

DP5 End of Arrays; DP on trees

CSE 421 Autumn 26
Lecture 15

Example

What's the distance between `babyyodas` and `tastysoda`?

<i>x</i>	B	A	B		Y	Y	O	D	A	S
op	Sub		sub	ins		sub				del
<i>y</i>	T	A	S	T	Y	S	O	D	A	

Distance: 5, one point for each colored box

Quick Checks – can you explain these?

If x has length n and y has length m , the edit distance is at most $\max(x, y)$

The distance from x to y is the same as from y to x (i.e. transforming x to y and y to x are the same)

Finding a recurrence

What information would let us simplify the problem?

What would let us “take one step” toward the solution?

“Handling” one character of x or y

i.e. choosing one of insert, delete, or substitution and increasing the “distance” by 1

OR realizing the characters are the same and matching for free.

$OPT(i, j)$ is the minimum number of insertions, deletions, and substitutions to transform $x_1x_2 \cdots x_i$ into $y_1y_2 \cdots y_j$. (we’re indexing strings from 1, it’ll make things a little prettier).

The recurrence

“Handling” one character of x or y

i.e. choosing one of insert, delete, or substitution and increasing the “distance” by 1

OR realizing the characters are the same and matching for free.

What does delete look like? $OPT(i - 1, j)$ (delete character from x match the rest)

Insert $OPT(i, j - 1)$ Substitution: $OPT(i - 1, j - 1)$

Matching characters? Also $OPT(i - 1, j - 1)$ but only if $x_i = y_j$

The recurrence (starting)

“Handling” one character of x or y

i.e. choosing one of insert, delete, or substitution and increasing the “distance” by 1

OR realizing the characters are the same and matching for free.

$$OPT(i, j) = \left\{ \begin{array}{l} \min \left\{ \begin{array}{l} \text{Delete} \\ \text{Insert} \\ \text{Substitution} \end{array} \right\} \\ \text{if } i = 0 \\ \text{if } j = 0 \end{array} \right. \left. \begin{array}{l} \min \{ 1 + OPT(i-1, j), 1 + OPT(i, j-1), 1 + OPT(i-1, j-1) \} \\ \text{if } i = 0 \\ \text{if } j = 0 \end{array} \right\}$$

TODO: Just Match

The recurrence (v1, we'll improve soon)

"Handling" one character of x or y

i.e. choosing one of insert, delete, or substitution and increasing the "distance" by 1

OR realizing the characters are the same and matching for free.

$$OPT(i, j) = \begin{cases} \min\{ \overset{\text{Delete}}{1 + OPT(i-1, j)}, \overset{\text{Insert}}{1 + OPT(i, j-1)}, \overset{\text{Substitution}}{1 + OPT(i-1, j-1)}, \overset{\text{Just Match}}{OPT(i-1, j-1) + \infty \cdot \mathbb{I}\{x_i \neq y_j\}} \} & \text{if } i > 0 \text{ and } j > 0 \\ j & \text{if } i = 0 \\ i & \text{if } j = 0 \end{cases}$$

Idea: only allow "just match" when you can just match.

Otherwise make it ∞ (will never be the min).

In code: if/else branch, probably. This is a math notation trick.

"Indicator"
like from 312

The recurrence (finalized)

“Handling” one character of x or y

i.e. choosing one of insert, delete, or substitution and increasing the “distance” by 1

OR realizing the characters are the same and matching for free.

$$OPT(i, j) = \begin{cases} \min\{ \overset{\text{Delete}}{1 + OPT(i - 1, j)}, \overset{\text{Insert}}{1 + OPT(i, j - 1)}, \overset{\text{Sub and matching}}{\mathbb{I}[x_i \neq y_j] + OPT(i - 1, j - 1)} \} & \text{if } i = 0 \\ j & \text{if } i = 0 \\ i & \text{if } j = 0 \end{cases}$$

When we could match, we will never substitute; matching will always give us a better score! Still have to check delete, insert (those could be better).

Computing (10)

Fill in the next two entries. Be careful with the sub/match distinction!

