Announcements

- Homework 5 online due Friday
- Exam 2 scores out

Last Time

- Dynamic Programming

Dynamic Programming

Our final general algorithmic technique:

1. Break problem into smaller subproblems.
2. Find recursive formula solving one subproblem in terms of simpler ones.
3. Tabulate answers and solve all subproblems.

Notes about DP

- General Correct Proof Outline:
  - Prove by induction that each table entry is filled out correctly
  - Use base-case and recursion
- Runtime of DP:
  - Usually [Number of subproblems]x[Time per subproblem]

More Notes about DP

- Finding Recursion
  - Often look at first or last choice and see what things look like without that choice
- Key point: Picking right subproblem
  - Enough information stored to allow recursion
  - Not too many

Today

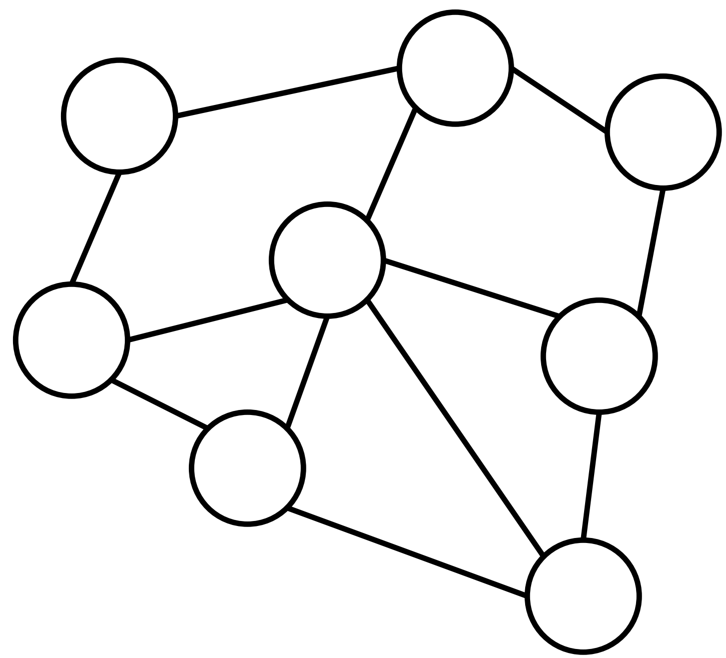
- Maximum Independent Set in Trees
- Travelling Salesman

Independent Set

Definition: In an undirected graph G, an independent set is a subset of the vertices of G, no two of which are connected by an edge.

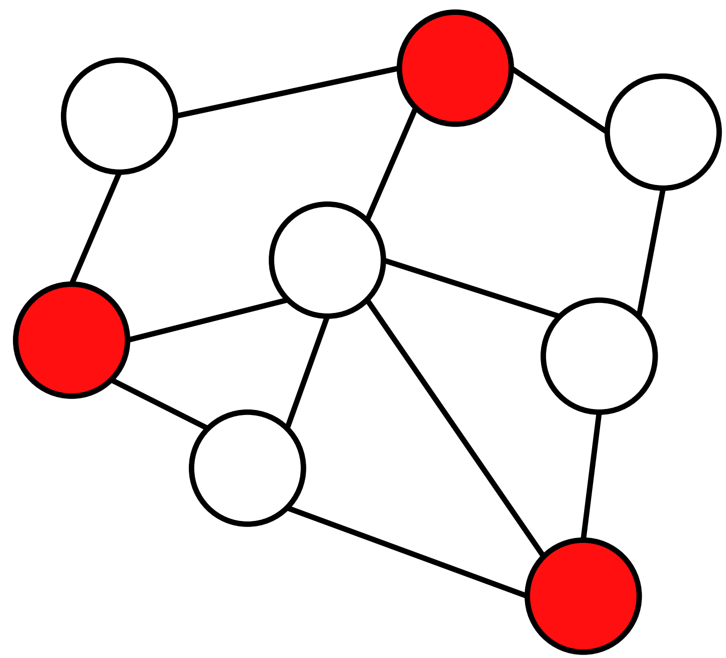
Independent Set

Definition: In an undirected graph G, an independent set is a subset of the vertices of G, no two of which are connected by an edge.



Independent Set

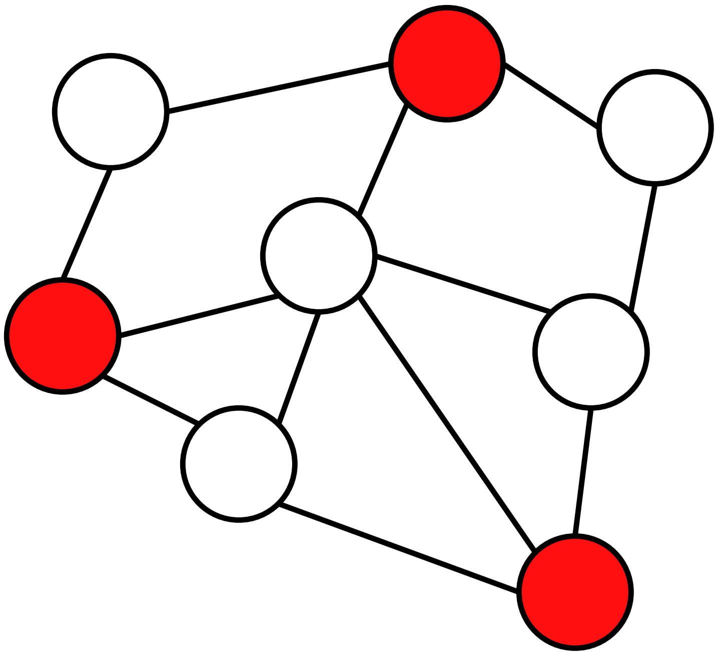
Definition: In an undirected graph G, an independent set is a subset of the vertices of G, no two of which are connected by an edge.



Independent Set

Definition: In an undirected graph G, an independent set is a subset of the vertices of G, no two of which are connected by an edge.

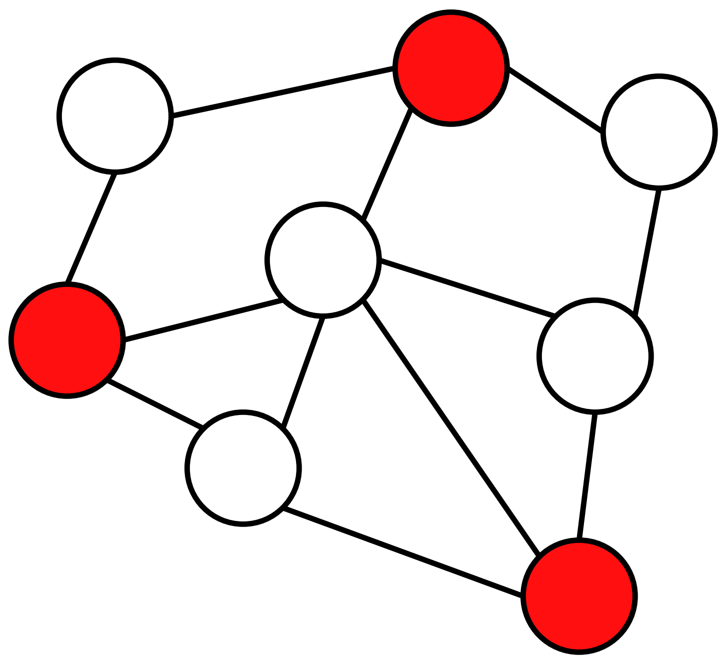
Problem: Given a graph G compute the largest possible size of an independent set of G.



Independent Set

<u>Definition:</u> In an undirected graph G, an <u>independent set</u> is a subset of the vertices of G, no two of which are connected by an edge.

<u>Problem:</u> Given a graph G compute the largest possible <u>size</u> of an independent set of G.



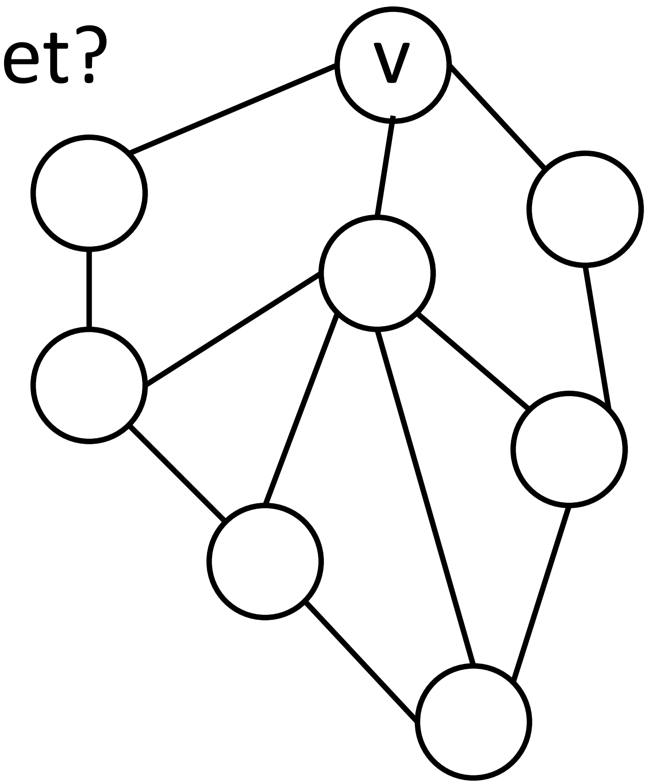
Call answer I(G).

Simple Recursion

Is vertex v in the independent set?

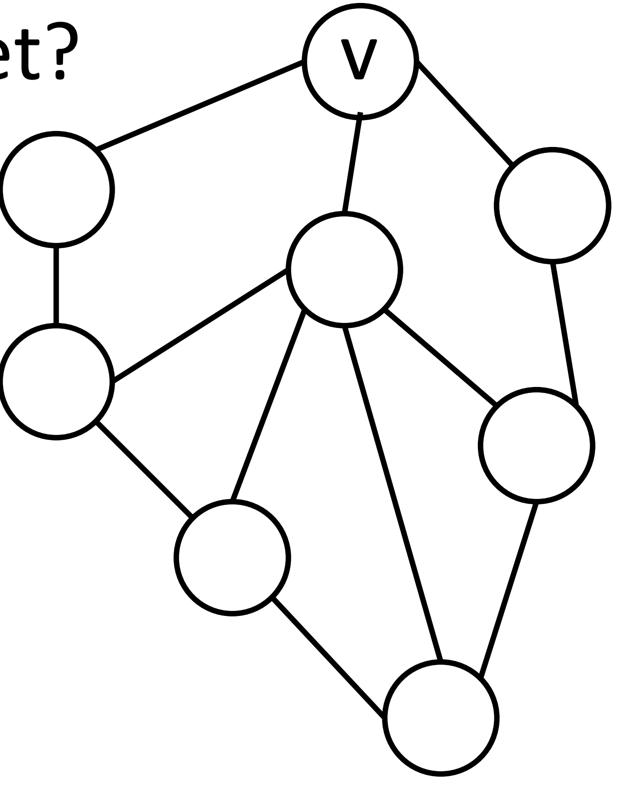
Simple Recursion

Is vertex v in the independent set?



