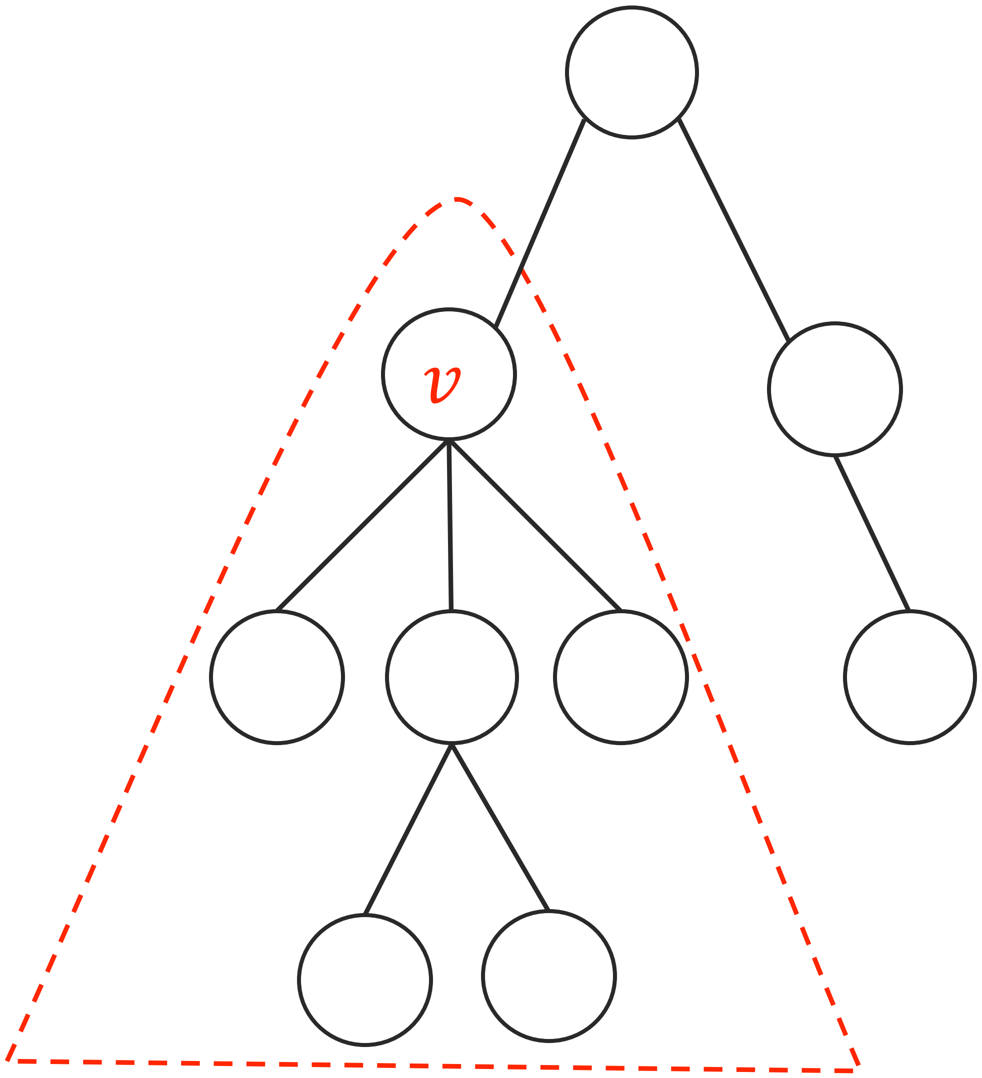
Output: Largest "independent set" of $G$.

**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)

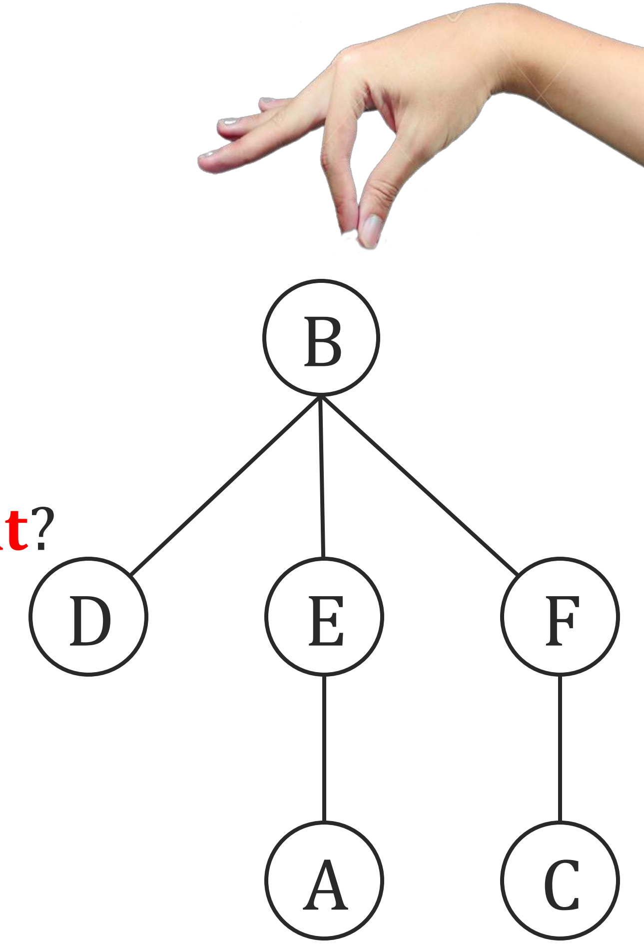
$$I[v] = \max\left\{1 + \sum_{u\text{:grandchild of } v} I[u]\,,\ \sum_{u\text{: child of } v} I[u]\right\}$$



# Step 3: Design the Algorithm

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

<ins>Output</ins>: 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 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** $i = 1, \dots, n$

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

**return** $I[v_n]$

# 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