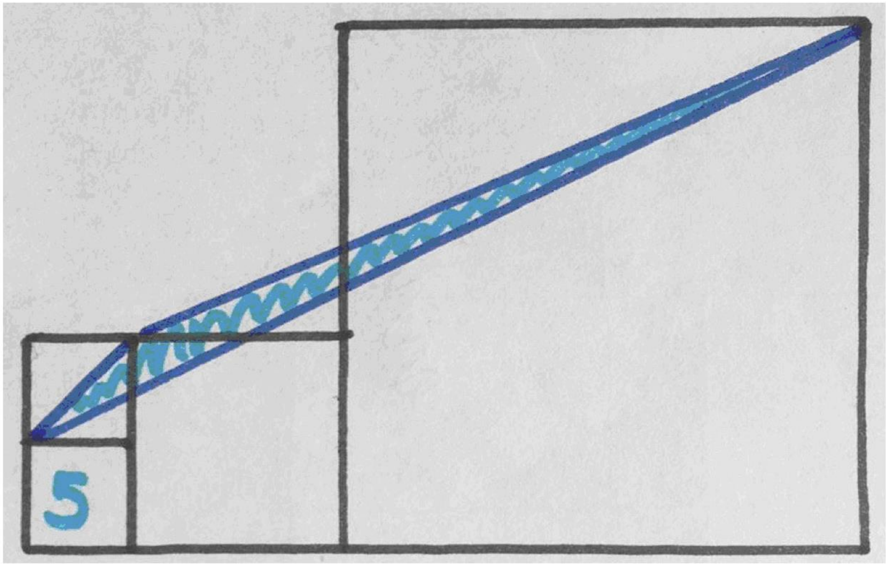
CS2040S
Data Structures and Algorithms

Dynamic Programming…

Puzzle of the Week:

The area of the bottom left square is 5. What's the area of the blue triangle?



Catriona Agg
https://twitter.com/cshearer41/status/1027844515338616832

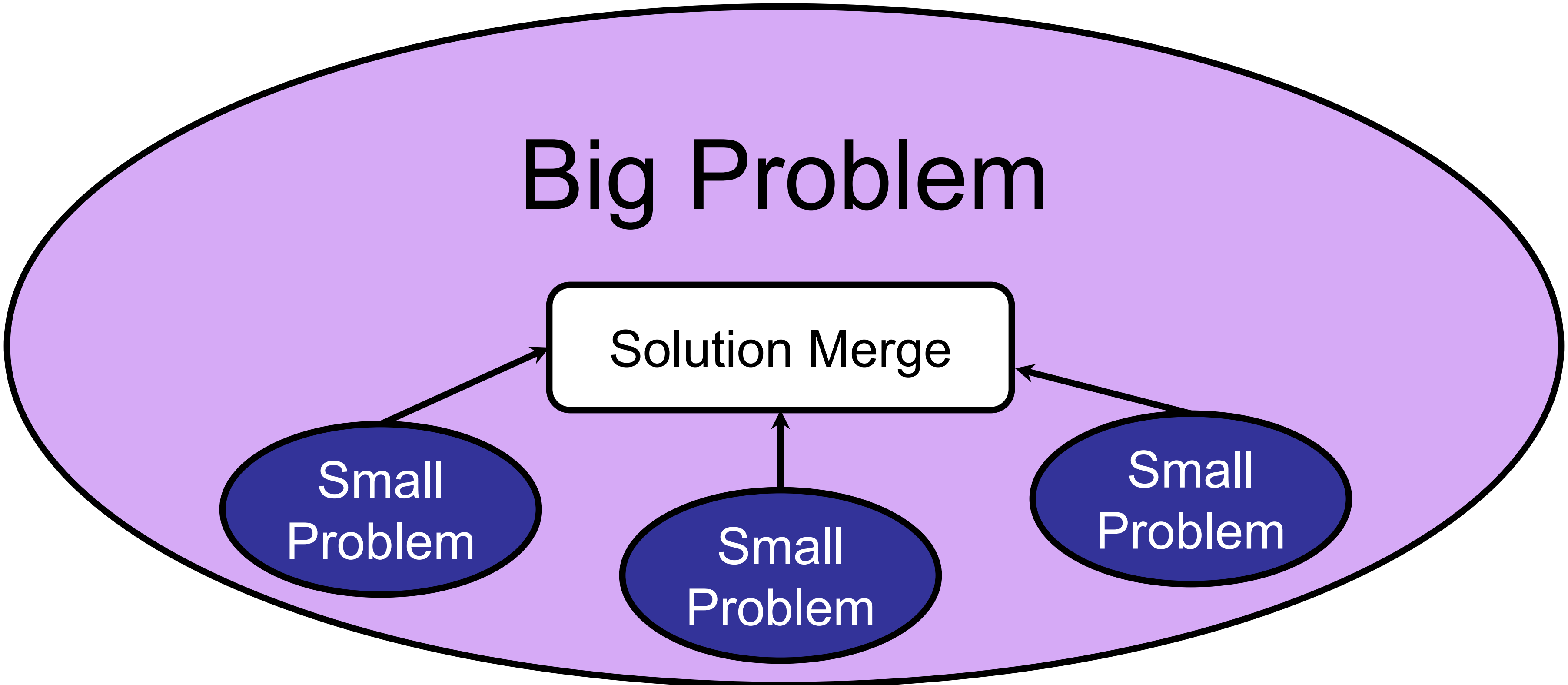
Housekeeping:

No recitation this week. Good luck for CS2030S PE2!

Dynamic Programming Basics

Optimal sub-structure:

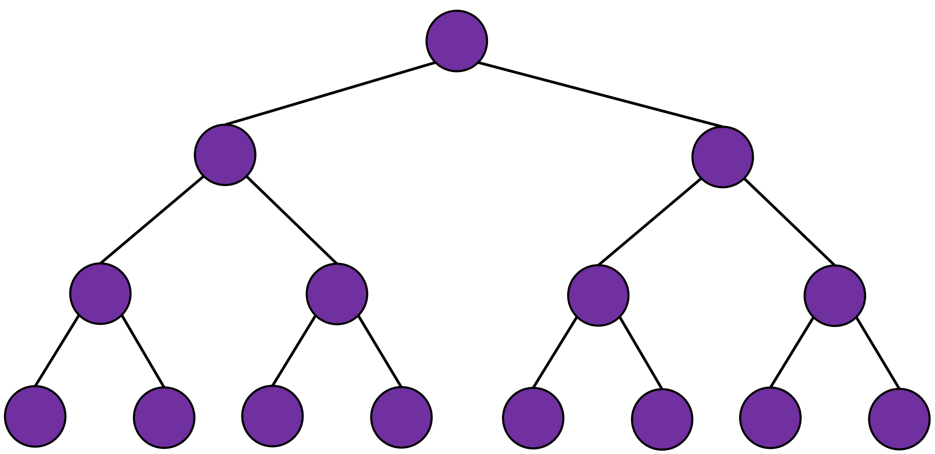
Optimal solution can be constructed from optimal solutions to smaller sub-problems.



Dynamic Programming

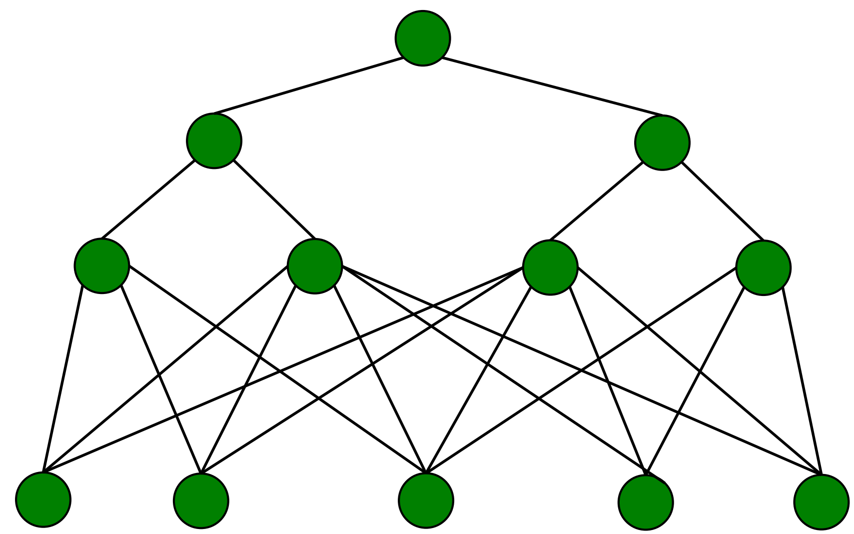
Contrast: Both have optimal substructure

No overlapping subproblems



Divide-and-Conquer

Overlapping subproblems



Dynamic Programming

Dynamic Programming

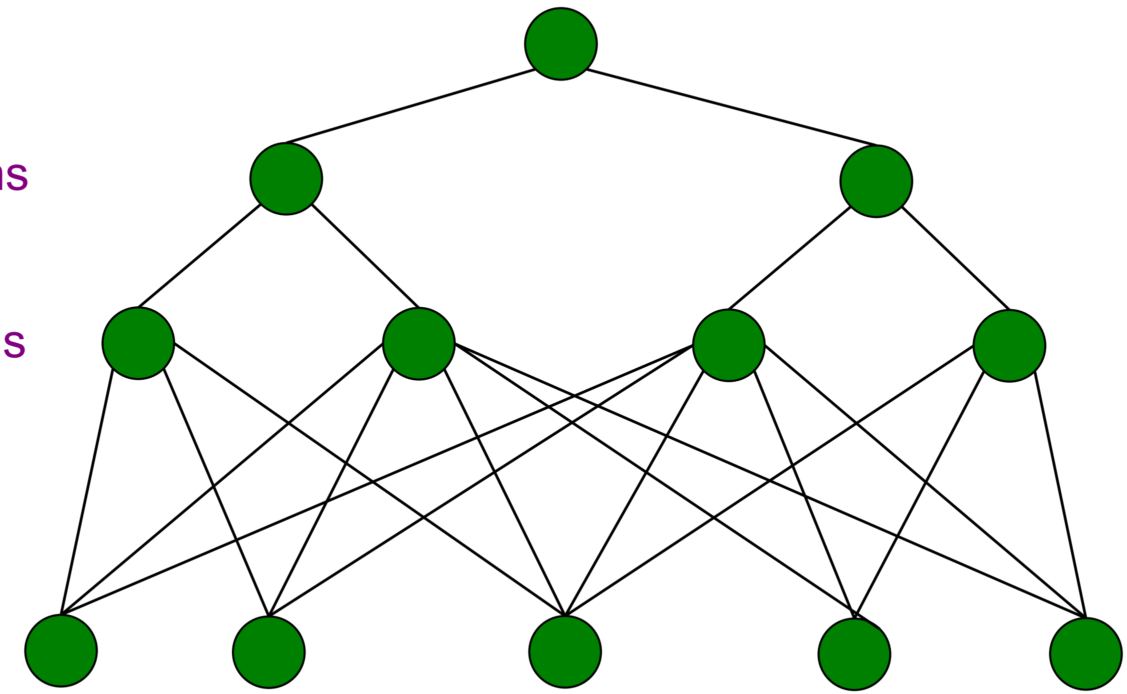
Basic strategy: *(bottom up dynamic programming)*

Step 4: solve root problem

Step 3: combine smaller problems

Step 2: combine smaller problems

Step 1: solve smallest problems

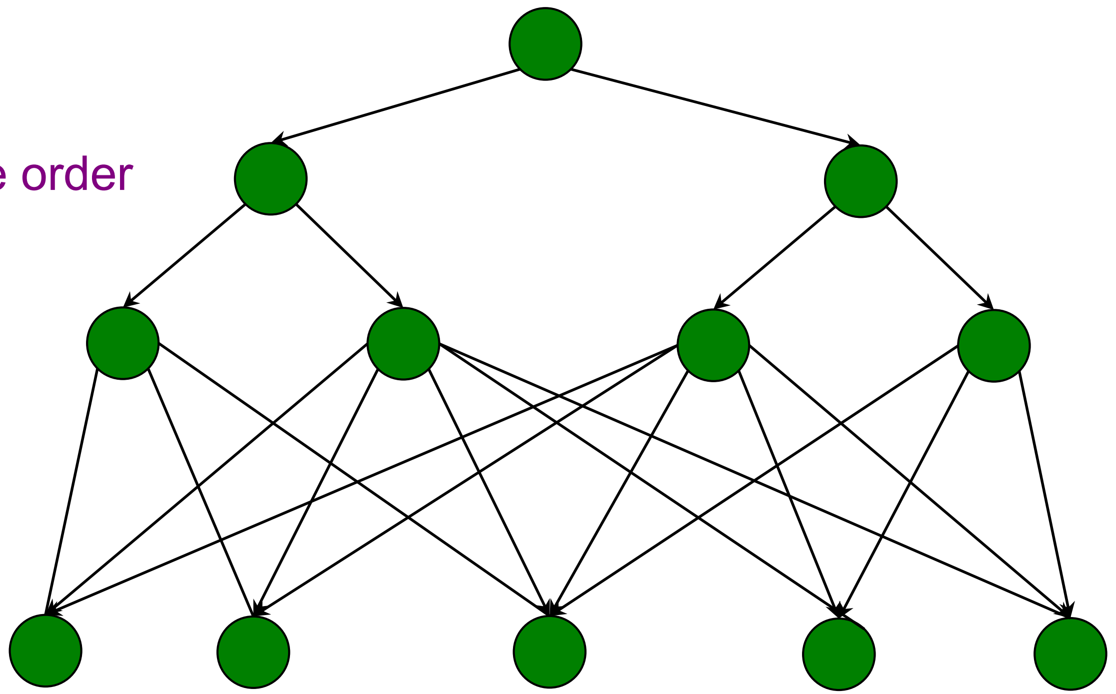


Dynamic Programming

Basic strategy: *(DAG + topological sort)*

Step 1: Topologically sort DAG

Step 2: Solve problems in reverse order



Dynamic Programming

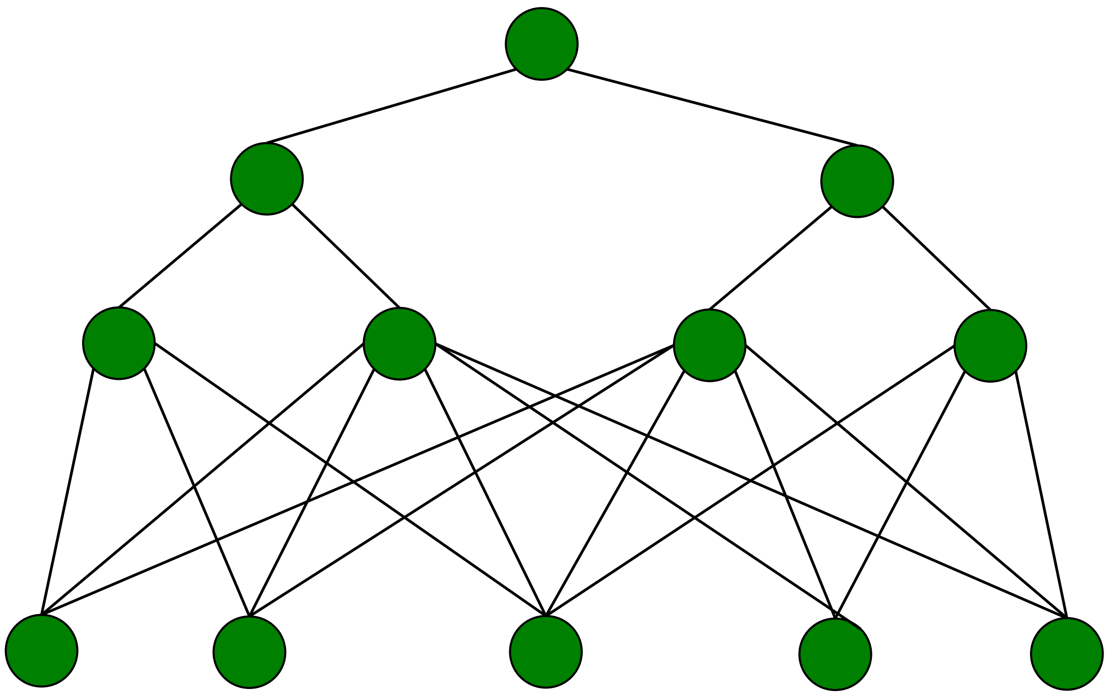
Basic strategy: *(top down dynamic programming)*

Step 1: Start at root and recurse.

Step 2: Recurse.

Step 3: Recurse.

Step 4: Solve and memoize.
Only compute each
solution once.



Roadmap

Dynamic Programming

- ☑ Basics of DP
- ☑ Example: Longest Increasing Subsequence
- ☑ Example: Bounded Prize Collecting
- – Example: Vertex Cover on a Tree
- – Example: All-Pairs Shortest Paths
- – Example: Knapsack

Recall Rough Idea:

A problem can be solved via dynamic programming if:

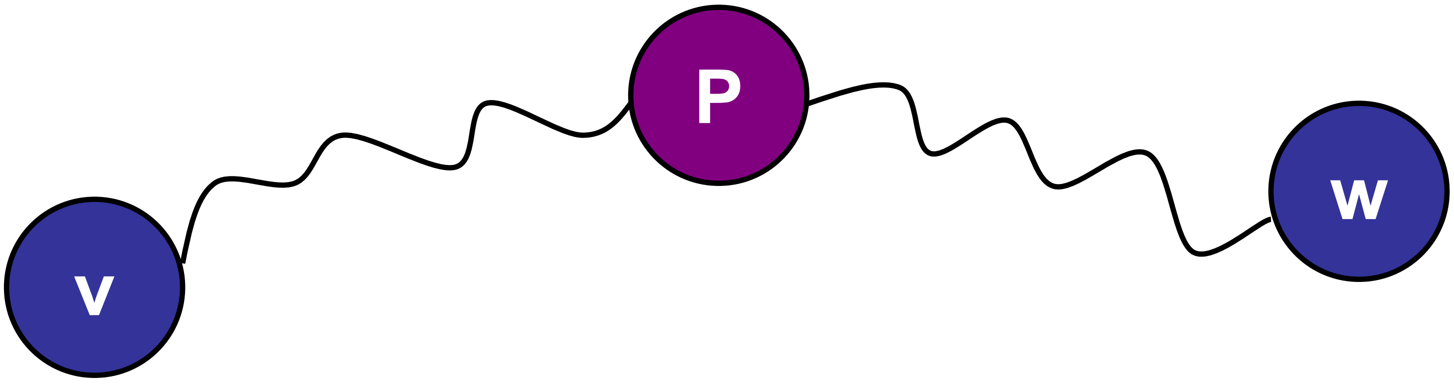
If exhibits optimal sub-structure.

Optimal Substructure:

Solution to a problem uses optimal solutions to its sub-problems.

Optimal Substructure:

Solution to a problem uses optimal solutions to its sub-problems.