Simple Recursion

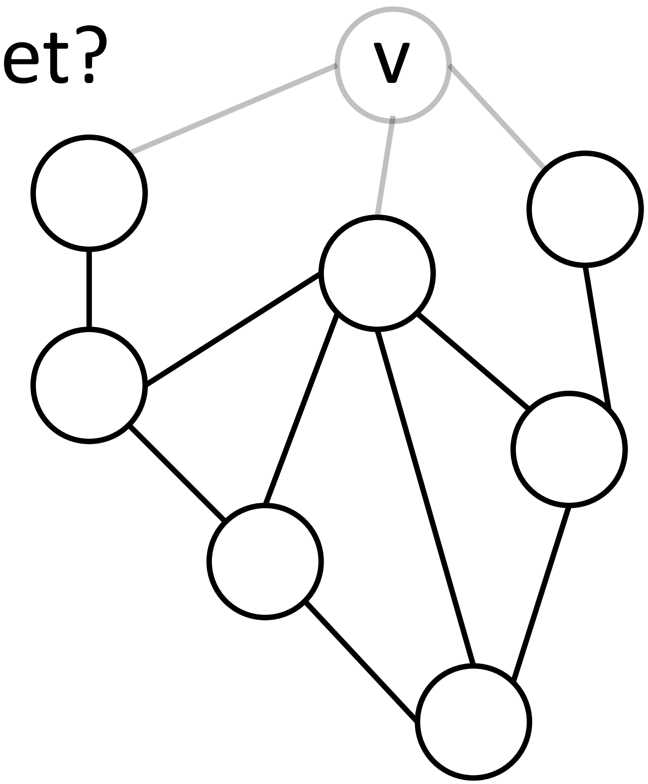
Is vertex v in the independent set?



<u>If not:</u> Maximum independent set is an independent set of G-v.
I(G) = I(G-v).

Simple Recursion

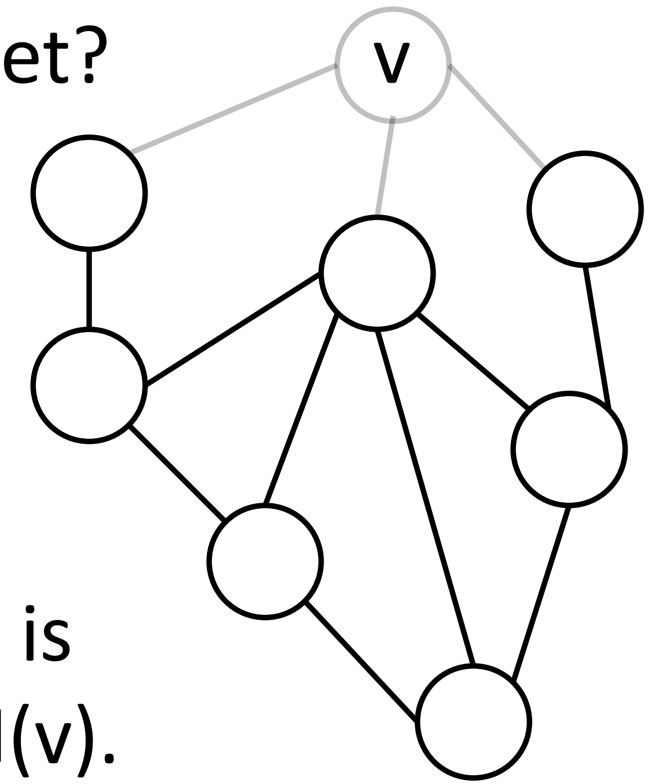
Is vertex v in the independent set?



If not: Maximum independent set is an independent set of G-v.
I(G) = I(G-v).

Simple Recursion

Is vertex v in the independent set?

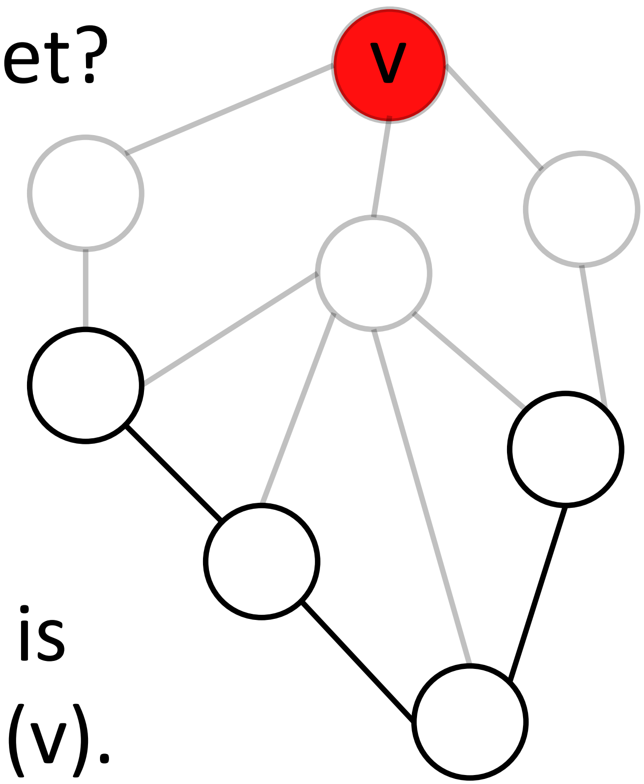


If not: Maximum independent set is an independent set of G-v.
I(G) = I(G-v).

If so: Maximum independent set is v plus an independent set of G-N(v).
I(G) = 1+I(G-N(v)).

Simple Recursion

Is vertex v in the independent set?

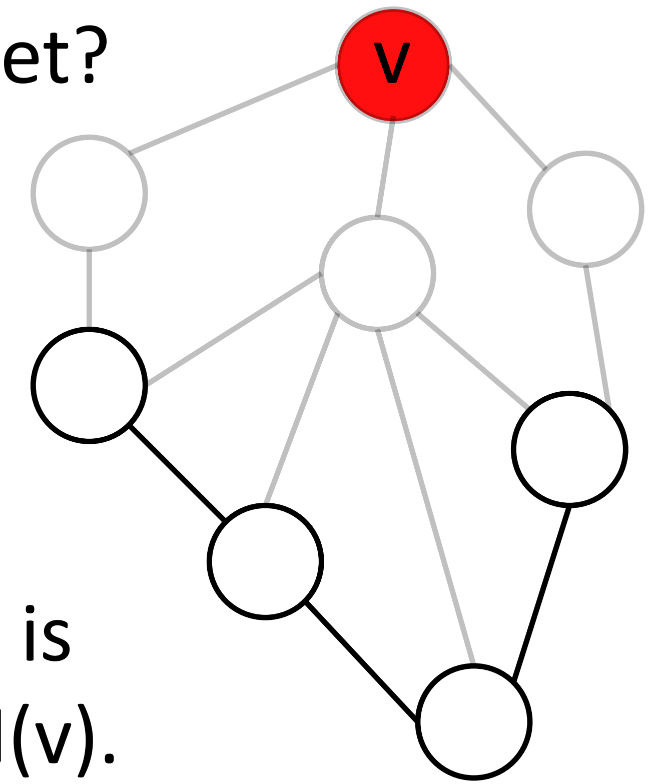


If not: Maximum independent set is an independent set of G-v.
I(G) = I(G-v).

If so: Maximum independent set is v plus an independent set of G-N(v).
I(G) = 1+I(G-N(v)).

Simple Recursion

Is vertex v in the independent set?



<u>If not:</u> Maximum independent set is an independent set of G-v.
$I(G) = I(G\text{-}v)$.

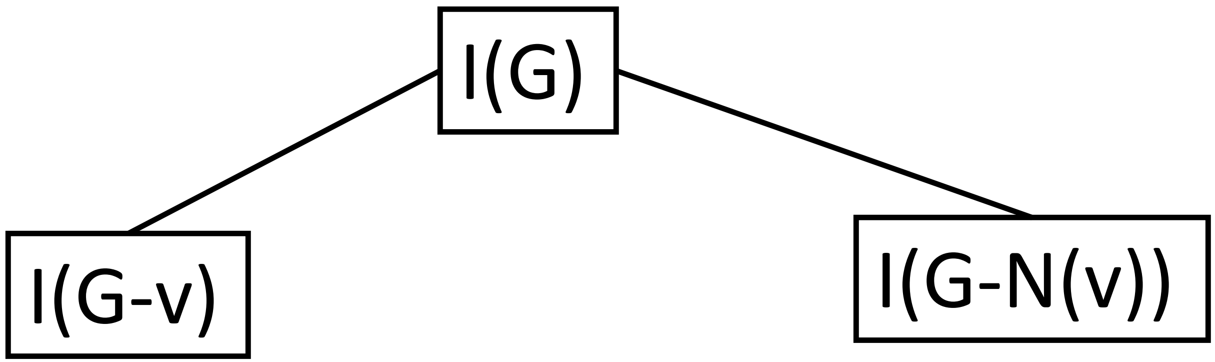
<u>If so:</u> Maximum independent set is v plus an independent set of G-N(v).
$I(G) = 1+I(G\text{-}N(v))$.

