



# DP Practice!

CSE 417 22AU  
Lecture 16

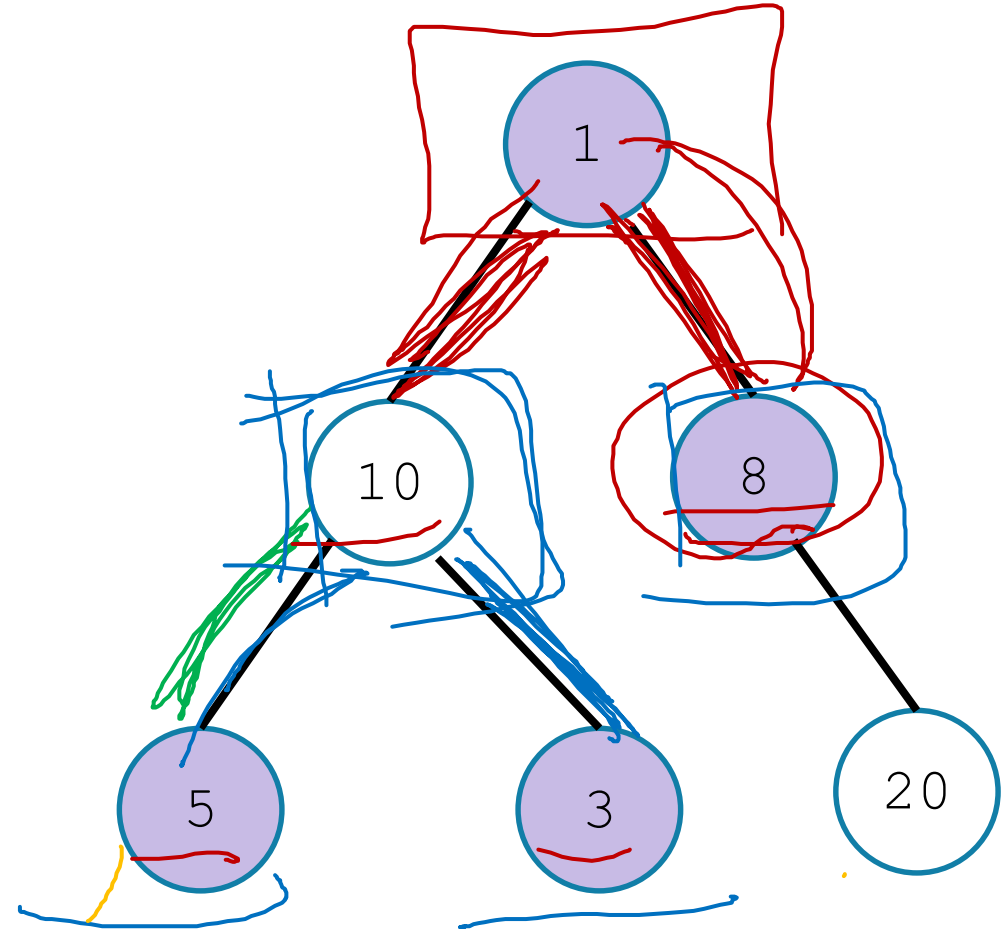
# Vertex Cover

## 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)

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.



# Step 1: Formulate your problem recursively

OPT(???)

Try top down and bottom up thinking.

(Top down: For one particular "element" of the input, what do I need to decide? What are all the possibilities?

(Bottom up: What is a smaller version of the problem?

What information do we need to solve the problem recursively?

# 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

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)

## Step 2: What do you need recursively

Write in English (not math. English.) what exactly the recursive call is giving you.

I know this sounds silly. Every time I am trying to solve a DP problem from scratch (or just remember how one worked) and can't do it, it's because I haven't written it down in English.

If you realize "I need an extra parameter" update the English description with that extra parameter.

# Recurrence



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

For this recurrence we are ignoring any edge from  $v$  to a parent (if one exists).

Write a recurrence for  $OPT()$

Then figure out how to calculate it.