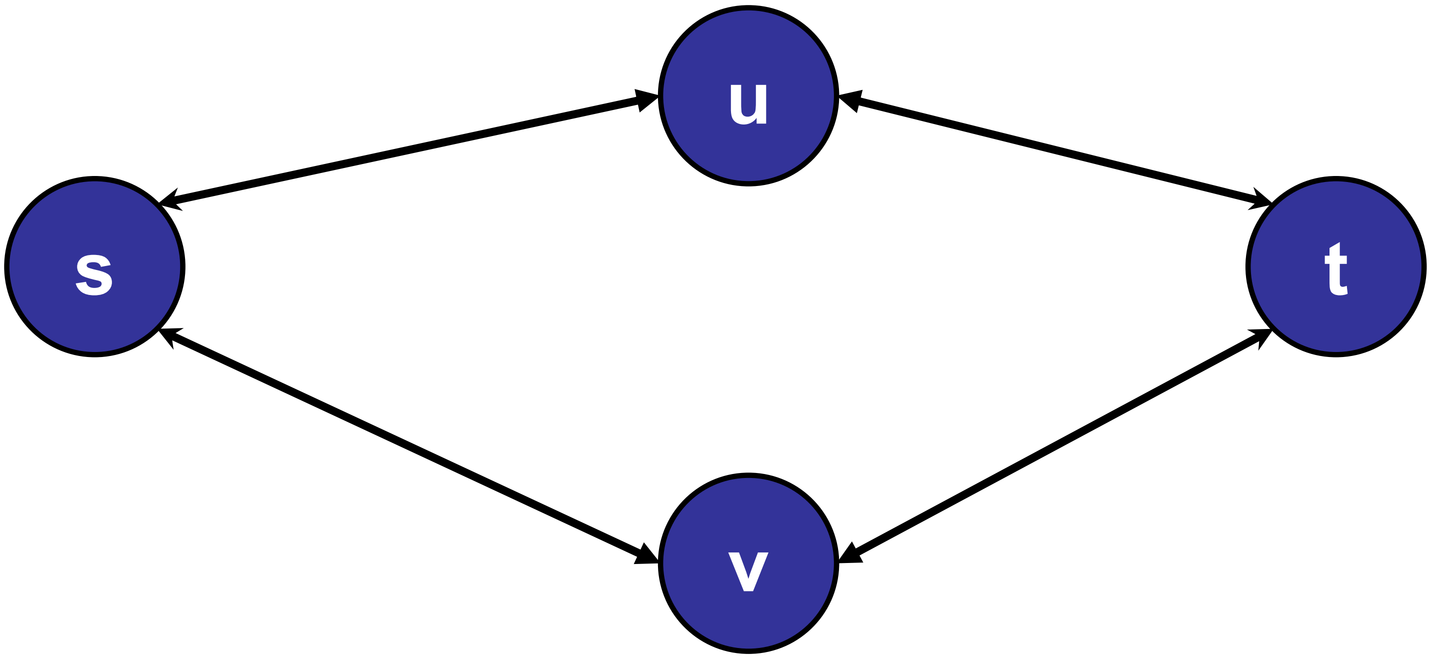
If node P lies on a shortest path from v to w,

then a shortest path is made of:

1. a shortest path from v to P
2. a shortest path from P to w

No Optimal Substructure:

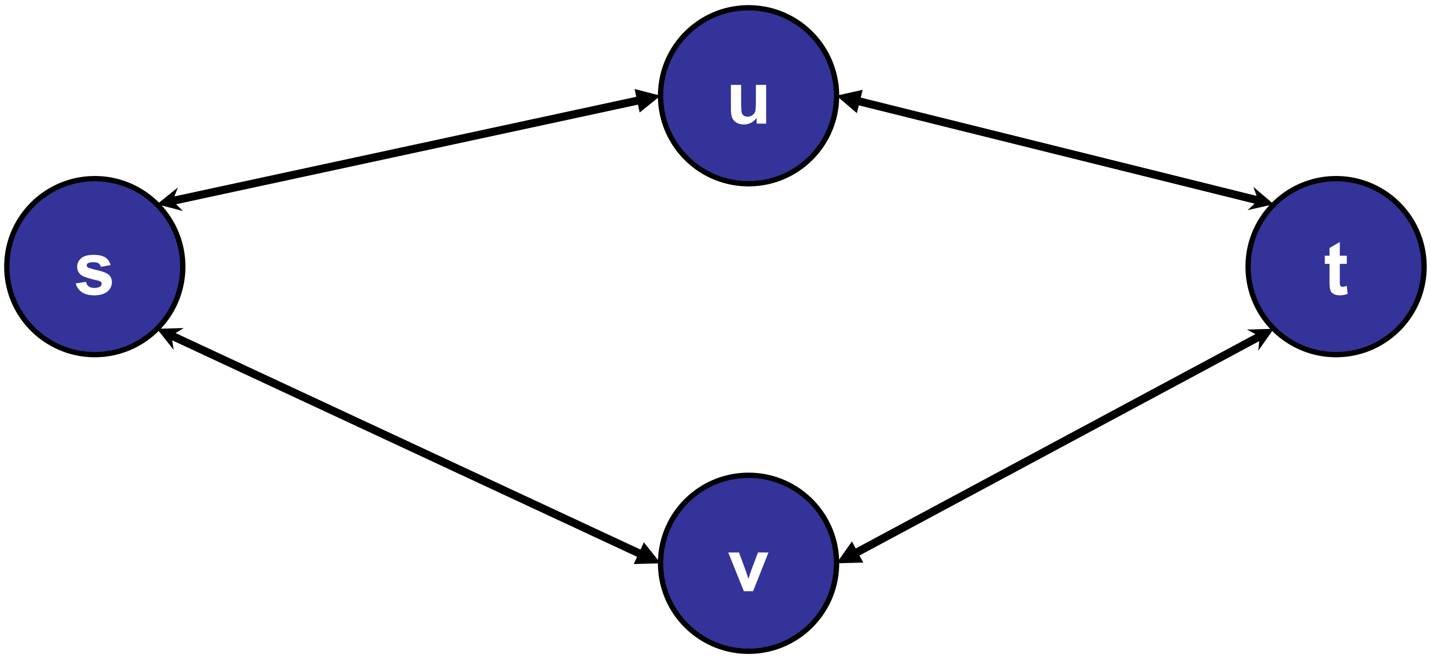
Longest (simple) path problem:



Find a longest simple path from node **s** to node **t**

No Optimal Substructure:

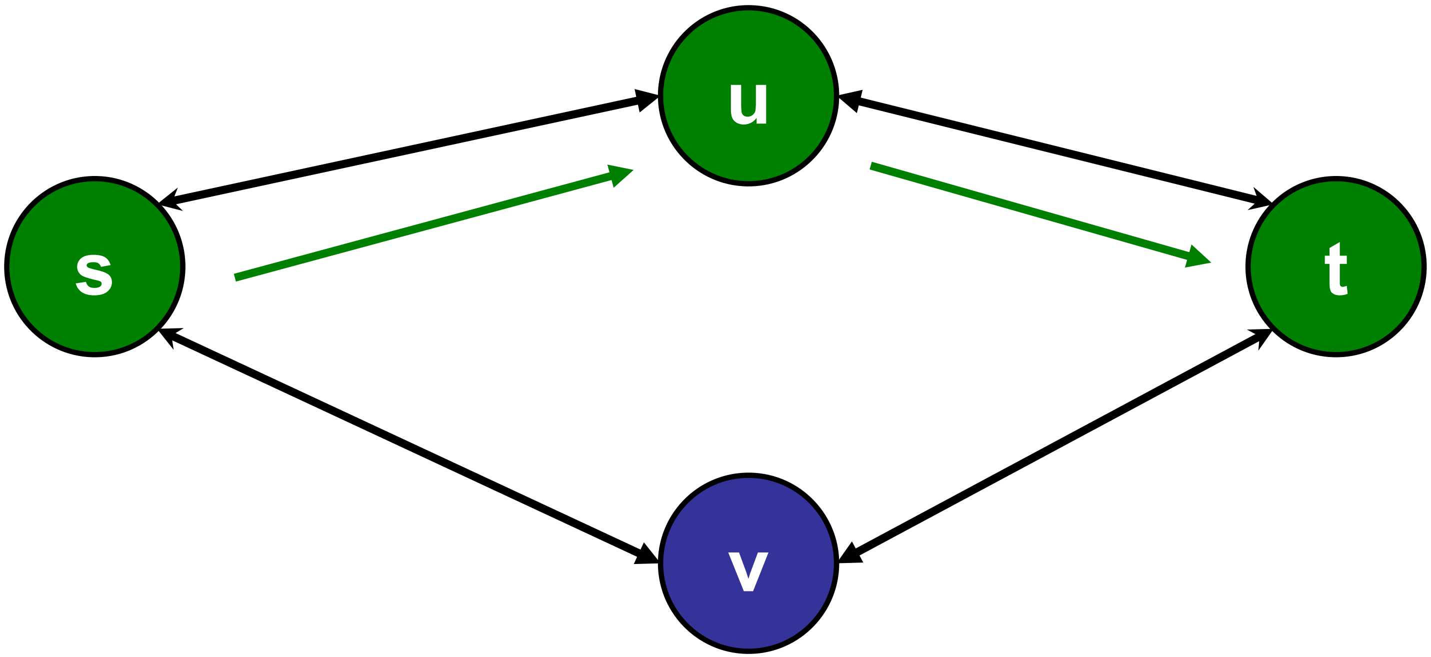
Longest (simple) path problem:



The longest path from **s** to **t** does not use longest paths from **s** to **v** nor **s** to **u**.

No Optimal Substructure:

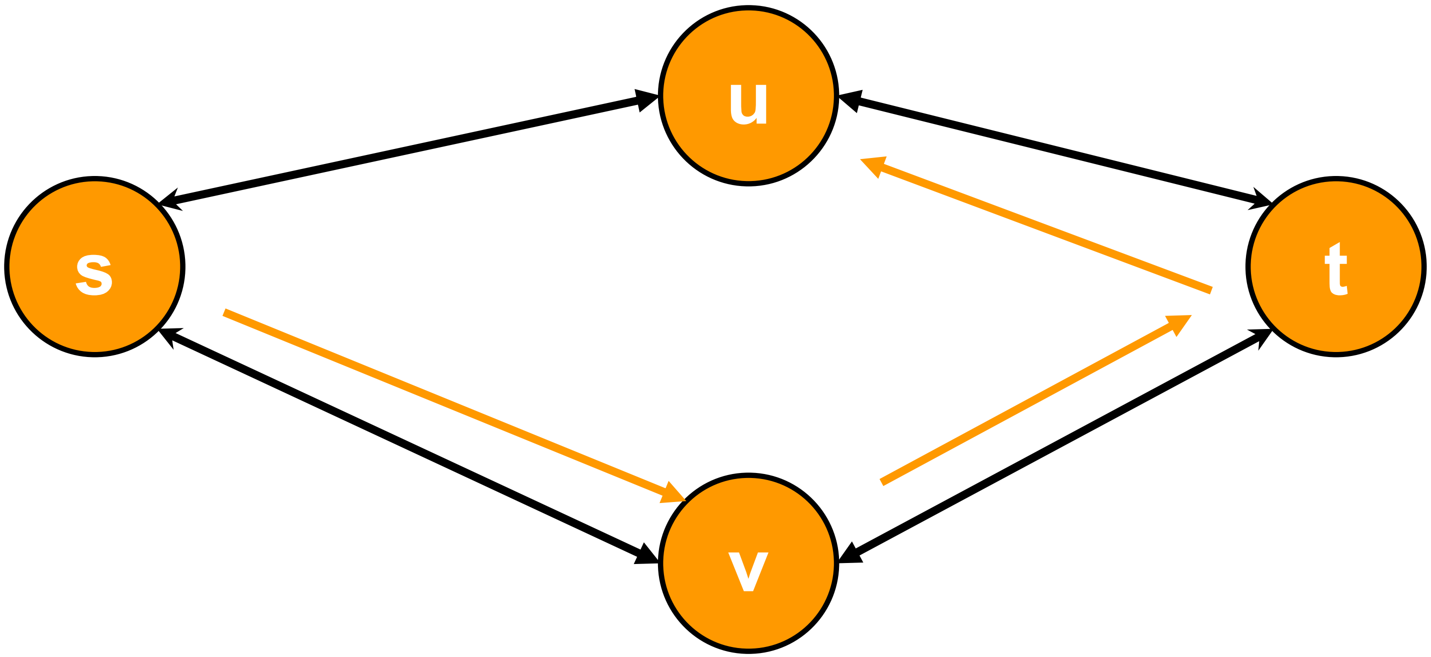
Longest (simple) path problem:



Longest path from **s** to **t**

No Optimal Substructure:

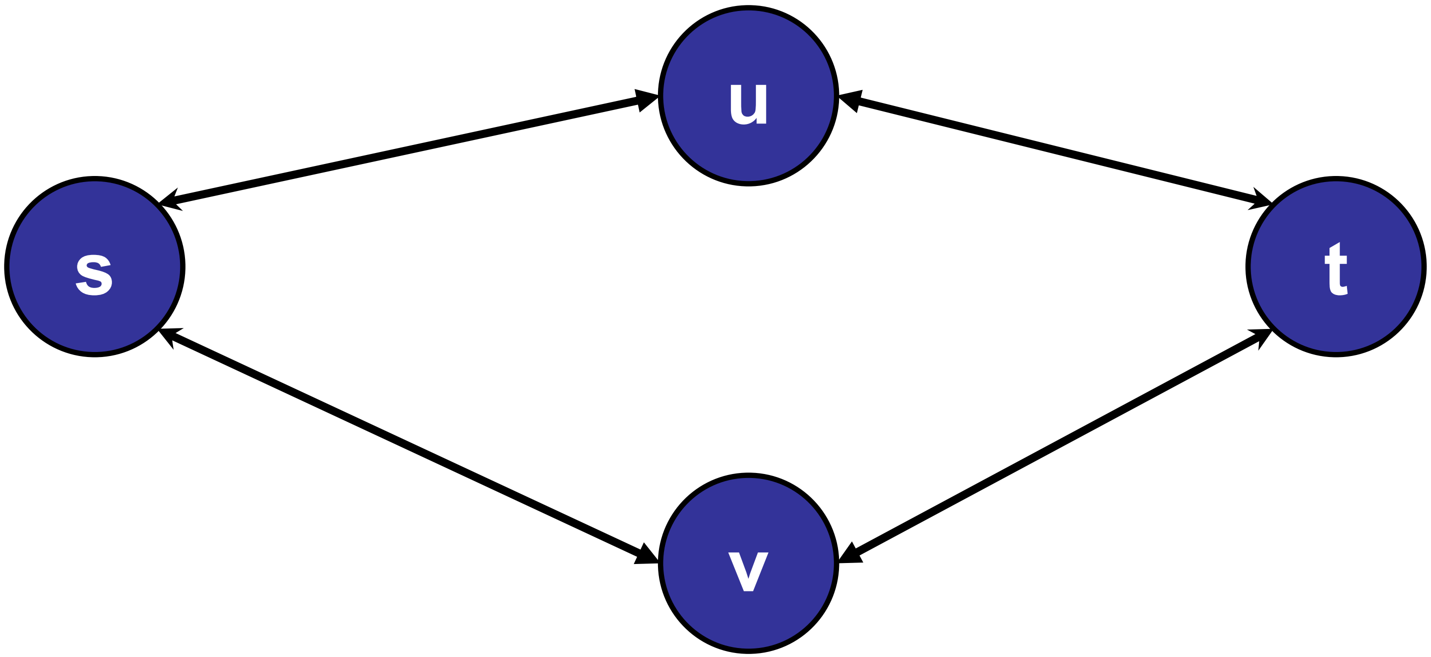
Longest (simple) path problem:



Longest path from **s** to **u**

No Optimal Substructure:

Longest (simple) path problem:

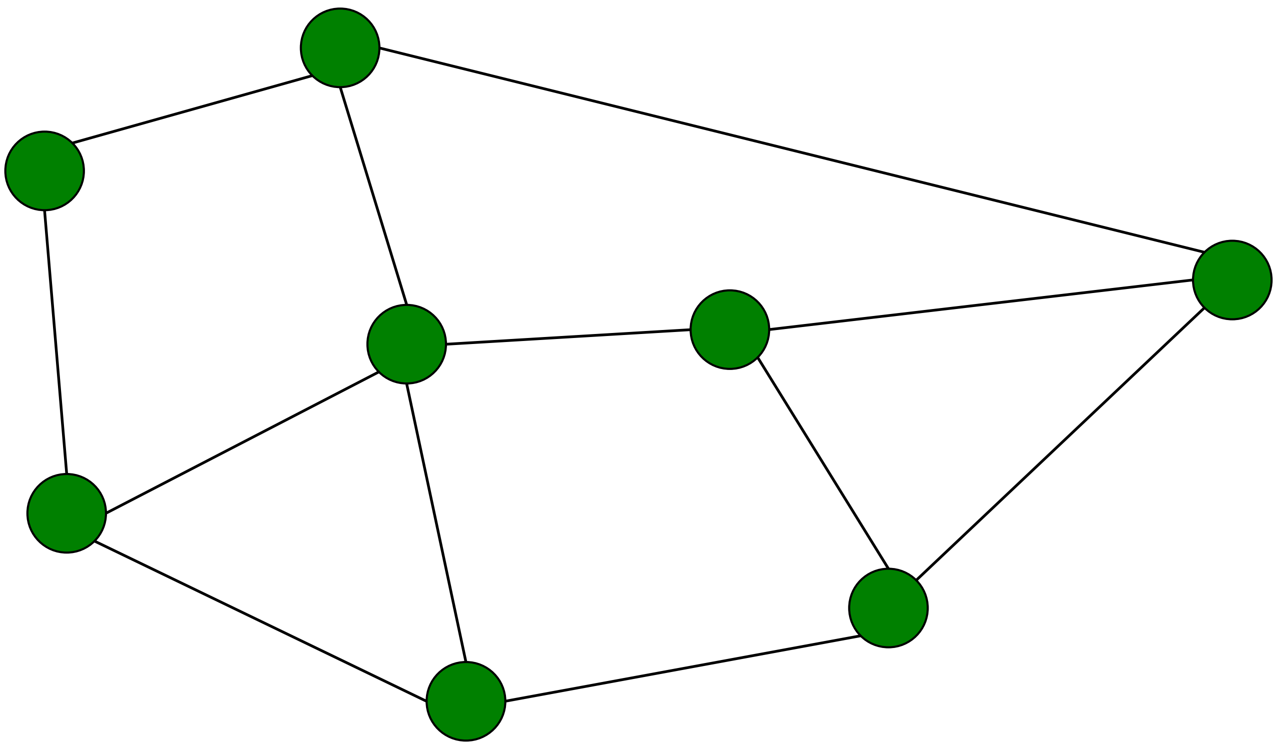


The longest path from **s** to **t** does not use longest paths from **s** to **v** nor **s** to **u**.

Vertex Cover

Input:

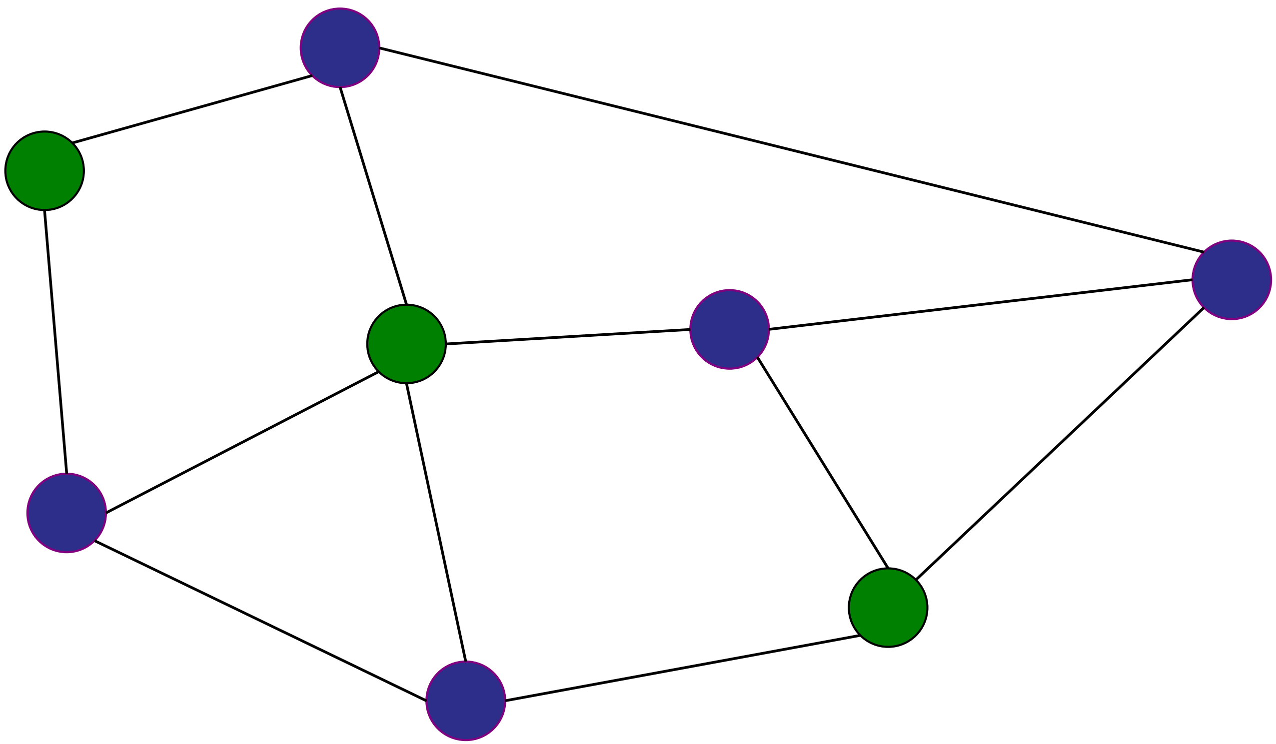
- Undirected, unweighted graph G = (V,E)



Vertex Cover

Output:

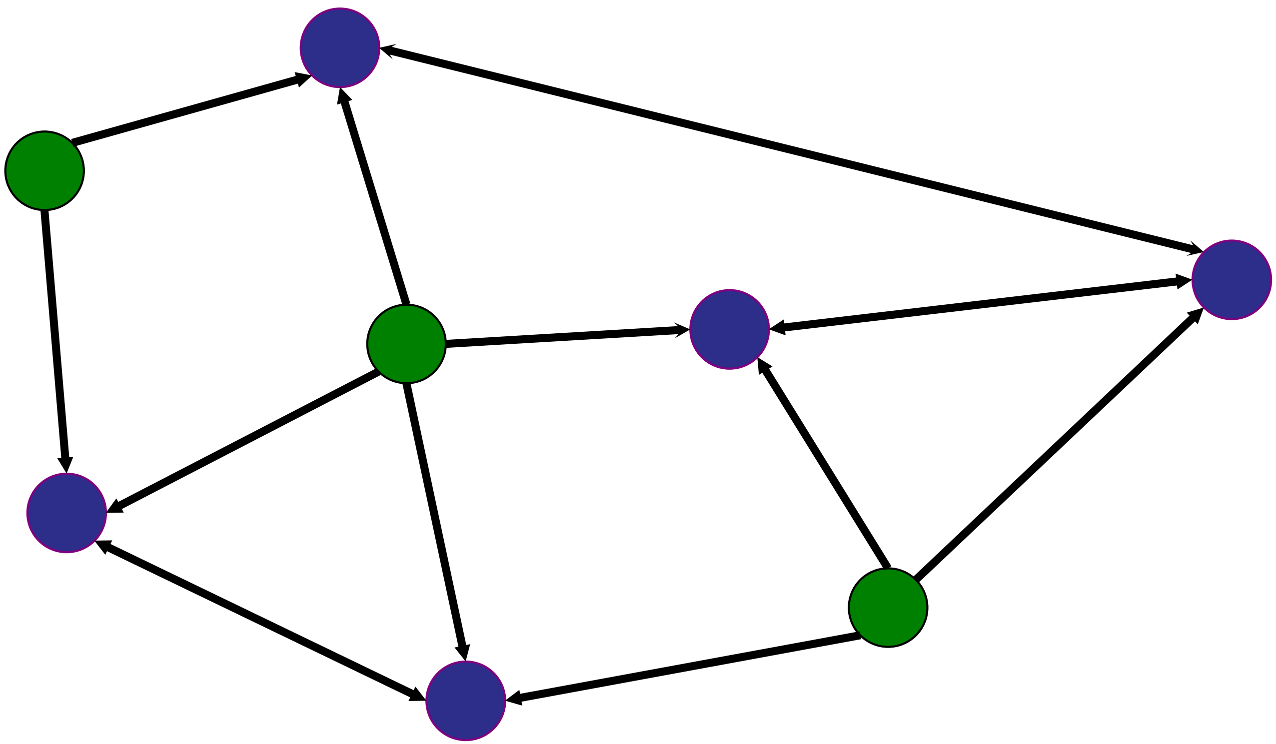
Set of nodes C where every edge is adjacent to at least one node in C.