<u>Recursion:</u> $I(G) = \max(I(G\text{-}v), 1+I(G\text{-}N(v)))$

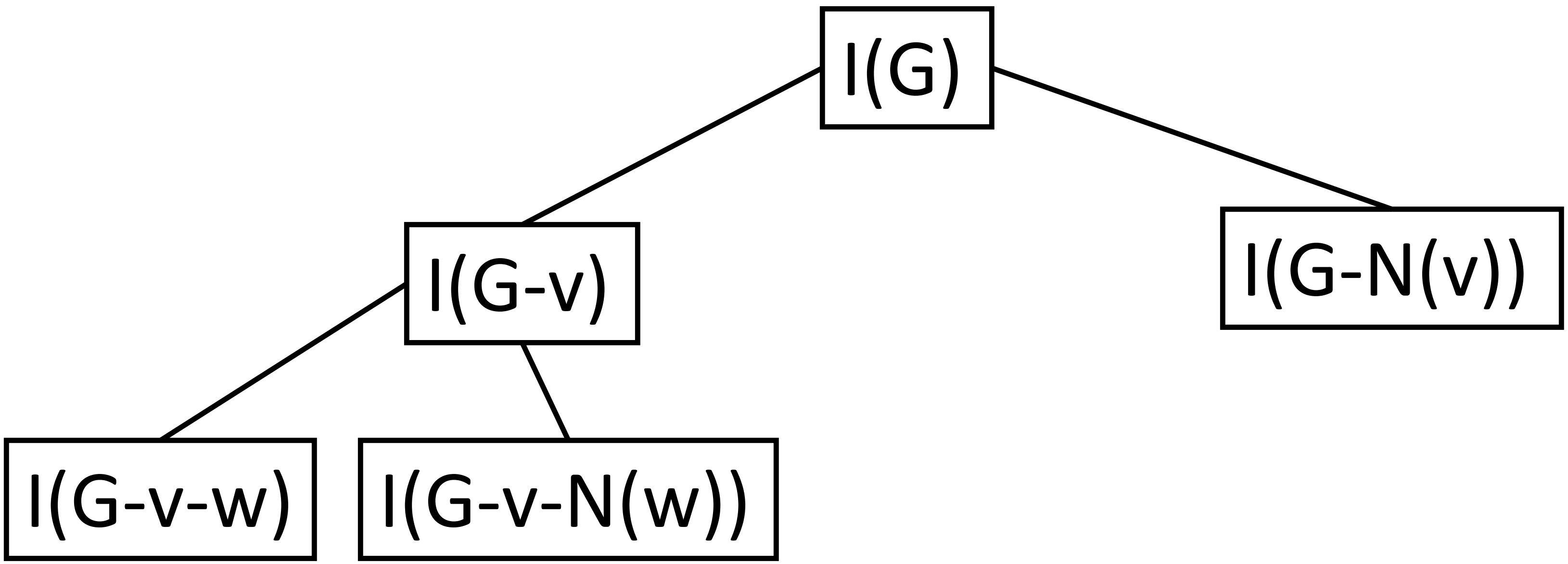
Subproblems

$\boxed{\text{I(G)}}$

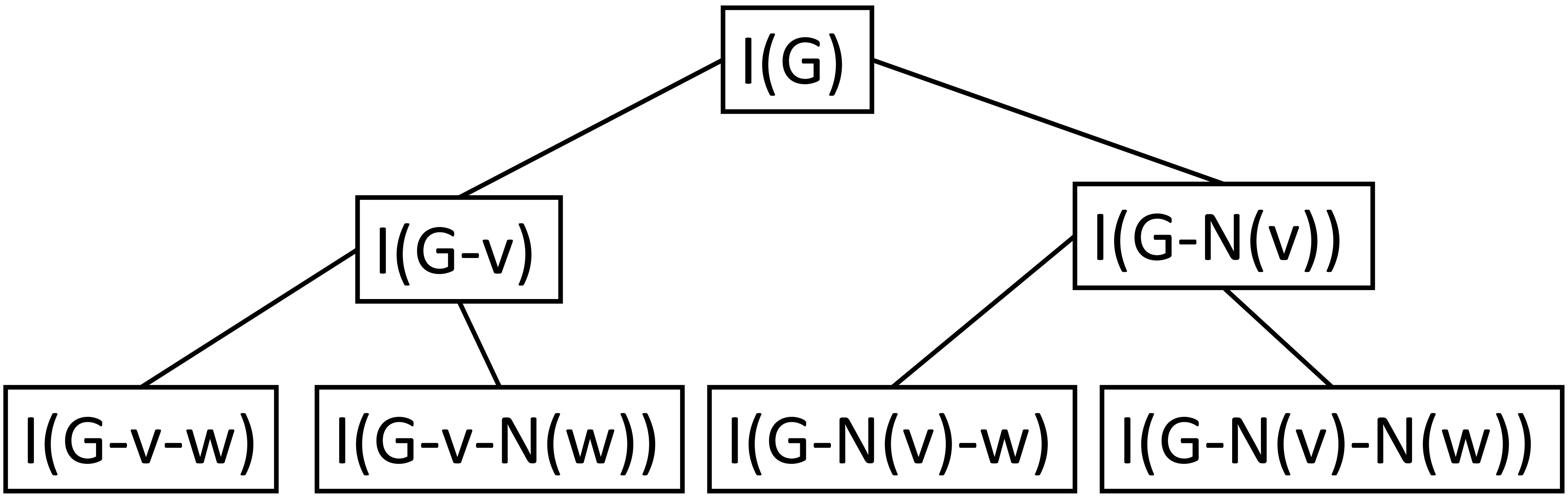
Subproblems



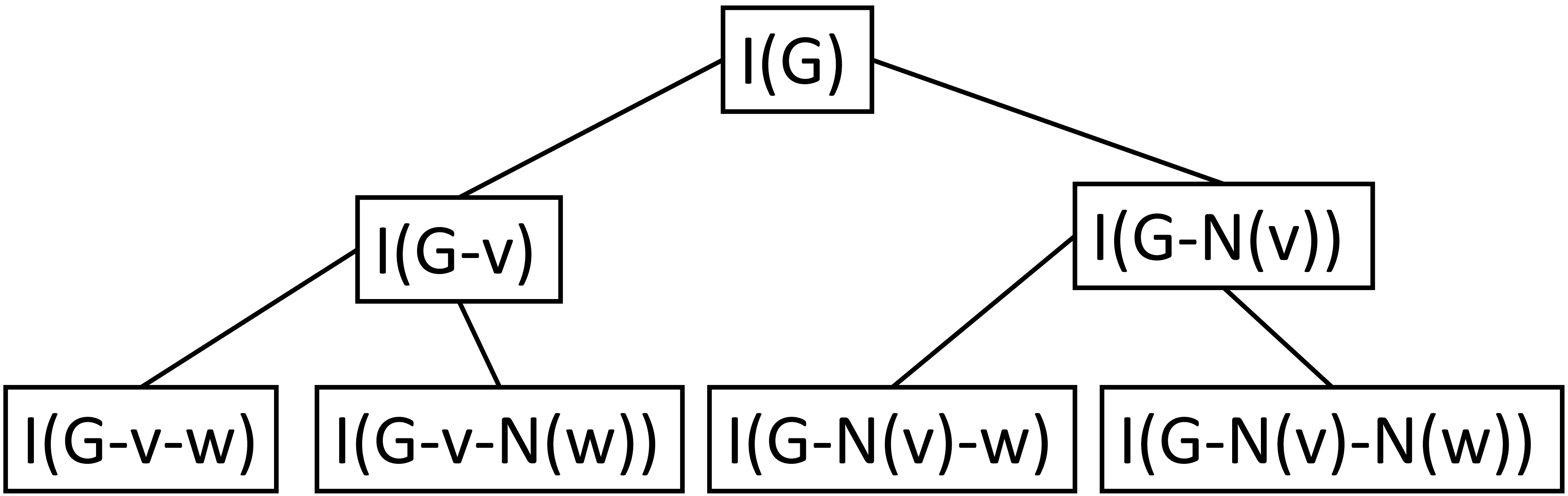
Subproblems



Subproblems

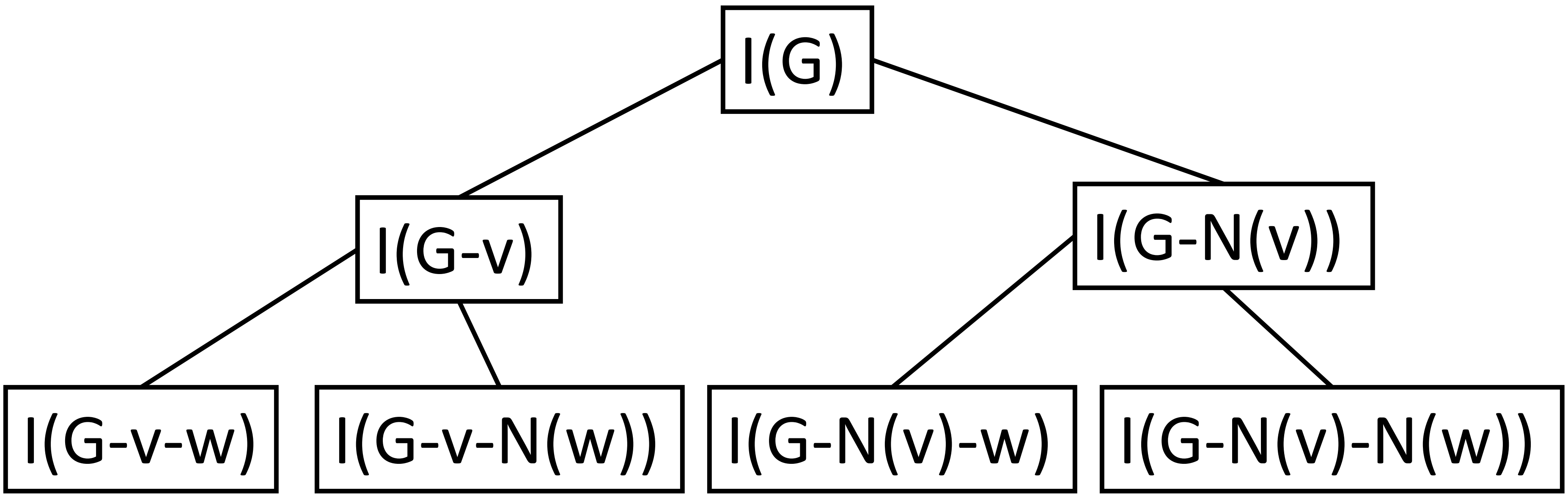


Subproblems



- Very little subproblem reuse.

Subproblems



- Very little subproblem reuse.
- Need subproblems I(G’) for every subgraph G’.
  - Number of subproblems $2^{|V|}$.

Hardness

Independent Set is what's known as an <u>NP-Hard</u> problem. This means that people believe that there may well be no efficient algorithm for it.

Hardness

Independent Set is what's known as an NP-Hard problem. This means that people believe that there may well be no efficient algorithm for it.

However, there are special cases where we can do better.

Independent Sets and Components

Lemma: If G has connected components $C_1, C_2, \ldots, C_k$ then

$$I(G) = I(C_1)+I(C_2)+\ldots+I(C_k).$$

Independent Sets and Components

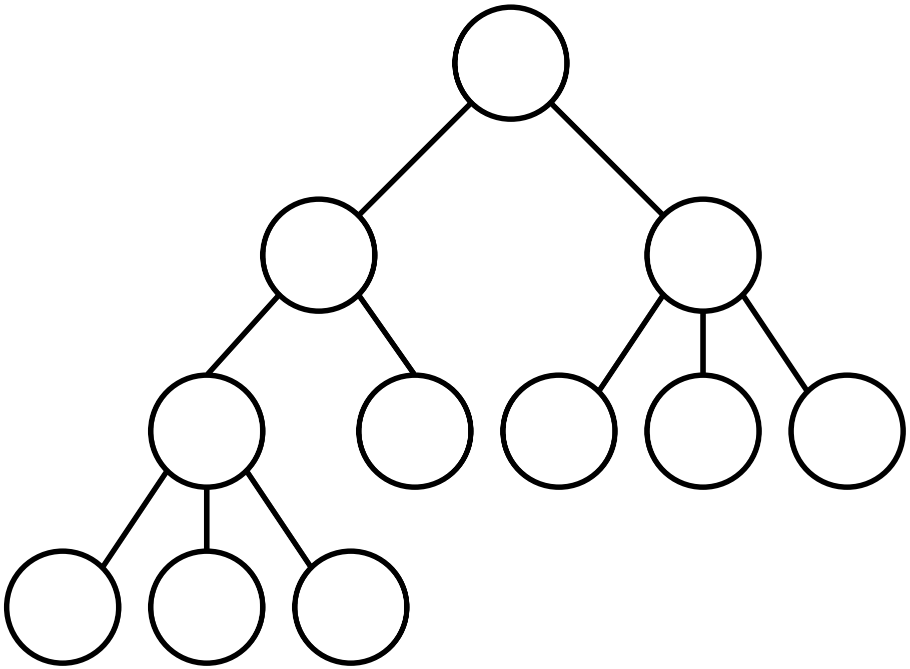
Lemma: If G has connected components $C_1, C_2, \ldots, C_k$ then

$$I(G) = I(C_1)+I(C_2)+\ldots+I(C_k).$$

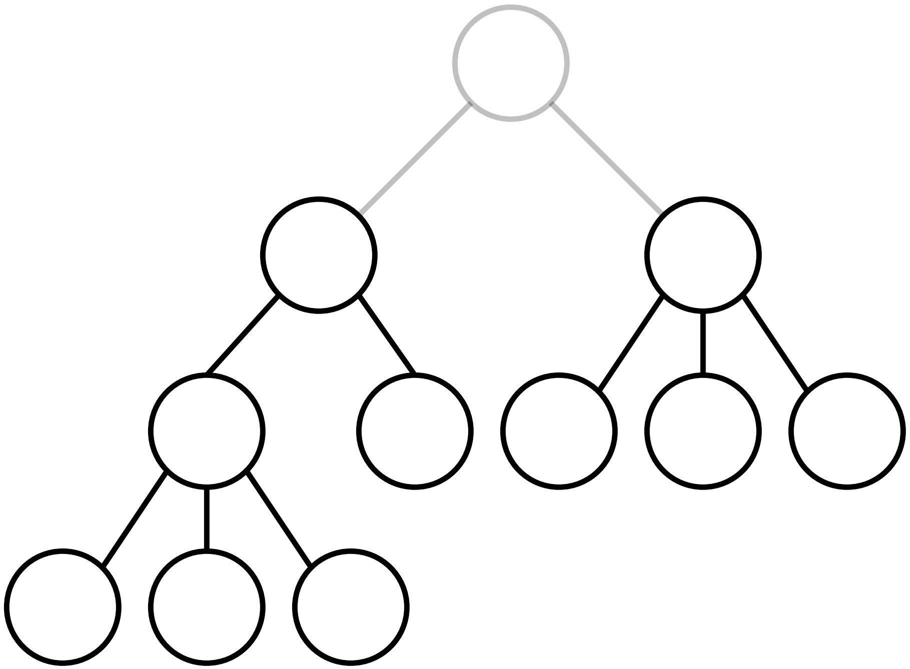
Proof: An independent set for G is exactly the union of an independent set for each of the C_i. Can pick the biggest set for each C_i.