Hint: you need to change or add something.

$$OPT(v) =$$

min

$$\left\{ \begin{array}{l} \sum OPT(u) + w(v) \\ \sum OPT(u) \end{array} \right.$$

# Recurrence

$OPT(\text{leaf})$

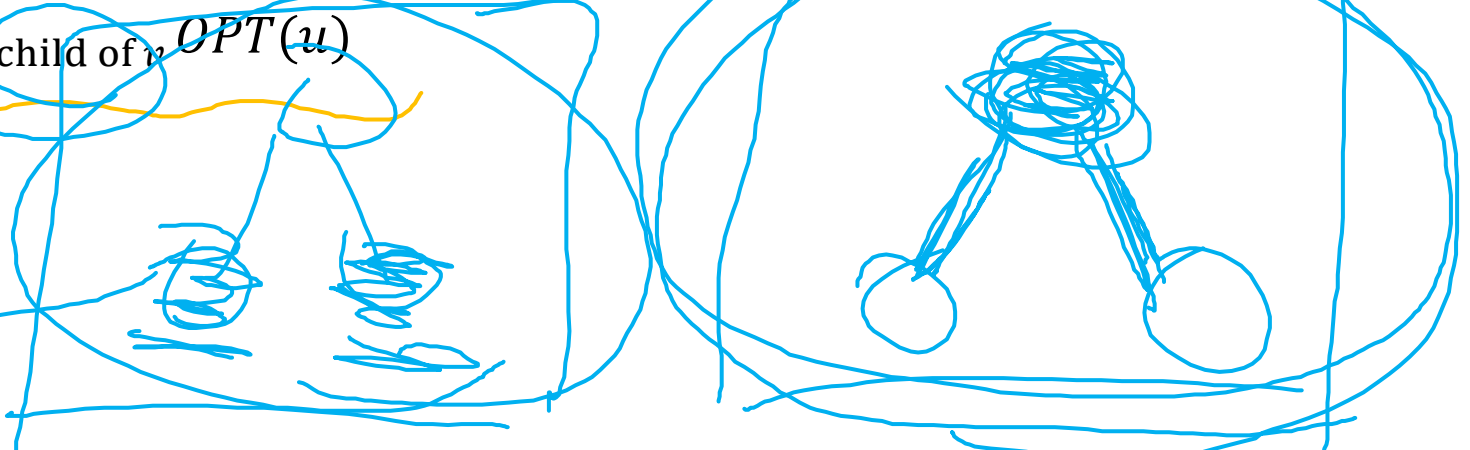
0

$OPT(v)$  – 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

What memoization structure should we use?

What code should we write?

What's the running time?

# Vertex Cover Dynamic Program

What memoization structure should we use?

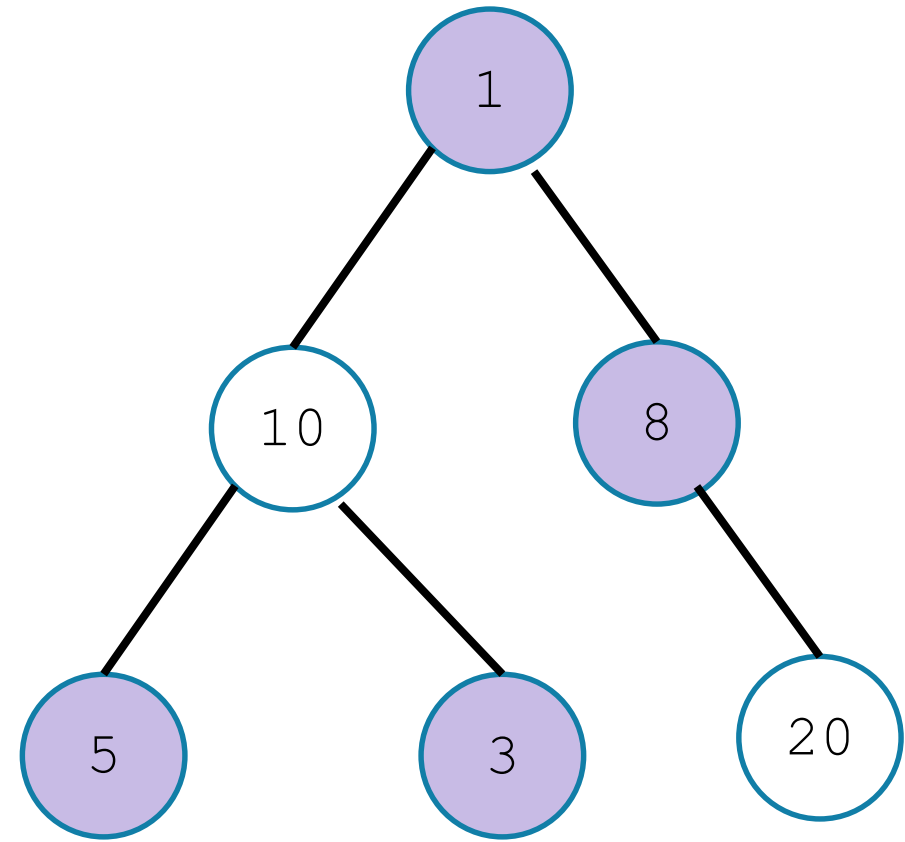
the tree itself!