Vertex Cover

Intuition:

Every edge is "covered" by at least one of its endpoints.



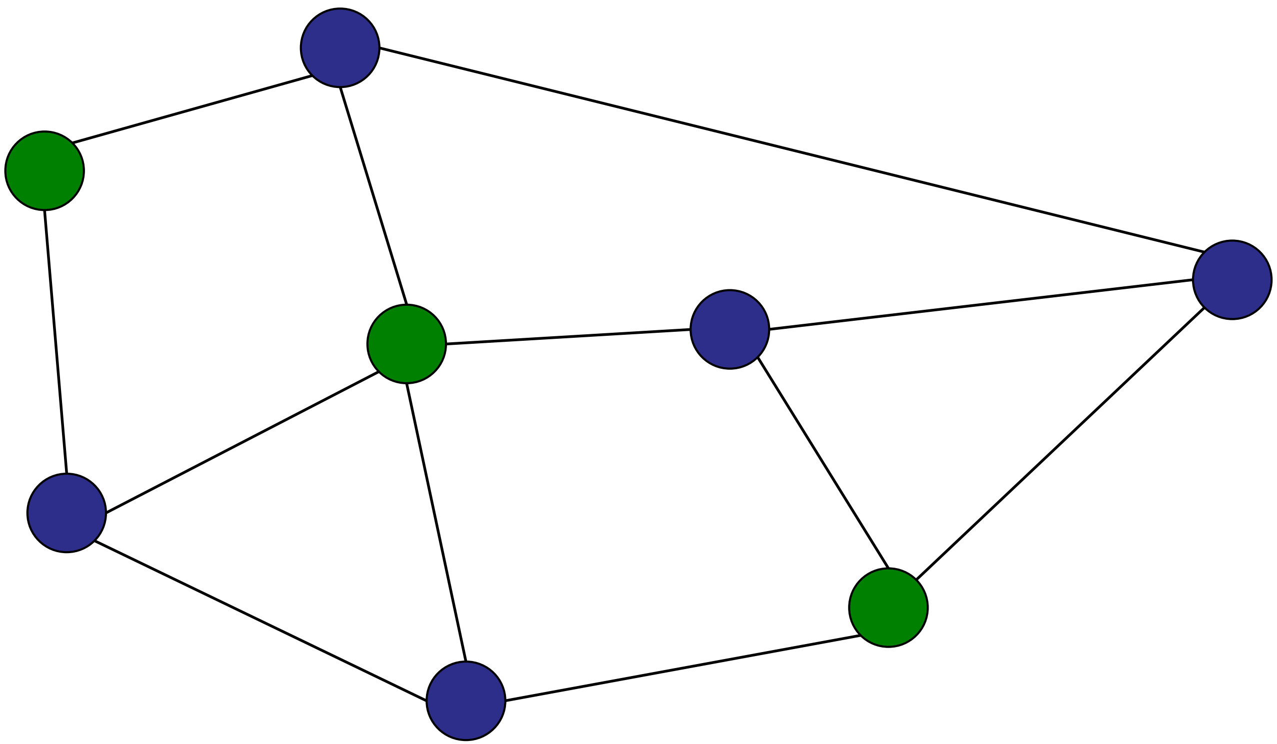
Minimum Vertex Cover

NP-complete:

No polynomial time algorithm (unless P=NP).

Easy 2-approximation (via matchings).

Nothing better known.

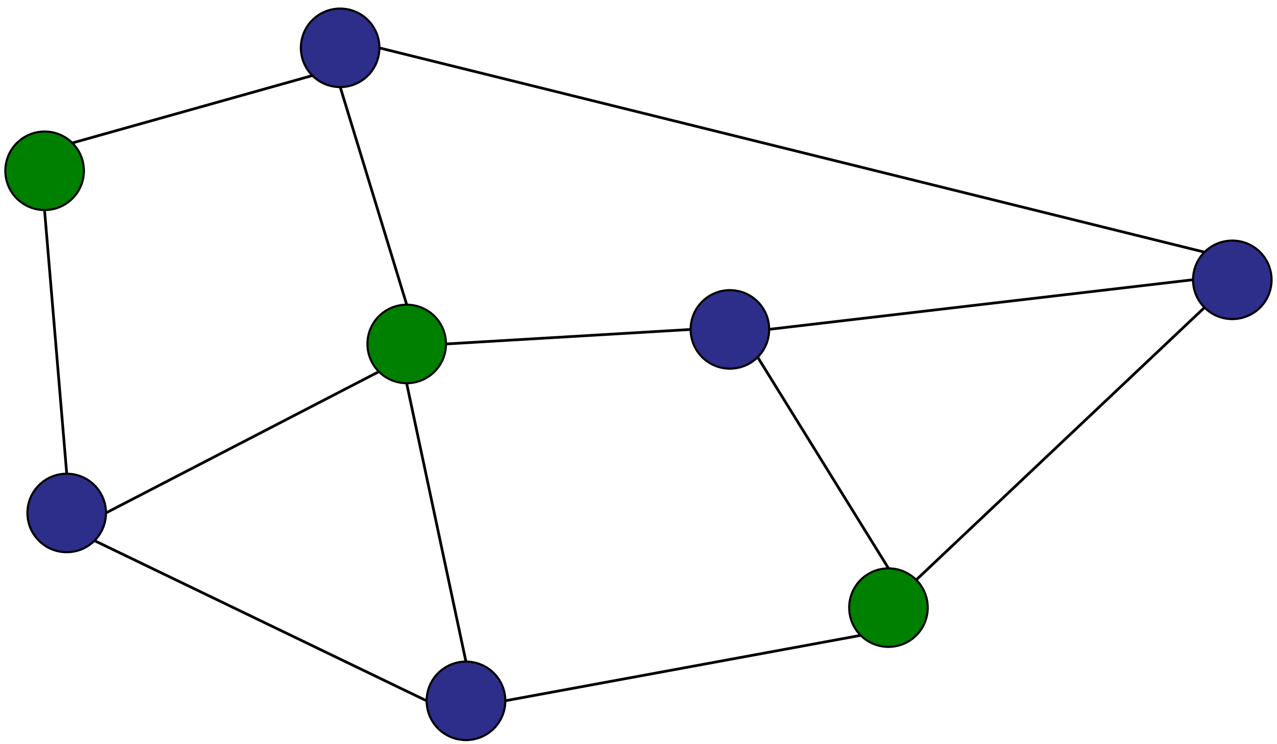


Minimum Vertex Cover

NP-complete:

Solve this problem, win a million US dollars!

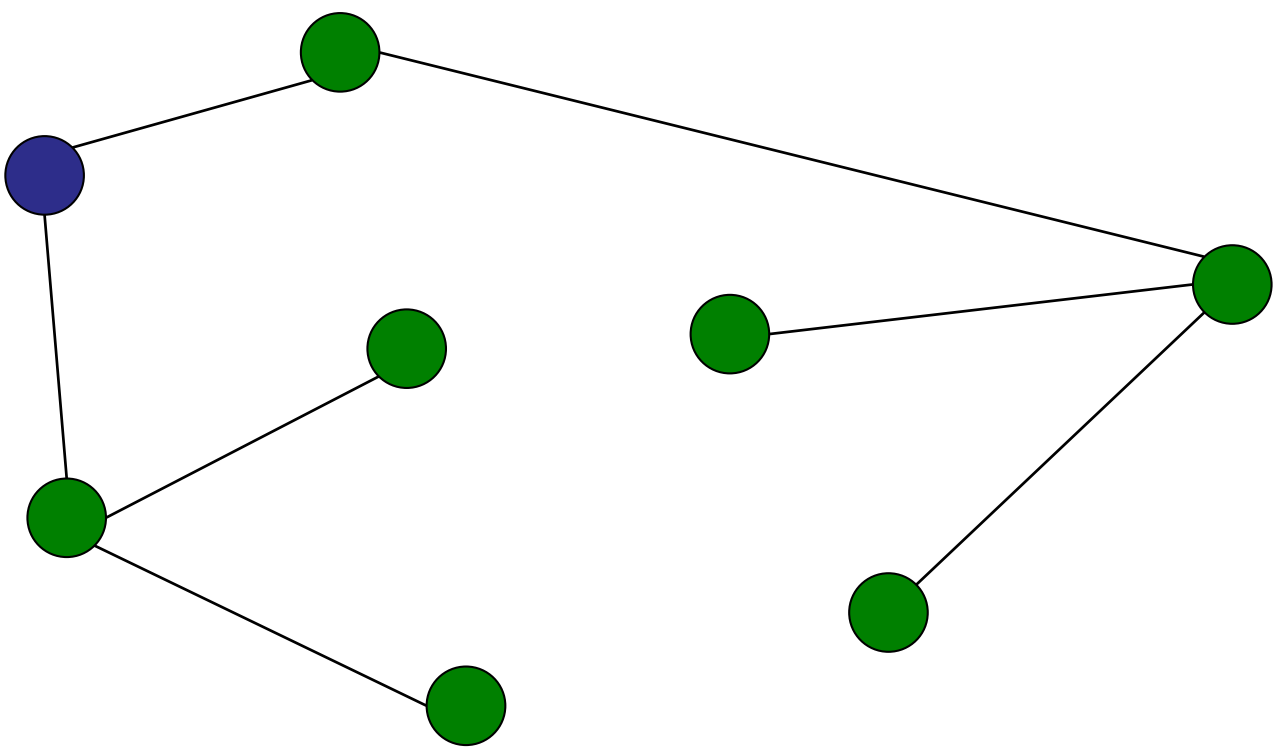
(Hurry before USD devalues)



Vertex Cover on a Tree

Input:

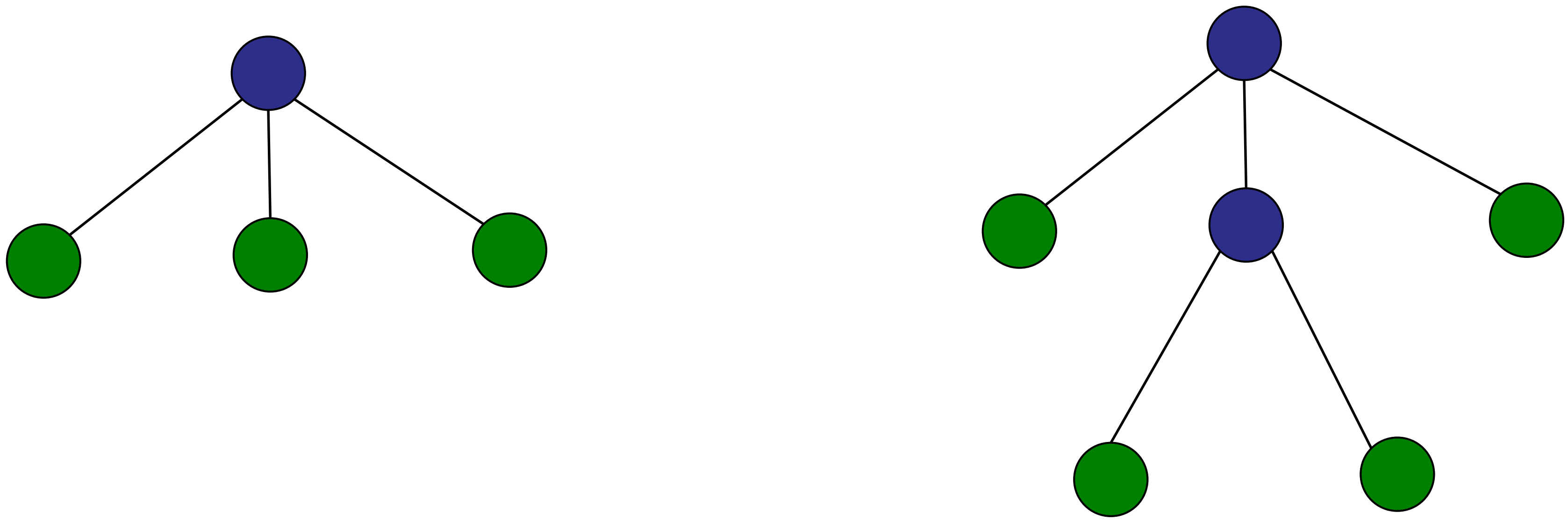
- Undirected, unweighted **tree** G = (V,E)
- Root of tree r



Vertex Cover on a Tree

Output:

- size of the minimum vertex cover



Dynamic Programming Recipe

Step 1: Identify optimal substructure

Step 2: Define sub-problems

Step 3: Solve problem using sub-problems

Step 4: Write (pseudo)code.

Dynamic Programming Analysis

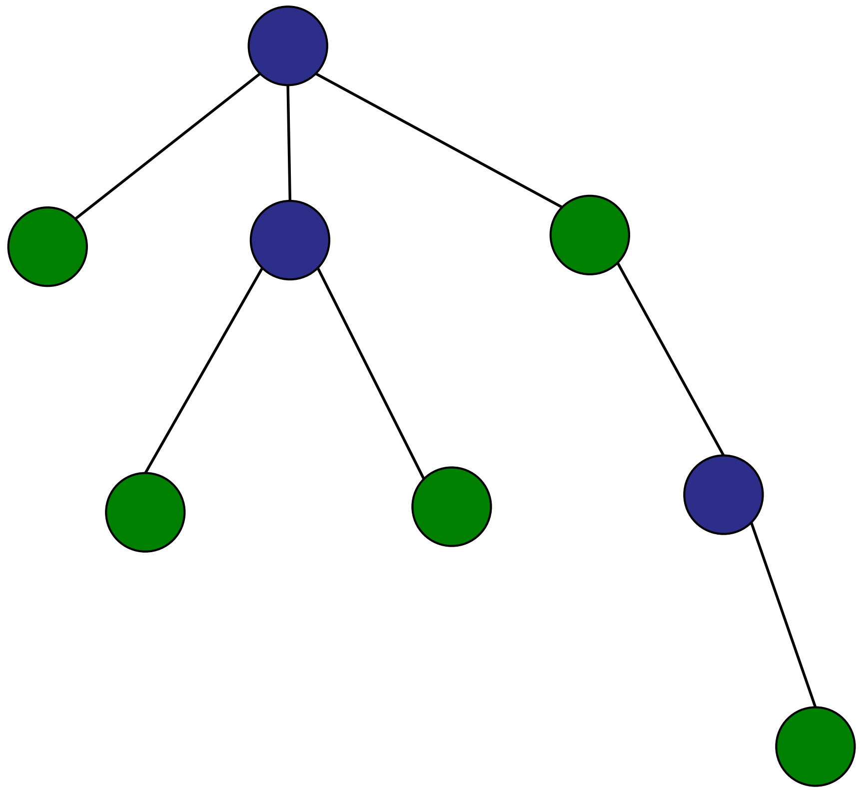
Step 1: Count Sub-problems

Step 2: Figure out total time taken to solve all sub-problems

Often times this is just:
number of sub-problems x time taken per sub-problem

Vertex Cover on a Tree

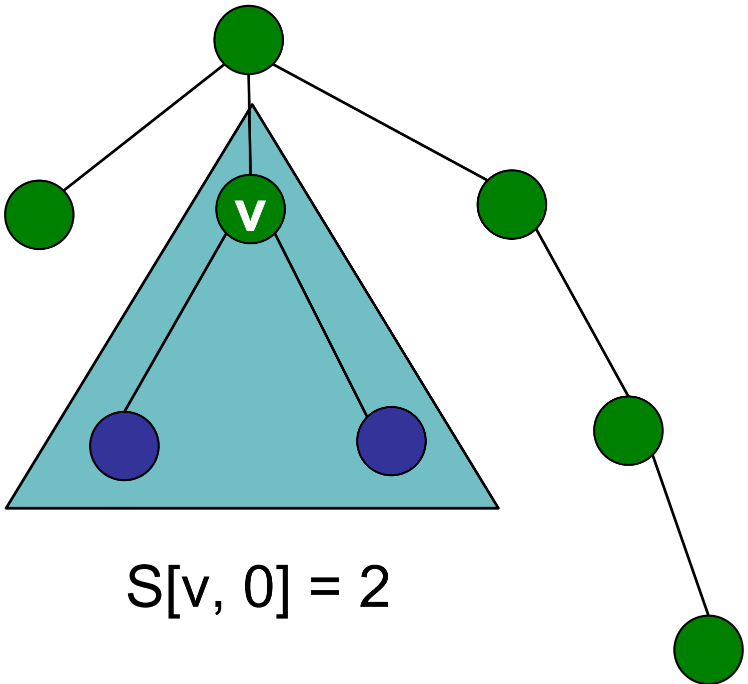
What are the subproblems?



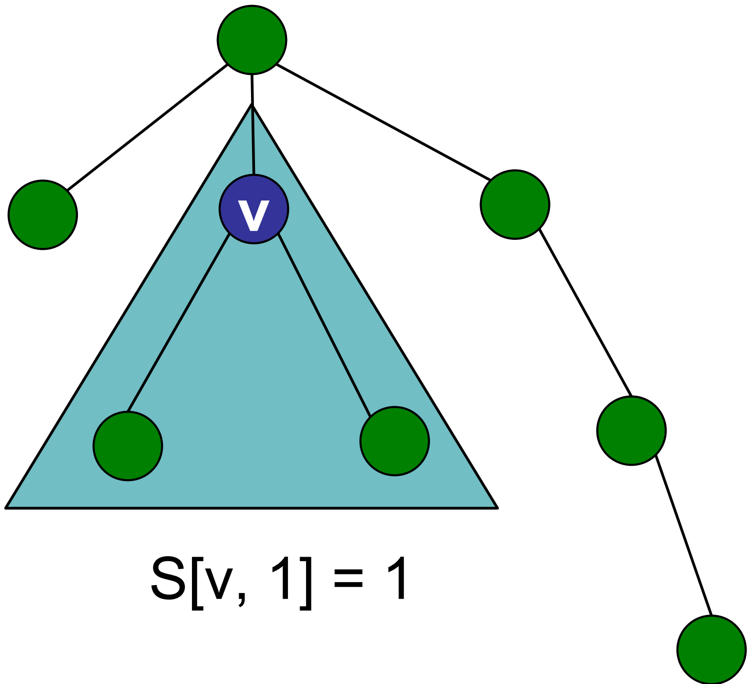
Vertex Cover on a Tree

S[v, 0] = size of vertex cover in subtree rooted at node v, if v is NOT covered.

S[v, 1] = size of vertex cover in subtree rooted at node v, if v IS covered.



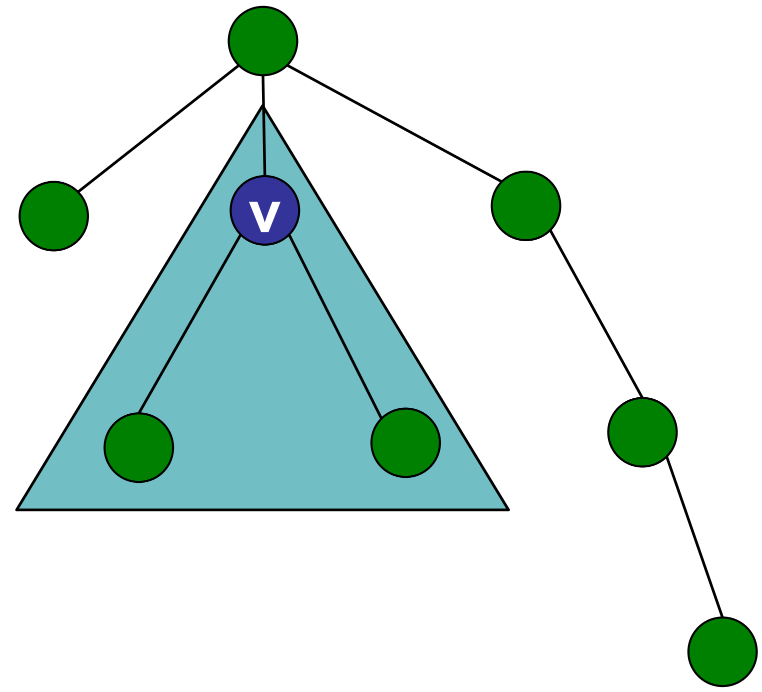
S[v, 0] = 2



S[v, 1] = 1

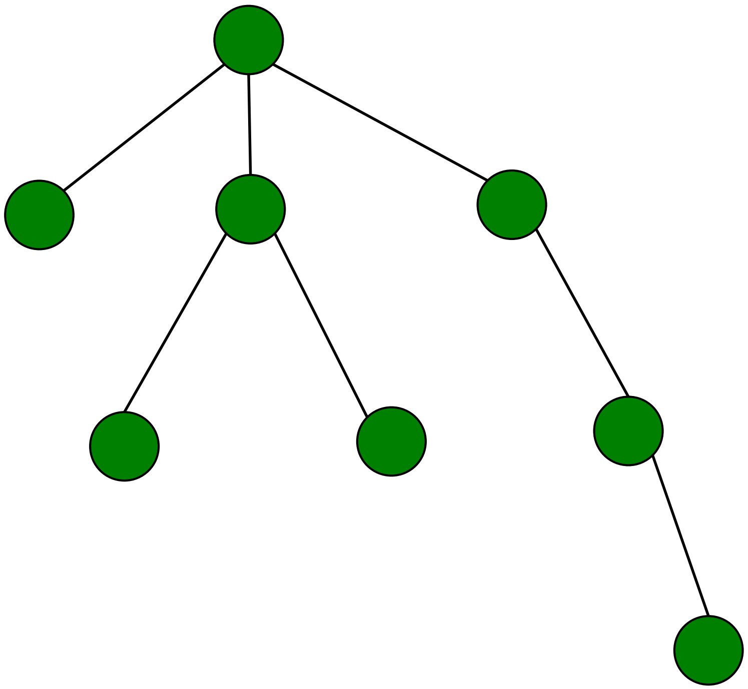
How many subproblems?