Independent Sets of Trees

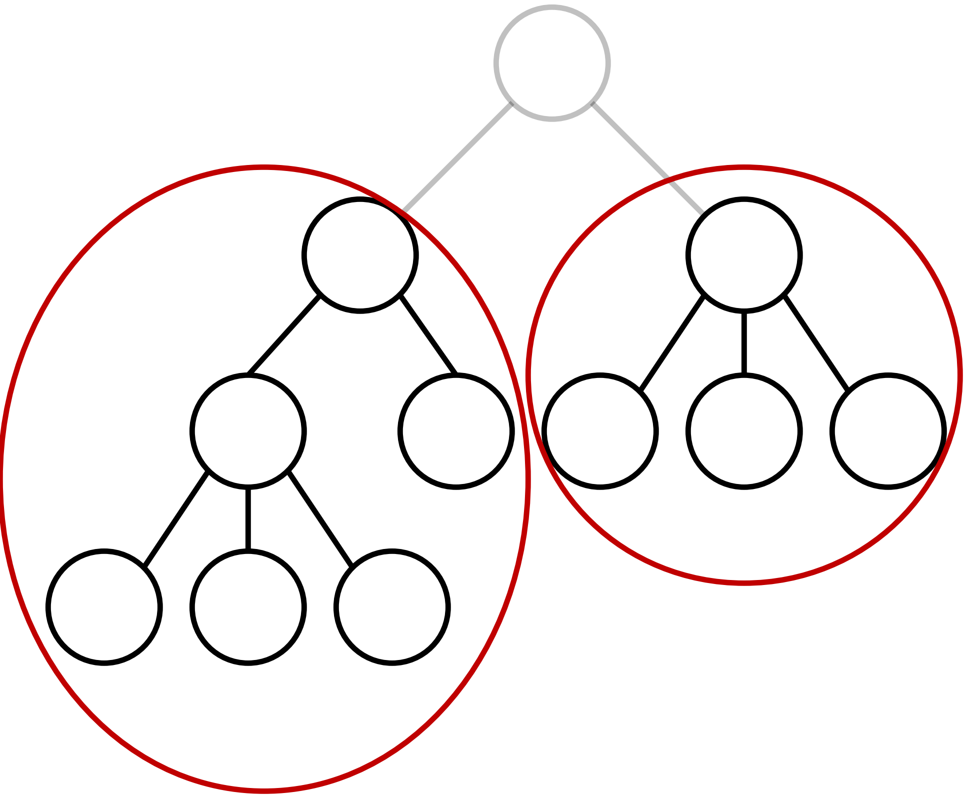
Independent Sets of Trees



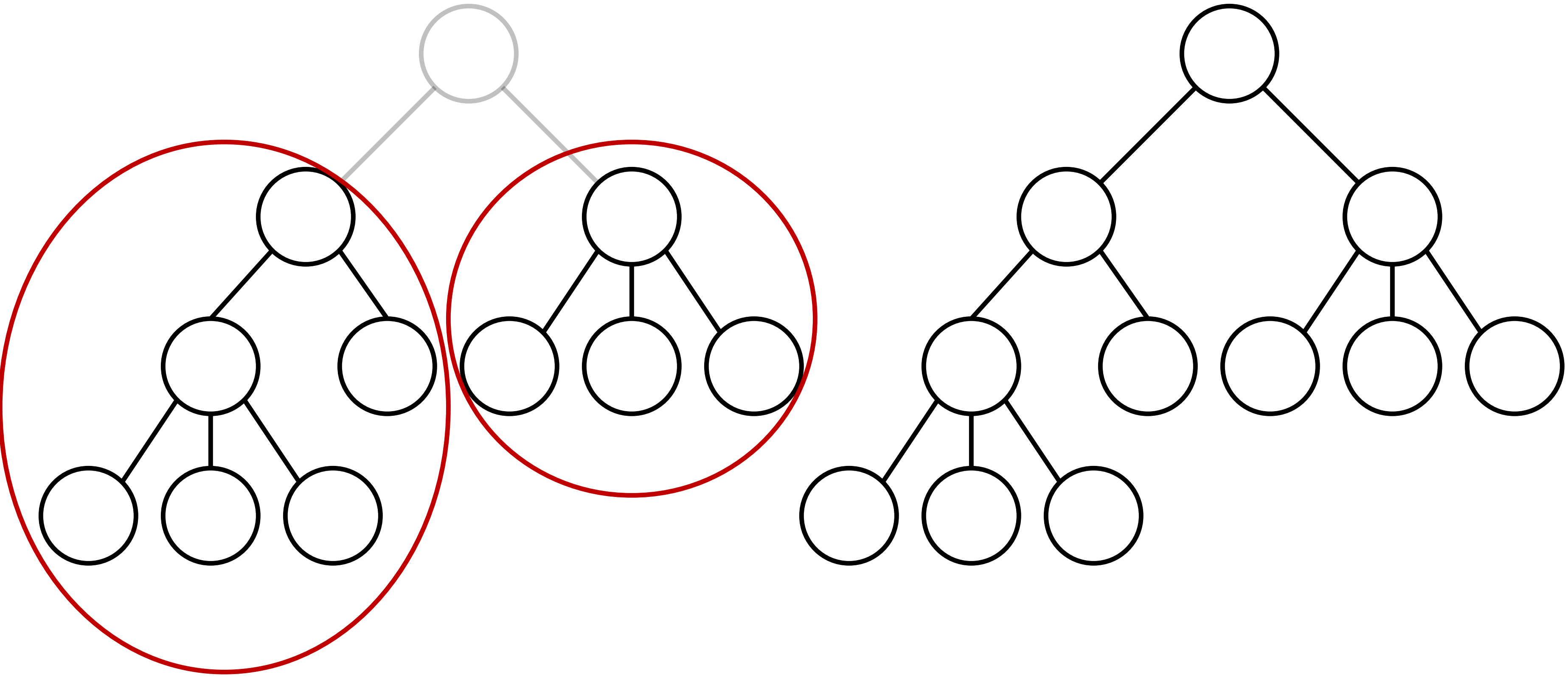
Independent Sets of Trees



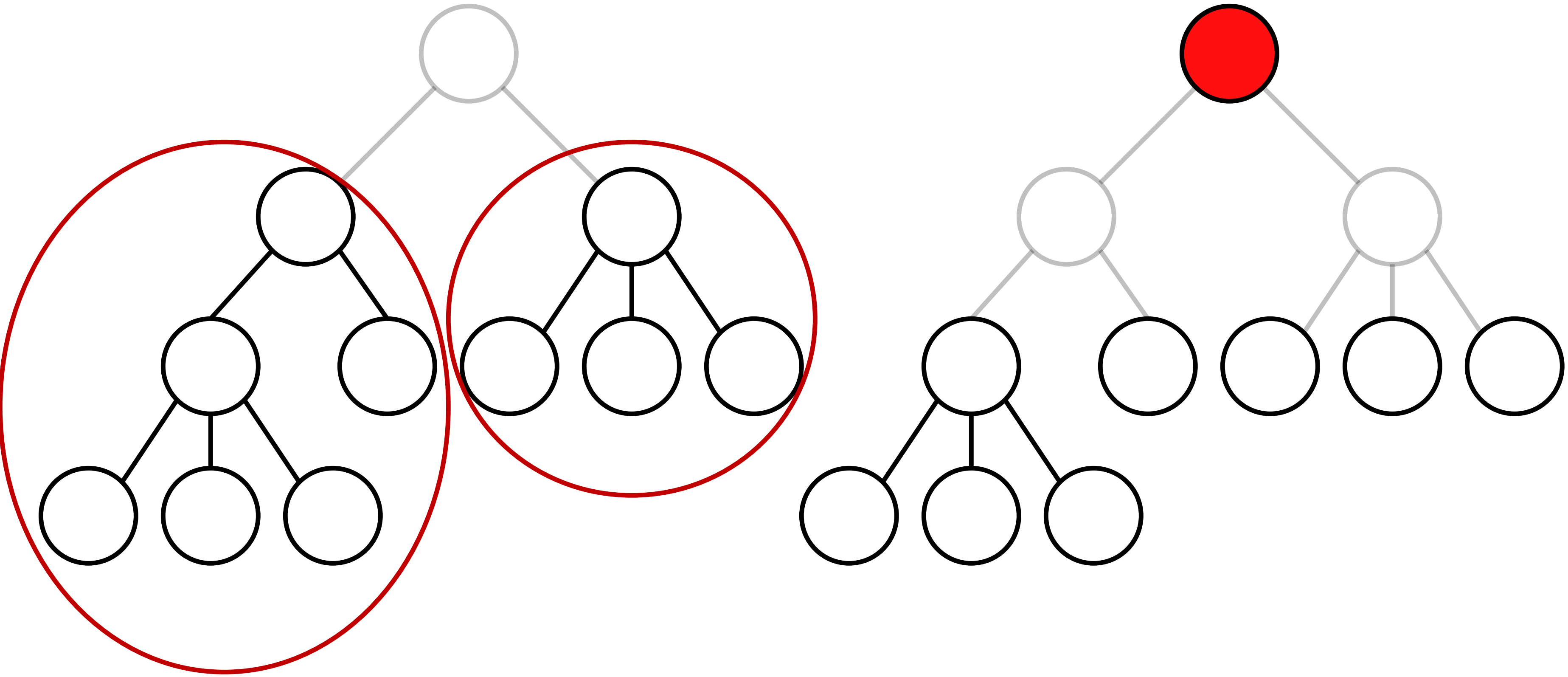
Independent Sets of Trees



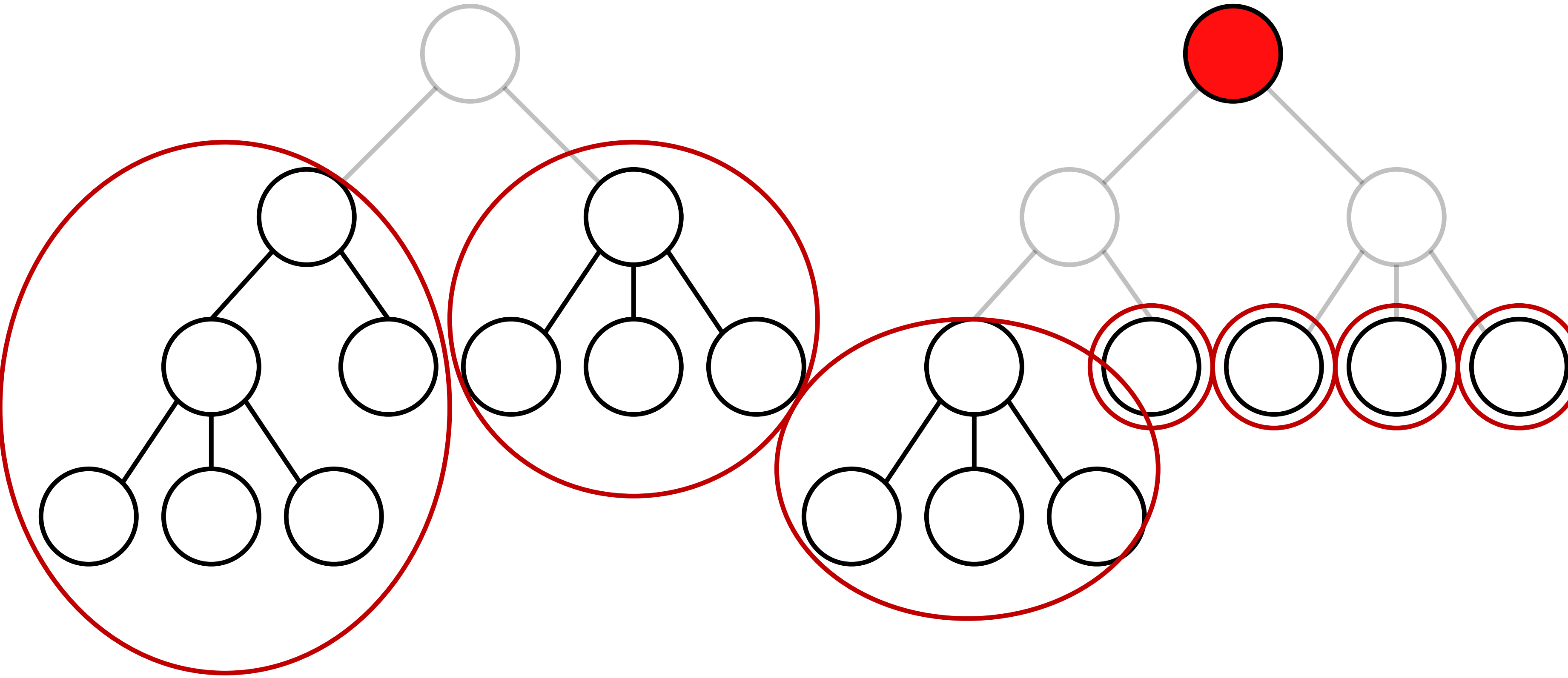
Independent Sets of Trees



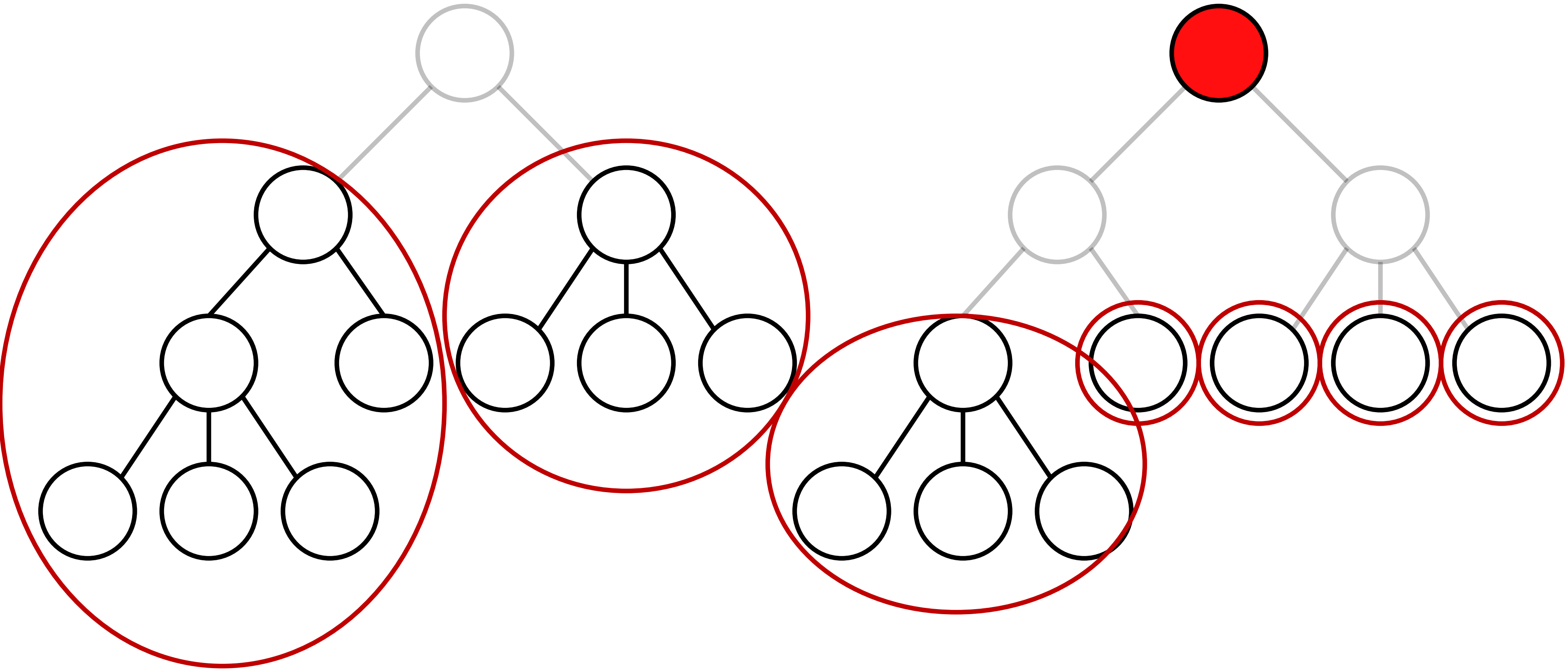
Independent Sets of Trees



Independent Sets of Trees



Independent Sets of Trees



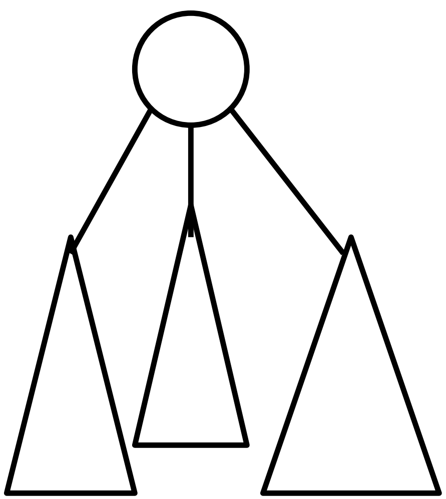