What code should we write?

What's the running time?

# Vertex Cover

What order do we do the calculation?



# Vertex Cover Dynamic Program

What memoization structure should we use?

the tree itself!

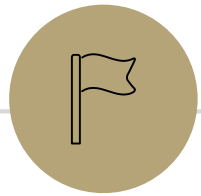
What code should we write?

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

What's the running time?

$\Theta(n)$





**More Practice**

---

# Longest Increasing Subsequence

0	1	2	3	4	5	6	7
5	-6	3	6	-5	2	8	10

Longest set of (not necessarily consecutive) elements that are increasing

5 is optimal for the array above  
(indices 1,2,3,6,7; elements -6,3,6,8,10)

Try it again!  
But mirrored

For simplicity – assume all array elements are distinct.

# Recurrence

0		1	2	3	4	5	6	7
5	-6	3	6	-5	2	8	10	
Parent recursive calls		Current $i$	Recursive call is best value in this area					

Need recursive answer to the right

Currently processing  $i$

Recursive calls to the right are needed to know optimum from  $i \dots n$

Will move  $i$  to the left in our iterative algorithm

# Recurrence

0	1	2	3	4	5	6	7
5	-6	3	6	-5	2	8	10
Parent recursive calls			Recursive call is best value in this area				

[pollev.com/robbie](http://pollev.com/robbie)

Try to write a different recurrence for longest increasing subsequence.

$LIS_{Alt}(i, j)$  is "Number of elements of the maximum increasing subsequence from  $i, \dots, n$  where smallest element of the sequence is  $A[j]$ "

# Longest Increasing Subsequence

Think left-to-right instead of right-to-left

$LISAlt(i, j)$  is "Number of elements of the maximum increasing subsequence from  $1, \dots, n$  where smallest element of the sequence is  $A[j]$ "

$$LISAlt(i, j) = \begin{cases} 0 & \text{if } i > n \text{ or } j > n \\ LISAlt(i + 1, j) & \text{if } A[i] < A[j] \\ \max\{1 + LISAlt(i + 1, i), LISAlt(i + 1, j)\} & \text{if } A[i] \geq A[j] \text{ o/w} \end{cases}$$

# Longest Increasing Subsequence

$LISAlt(i, j)$  is "Number of elements of the maximum increasing subsequence from  $1, \dots, n$  where smallest element of the sequence is  $A[j]$ "

$$LISAlt(i, j) = \begin{cases} 0 & \text{if } i > n \text{ or } j > n \\ LISAlt(i + 1, j) & \text{if } A[i] > A[j] \\ \max\{1 + LISAlt(i + 1, i), LISAlt(i + 1, j)\} & \text{o/w} \end{cases}$$

Memoization structure?  $n \times n$  array.

Filling order? Multiple possible

Outer loop:  $i$  from  $n - 1$  to  $0$

Inner loop:  $j$  from  $n - 1$  to  $0$

# How do you choose your base cases?

Ask “what’s the smallest my problem could be?”

Usually, 0 or 1 elements. Ask “what’s the answer for this tiny problem?”

**Then** check against the English description, is it right?

**And then** run it on a small example, make sure it ‘fits right’ with the recursive definition you’re using.

Finally ask “what about edge/invalid cases?”

Baby Yoda could walk off the map. Usually choose the value that “is never chosen recursively” ( $\infty$  for taking a min,  $-\infty$  for taking a max)

There’s multiple options!!

Usually “empty” or “one thing left” are both options.

# Summing Up

The two recurrences have the same idea (add/don't add, record the end of the array closest to your next decision)

But thinking left-to-right vs. right-to-left

Both end up with an  $n \times n$  memoization structure (both of which could be cut down  $O(n)$  memory if needed)

And  $O(n^2)$  running time.