1. 2
2. V
3. 2V
4. E
5. 2E
6. VE



Vertex Cover on a Tree

What is the base case?



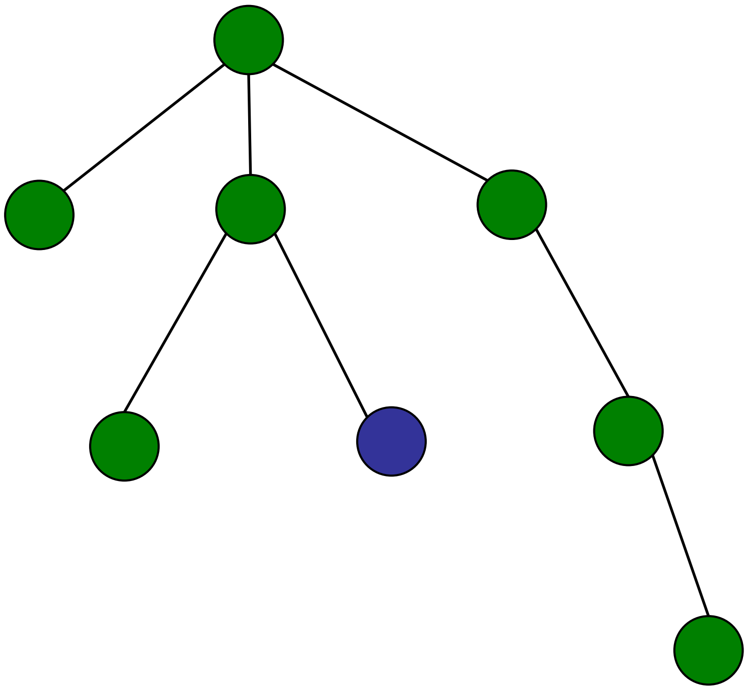
Vertex Cover on a Tree

What is the base case?

Start at the leaves!

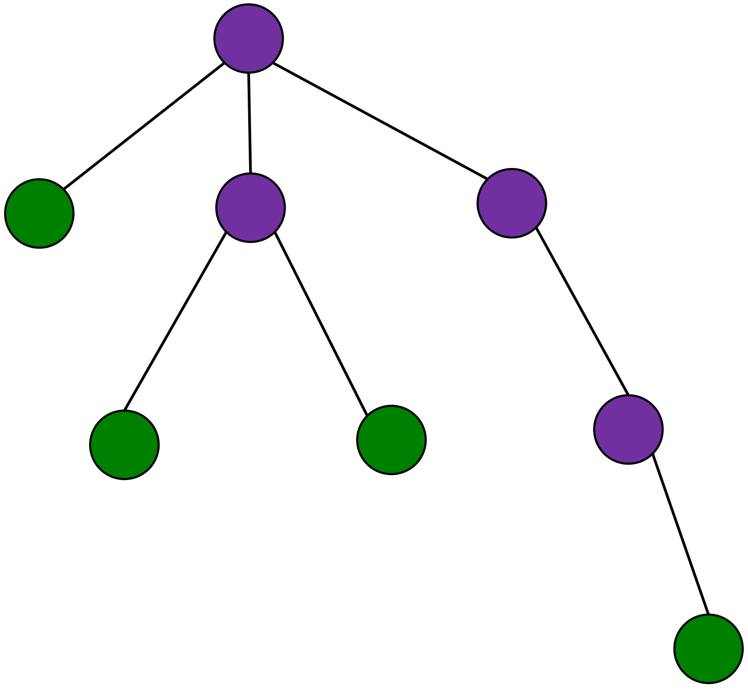
S[leaf, 0] = 0

S[leaf, 1] = 1



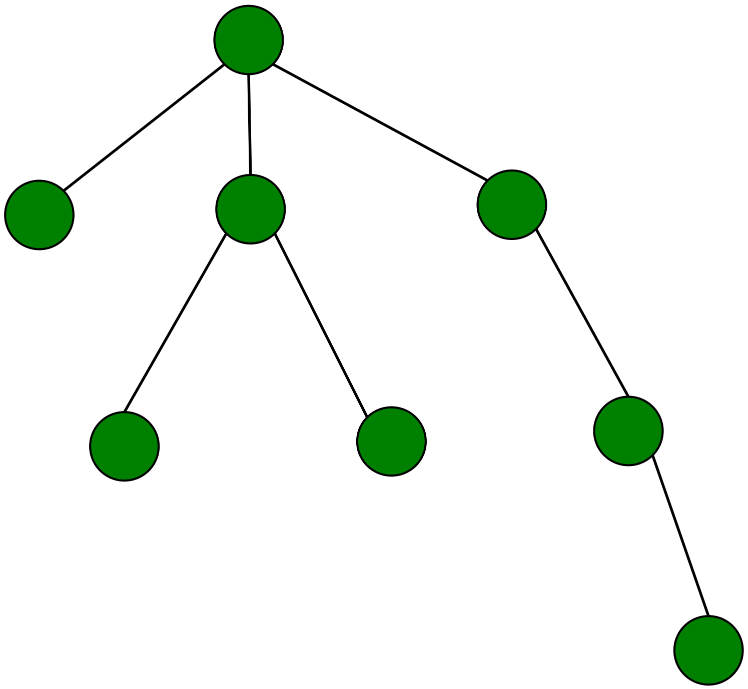
Vertex Cover on a Tree

What about the internal nodes?



Vertex Cover on a Tree

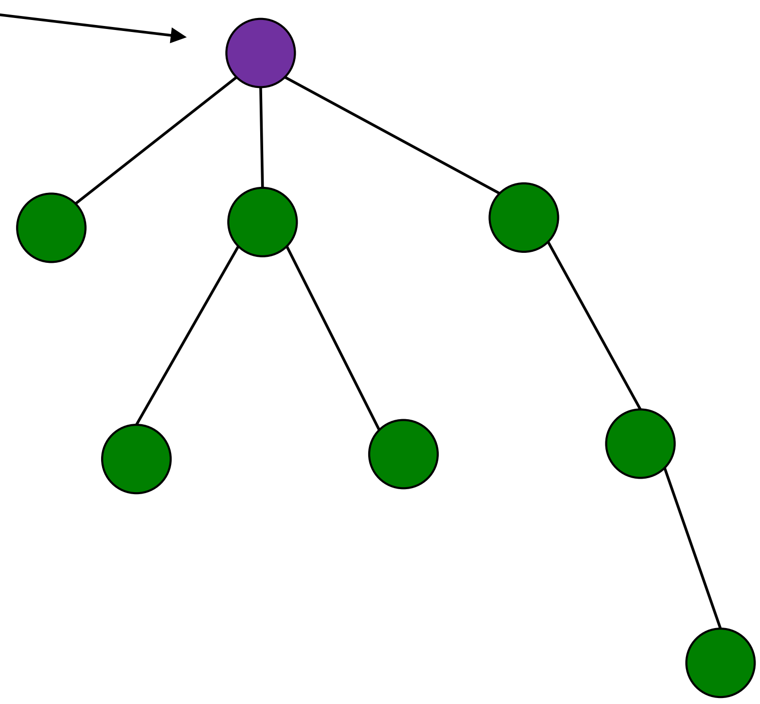
How do we calculate S[v, 0]? (For internal node v)



Vertex Cover on a Tree

How do we calculate $S[v, 0]$? (For internal node v)

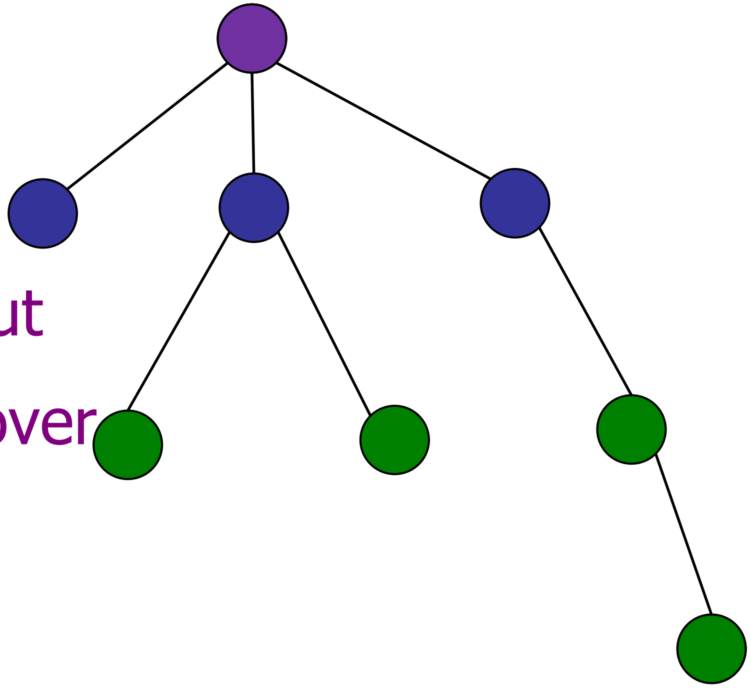
E.g. this node:
What happens if we don't include it in the cover?



Vertex Cover on a Tree

How do we calculate S[v, 0]?

If v is not in the cover, then v's children **needs to be** in the cover.

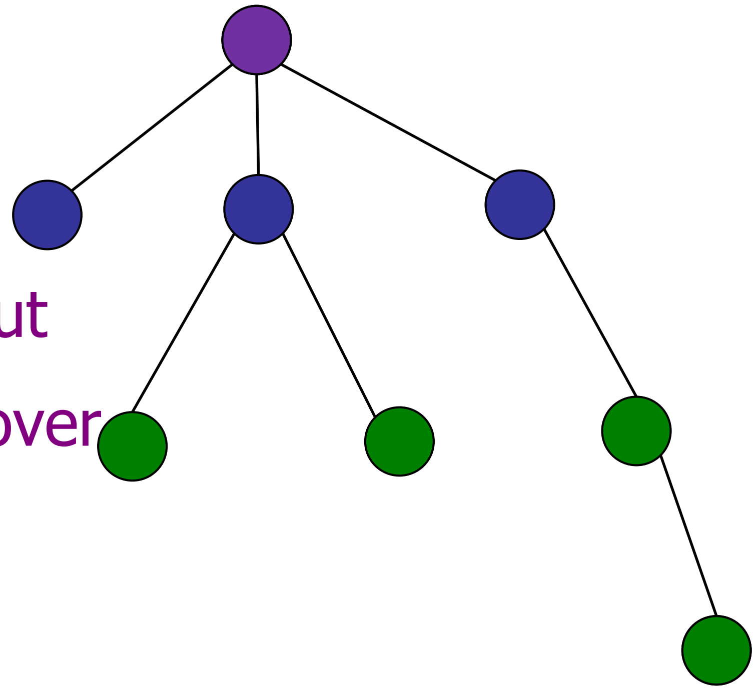


Since S[v, 0] should output the size of the smallest cover without using node v

Vertex Cover on a Tree

How do we calculate S[v, 0]?

If v is not in the cover, then v's children **needs to be** in the cover.



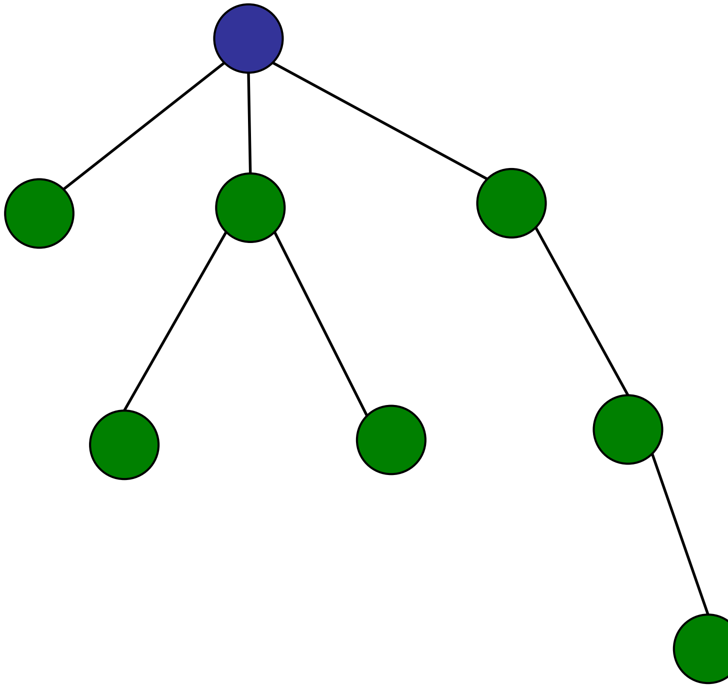
Since S[v, 0] should output the size of the smallest cover without using node v

S[v, 0] = sum(S[n, 1]) for all n that are v's neighbours.

Vertex Cover on a Tree

What about S[v, 1]?

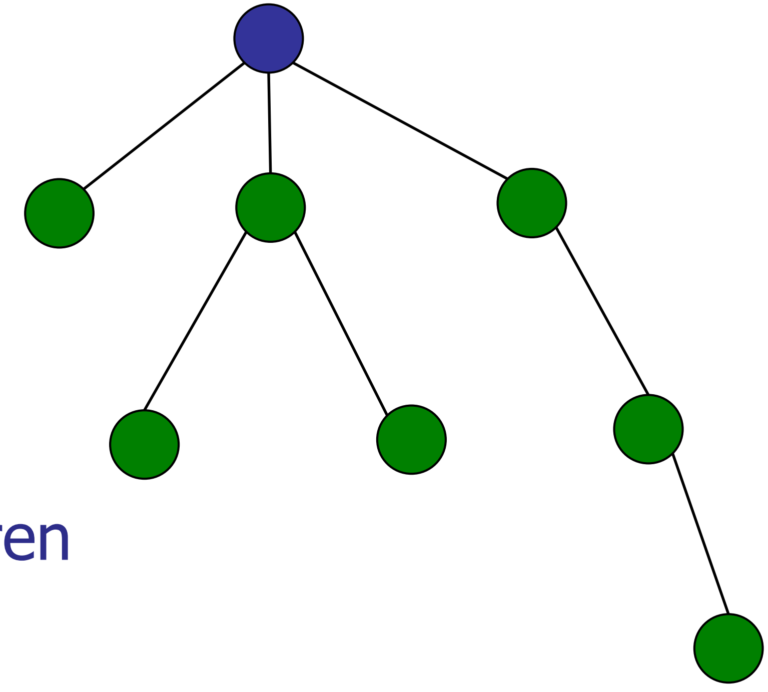
This solution corresponds to including node v in the cover.



Vertex Cover on a Tree

What about S[v, 1]?

This solution corresponds to including node v in the cover.



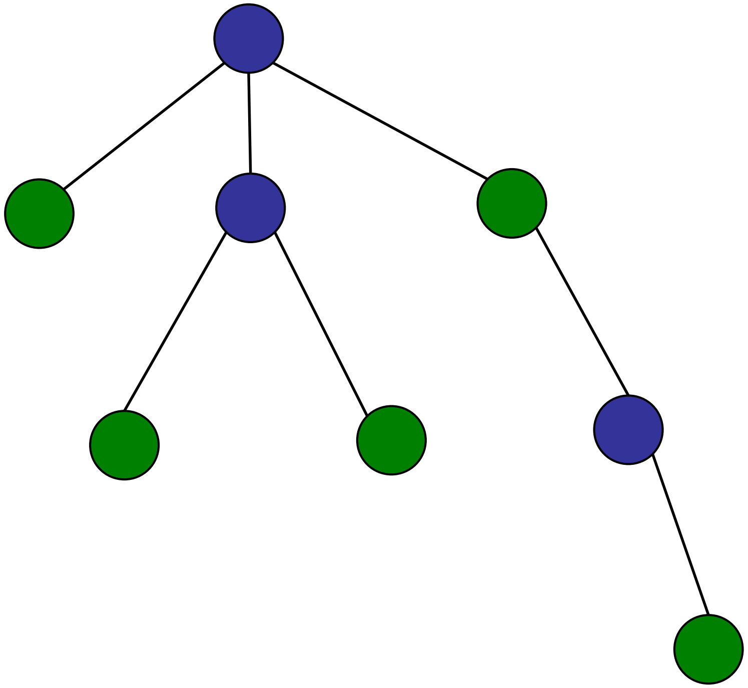
We should consider:

1. Solutions that include our children
2. Solutions that don't

Vertex Cover on a Tree

What about S[v, 1]?

This solution corresponds to including node v in the cover.

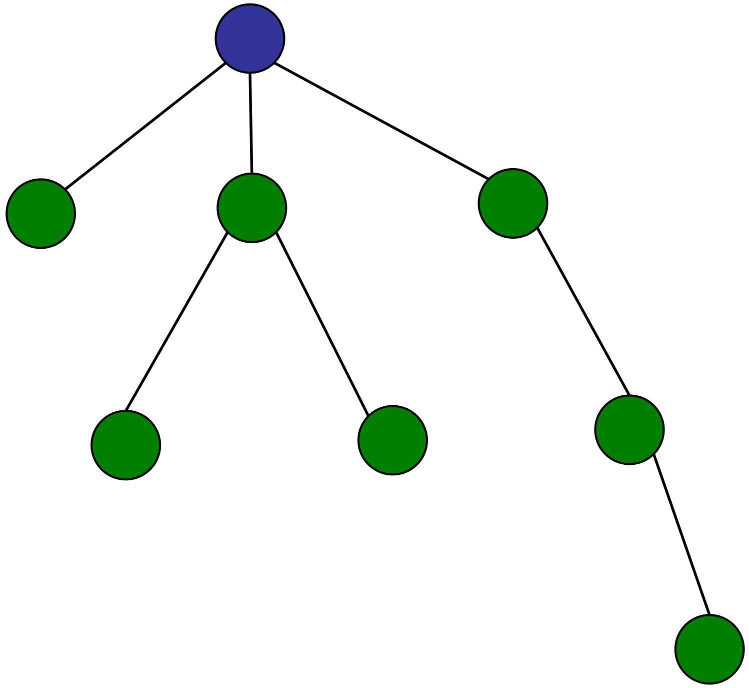


E.g. optimal solution for this tree includes 2 adjacent nodes in the cover.

Vertex Cover on a Tree

How do we calculate S[v, 1]?

We can either cover or uncover v's children (either is fine).