$OPT(i, j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1	1	2	3	4	5	6	7	8	9
A 2	2	2	1	2	3	4	5	6	7	8
S 3	3	3	2	2	3	4	5	6	7	7
T 4	4	4	3	3	3	4	5	6	7	8
Y 5	5	5	4	4	3	3	4			
S 6										
O 7										
D 8										
A 9										

Y's match, so sub is free!

Computing (11)

$OPT(i, j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1	1	2	3	4	5	6	7	8	9
A 2	2	2	1	2	3	4	5	6	7	8
S 3	3	3	2	2	3	4	5	6	7	7
T 4	4	4	3	3	3	4	5	6	7	8
Y 5	5	5	4	4	3	3	4	5	6	7
S 6	6	6	5	5	4	4	4	5	6	6
O 7	7	7	6	6	5	5	4	5	6	7
D 8	8	8	7	7	6	6	5	4	5	6
A 9	9	9	8	8	7	7	6	6	4	5

What operations?

$OPT(i, j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1	1	2	3	4	5	6	7	8	9
A 2	2	2	1	2	3	4	5	6	7	8
S 3	3	3	2	2	3	4	5	6	7	7
T 4	4	4	3	3	3	4	5	6	7	8
Y 5	5	5	4	4	3	3	4	5	6	7
S 6	6	6	5	5	4	4	4	5	6	6
O 7	7	7	6	6	5	5	4	5	6	7
D 8	8	8	7	7	6	6	5	4	5	6
A 9	9	9	8	8	7	7	6	6	4	5

Dynamic Programming Process

1. Define the object you're looking for

$OPT(i, j)$ is the minimum number of insertions, deletions, and substitutions to transform $x_1x_2 \cdots x_i$ into $y_1y_2 \cdots y_j$.

2. Write a recurrence to say how to find it



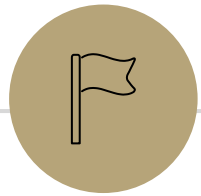
3. Design a memoization structure

$m \times n$ Array

4. Write an iterative algorithm

Outer loop: increasing j (starting from 1)

Inner loop: increasing i (starting from 1)



DP Thought Process



Goal of DP

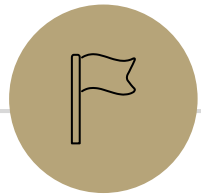
Just try all the (reasonable) possibilities.

Don't worry about greedily choosing the best, use recursion to "look ahead" for all the best options, and pick the best one.

There is a "greedy-ish" alteration to the Edit Distance recurrence...

It turns out, if the two characters match, that will always be at least as good as the insert/delete options.

But it's fine to **not** notice! And if you thought it was safe but wasn't, well....



DP running times



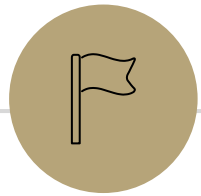
Analyzing Running Times

We haven't analyzed the running times of the DP algorithms we've done so far very closely.

Usually much easier with the iterative version than recursive.

Once you make the switch, the code is usually "just" a few levels of embedded for-loops, each doing constant work.

But be careful---sometimes you take a max of n things, which takes $\Theta(n)$ time.



Proofs of DP algorithms

DP Proofs (1)

We generally **won't** ask you for proofs of correctness on dynamic programming problems. (maybe one after the midterm)

Why?

The proofs are always inductive proofs where you say

“my recurrence checks all the possibilities” or, equivalently

“The maximum thing has to be made up of the best thing for all these other subproblems.”

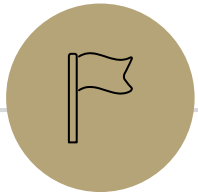
The proof itself is a lot of boilerplate and hard to write clearly (you have to differentiate between your recurrence and what your recurrence intends to calculate, which can be tricky).

DP Proofs (2)

There's one example proof (see lect. 14) so you know what you're (not) missing.

Instead, we're going to ask for your intuition on what your recurrence is doing (what do all the cases correspond to/why are they exhaustive)?

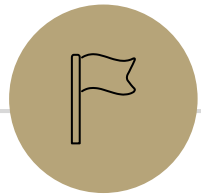
The proof is just a lot of formalism on that key idea. So we're going to have you focus on the idea, not the formalism.