Subproblems are all subtrees!

Recursion

Root not used:

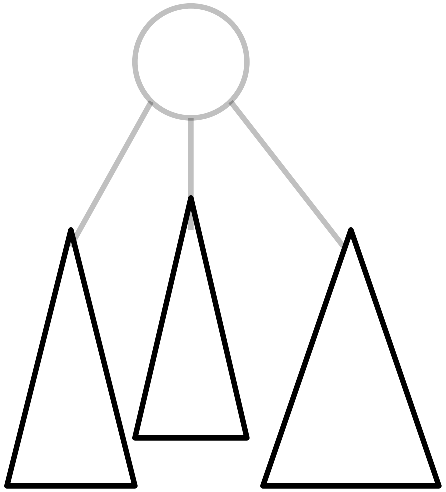
Recursion

Root not used:



Recursion

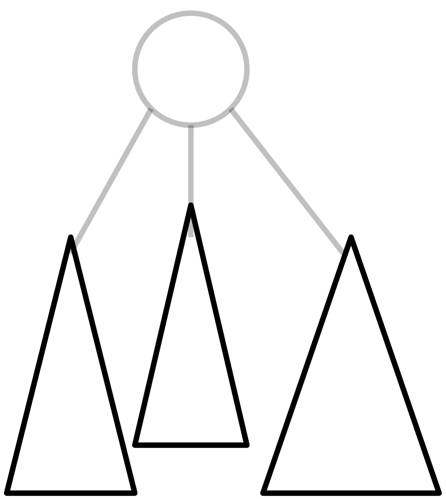
Root not used:



Recursion

Root not used:

I(G) = Σ I(children's subtrees)

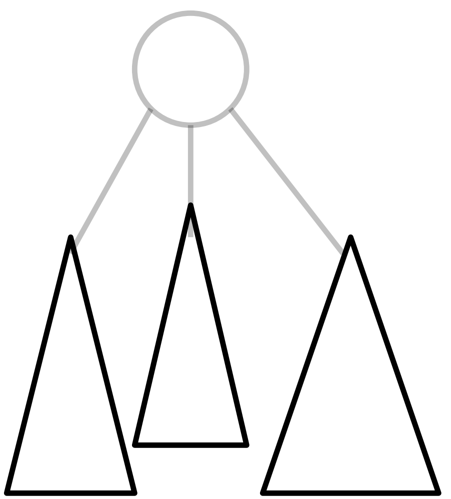


Recursion

Root not used:

I(G) = Σ I(children's subtrees)

Root is used:

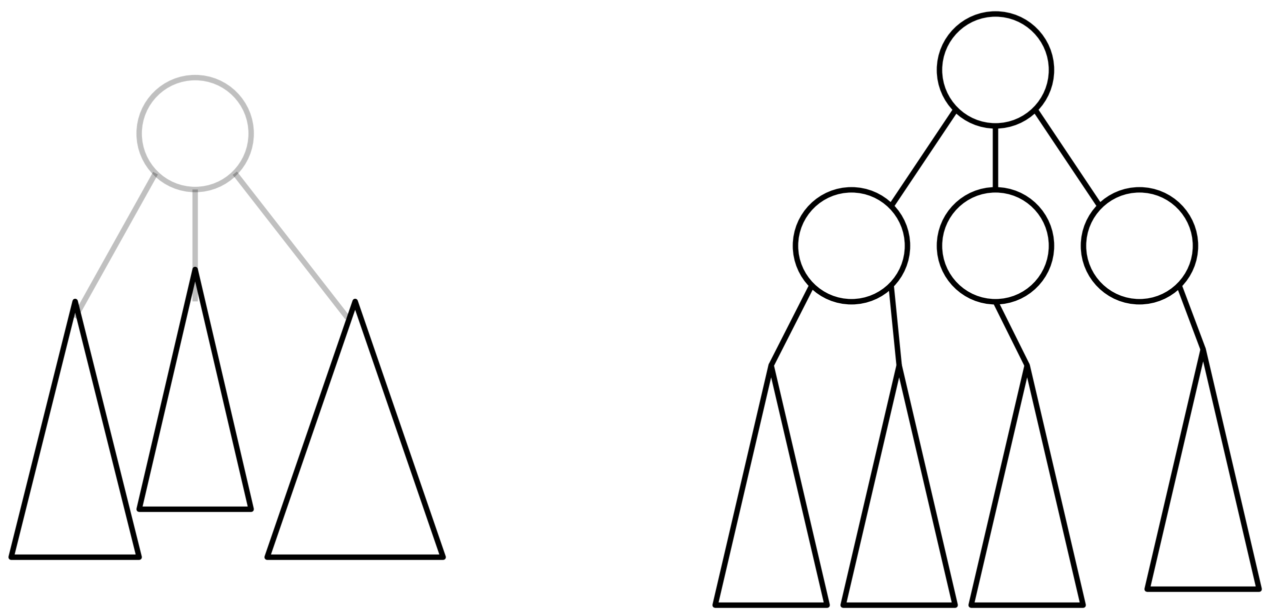


Recursion

<u>Root not used:</u>

I(G) = Σ I(children’s subtrees)