$W_1 = \min(S[w_1, 0], S[w_1, 1])$

$W_2 = \min(S[w_2, 0], S[w_2, 1])$

$W_3 = \min(S[w_3, 0], S[w_3, 1])$

$S[v, 1] = 1 + W_1 + W_2 + W_3 + \ldots$

v.nbrList() = $\{w_1, w_2, w_3, \ldots\}$

```
int treeVertexCover(V) { //Assume tree is ordered from root-to-leaf
    int[][] S = new int[V.length][2]; // create memo table S

    for (int v = V.length - 1; v >= 0; v--) { //From the leaf to the root
        if (v.childList().size() == 0) { // If v is a leaf…
            S[v][0] = 0;
            S[v][1] = 1;
        } else { // Calculate S from v’s children.
            int S[v][0] = 0;
            int S[v][1] = 1;
            for (int w: V[v].childList()) {
                S[v][0] += S[w][1];
                S[v][1] += Math.min(S[w][0], S[w][1]);
            }
        }
    }
    return Math.min(S[0][0], S[0][1]); // returns min at root
}
```

```
int treeVertexCover(V) { //Assume tree is ordered from root-to-leaf
    int[][] S = new int[V.length][2]; // create memo table S

    for (int v = V.length - 1; v >= 0; v--) { //From the leaf to the root
        if (v.childList().size() == 0) { // If v is a leaf…
            S[v][0] = 0;
            S[v][1] = 1;
        } else { // Calculate S from v’s children.
            int S[v][0] = 0;
            int S[v][1] = 1;
            for (int w: V[v].childList()) {
                S[v][0] += S[w][1];
                S[v][1] += Math.min(S[w][0], S[w][1]);
            }
        }
    }
    return Math.min(S[0][0], S[0][1]); // returns min at root
}
```

Store all 2 x V possible solutions to all the possible sub-problems

```
int treeVertexCover(V) { //Assume tree is ordered from root-to-leaf
    int[][] S = new int[V.length][2]; // create memo table S

    for (int v = V.length - 1; v >= 0; v--) { //From the leaf to the root
        if (v.childList().size() == 0) { // If v is a leaf…
            S[v][0] = 0;
            S[v][1] = 1;
        } else { // Calculate S from v’s children.
            int S[v][0] = 0;
            int S[v][1] = 1;
            for (int w: V[v].childList()) {
                S[v][0] += S[w][1];
                S[v][1] += Math.min(S[w][0], S[w][1]);
            }
        }
    }
    return Math.min(S[0][0], S[0][1]); // returns min at root
}
```

Solve for the base cases

```
int treeVertexCover(V) { //Assume tree is ordered from root-to-leaf
    int[][] S = new int[V.length][2]; // create memo table S

    for (int v = V.length - 1; v >= 0; v--) { //From the leaf to the root
        if (v.childList().size() == 0) { // If v is a leaf…
            S[v][0] = 0;
            S[v][1] = 1;
        } else { // Calculate S from v’s children.
            int S[v][0] = 0;
            int S[v][1] = 1;
            for (int w: V[v].childList()) {
                S[v][0] += S[w][1];
                S[v][1] += Math.min(S[w][0], S[w][1]);
            }
        }
    }
    return Math.min(S[0][0], S[0][1]); // returns min at root
}
```

Inductive case

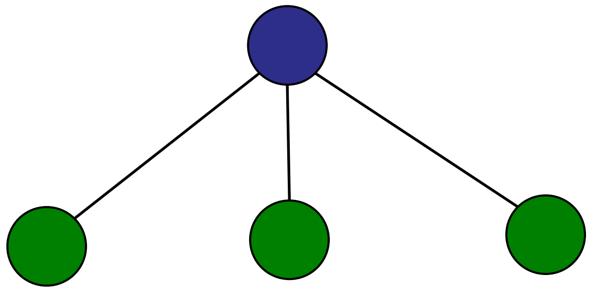
```
int treeVertexCover(V) { //Assume tree is ordered from root-to-leaf
    int[][] S = new int[V.length][2]; // create memo table S

    for (int v = V.length - 1; v >= 0; v--) { //From the leaf to the root
        if (v.childList().size() == 0) { // If v is a leaf…
            S[v][0] = 0;
            S[v][1] = 1;
        } else { // Calculate S from v's children.
            int S[v][0] = 0;
            int S[v][1] = 1;
            for (int w: V[v].childList()) {
                S[v][0] += S[w][1];
                S[v][1] += Math.min(S[w][0], S[w][1]);
            }
        }
    }
    return Math.min(S[0][0], S[0][1]); // returns min at root
}
```

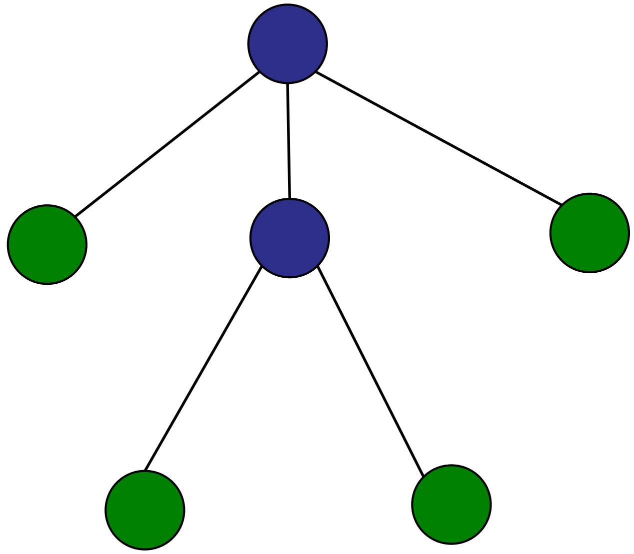
The solution we care about.

Vertex Cover on a Tree

Running time:

- 2V sub-problems
- O(V) time to solve all sub-problems.
  - Each edge explored once.
  - Each sub-problem involves exploring children edges.





Roadmap

Dynamic Programming

- ☑ Basics of DP
- ☑ Example: Longest Increasing Subsequence
- ☑ Example: Bounded Prize Collecting
- ☑ Example: Vertex Cover on a Tree
- – Example: All-Pairs Shortest Paths
- – Example: Knapsack

All Pairs Shortest Path

Input:

- Directed, weighted graph $G = (V,E)$

Goal:

- Preprocess G
- Answer queries: min-distance(v, w)?

Example:

- On-line map service

All Pairs Shortest Path

Input:

- Directed, weighted graph G = (V,E)

Goal:

- Preprocess G
- Answer queries for any pair of nodes v, and w what is min-distance(v, w)?

Example:

- On-line map service

All Pairs Shortest Path

Input:

- Directed, weighted graph G = (V,E)

Goal:

- Preprocess G
- Answer queries for any pair of nodes v, and w what is min-distance(v, w)?

Note:

- When we pre-process G, we don't know what queries we might get.

All Pairs Shortest Path

Simple solution:

- On query (v, w), run SSSP from source node v.

Cost:

- Preprocessing: 0
- Responding to q queries: O(q*E*log V)

All Pairs Shortest Path

Simple solution:

- For every node v, run SSSP, and store its distance to every other node. Total cost: O(VE log V)

Cost:

- Preprocessing: O(VE log V)
- Responding to q queries: O(q) time

What is the running time of running SSSP for every vertex in V on a connected graph with positive weights?

1. $O(VE)$
2. $O(V^2E)$
3. $O(V^2 + E^2)$
4. $O(E \log V)$
5. $O(V^2 \log E)$
6. $O(VE \log V)$

All Pairs Shortest Path

Preprocessing solution:

On preprocessing:

- For all (v,w): calculate distance(v,w)

On query:

- Return precalculated value.

Cost:

- Preprocessing: all-pairs-shortest-paths
- Responding to q queries: O(q)

Diameter of a Graph

Input:

Undirected, weighted graph G=(V, E)

Output:

The longest shortest path possible in the graph.

Diameter of a Graph

Input:

Undirected, weighted graph G=(V, E)

Output:

The longest shortest path possible in the graph.

max across all possible (u, v) { shortest-dist(u, v) }

Diameter of a Graph

Input:

Undirected, weighted graph G=(V, E)

Output:

The longest shortest path possible in the graph.

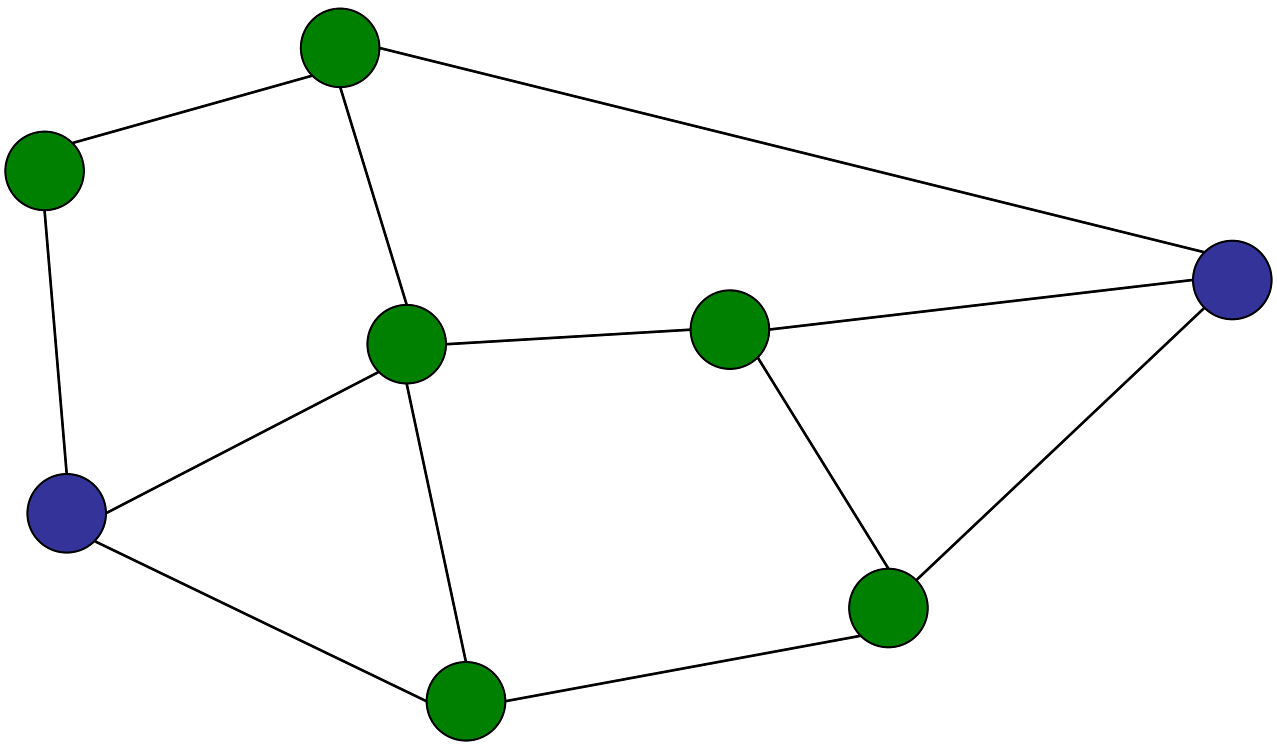
max across all possible (u, v) { shortest-dist(u, v) }

Note: Not the longest path problem!

Diameter of a Graph

Example:

diameter = 3



Diameter of a Graph

Input:

Undirected, weighted graph G=(V, E)

Output:

The longest shortest path possible in the graph.

max across all possible (u, v) { shortest-dist(u, v) }

Note: Not the longest path problem!

All Pairs Shortest Paths

If we knew the shortest distances between any pair of nodes u, v:

Then we can just find the maximum possible shortest distance, and output that!

All Pairs Shortest Paths

Input:

- Weighted, directed graph G = (V,E)

Output:

- dist[v,w] : shortest distance from v to w, for all pairs of vertices (v,w)

All Pairs Shortest Paths

Input:

- Weighted, directed graph G = (V,E)

Output:

- dist[v,w] : shortest distance from v to w, for all pairs of vertices (v,w)

"Straightforward" Solution:

- Run single-source-shortest paths once for every vertex v in the graph.

All Pairs Shortest Paths

Solution:

- Run single-source-shortest paths once for every vertex v in the graph .
- Assume weights are all positive…

Note:

- In a sparse graph where E = O(V): $O(V^2 \log V)$
  - We don't know how to do any better.

What is the running time of running SSSP for every vertex in V on a connected graph with **all identical weights**?

1. $O(VE)$
2. $O(V^2E)$
3. $O(V^2 + E^2)$
4. $O(E \log V)$
5. $O(V^2 \log E)$
6. $O(VE \log V)$

All Pairs Shortest Paths

Solution:

- Run single-source-shortest paths once for every vertex v in the graph .
- Assume weights are all positive…

Note:

- In a sparse graph where $E = O(V)$: $O(V^2 \log V)$
  - We don't know how to do any better.
- Identical weights, use BFS: $O(V(E+V)) = O(VE)$
  - In dense graph: $O(V^3)$
  - In sparse graph: $O(V^2)$

Dynamic Programming Recipe

Step 1: Identify optimal substructure

Step 2: Define sub-problems

Step 3: Solve problem using sub-problems

Step 4: Write (pseudo)code.

Floyd-Warshall

- Dynamic programming:
  - Shortest paths have optimal sub-structure:
    - If P is a shortest path ($u \rightarrow v \rightarrow w$), then P contains a shortest path from ($u \rightarrow v$) and from ($v \rightarrow w$).

Floyd-Warshall

- Dynamic programming:

  Shortest paths have optimal sub-structure:

  If P is a shortest path ($u \rightarrow v \rightarrow w$), then P contains a shortest path from ($u \rightarrow v$) and from ($v \rightarrow w$).

  Shortest paths have overlapping subproblems

  Many shortest path calculations depends on the same sub-pieces.

Floyd-Warshall

- Dynamic programming:

  Shortest paths have optimal sub-structure:

  If P is a shortest path ($u \rightarrow v \rightarrow w$), then P contains a shortest path from ($u \rightarrow v$) and from ($v \rightarrow w$).

  Shortest paths have overlapping subproblems

  Many shortest path calculations depends on the same sub-pieces.

  To solve shortest path, we're solving "smaller" shortest path problems.

Floyd-Warshall

- Dynamic programming:

  Shortest paths have optimal sub-structure:

  If P is a shortest path ($u \rightarrow v \rightarrow w$), then P contains a shortest path from ($u \rightarrow v$) and from ($v \rightarrow w$).

  Shortest paths have overlapping subproblems

  Many shortest path calculations depends on the same sub-pieces.

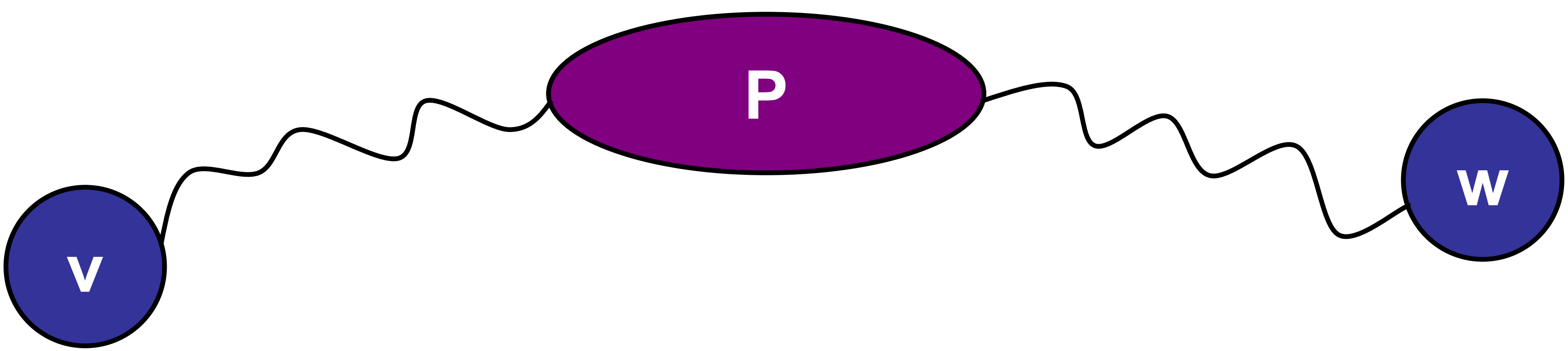
  Hard question: what are the right subproblems?

Floyd-Warshall

Dynamic programming:

Actually, we store distance

Let S[v,w,P] be a shortest path from v to w that only uses intermediate nodes in the set P.

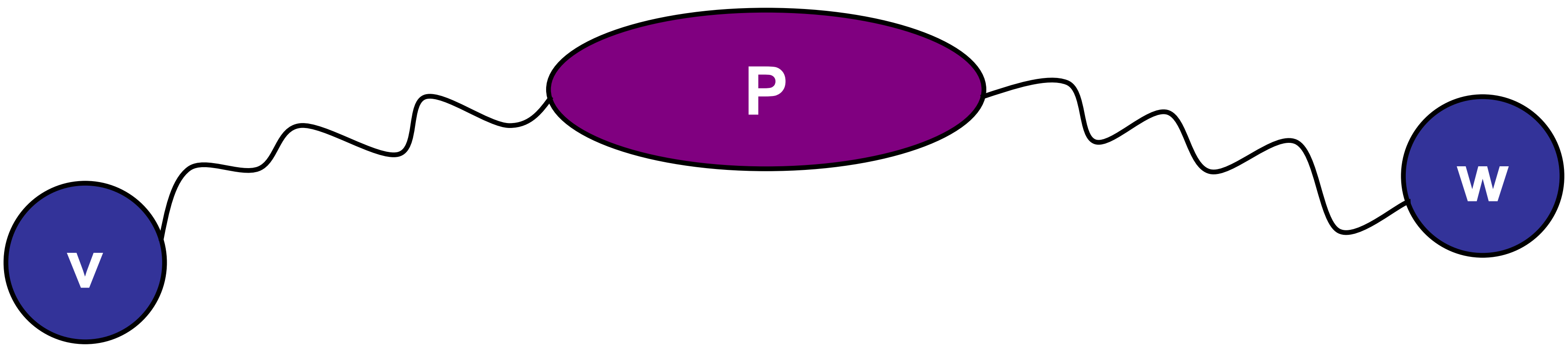


Floyd-Warshall

Dynamic programming:

Let S[v,w,P] be a shortest path from v to w that only uses intermediate nodes in the set P.

We will try to be efficient about how to represent P later.



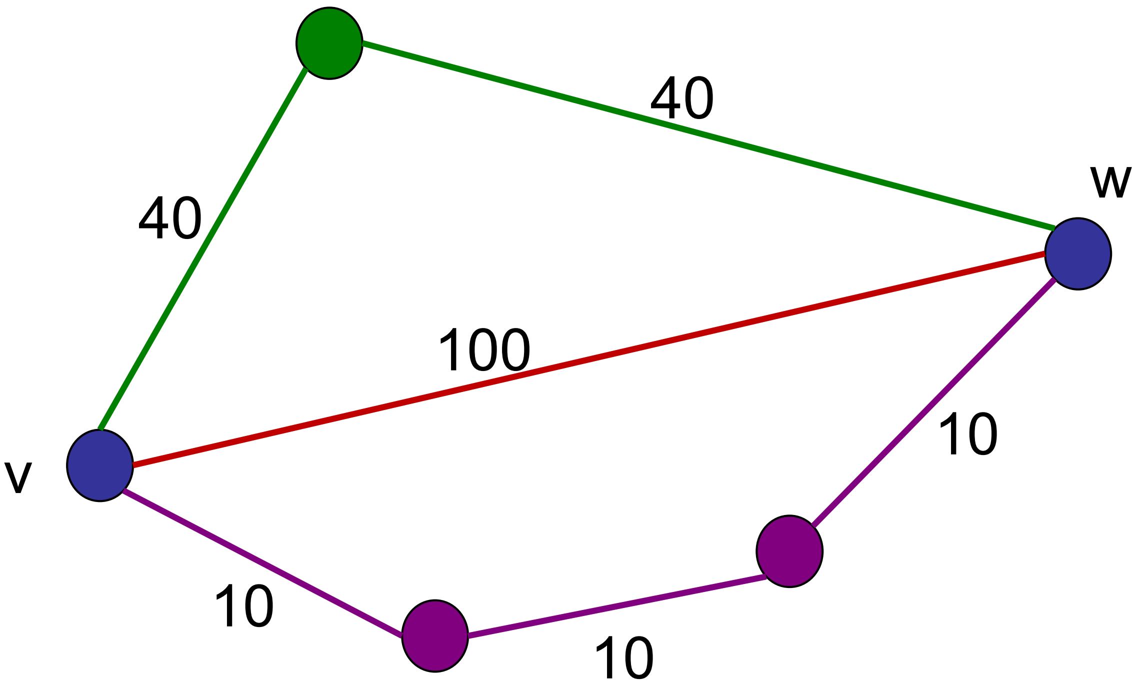
Floyd-Warshall

Let S[v,w,P] be a shortest path from v to w that only uses intermediate nodes only in the set P.

P1 = no nodes (empty set)

P2 = green nodes

P3 = purple nodes



Floyd-Warshall

Let S[v,w,P] be a shortest path from v to w that only uses intermediate nodes only in the set P.