Look at the process once more

Any Questions on DP Process

1. Define the object you're looking for
2. Write a recurrence to say how to find it
3. Design a memoization structure
4. Write an iterative algorithm



DP on Trees

DP: Not just for arrays and strings

Trees are recursive structures

A tree is a root node, with zero or more children

Each of which are roots of trees

Since DP is “smart recursion” (recursion where we save values)

Recursive functions/calculations are really common.

A DP problem for trees

Find the minimum vertex cover in a tree.

Give every **vertex** a weight, find the minimum weight vertex cover

Vertex Cover

A set S of vertices is a vertex cover if for every edge (u, v) :
 u is in S , or v is in S , (or both)

The weight of a vertex cover is just the sum of the weights of the vertices in the set.

We want to find the minimum weight vertex cover.

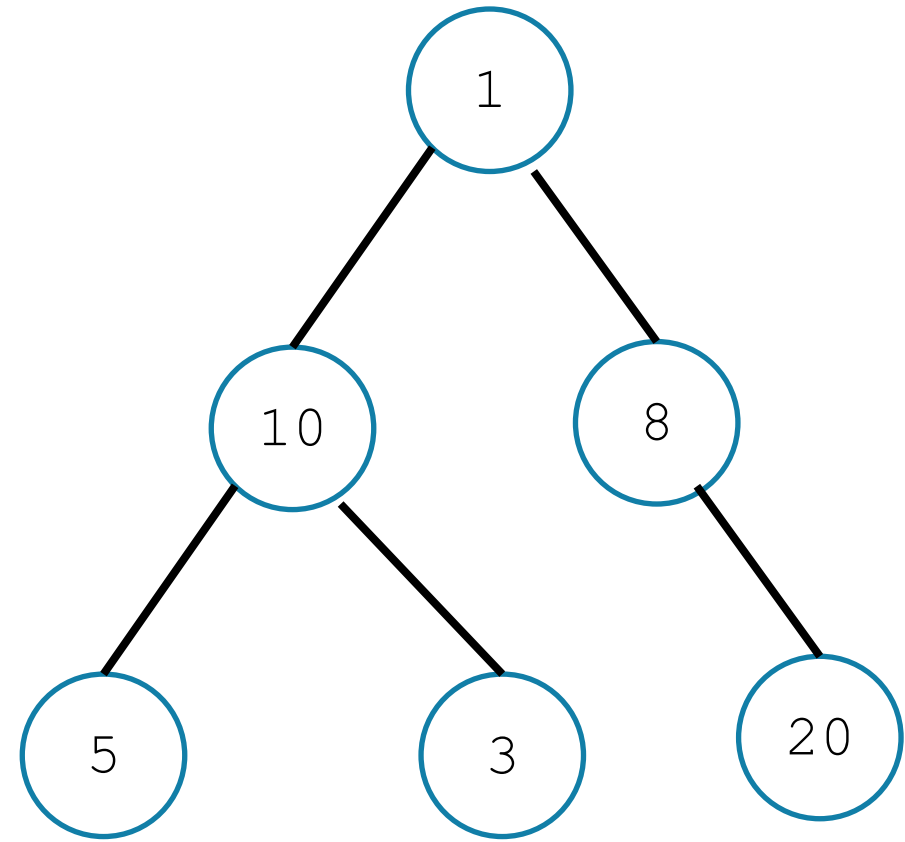
Vertex Cover

Vertex Cover

A set S of vertices is a vertex cover if for every edge (u, v) : u is in S , or v is in S , (or both)

Find the minimum vertex cover in a tree.

Give every **vertex** a weight, find the minimum weight vertex cover



Vertex Cover (1)