<u>Root is used:</u>

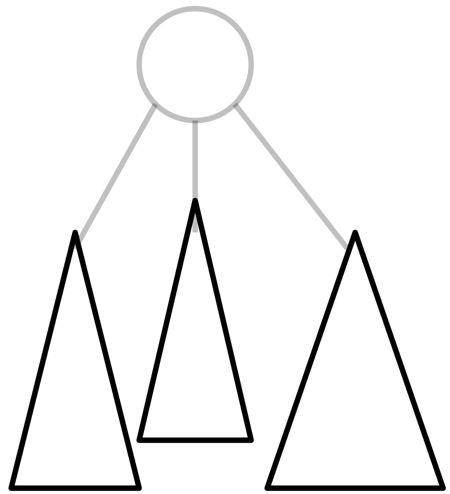


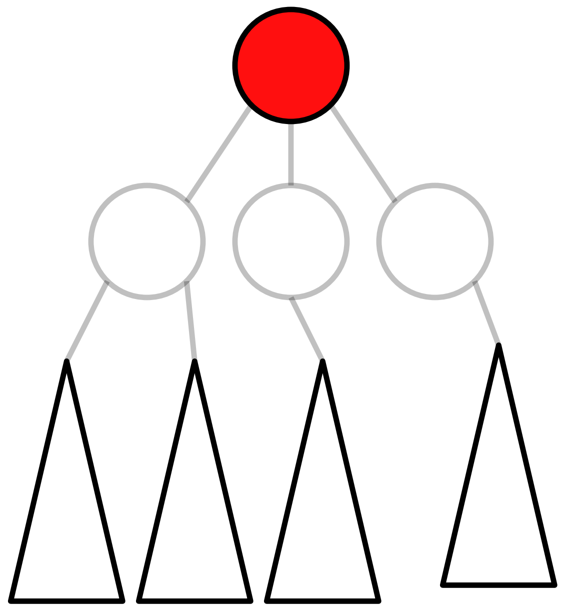
Recursion

Root not used:

I(G) = Σ I(children’s subtrees)

Root is used:





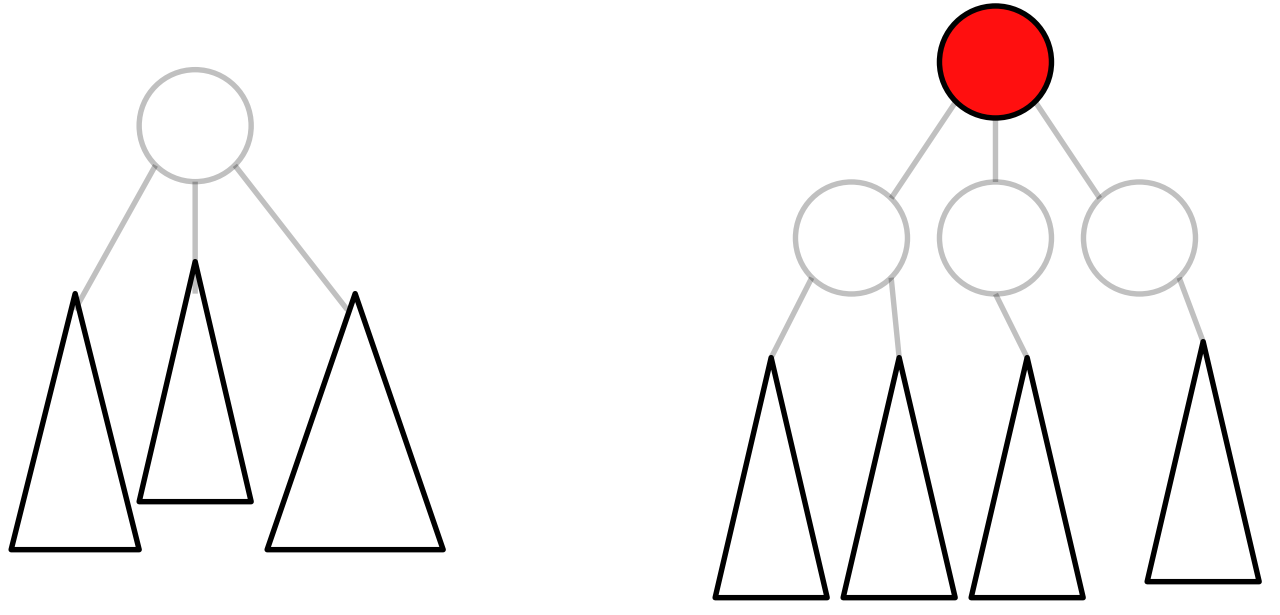
Recursion

Root not used:

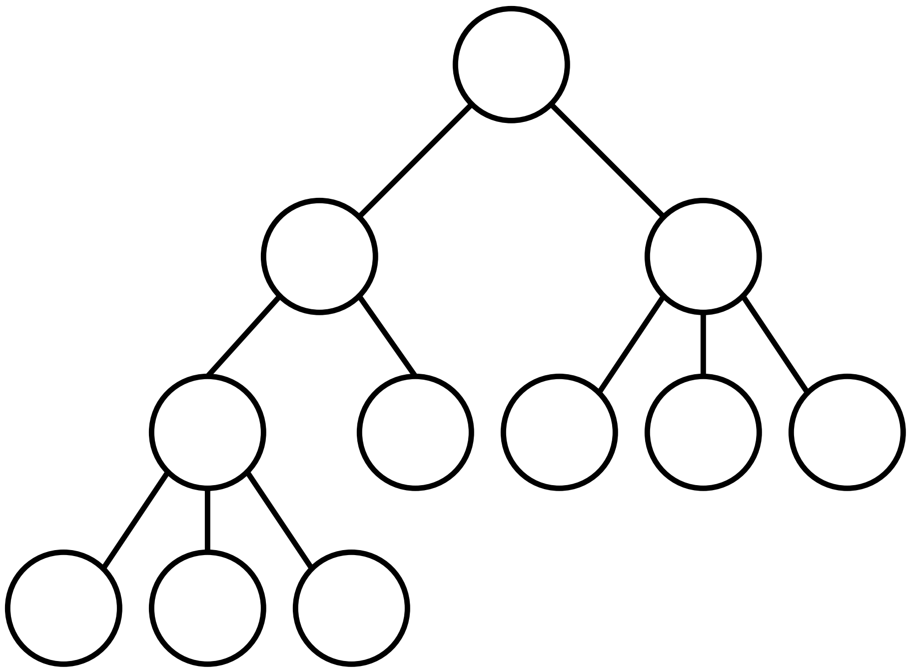
I(G) = Σ I(children's subtrees)

Root is used:

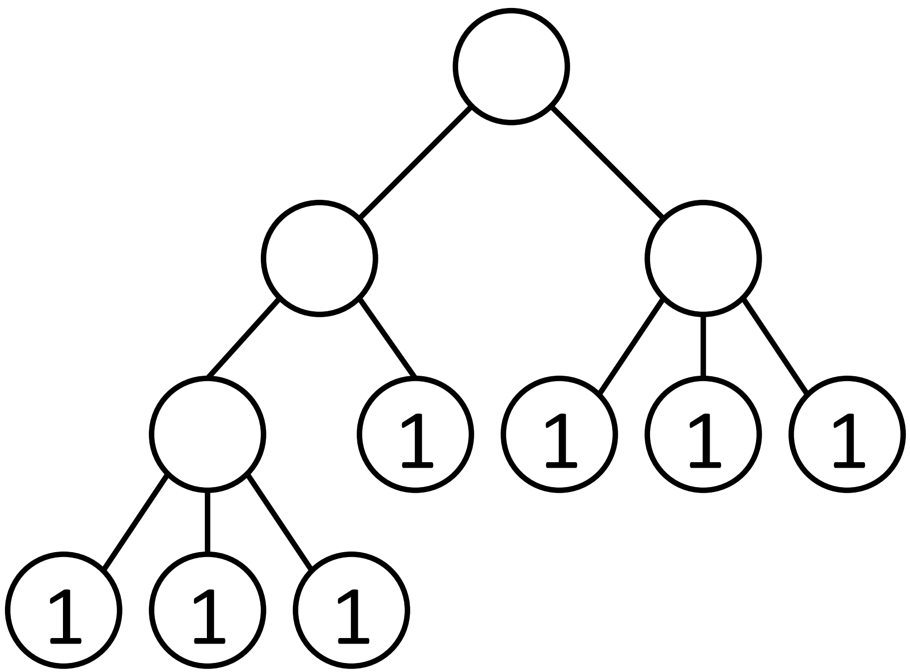
I(G) = 1+Σ I(grandchildren's subtrees)