# But Wait! There's more

Another recurrence at the end of these slides for more practice.

Instead of thinking "do I include this element or not?" for each element,  
Ask "what's the next element" or equivalently "what's the longest  
subsequence starting from me"

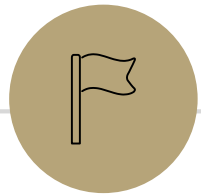
Get a different recurrence, but not a better running time.

# Takeaways

When designing a dynamic program, we sometimes need to introduce a second variable, that doesn't appear in the program

Or a second recurrence that mixes with the first if other decisions affect what's optimal (beyond which problem you look at)

There might be more than one program available.



## Extra Practice



# Subset Sum

Given an array  $A[]$  of positive integers, and a number  $t$  find whether there is a subset of  $A[]$  that sums to exactly  $t$ .

0	1	2	3	4	5	6	7
5	6	3	6	5	2	8	10

If  $t = 30$ , answer is "yes" (for example,  $5 + 5 + 2 + 8 + 10$ )

If  $t = 100$ , answer is "no" (not allowed to repeat elements beyond the number of copies in the array, e.g. can't say "10 copies of 10")

# Subset Sum

Write an English description of what you want to calculate

Write a recurrence

Give a sentence or two (in English) of why your recurrence should work.

# Subset Sum

Write an English description of what you want to calculate

Let  $SUBSUM(i, t)$  be true if and only if a subset of  $A[0], \dots, A[i]$  can sum to  $t$ .

Write a recurrence

$$SUBSUM(i, t) = \begin{cases} True & \text{if } t = 0 \\ False & \text{if } i < 0 \text{ and } t \neq 0 \\ SUBSUM(i - 1, t) \text{ || } SUBSUM(i - 1, t - A[i]) & \text{o/w} \end{cases}$$

Give a sentence or two (in English) of why your recurrence should work.

Element  $i$  is either included or it isn't – if  $i$  appears in a valid subset, then we need to have the remaining elements sum to  $t - A[i]$ . If  $i$  doesn't appear then the remaining elements will get to  $t$ . We "or" together because either could be a valid path to getting the right sum.

# Subset Sum

What memorization structure will you use?

A 2D Boolean array  $SUBSUM(i, j)$ . Array will be  $n \times T$

Write the pseudocode to fill up the structure iteratively.

```
SubSum(int[] A, int T)
Bool[][] SubSum = new Bool[n][T+1]
for(int j=0; j<T+1; j++){ SubSum[0][j]=False;}
SubSum[0][A[0]]=True;
for(int i=1; i<n; i++){
    for(int j=0; j<T+1; j++){
        if(SubSum[i-1][j]){
            SubSum[i][j]=True;
            SubSum[i][j+A[i]]=True; //need to catch Array index errors. Don't do
                                   //this in real code.
        }
    }
}
return SubSum[n][T-1];
```

# Longest Increasing Subsequence, Round 3

Let's ask "what's the best choice for the next element" (instead of just "is this the next element")

What's the best choice?

It has to be greater than our current element, after that it's the one that can lead to the longest subsequence.

So, (since we're starting with our current element), the question is "what's the longest increasing subsequence, starting at index  $i$ "

# Longest Increasing Subsequence, Round 3

Let  $LISStart(i)$  be the length of the longest increasing subsequence among indices  $i \dots n$ , that starts at index  $i$ .

Call an index "valid" if  $A[j] > A[i]$  (it's "valid" to add  $j$  to a sequence starting at  $i$ )

$$LISStart(i) = \max\{1, \max_{j: j \text{ is valid and } j > i} \{LISStart(j)\} \text{ if } i \leq n\}$$

i.e. have a single entry (yourself) or prepend yourself to the longest subsequence starting after you (that you can prepend yourself to)

# Longest Increasing Subsequence, Round 3

Memoization? 1D array of size  $n$

Iteration? Outer-loop:  $i$  decreasing

Inner-loop: calculate  $LISStart(i)$  by iterating over previous calculations.

Checking  $n$  values for each new calculation, not  $O(1)$

Still  $O(n^2)$  time.

**Be careful!**

Final answer is **not**  $LISStart(i)$ .

It's the maximum entry among  $LISStart()$  array