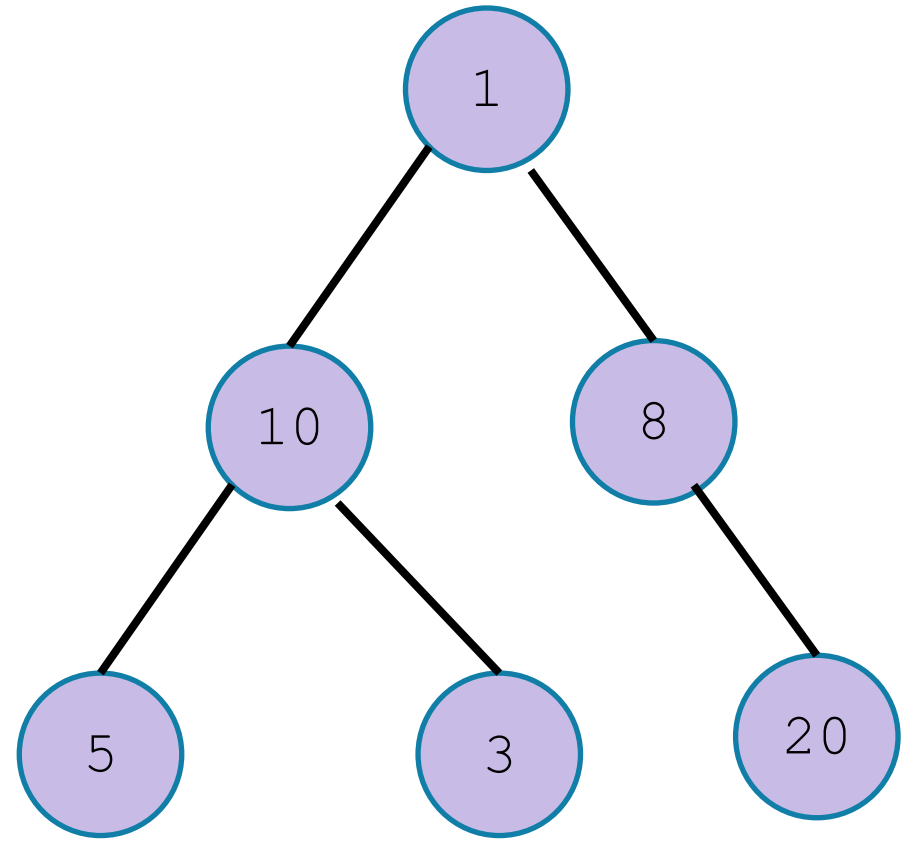
Vertex Cover

A set S of vertices is a vertex cover if for every edge (u, v) : u is in S , or v is in S , (or both)

Find the minimum vertex cover in a tree.

Give every **vertex** a weight, find the minimum weight vertex cover

A valid vertex cover! (just take everything)
Definitely not the minimum though.



Vertex Cover (2)