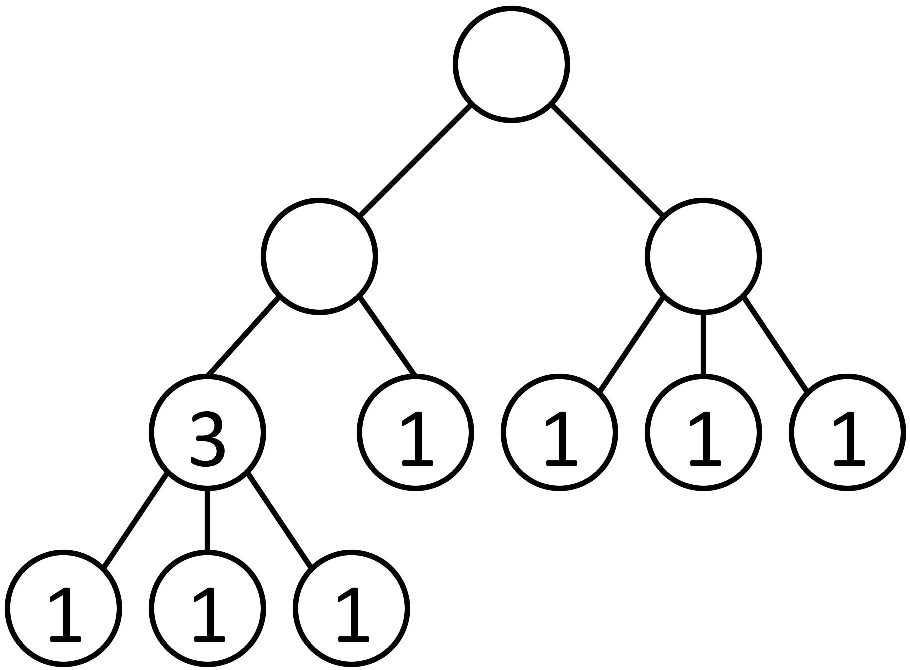
Example



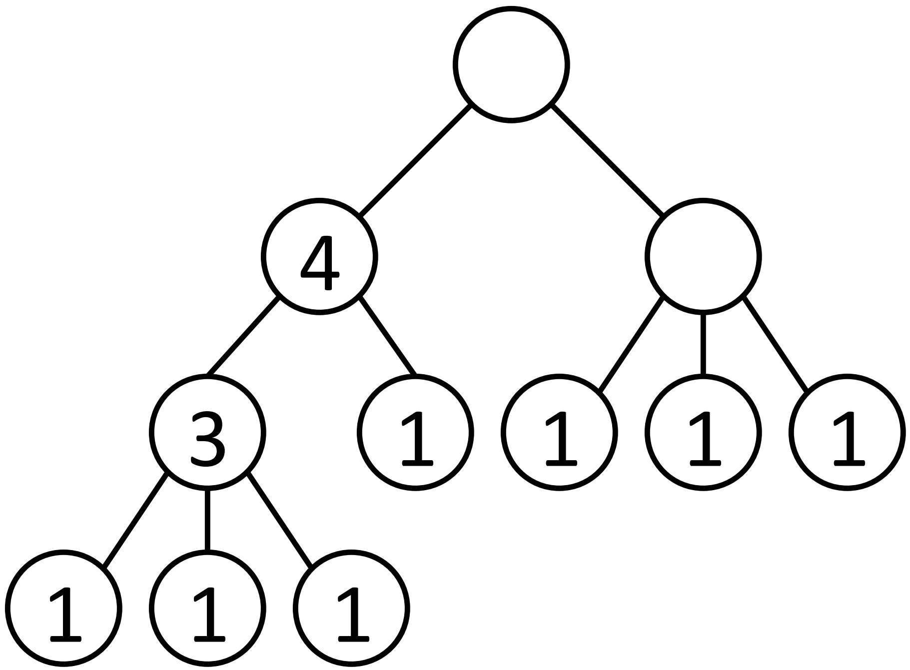
Example



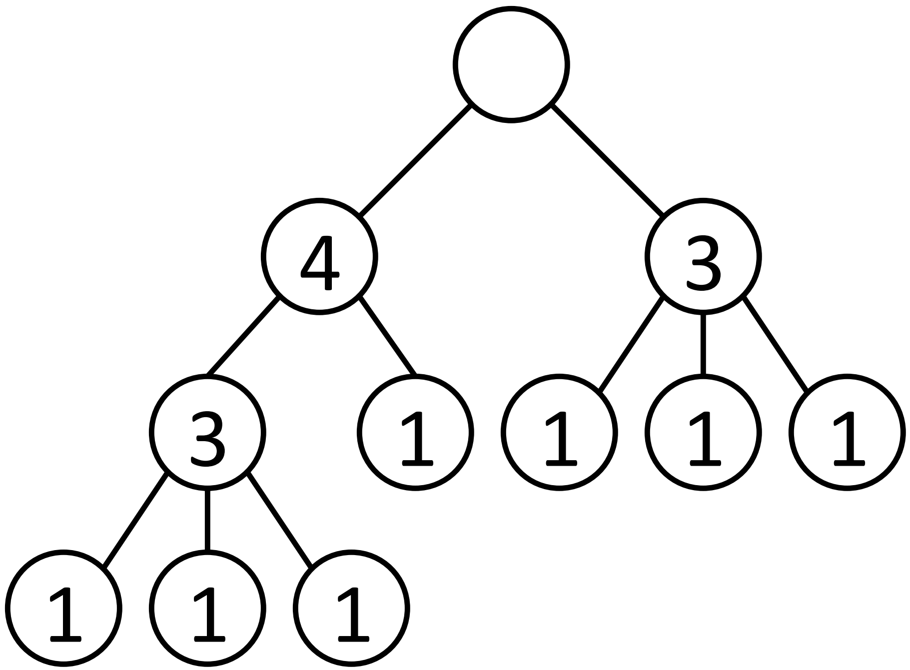
Example



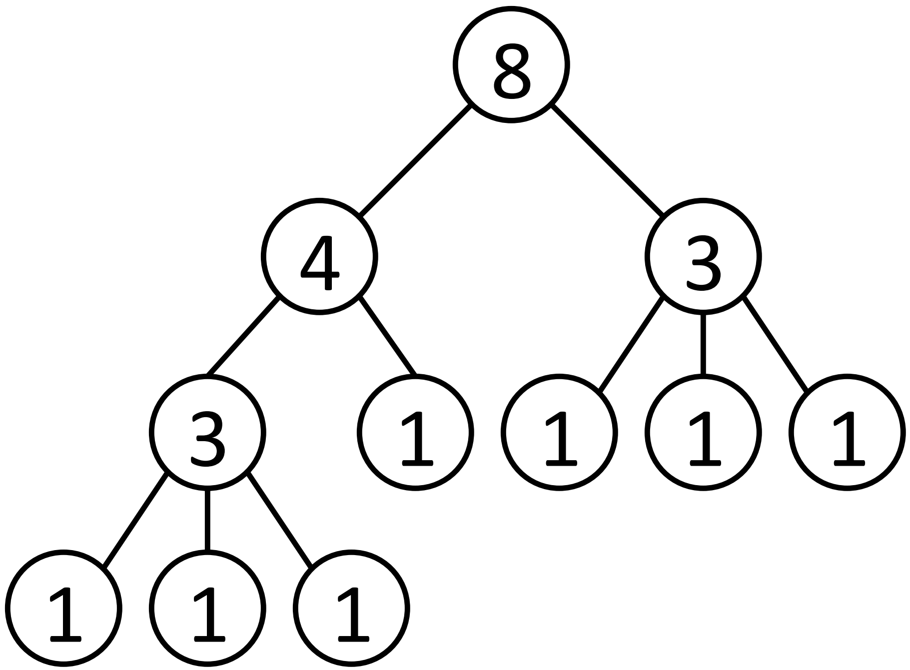
Example



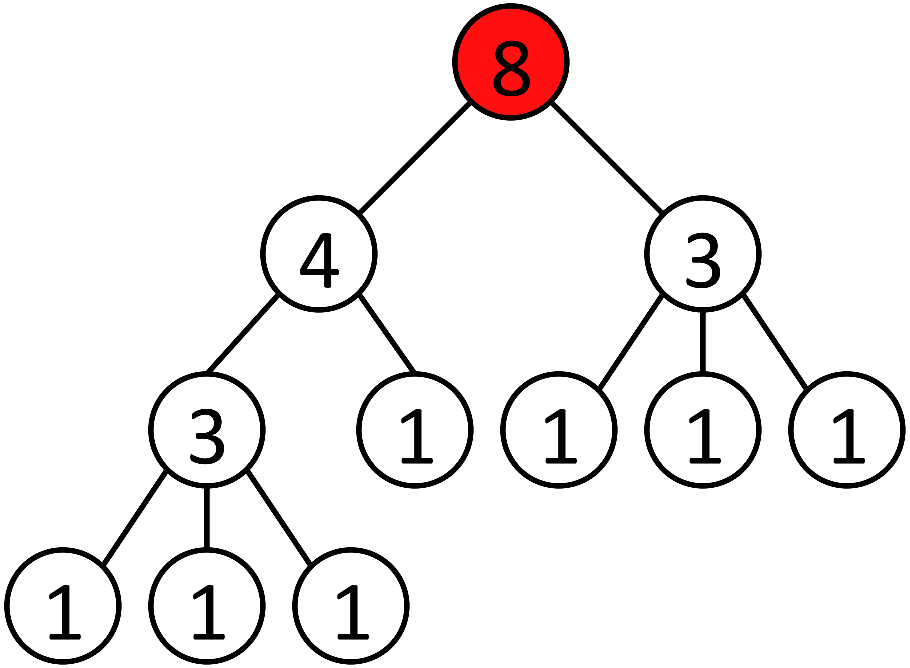
Example



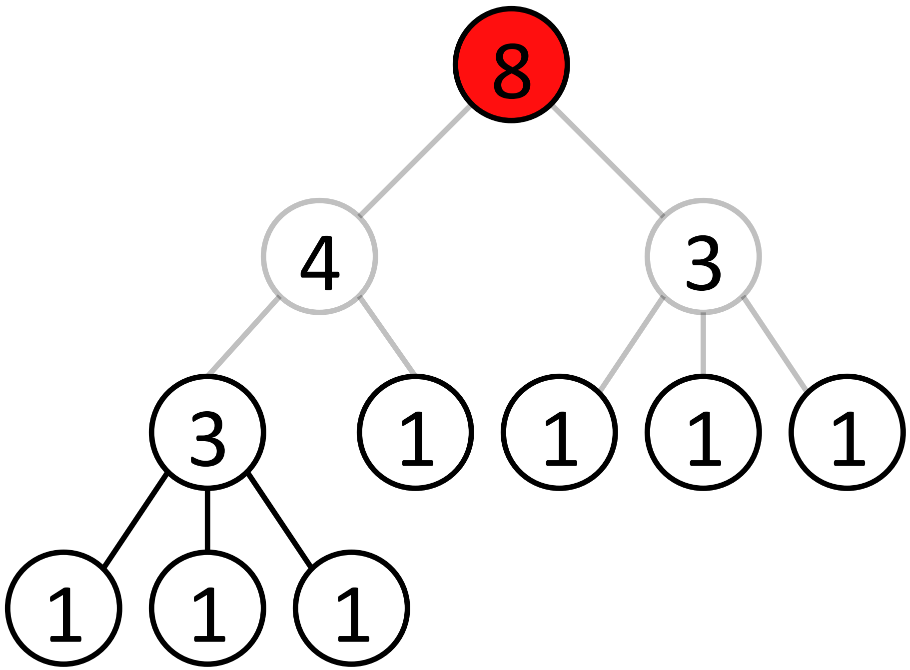
Example



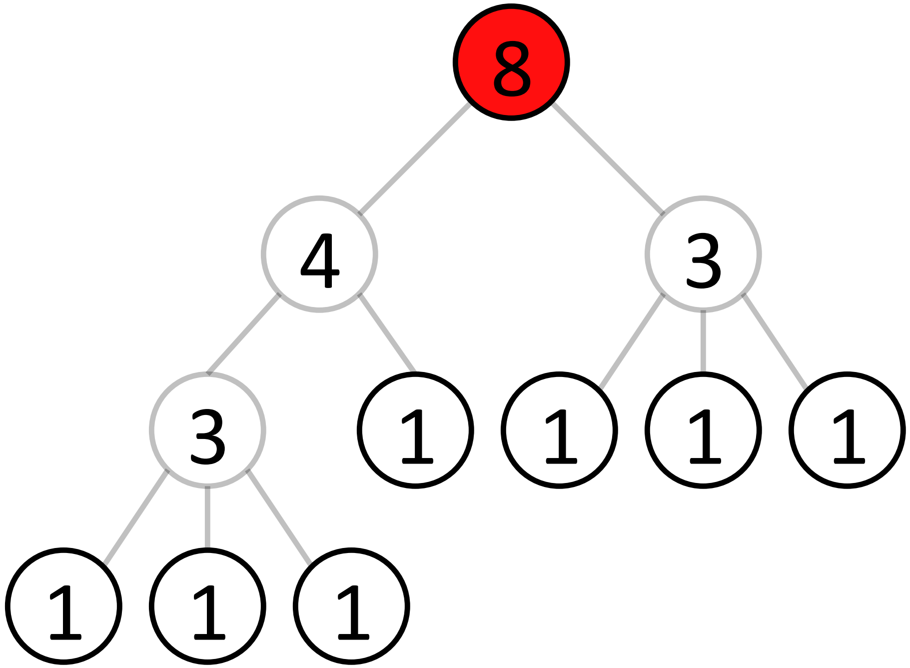
Example



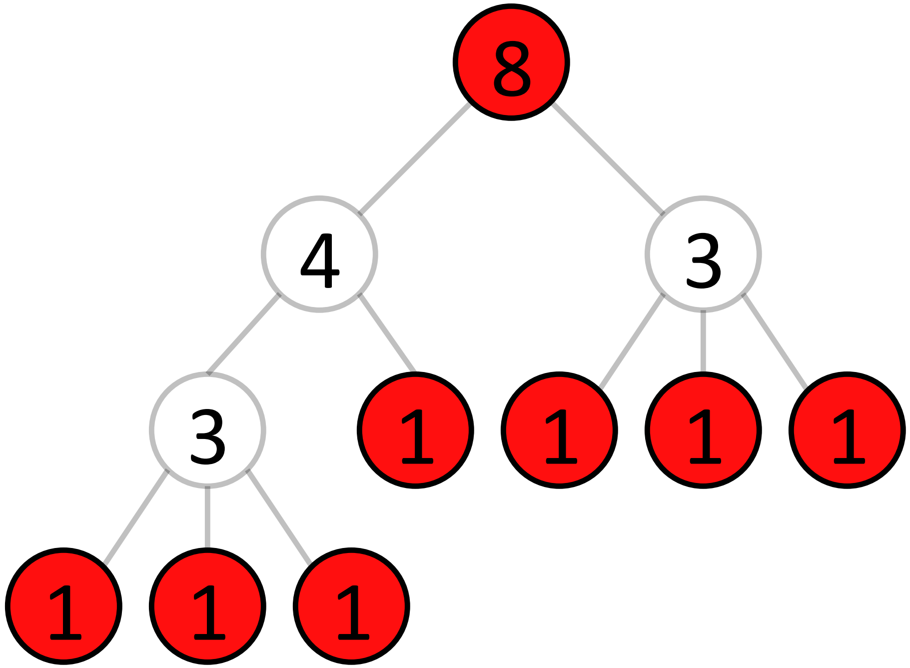
Example



Example



Example



Travelling Salesman Problem

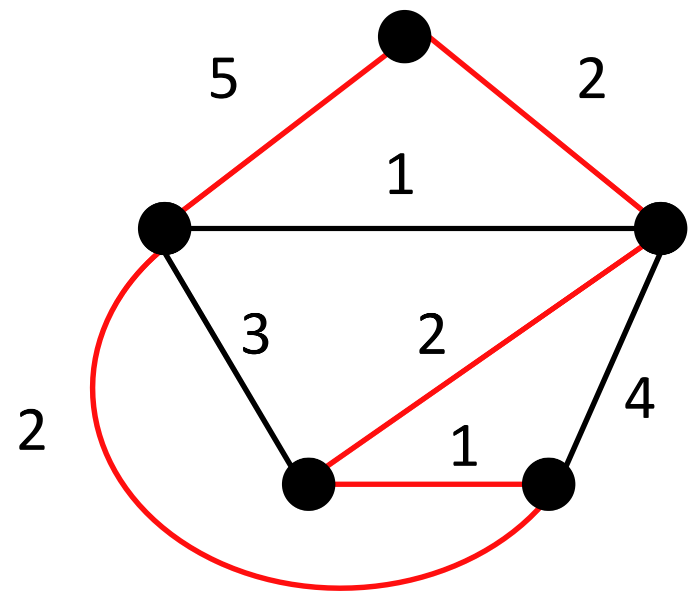
In your job as a door-to-door vacuum salesperson, you need to plan a route that takes you through n different cities. In order to space things out, you do not want to get back to the start until you have visited all cities. You also want to do so with as little travel as possible.

Formal Definition

Problem: Given a weighted (undirected) graph G with n vertices find a cycle that visits each vertex exactly once whose total weight is as small as possible.

Formal Definition

Problem: Given a weighted (undirected) graph G with n vertices find a cycle that visits each vertex exactly once whose total weight is as small as possible.



2+1+2+2+5 = 12

Naïve Algorithm

- Try all possible paths and see which is cheapest.

Naïve Algorithm

- Try all possible paths and see which is cheapest.
- How many paths?
  - n possible options for first city.
  - (n-1) possible options for second city.
  - (n-2) for third city
  - …
  - Total n!

Naïve Algorithm

- Try all possible paths and see which is cheapest.
- How many paths?
  - n possible options for first city.
  - (n-1) possible options for second city.
  - (n-2) for third city
  - …
  - Total n!
- Runtime ≈ n!

Note on Difficulty

The Travelling Salesman problem is a difficult problem. In fact it is widely believed that there is <u>no</u> polynomial time algorithm for it (more on this in chapter 8).

Note on Difficulty

The Travelling Salesman problem is a difficult problem. In fact it is widely believed that there is no polynomial time algorithm for it (more on this in chapter 8).

However, there is an algorithm that beats the n! naïve algorithm.

Setup

Need partial solutions for subproblems.

- Look for s-t paths instead of cycles.

Setup

Need partial solutions for subproblems.

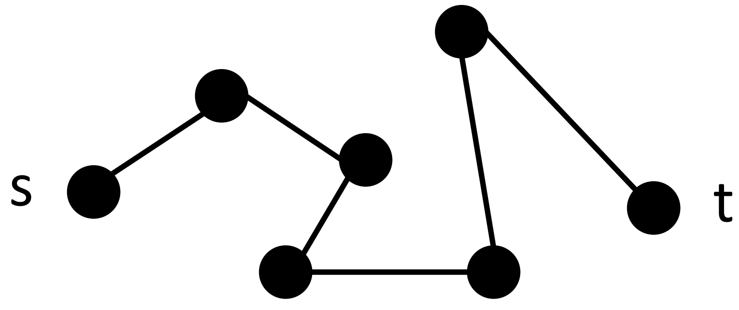
- Look for s-t paths instead of cycles.

$\text{Best}_{st}(G)$ = Best value of a path starting at s and ending at t that visits each vertex exactly once.

- Answer is minimum of $\text{Best}_{st}(G)+\ell(s,t)$.

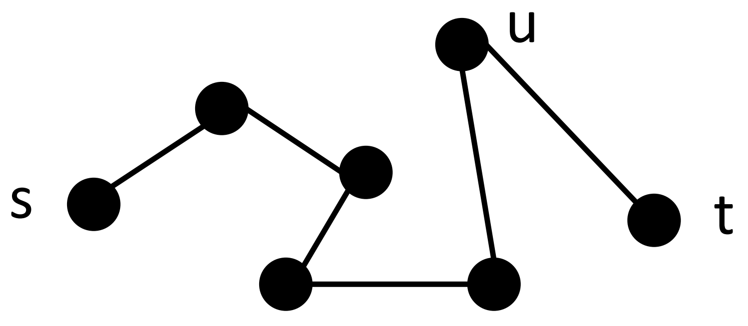
Recursion

- What happens if we undo the last step?



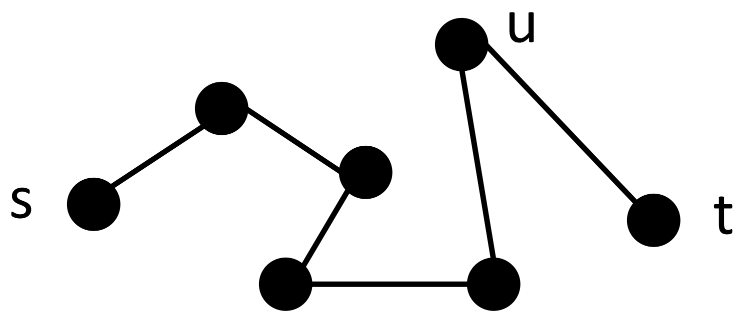
Recursion

- What happens if we undo the last step?
  - Last step some edge (u,t).



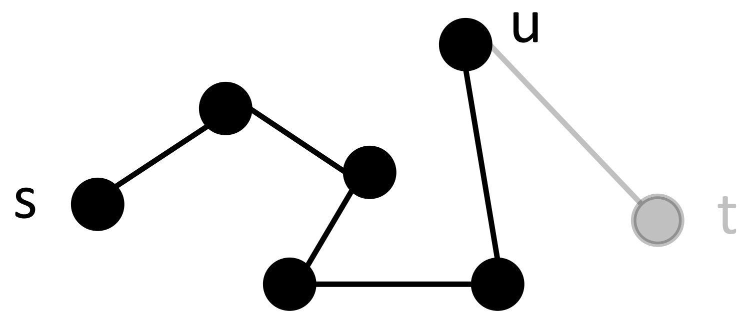
Recursion

- What happens if we undo the last step?