P1 = no nodes (empty set)

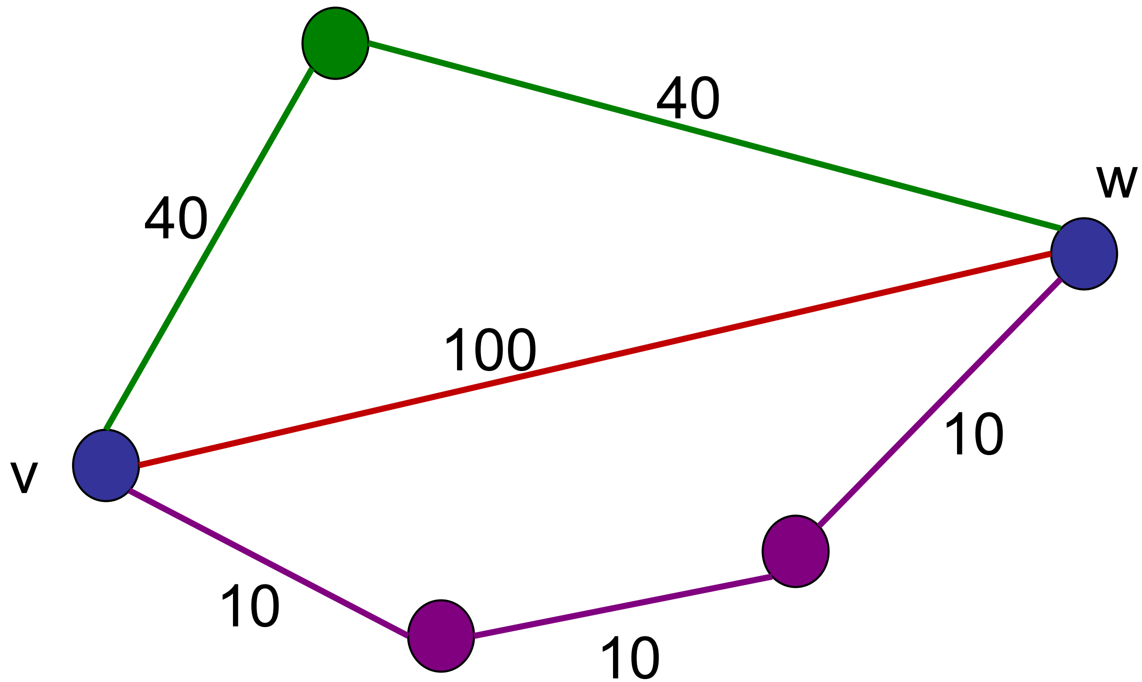
P2 = green nodes

P3 = purple nodes

S(v,w,P1) = 100

S(v,w,P2) = 80

S(v,w,P3) = 30



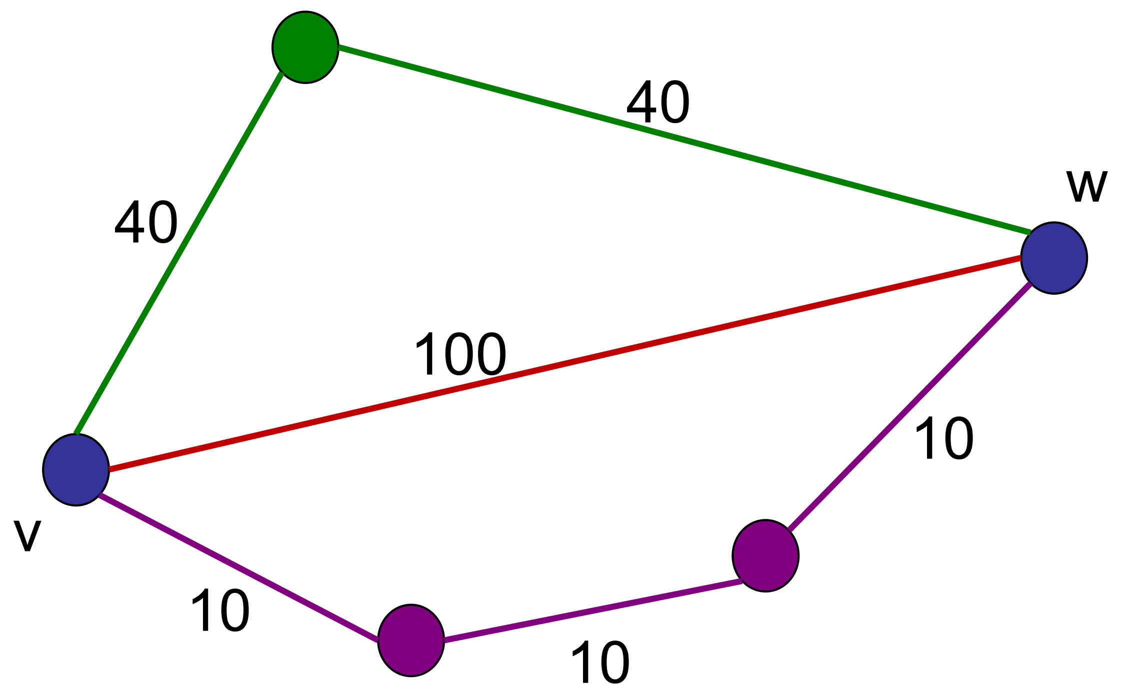
Floyd-Warshall

Let S[v,w,P] be a shortest path from v to w that only uses intermediate nodes only in the set P.

Base case:

S[v, w, Ø] = E[v,w]

E[v,w] = weight of edge from v to w.



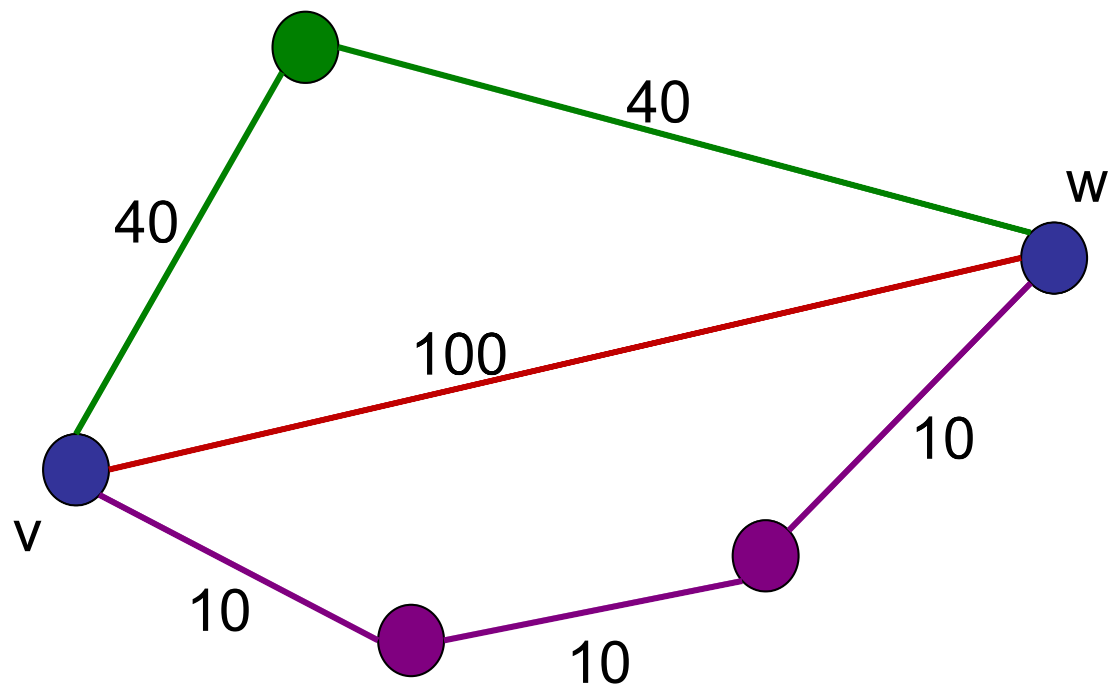
Floyd-Warshall

Let S[v,w,P] be a shortest path from v to w that only uses intermediate nodes only in the set P.

Base case:

S[v, w, Ø] = E[v,w]

Can't take any intermediate nodes so we have to rely on single edges



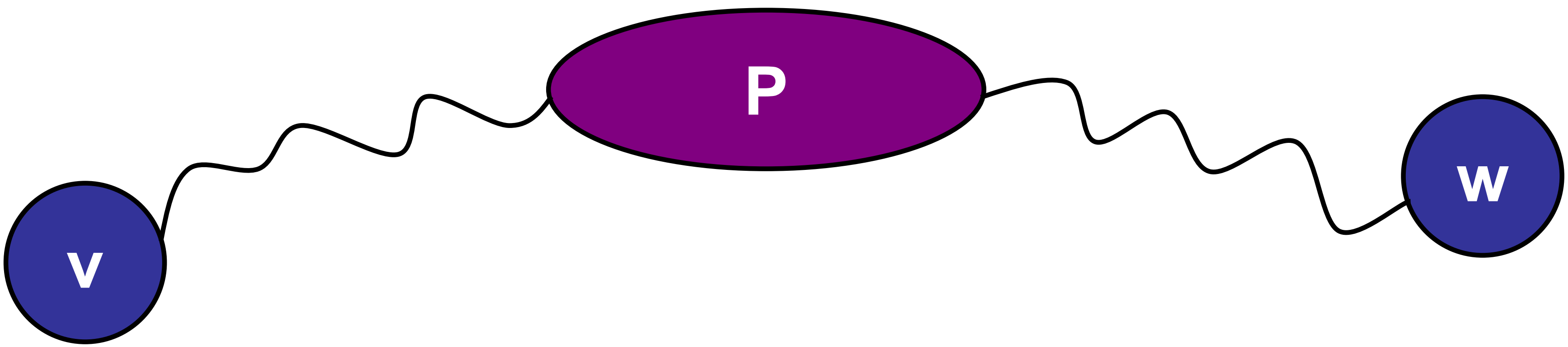
E[v,w] = weight of edge from v to w.

Floyd-Warshall

Dynamic programming:

Let S[v,w,P] be a shortest path from v to w that only uses intermediate nodes in the set P.

How many subproblems do we have?



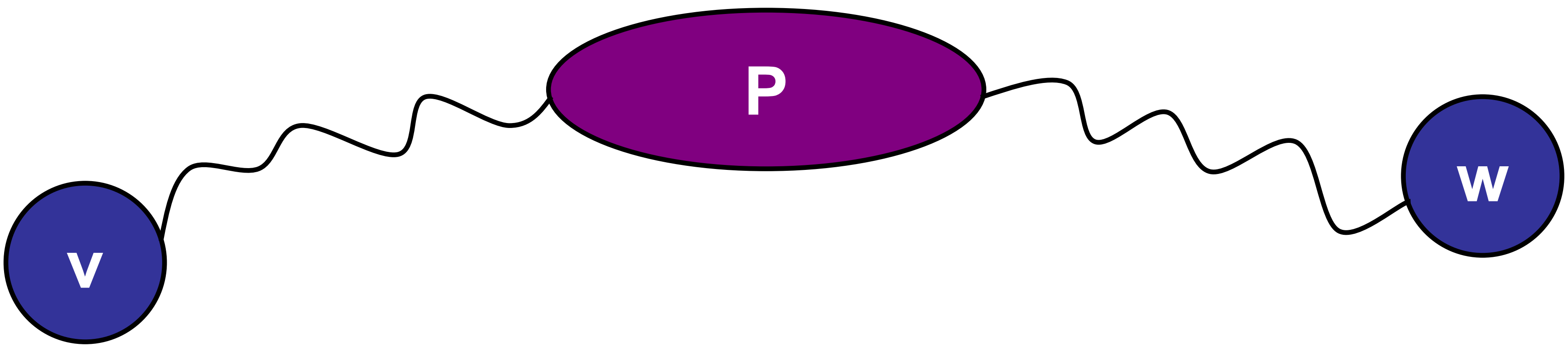
Floyd-Warshall

- Dynamic programming:

  Let S[v,w,P] be a shortest path from v to w that only uses intermediate nodes in the set P.

  Problem: 2^n possible sets P

  → slow to solve *all* $n^2 2^n$ subproblems



Floyd-Warshall

What if we limit ourselves to $n+1$ different sets P:

$P_0 = \emptyset$

$P_1 = \{1\}$

$P_2 = \{1, 2\}$

$P_3 = \{1, 2, 3\}$

$P_4 = \{1, 2, 3, 4\}$

…

$P_n = \{1, 2, 3, 4, \ldots, n\}$

Dynamic Programming Recipe

Step 1: Identify optimal substructure

- Shortest paths are built out of shortest paths.

Step 2: Define sub-problems

- S(u,v,P) = shortest path from u to v using nodes in P.
- Consider only (n+1) sets P of increasing size.

Step 3: Solve problem using sub-problems

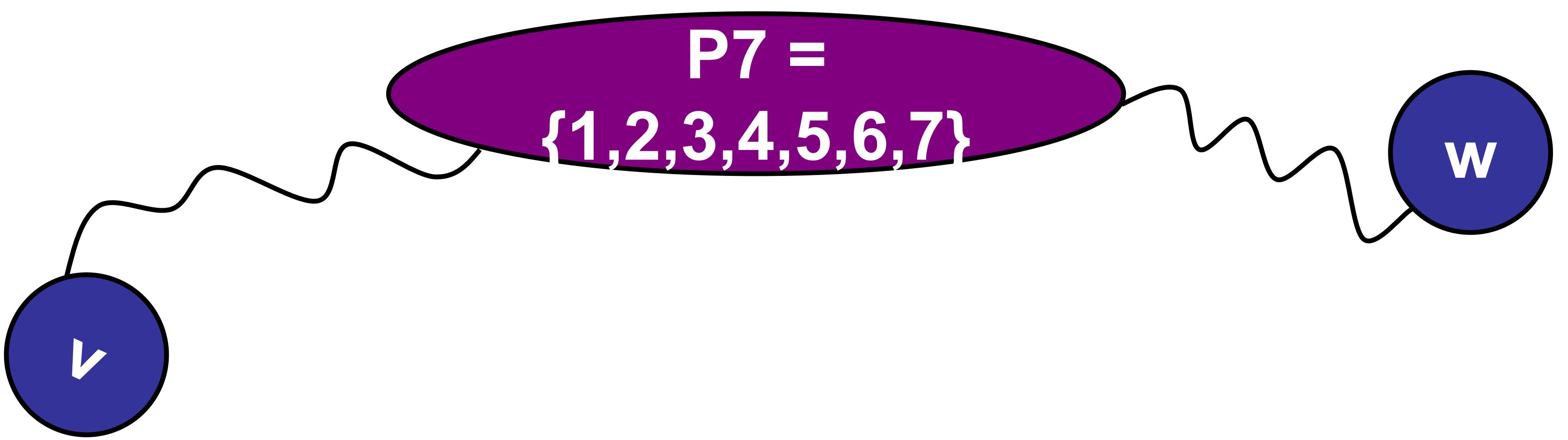
Step 4: Write (pseudo)code.

Floyd-Warshall

Use the precalculated subproblems:

Assume we have calculated $S[v,w,P_7] = 42$.

How do we calculate $S[v,w,P_8]$?



Floyd-Warshall

Use the precalculated subproblems:

Assume we have calculated $S[v,w,P_7] = 42$.

How do we calculate $S[v,w,P_8]$?

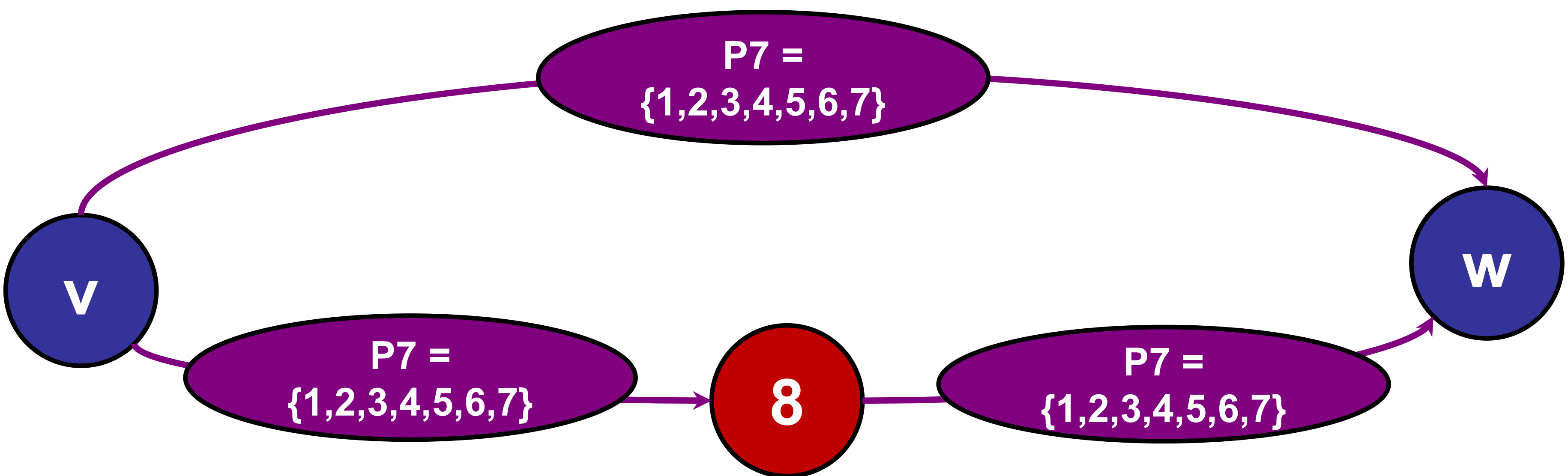
Two possibilities:

1. Shortest path using nodes P_8 includes node 8.
2. Shortest path using nodes P_8 does not include node 8.

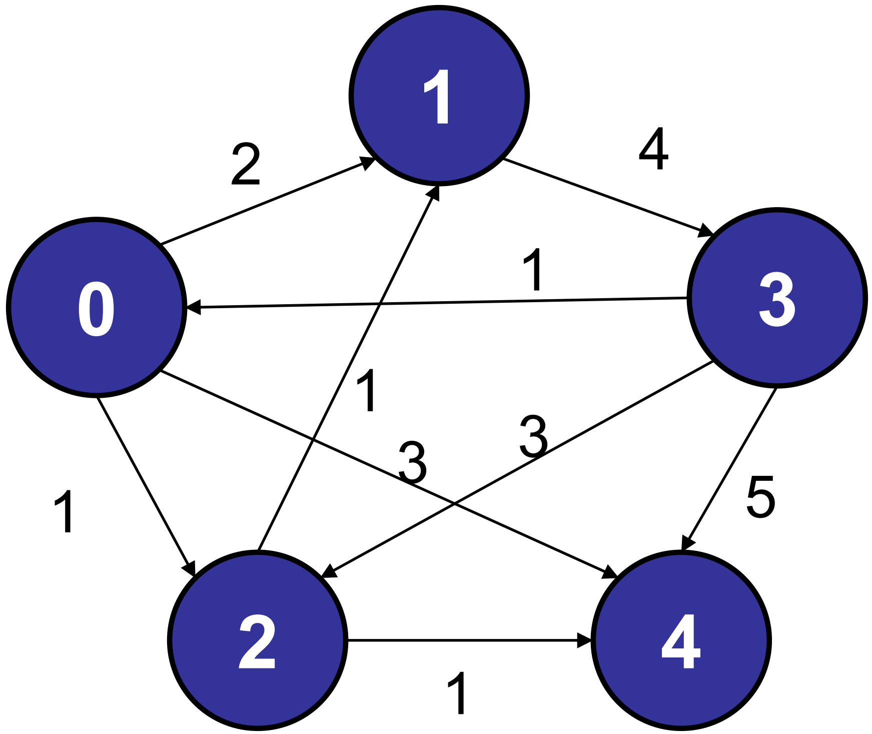
Floyd-Warshall

Use the precalculated subproblems:

$$S[v,w,P_8] = \min(S[v, w, P_7] , S[v, 8, P_7] + S[8, w, P_7])$$

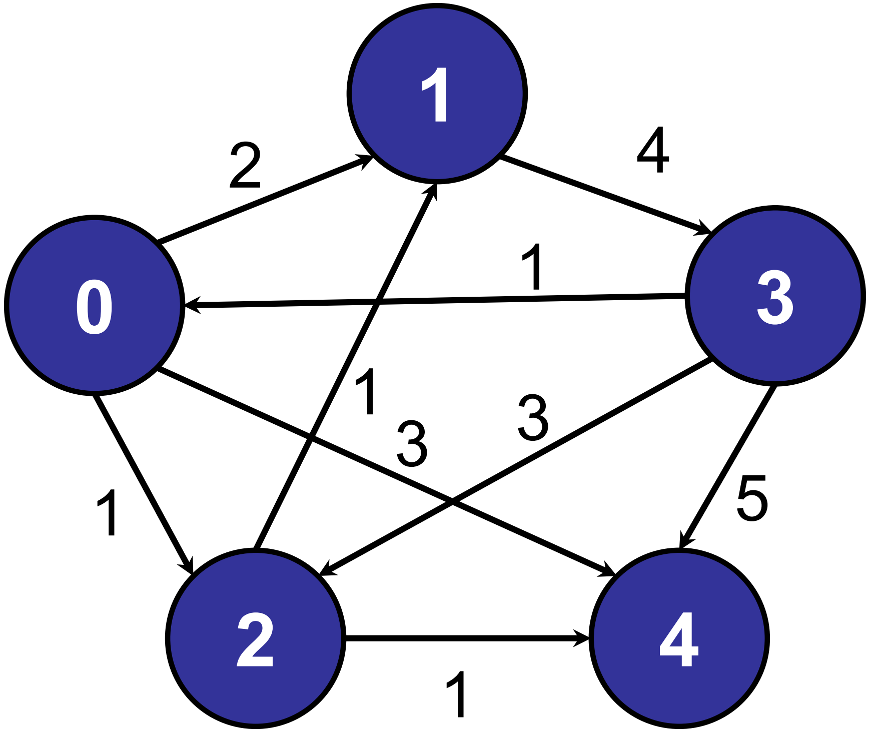


Example:

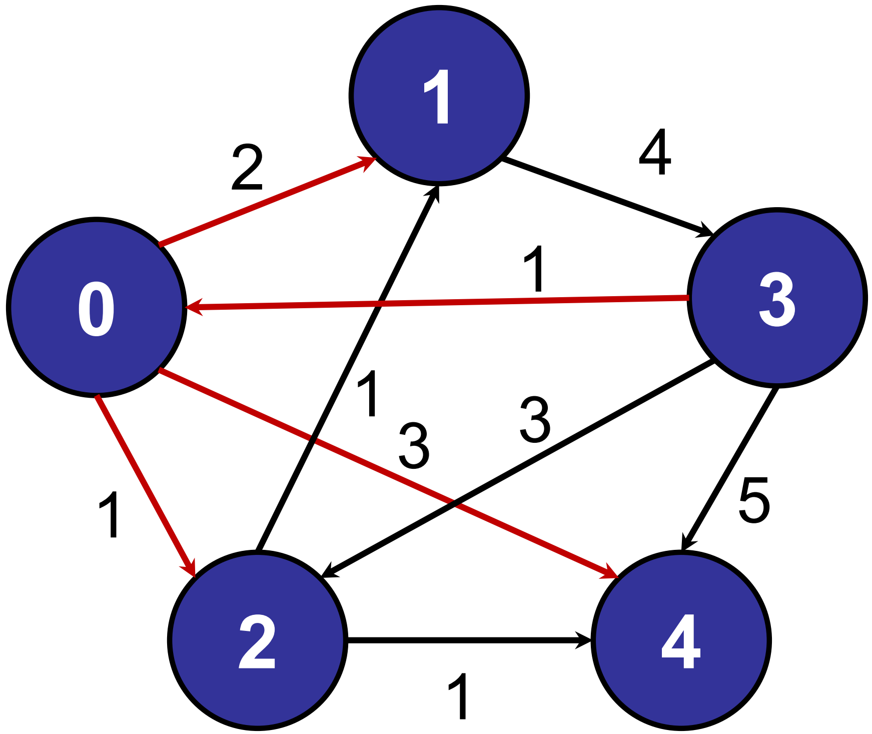


Initially:

	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	5
4	∞	∞	∞	∞	0



Step: P = {0}

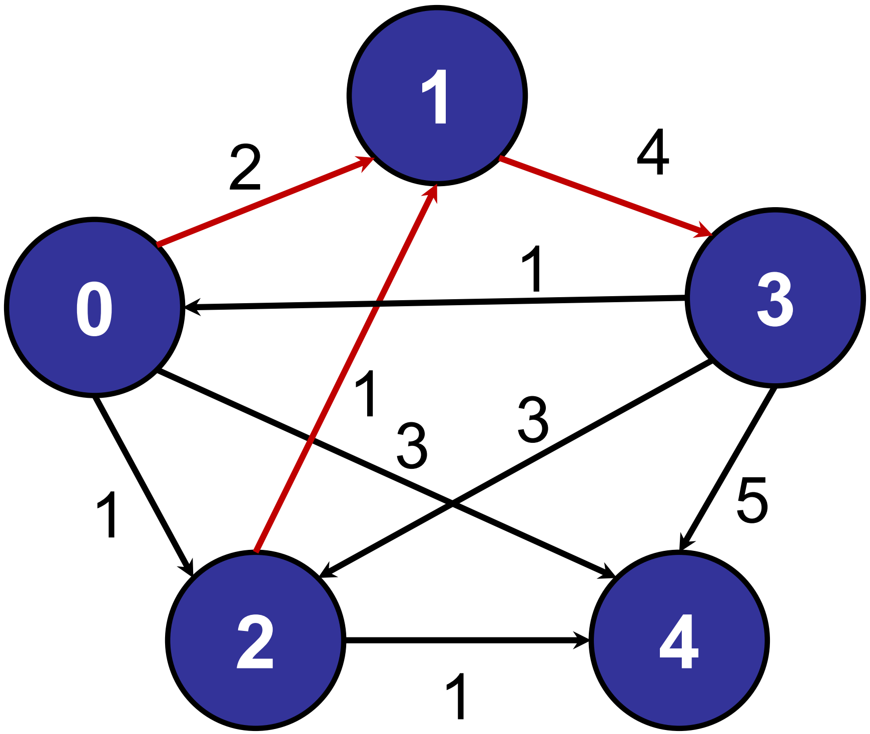


	0	1	2	3	4
0	0	**2**	**1**	∞	**3**
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	**1**	∞	3	0	5
4	∞	∞	∞	∞	0

⇓

	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	**3**	**2**	0	**4**
4	∞	∞	∞	∞	0

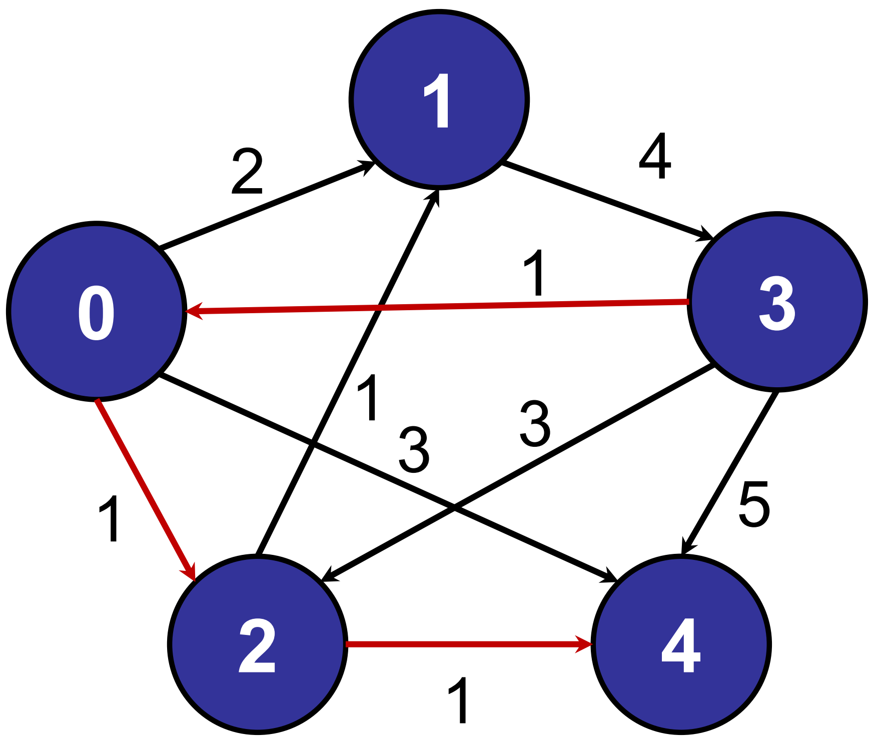
Step: P = {0, 1}



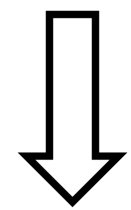
	0	1	2	3	4
0	0	**2**	1	∞	3
1	∞	0	∞	**4**	∞
2	∞	**1**	0	∞	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

	0	1	2	3	4
0	0	2	1	**6**	3
1	∞	0	∞	4	∞
2	∞	1	0	**5**	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

Step: P = {0, 1, 2}

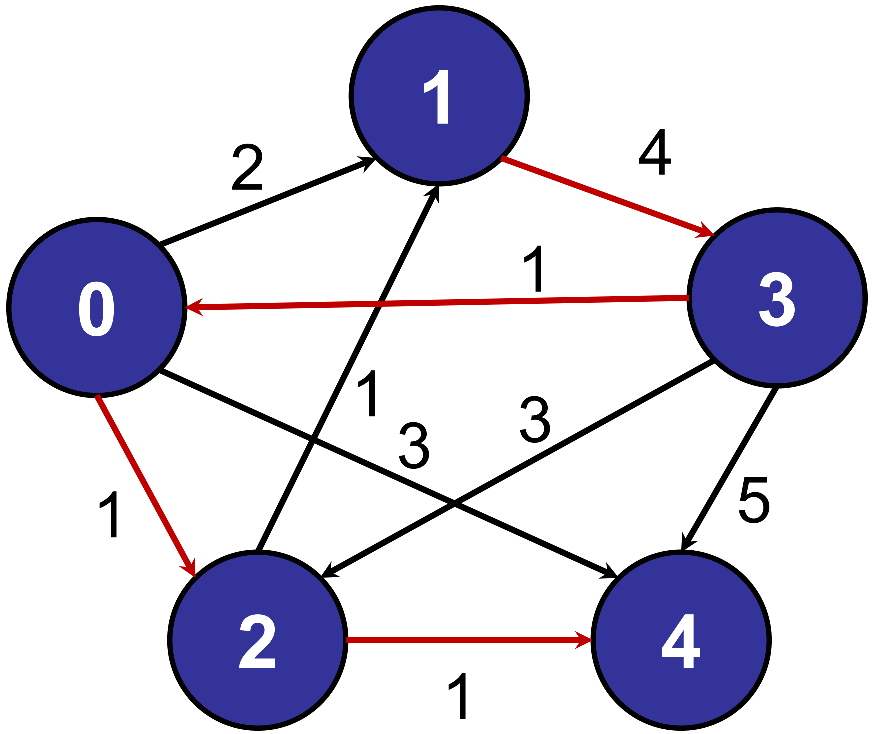


	0	1	2	3	4
0	0	2	**1**	6	3
1	∞	0	∞	4	∞
2	∞	1	0	5	**1**
3	1	3	**2**	0	4
4	∞	∞	∞	∞	0



	0	1	2	3	4
0	0	2	1	6	**2**
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	**3**
4	∞	∞	∞	∞	0

Step: P = {0, 1, 2, 3}

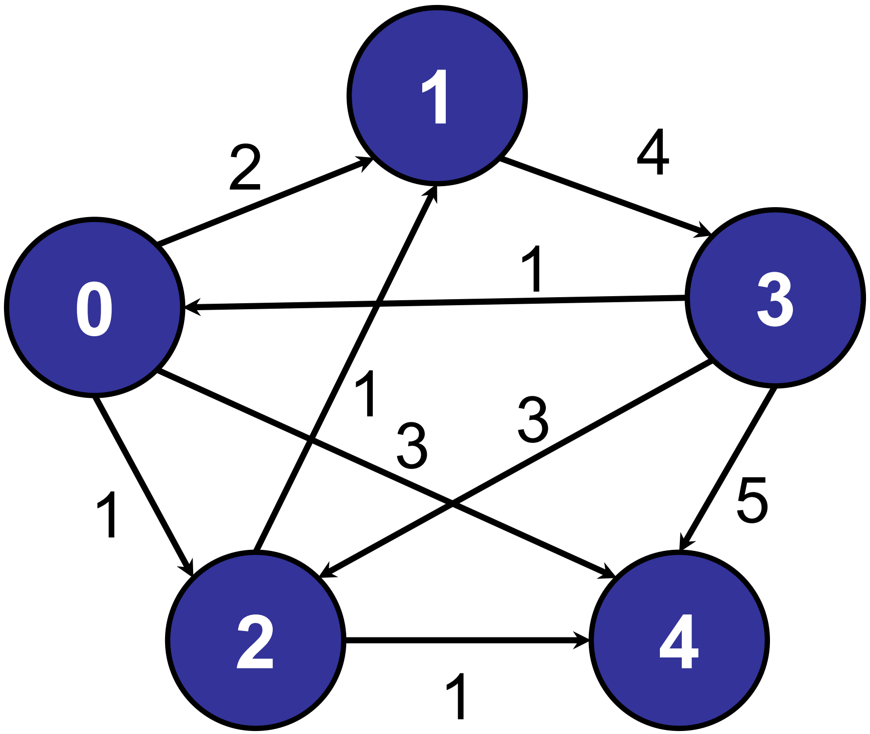


	0	1	2	3	4
0	0	2	1	6	2
1	∞	0	∞	**4**	∞
2	∞	1	0	**5**	1
3	**1**	3	**2**	0	**3**
4	∞	∞	∞	∞	0

⇓

	0	1	2	3	4
0	0	2	1	6	2
1	**5**	0	**6**	4	**7**
2	**6**	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	0

Done: P = {0, 1, 2, 3, 4}

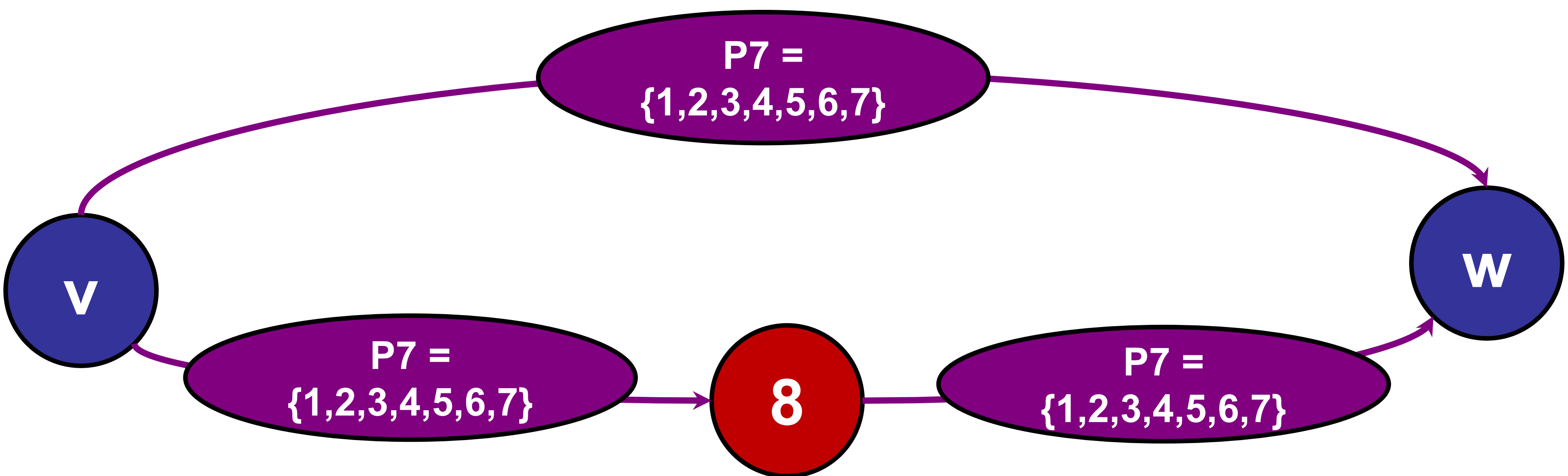


	0	1	2	3	4
0	0	2	1	6	2
1	5	0	6	4	7
2	6	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	0

Floyd-Warshall

Use the precalculated subproblems:

$$S[v,w,P_8] = \min(S[v, w, P_7] , S[v, 8, P_7] + S[8, w, P_7])$$



```
int[][] APSP(E) { // Adjacency matrix E
    int[][] S = new int[V.length][V.length]; //create memo table S

    // Initialize every pair of nodes
    for (int v = 0; v < V.length; v++)
        for (int w = 0; w < V.length; w++)
            S[v][w] = E[v][w];

    // For sets P0, P1, P2, P3, …, for every pair (v,w)
    for (int k = 0; k < V.length; k++)
        for (int v = 0; v < V.length; v++)
            for (int w = 0; w < V.length; w++)
              S[v][w] = min(S[v][w], S[v][k] + S[k][w]);
    return S;
}
```

```
int[][] APSP(E) { // Adjacency matrix E
    int[][] S = new int[V.length][V.length]; //create memo table S

    // Initialize every pair of nodes
    for (int v = 0; v < V.length; v++)
        for (int w = 0; w < V.length; w++)
            S[v][w] = E[v][w];

    // For sets P0, P1, P2, P3, …, for every pair (v,w)
    for (int k = 0; k < V.length; k++)
        for (int v = 0; v < V.length; v++)
            for (int w = 0; w < V.length; w++)
              S[v][w] = min(S[v][w], S[v][k] + S[k][w]);
    return S;
}
```

```
int[][] APSP(E) { // Adjacency matrix E
    int[][] S = new int[V.length][V.length]; //create memo table S

    // Initialize every pair of nodes
    for (int v = 0; v < V.length; v++)
        for (int w = 0; w < V.length; w++)
            S[v][w] = E[v][w];

    // For sets P0, P1, P2, P3, …, for every pair (v,w)
    for (int k = 0; k < V.length; k++)
        for (int v = 0; v < V.length; v++)
            for (int w = 0; w < V.length; w++)
              S[v][w] = min(S[v][w], S[v][k] + S[k][w]);
    return S;
}
```

```
int[][] APSP(E) { // Adjacency matrix E
    int[][] S = new int[V.length][V.length]; //create memo table S

    // Initialize every pair of nodes
    for (int v = 0; v < V.length; v++)
        for (int w = 0; w < V.length; w++)
            S[v][w] = E[v][w];

    // For sets P0, P1, P2, P3, …, for every pair (v,w)
    for (int k = 0; k < V.length; k++)
        for (int v = 0; v < V.length; v++)
            for (int w = 0; w < V.length; w++)
              S[v][w] = min(S[v][w], S[v][k] + S[k][w]);
    return S;
}
```

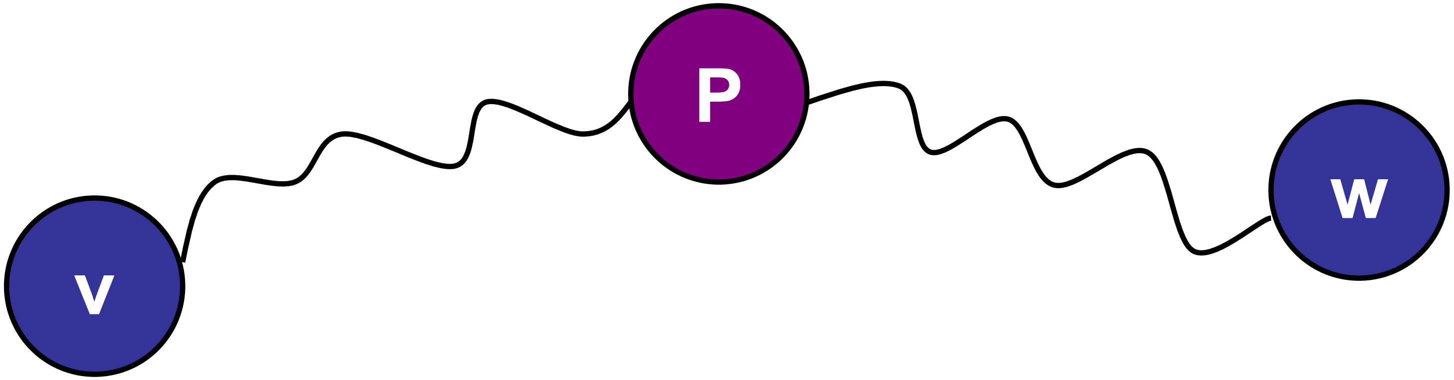
What is the running time of Floyd Warshall?

1. $O(VE)$
2. $O(VE^2)$
3. $O(V^2E)$
4. $O(V^3)$
5. $O(V^3 \log E)$
6. $O(V^4)$

Floyd-Warshall

Dynamic programming:

Let S[v,w,P] be a shortest path from v to w that only uses intermediate nodes only in the set P.



Dynamic Programming Recipe

Step 1: Identify optimal substructure

- Shortest paths are built out of shortest paths.

Step 2: Define sub-problems

- S(u,v,P) = shortest path from u to v using nodes in P.
- Consider only (n+1) sets P of increasing size.

Step 3: Solve problem using sub-problems

- $S(u,v,P_8) = \min(S[v,w,P_7], S[v, 8, P_7] + S[8, w, P_7])$.

Step 4: Write (pseudo)code.

Floyd-Warshall Variants

Path Reconstruction:

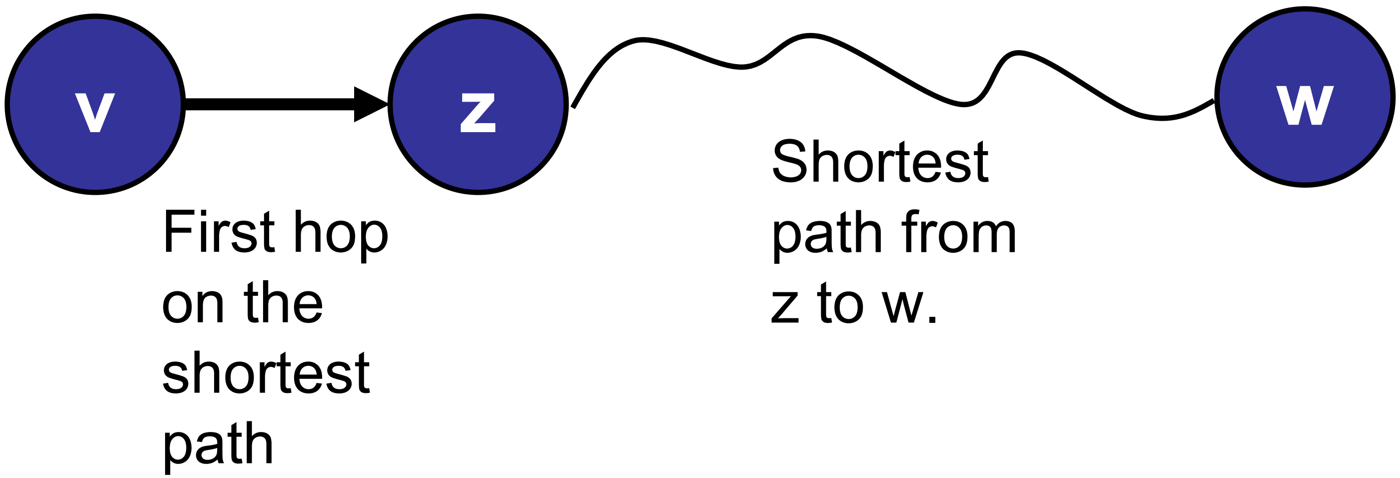
- Return the **actual** path from (v,w).
- Storing *all the shortest paths* requires (potentially) n^3 space!

  (n choose 2) pairs * n hops on the path

- How to represent it succinctly?
- How to store it efficiently?