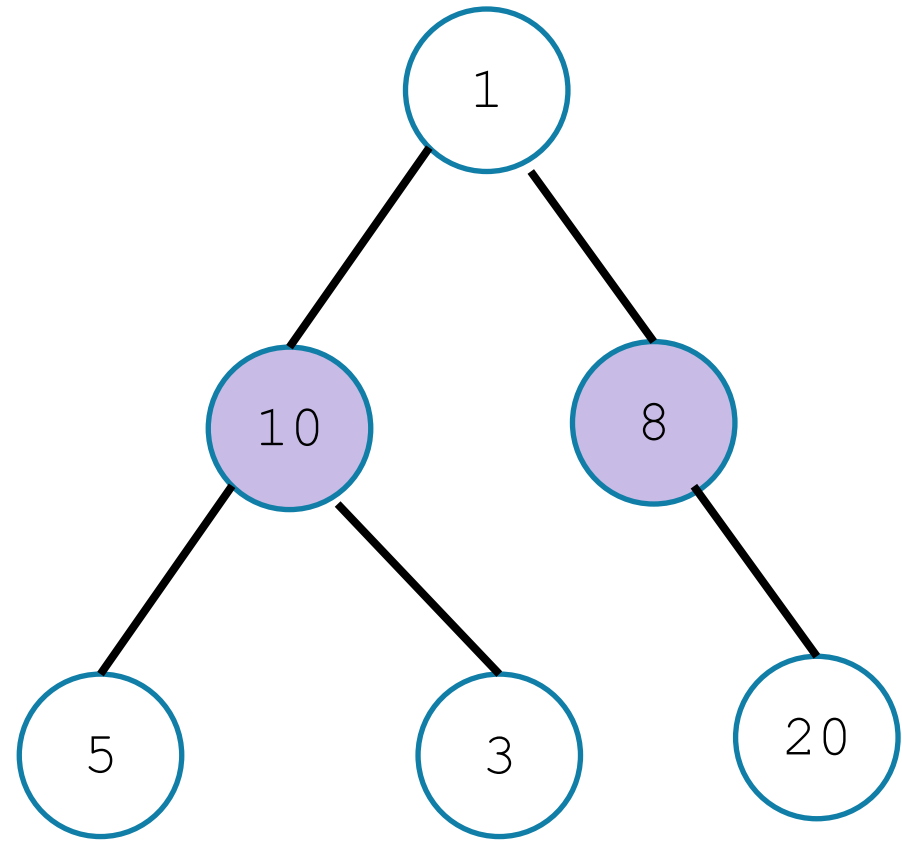
Vertex Cover

A set S of vertices is a vertex cover if for every edge (u, v) : u is in S , or v is in S , (or both)

Find the minimum vertex cover in a tree.

Give every **vertex** a weight, find the minimum weight vertex cover

A better vertex cover – weight 18



Vertex Cover (3)

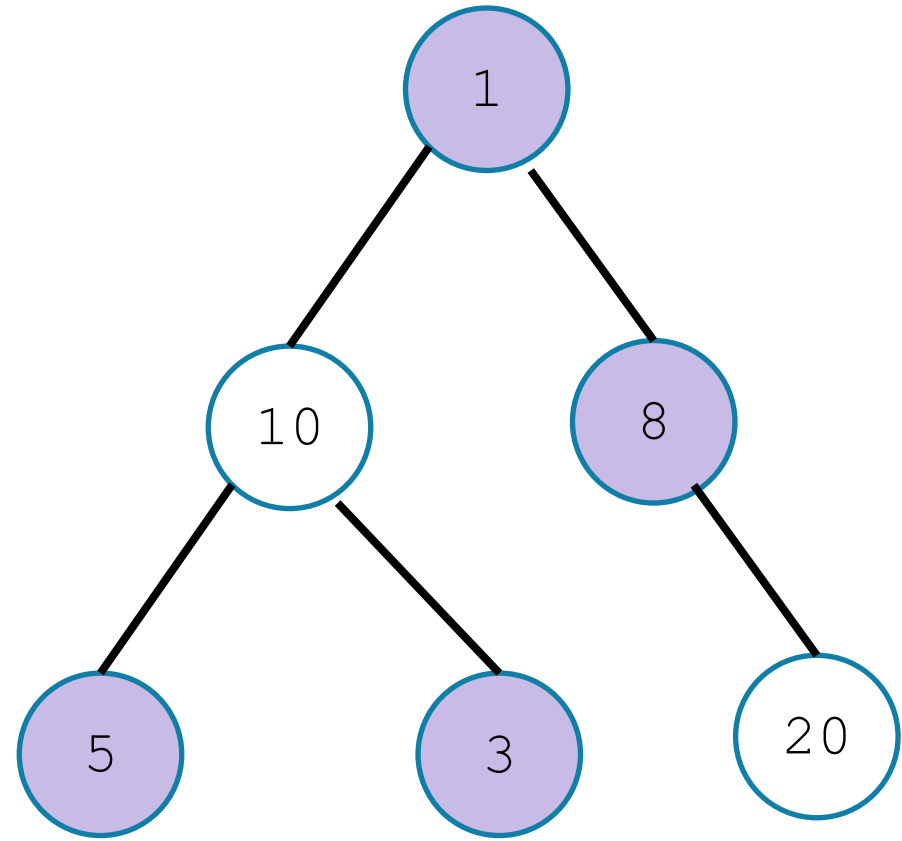
Vertex Cover

A set S of vertices is a vertex cover if for every edge (u, v) : u is in S , or v is in S , (or both)

Find the minimum vertex cover in a tree.