  - Last step some edge (u,t).
  - Value is ℓ(u,t)+length of rest of path.



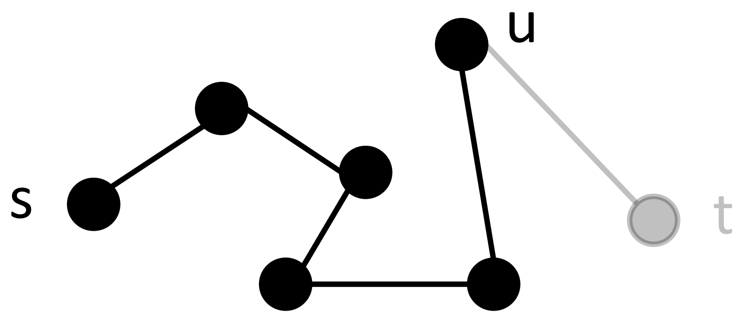
Recursion

- What happens if we undo the last step?
  - Last step some edge (u,t).
  - Value is ℓ(u,t)+length of rest of path.
  - Want best s-u that uses every vertex <u>except for t</u>.



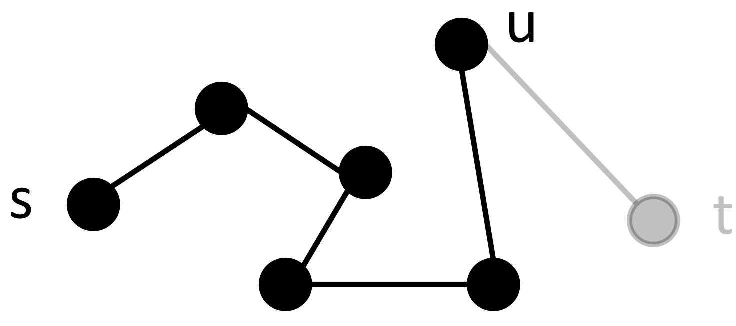
Recursion

- What happens if we undo the last step?
  - Last step some edge (u,t).
  - Value is ℓ(u,t)+length of rest of path.
  - Want best s-u that uses every vertex <u>except for t</u>.
- $\text{Best}_{st}(G) = \min_u[\text{Best}_{su,-t}(G)+\ell(u,t)]$.



Recursion

- What happens if we undo the last step?
  - Last step some edge (u,t).
  - Value is $\ell(u,t)$+length of rest of path.
  - Want best s-u that uses every vertex <u>except for t</u>.
- $\text{Best}_{st}(G) = \min_u[\text{Best}_{su,-t}(G)+\ell(u,t)]$.
- Now we need a recursion for $\text{Best}_{su,-t}(G)$.



Recursion II

- How do we solve for $Best_{su,-t}(G)$?
- Remove last edge (v,u).
- Need best s-v path that uses all vertices except for t and u.
- Need more complicated subproblems to solve for that.

Recursion III

$\text{Best}_{st,L}(G)$ = Best s-t path that uses exactly the vertices in L.

Recursion III

$Best_{st,L}(G)$ = Best s-t path that uses exactly the vertices in L.

- Last edge is some $(v,t) \in E$ for some $v \in L$.
- Cost is $Best_{sv,L-t}(G) + \ell(v,t)$.

Recursion III

$\text{Best}_{st,L}(G)$ = Best s-t path that uses <u>exactly</u> the vertices in L.

- Last edge is some $(v,t) \in E$ for some $v \in L$.
- Cost is $\text{Best}_{sv,L-t}(G) + \ell(v,t)$.

<u>Full Recursion:</u>

$\text{Best}_{st,L}(G) = \min_v[\text{Best}_{sv,L-t}(G) + \ell(v,t)]$.

Runtime Analysis

<u>**Number of Subproblems:**</u>

L can be any subset of vertices (2^n possibilities)

s and t can be any vertices (n^2 possibilities)

$n^2 2^n$ total.

Runtime Analysis

<u>Number of Subproblems:</u>

L can be any subset of vertices (2^n possibilities)

s and t can be any vertices (n^2 possibilities)

$n^2 2^n$ total.

<u>Time per Subproblem:</u>

Need to check every v (O(n) time).

Runtime Analysis

Number of Subproblems:

L can be any subset of vertices (2^n possibilities)

s and t can be any vertices (n^2 possibilities)

$n^2 2^n$ total.

Time per Subproblem:

Need to check every v (O(n) time).

Final Runtime:

$O(n^3 2^n)$

[can improve to $O(n^2 2^n)$ with a bit of thought]

Runtime Comparison

We have $O(n^3 2^n)$ time.

Naïve algorithm is $O(n!)$.

Runtime Comparison

We have $O(n^3 2^n)$ time.

Naïve algorithm is $O(n!)$.

Note: $n! > (n/e)^n >> 2^n$.

Runtime Comparison

We have $O(n^3 2^n)$ time.

Naïve algorithm is $O(n!)$.

Note: $n! > (n/e)^n >> 2^n$.

Dynamic programming doesn't make this problem fast, but it is much better than what we had before.