Floyd-Warshall Variants

- Optimal substructure:

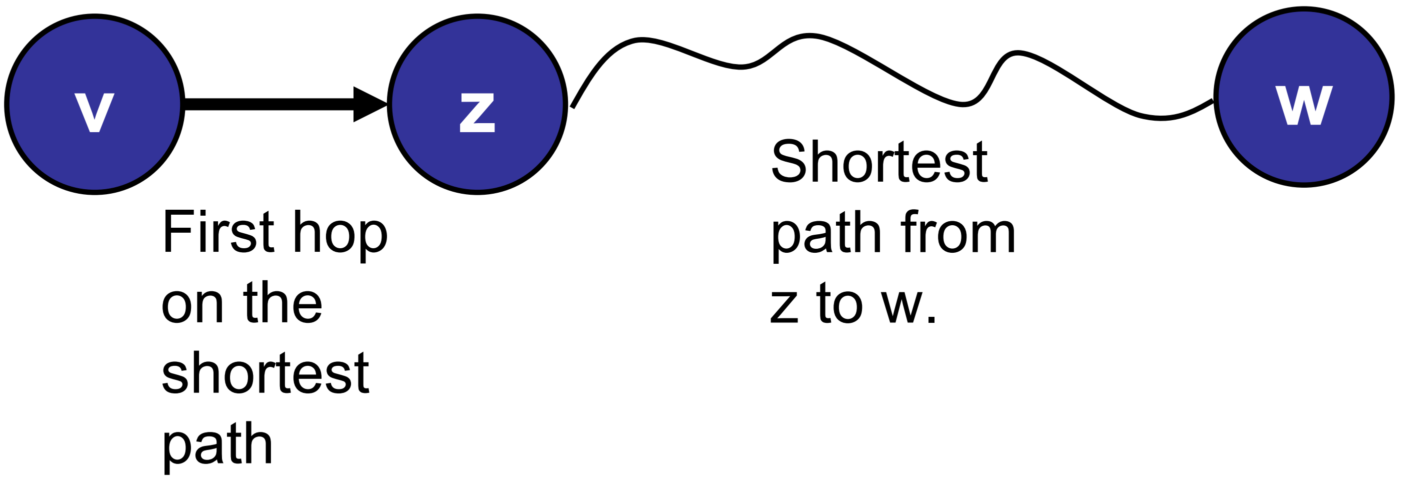


Shortest path from $(v \rightarrow w)$ is:

$(z$ + shortest path $(z \rightarrow w))$.

Floyd-Warshall Variants

- Optimal substructure:



Only store first hop for each destination.

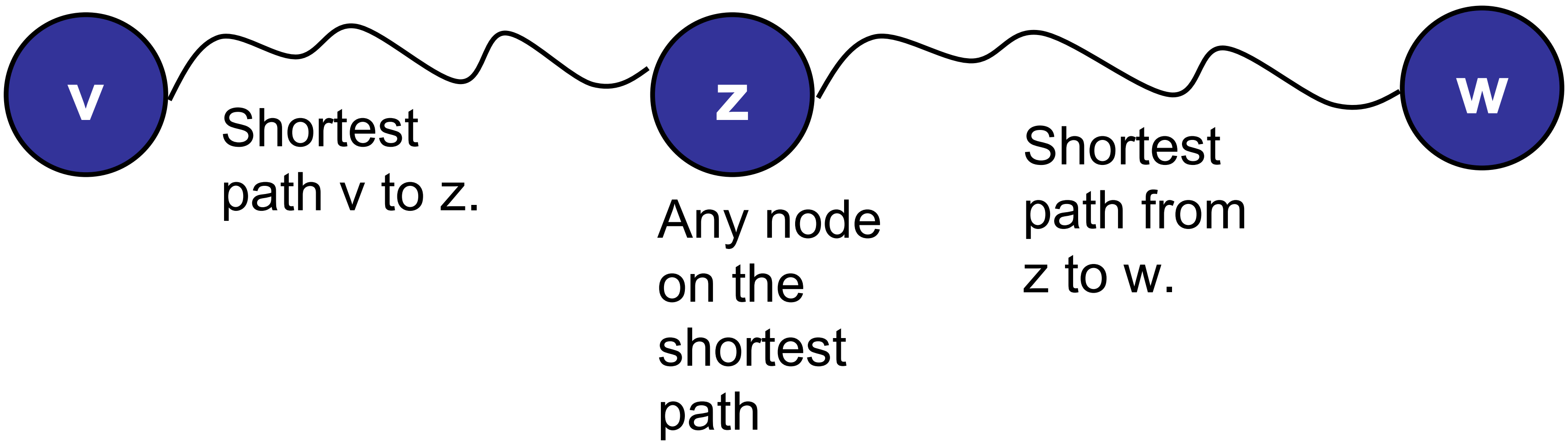
→ routing table!

How much space to store all shortest paths in a routing table?

1. $O(V^2)$
2. $O(VE)$
3. $O(VE^2)$
4. $O(V^2E)$
5. $O(V^3)$
6. $O(V^3 \log E)$

Floyd-Warshall Variants

- Optimal substructure:

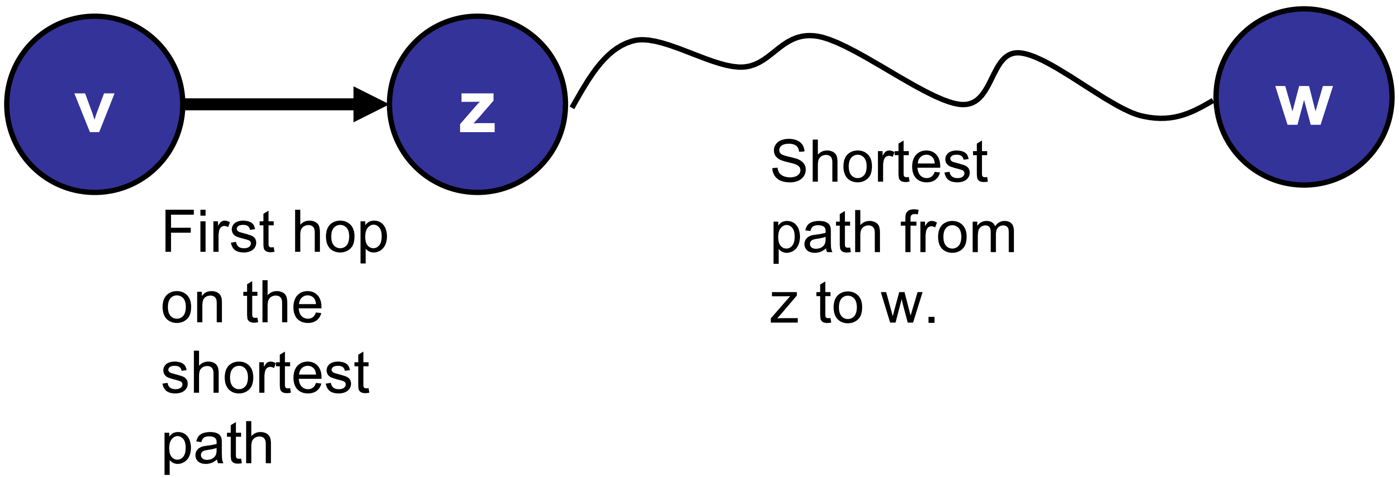


Store some node z on the shortest path from v to w.

Recursively find shortest path from v → z and z → w.

Floyd-Warshall Variants

Optimal substructure:



In Floyd-Warshall, store “intermediate node” whenever you modify/update the matrix entry for a pair.

Floyd-Warshall Variants

Transitive Closure:

Return a matrix M where:

- M[v,w] = 1 if there exists a path from v to w;
- M[v,w] = 0, otherwise.

Floyd-Warshall Variants

Minimum Bottleneck Edge:

- For (v,w), the bottleneck is the heaviest edge on a path between v and w.
- Return a matrix B where:

  B[v,w] = weight of the minimum bottleneck.

Roadmap

Dynamic Programming

- ☑ Basics of DP
- ☑ Example: Longest Increasing Subsequence
- ☑ Example: Bounded Prize Collecting
- ☑ Example: Vertex Cover on a Tree
- ☑ Example: All-Pairs Shortest Paths
- – Example: Knapsack

Knapsack

Given a set of n items, each item has a weight and a value.

weight: 9000
value: 1000

weight: 2
value: 500

weight: 1
value: 600

Knapsack

Given a set of n items, each item has a weight and a value.
Limited knapsack weight limit: C

weight: 9000
value: 1000

weight: 2
value: 500

weight: 1
value: 600

Knapsack

Given a set of n items, each item has a weight and a value.
Limited knapsack weight limit: C

weight: 9000
value: 1000

weight: 2
value: 500

weight: 1
value: 600

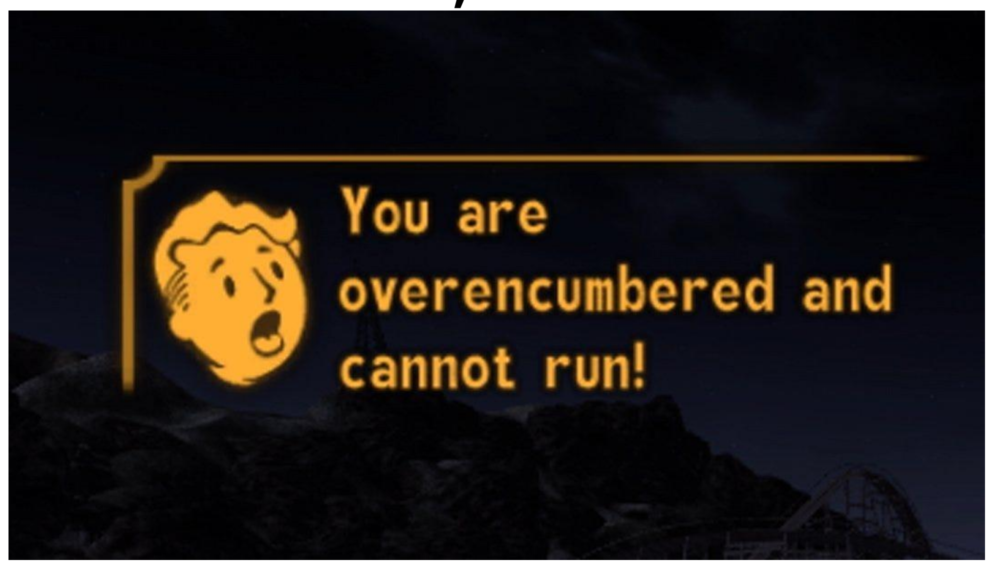
Total value: 1000 + 500

Total weight: 9000 + 2

Knapsack

Given a set of n items, each item has a weight and a value.
Limited knapsack weight limit: C

Goal: Want to maximise value, but cannot exceed limit.



Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

How should we formulate Value(S, L) recursively?

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

How should we formulate Value(S, L) recursively?

Let x = (v, w), be an item with value v and weight w.

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

How should we formulate Value(S, L) recursively?

Let x = (v, w), be an item with value v and weight w.

What happens if x **is part** of the optimal solution?

Value(S, L) = ???

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

How should we formulate Value(S, L) recursively?

Let x = (v, w), be an item with value v and weight w.

What happens if x **is part** of the optimal solution?

Value(S, L) = Value(S \ {x}, L - w) + v

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

How should we formulate Value(S, L) recursively?

Let x = (v, w), be an item with value v and weight w.

What happens if x **is part** of the optimal solution?

$$\text{Value}(S, L) = \text{Value}(S \setminus \{x\}, L - w) + v$$

Since x is part of the optimal solution, to include the item,

recurse on L - w as our new limit, and v to our earned value.

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

How should we formulate Value(S, L) recursively?

Let x = (v, w), be an item with value v and weight w.

What happens if x **is not part of** of the optimal solution?

Value(S, L) = ??

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

How should we formulate Value(S, L) recursively?

Let x = (v, w), be an item with value v and weight w.

What happens if x **is not part of** the optimal solution?

Value(S, L) = Value(S \ {x}, L)

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

How should we formulate Value(S, L) recursively?

Let x = (v, w), be an item with value v and weight w.

What happens if x **is not part part** of the optimal solution?

$Value(S, L) = Value(S \setminus \{x\}, L)$

Since x is not part of the optimal solution, we can ignore it.

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

How should we formulate Value(S, L) recursively?

Let x = (v, w), be an item with value v and weight w.

Since we don’t know whether x is in the solution or not:

Value(S, L) = max(Value(S \ {x}, L), Value(S \ {x}, L - w) + v)

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

Value(S, L) = max(Value(S \ {x}, L), Value(S \ {x}, L - w) + v)

How many states do we have?

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

$$\text{Value}(S, L) = \max(\text{Value}(S \setminus \{x\}, L), \text{Value}(S \setminus \{x\}, L - w) + v)$$

How many states do we have?

$(L+1) \times 2^n$

Knapsack

Step 1: Formulate recurrence.

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

$$\text{Value}(S, L) = \max(\text{Value}(S \setminus \{x\}, L), \text{Value}(S \setminus \{x\}, L - w) + v)$$

How many states do we have?

$(L+1) \times 2^n$: L+1 possible limit values, 2^n possible subsets.

Knapsack

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

Value(S, L) = max(Value(S \ {x}, L), Value(S \ {x}, L - w) + v)

How about we order the set items:

$S = \{s_1, s_2, s_3, \ldots, s_n\}$

Knapsack

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

$$Value(S, L) = \max(Value(S \setminus \{x\}, L), Value(S \setminus \{x\}, L - w) + v)$$

How about we order the set items:

$S = \{s_1, s_2, s_3, \ldots, s_n\}$

$$Value(S, L) = \max(Value(S \setminus \{s_n\}, L), Value(S \setminus \{s_n\}, L - w) + v)$$

Knapsack

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

Value(S, L) = max(Value(S \ {x}, L), Value(S \ {x}, L - w) + v)

How about we order the set items:

$S = \{s_1, s_2, s_3, \ldots, s_n\}$

Value(S, L) = max(Value(S \ {s_n}, L), Value(S \ {s_n}, L - w) + v)

Solving for the first n items

Knapsack

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

Value(S, L) = max(Value(S \ {x}, L), Value(S \ {x}, L - w) + v)

How about we order the set items:

$S = \{s_1, s_2, s_3, \ldots, s_n\}$

Value(S, L) = max(Value(S \ {s_n}, L), Value(S \ {s_n}, L - w) + v)

Solving for the first n items

solving for the first n - 1 items

Knapsack

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

$$Value(S, L) = \max(Value(S \setminus \{x\}, L), Value(S \setminus \{x\}, L - w) + v)$$

How about we order the set items:
$S = \{s_1, s_2, s_3, \ldots, s_n\}$

$$Value(S, L) = \max(Value(S \setminus \{s_n\}, L), Value(S \setminus \{s_n\}, L - w) + v)$$

What are we missing?

Knapsack

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

Value(S, L) = max(Value(S \ {x}, L), Value(S \ {x}, L - w) + v)

How about we order the set items:

$S = \{s_1, s_2, s_3, \ldots, s_n\}$

Value(S, L) = max(Value(S \ $\{s_n\}$, L), Value(S \ $\{s_n\}$, L - w) + v)

What are we missing?

What is Value(S, 0)?

Knapsack

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

Value(S, L) = max(Value(S \ {x}, L), Value(S \ {x}, L - w) + v)

How about we order the set items:

$S = \{s_1, s_2, s_3, \ldots, s_n\}$

Value(S, L) = max(Value(S \ {s_n}, L), Value(S \ {s_n}, L - w) + v)

What are we missing?

What is Value(S, 0)? We should return 0.

Knapsack

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

Value(S, L) = max(Value(S \ {x}, L), Value(S \ {x}, L - w) + v)

How about we order the set items:

$S = \{s_1, s_2, s_3, \ldots, s_n\}$

Value(S, L) = max(Value(S \ {s_n}, L), Value(S \ {s_n}, L - w) + v)

What else are we missing?

Knapsack

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

Value(S, L) = max(Value(S \ {x}, L), Value(S \ {x}, L - w) + v)

How about we order the set items:

$S = \{s_1, s_2, s_3, \ldots, s_n\}$

Value(S, L) = max(Value(S \ $\{s_n\}$, L), Value(S \ $\{s_n\}$, L - w) + v)

What else are we missing?

What happens if the weight of s_n is larger than L?

Knapsack

Value(S, L): Outputs the maximum attainable value using items from set S, subject to not exceeding limit L.

Value(S, L) = max(Value(S \ {x}, L), Value(S \ {x}, L - w) + v)

How about we order the set items:

$S = \{s_1, s_2, s_3, \ldots, s_n\}$

Value(S, L) = max(Value(S \ {s_n}, L), Value(S \ {s_n}, L - w) + v)

What happens if the weight of s_n is larger than L?

Should not recurse on Value(S \ {s_n}, L - w)!

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)							
s2 = (10, 1)							
s3 = (8, 6)							
s4 = (1, 2)							

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)							
s2 = (10, 1)							
s3 = (8, 6)							
s4 = (1, 2)							

L+1 possible columns

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)							
s2 = (10, 1)							
s3 = (8, 6)							
s4 = (1, 2)							

n possible rows

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)							
s2 = (10, 1)							
s3 = (8, 6)							
s4 = (1, 2)							

Want to store at each table element: the optimal solution.

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)							
s2 = (10, 1)							
s3 = (8, 6)							
s4 = (1, 2)							

E.g. 3rd row, 5th column represents:

Value({s1, s2, s3}, 5) - optimal solution using first 3 items, with limit 5.

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)							
s2 = (10, 1)							
s3 = (8, 6)							
s4 = (1, 2)							

What should we do on the first row?

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)							
s2 = (10, 1)							
s3 = (8, 6)							
s4 = (1, 2)							

What should we do on the first row?

That's the base case.

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)							
s2 = (10, 1)							
s3 = (8, 6)							
s4 = (1, 2)							

E.g. first item has a weight of 3.

How should we fill out our first row?

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	0	0	0	0

1. Set everything to 5
2. Set everything to 0
3. Set columns 0, 1, 2 to 0
   Set columns 3, 4, 5, 6 to 5

How should we fill out our first row?

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	0	0	0	0

1. Set everything to 5

2. Set everything to 0

3. Set columns 0, 1, 2 to 0
   Set columns 3, 4, 5, 6 to 5

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	**0**	**0**	**0**	**5**	**5**	**5**	**5**
s2 = (10, 1)							
s3 = (8, 6)							
s4 = (1, 2)							

What about the 0th column?

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0						
s3 = (8, 6)	0						
s4 = (1, 2)	0						

What about the 0th column?

Base case also, set it to 0

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0						
s3 = (8, 6)	0						
s4 = (1, 2)	0						

What about 2nd row, 1st column?

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0						
s3 = (8, 6)	0						
s4 = (1, 2)	0						

What about 2nd row, 1st column?

either don't take the current item,

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0						
s3 = (8, 6)	0						
s4 = (1, 2)	0						

+10

What about 2nd row, 1st column?

either don't take the current item,

or take current item, add 10 to it

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10					
s3 = (8, 6)	0						
s4 = (1, 2)	0						

+10

What about 2nd row, 1st column?

either don't take the current item,

or take current item, add 10 to it. Take max.

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10					
s3 = (8, 6)	0						
s4 = (1, 2)	0						

What about 2nd row, 1st column?

either don't take the current item,

or take current item, add 10 to it. Take max.

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10					
s3 = (8, 6)	0						
s4 = (1, 2)	0						

The column we check depends on the weight.

E.g. weight 1, check 1 column to the left.

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10					
s3 = (8, 6)	0						
s4 = (1, 2)	0						

The column we check depends on the weight.

E.g. weight 1, check 1 column to the left.

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	**0**	**0**	**0**	**5**	**5**	**5**	**5**
s2 = (10, 1)	**0**	**10**	+10				
s3 = (8, 6)	**0**						
s4 = (1, 2)	**0**						

Similarly, 2nd row, 2nd column is max of:

1. 1st row, 2nd column
2. 1st row, 1st column + 10

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10			
s3 = (8, 6)	0						
s4 = (1, 2)	0						

Notice:
With limit 4, we can take both item 1 and item 2.

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10	15	15	15
s3 = (8, 6)	0						
s4 = (1, 2)	0						

Notice:
With limit 4, we can take both item 1 and item 2.

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10	15	15	15
s3 = (8, 6)	0						
s4 = (1, 2)	0						

What about third row, first column?
(Item has value 8, weight 6)

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10	15	15	15
s3 = (8, 6)	0						
s4 = (1, 2)	0						

What about third row, first column?
(Item has value 8, weight 6)

Item is too heavy! We cannot take the item!

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10	15	15	15
s3 = (8, 6)	0	10					
s4 = (1, 2)	0						

What about third row, first column?
(Item has value 8, weight 6)

Item is too heavy! We cannot take the item!

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10	15	15	15
s3 = (8, 6)	0	10	10	10	15	15	
s4 = (1, 2)	0						

What about third row, first column?
(Item has value 8, weight 6)

Same for all columns before 6

How should we fill out this cell?

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10	15	15	15
s3 = (8, 6)	0	10	10	10	15	15	
s4 = (1, 2)	0						

1. 15
2. 8
3. 23
4. 18

How should we fill out this cell?

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10	15	15	15
s3 = (8, 6)	0	10	10	10	15	15	15
s4 = (1, 2)	0						

1. 15
2. 8
3. 23
4. 18

Knapsack

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10	15	15	15
s3 = (8, 6)	0	10	10	10	15	15	15
s4 = (1, 2)	0	10	10	11	15	15	16

Filling out the last row:

Knapsack Analysis:

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10	15	15	15
s3 = (8, 6)	0	10	10	10	15	15	15
s4 = (1, 2)	0	10	10	11	15	15	16

O(n L) subproblems, each sub-problem takes O(1) to compute.

Knapsack Analysis:

(value, weight)	0	1	2	3	4	5	6
s1 = (5, 3)	0	0	0	5	5	5	5
s2 = (10, 1)	0	10	10	10	15	15	15
s3 = (8, 6)	0	10	10	10	15	15	15
s4 = (1, 2)	0	10	10	11	15	15	16

O(n L) subproblems, each sub-problem takes O(1) to compute.

O(nL) time!

Roadmap

Dynamic Programming

- ☑ Basics of DP
- ☑ Example: Longest Increasing Subsequence
- ☑ Example: Bounded Prize Collecting
- ☑ Example: Vertex Cover on a Tree
- ☑ Example: All-Pairs Shortest Paths
- ☑ Example: Knapsack