Give every **vertex** a weight, find the minimum weight vertex cover

The minimum vertex cover: weight 17



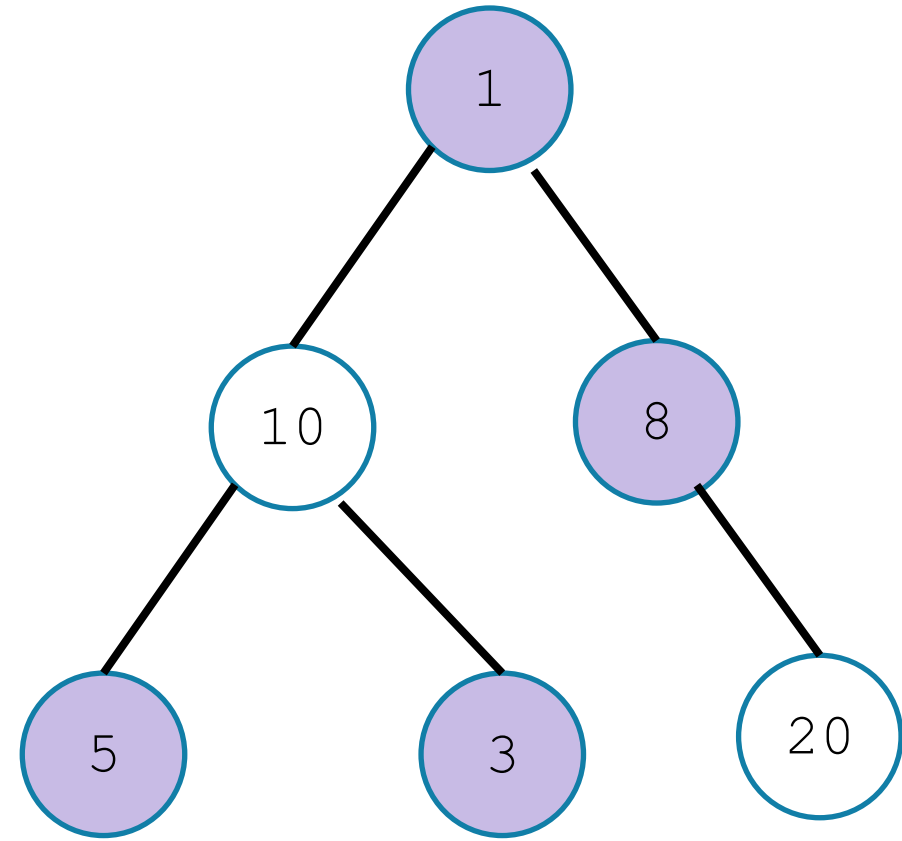
Vertex Cover (4)

Vertex Cover

A set S of vertices is a vertex cover if for every edge (u, v) : u is in S , or v is in S , (or both)

Notice, the minimum weight vertex cover might have both endpoints of some edges

Even though only one of 1, 8 is required on the edge between them, they are both required for other edges.



Vertex Cover – Recursively

Let's try to write a recursive algorithm first.

What information do we need to decide if we include u ?

If we don't include u then to be a valid vertex cover we need...

If we do include u then to be a valid vertex cover we need...

Vertex Cover – Recursively (ans)

Let's try to write a recursive algorithm first.

What information do we need to decide if we include u ?

If we don't include u then to be a valid vertex cover we need...

to include **all** of u 's children, and vertex covers for each subtree

If we do include u then to be a valid vertex cover we need...

just vertex covers in each subtree (whether children included or not)

Recurrence: try to find it

Let $OPT(v)$ be the weight of a minimum weight vertex cover for the subtree rooted at v .

Write a recurrence for $OPT()$

Then figure out how to calculate it

Recurrence

$OPT(v)$ – the weight of the minimum weight vertex cover for the tree rooted at v (whether or not v is included).

$INCLUDE(v)$ – the weight of the minimum weight vertex cover for the tree rooted at v where v is included in the vertex cover.

$$OPT(v) = \begin{cases} \min\{\sum_{u:u \text{ is a child of } v} INCLUDE(u), weight(v) + \sum_{u:u \text{ is a child of } v} OPT(u)\} & \text{if } v \text{ is not a leaf} \\ 0 & \text{if } v \text{ is a leaf} \end{cases}$$

$$INCLUDE(v) = weight(v) + \sum_{u:u \text{ is a child of } v} OPT(u)$$

Vertex Cover Dynamic Program (loose ends, 1)

What memoization structure should we use?

What code should we write?

What's the running time?

Vertex Cover Dynamic Program (loose ends,2)

What memoization structure should we use?

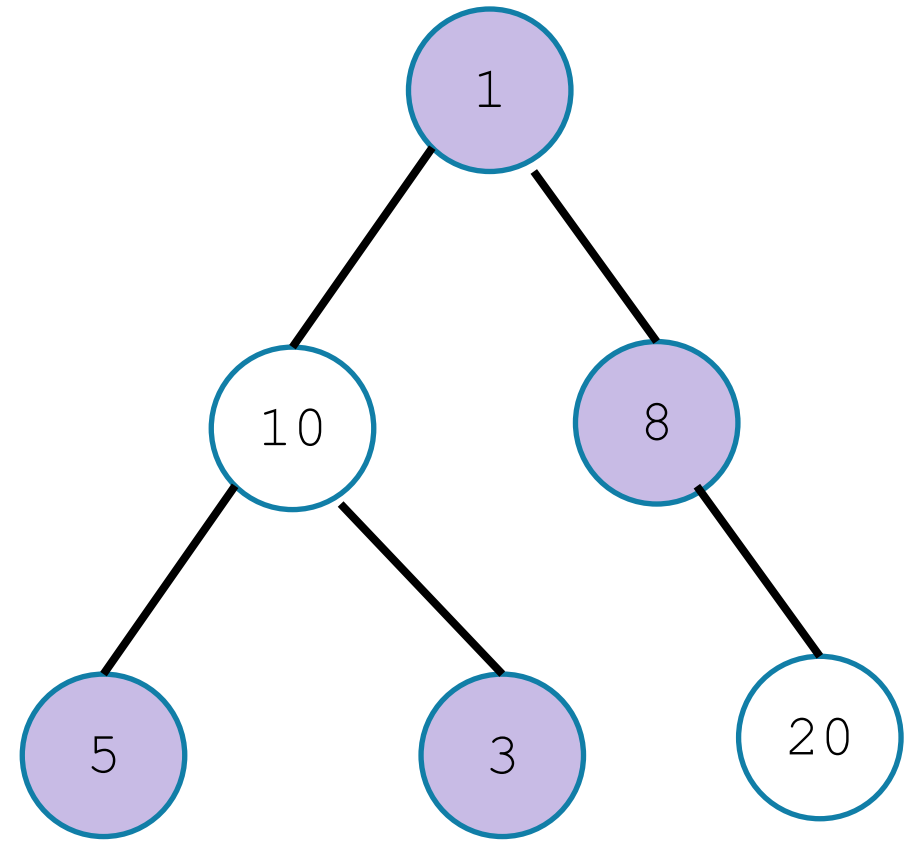
the tree itself!

What code should we write?

What's the running time?

What's our evaluation order?

What order do we do the calculation?



Vertex Cover Dynamic Program (loose ends tied)

What memoization structure should we use?

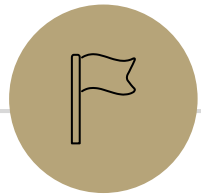
the tree itself!

What code should we write?

A post-order traversal (make recursive calls, then look up values in children to do calculations)

What's the running time?

$\Theta(n)$



Generalizing: DAGs

Coming Up

Dynamic Programming on Graphs

Getting more complicated: Trees → DAGs → General graphs

We're building up to "Bellman-Ford" and "Floyd-Warshall"

Two very clever algorithms.

Tree and DAG DPs are common; general graph DPs are less common, but these are standard library functions, so good to know.

And deriving them together is good for practicing DP skills.

Shortest Paths

Shortest Path Problem

Given: A directed graph and a vertex s

Find: The length of the shortest path from s to t .

The length of a path is the sum of the edge weights.

Baseline: Dijkstra's Algorithm

Dijkstra's Algorithm

```
Dijkstra(Graph G, Vertex source)
  initialize distances to  $\infty$ 
  mark source as distance 0
  mark all vertices unprocessed
  while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    foreach(edge (u,v) leaving u){
      if(u.dist+weight(u,v) < v.dist){
        v.dist = u.dist+weight(u,v)
        v.predecessor = u
      }
    }
    mark u as processed
  }
```

In 332, we said the running time was $O(m \log n + n \log n)$

Can be sped up to $O(m + n \log n)$ by inserting a different heap implementation.

A recurrence

Suppose you have a directed acyclic graph G .

How could you find distances from s ?

What's one step in this problem?

A recurrence

Suppose you have a directed acyclic graph G .

How could you find distances from s ?

What's one step in this problem?

Choosing the predecessor, i.e. "the last edge" on a path.

A recurrence

$$\mathit{dist}(v) = \begin{cases} 0 & \text{if } v \text{ is the source} \\ \min_{u:(u,v) \in E} \{\mathit{dist}(u) + \mathit{weight}(u, v)\} & \text{otherwise} \end{cases}$$

Our memoization structure can be the graph itself.

What's an evaluation order? (Remember we're in a DAG!)

A recurrence

$$\mathit{dist}(v) = \begin{cases} 0 & \text{if } v \text{ is the source} \\ \min_{u:(u,v) \in E} \{\mathit{dist}(u) + \mathit{weight}(u,v)\} & \text{otherwise} \end{cases}$$

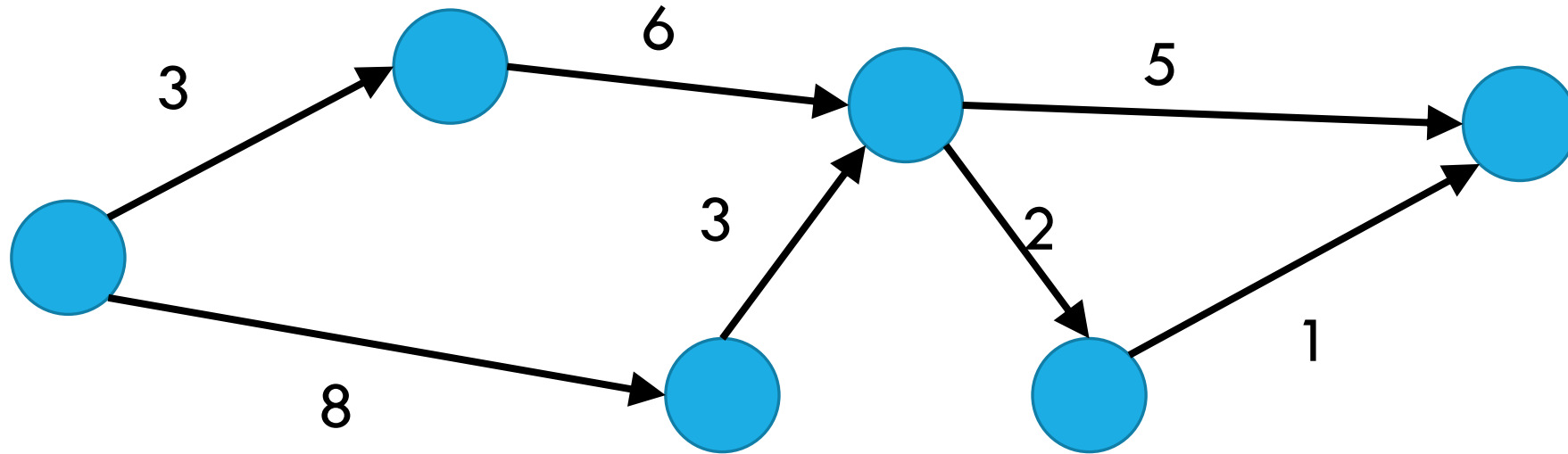
Our memoization structure can be the graph itself.

What's an evaluation order? (Remember we're in a DAG!)

A topological sort! – we need to have distances for all incoming edges calculated.

In a DAG

$$\text{dist}(v) = \begin{cases} 0 & \text{if } v \text{ is the source} \\ \min_{u:(u,v) \in E} \{\text{dist}(u) + \text{weight}(u,v)\} & \text{otherwise} \end{cases}$$



What about cycles?

$$\text{dist}(v) = \begin{cases} 0 & \text{if } v \text{ is the source} \\ \min_{u:(u,v) \in E} \{\text{dist}(u) + \text{weight}(u,v)\} & \text{otherwise} \end{cases}$$

