Activity slide is up

main slides will be connected in a few minutes

for now add a hyphen in URL between dp and 4

e.g. /14-dp-4.pdf

Reminders

- HW 3 due tonight (written)
- HW 4 out tonight

Dynamic Programming on Trees

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 $-6,3,6,8,10$)

For simplicity – assume all array elements are distinct.

Longest Increasing Subsequence

What do we need to know to decide on element i?

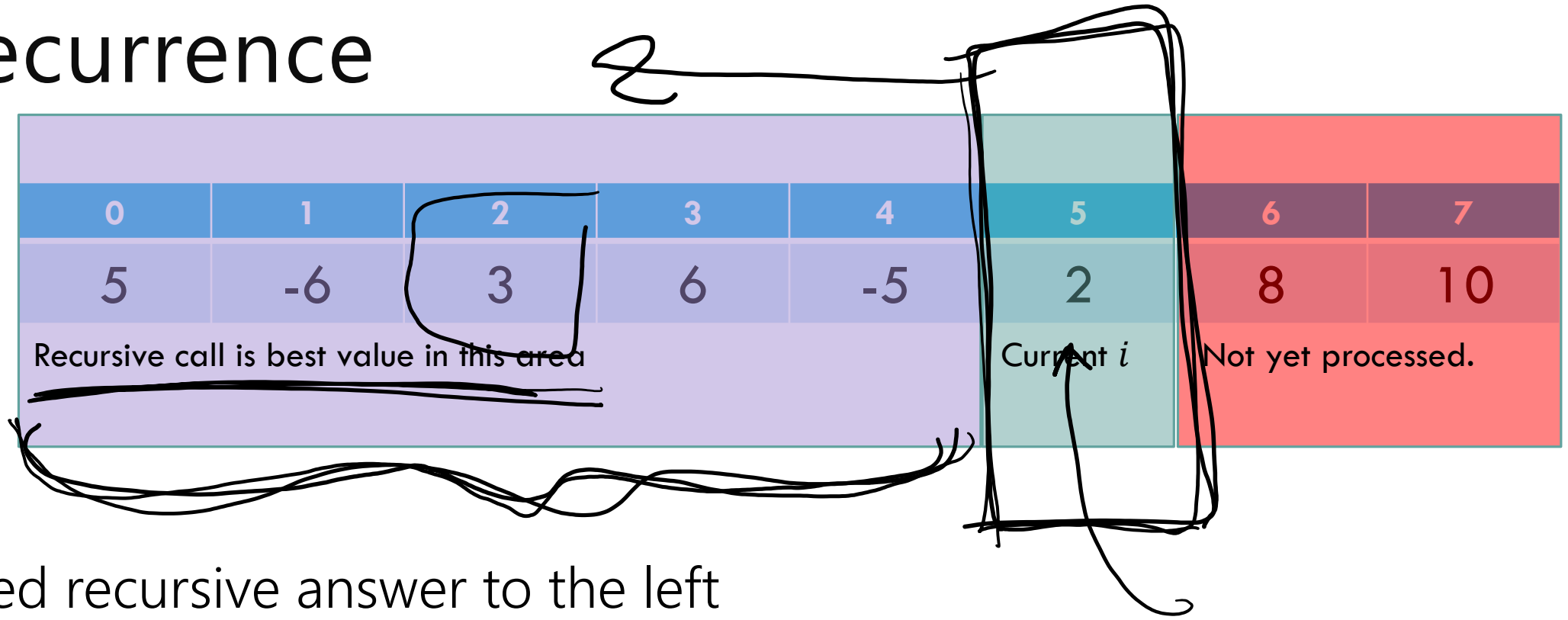
Is it allowed?

Will the sequence still be increasing if it's included?

Still thinking right to left --

Two indices: index we're looking at, and index of upper bound on elements (i.e. the value we need to decide if we're still increasing).

Recurrence



Need recursive answer to the left

Currently processing i

Recursive calls to the left are needed to know optimum from $1 \ldots i$

Will move i to the right in our iterative algorithm

Longest Increasing Subsequence

$LIS(i, j)$ is "Number of elements of the maximum increasing subsequence from $1, \ldots, i$ where every element of the sequence is at most $A[j]$"

Need a recurrence

$$LIS(i,j) = \begin{cases} 0 & \text{if } i < 0 \\ \mathbb{I}[A[i] \leq A[j]] & \text{if } i = 0 \\ LIS(i-1,j) & \text{if } A[i] > A[j] \\ \max\{1 + LIS(i-1,i), LIS(i-1,j)\} & \text{otherwise} \end{cases}$$

If $A[i] > A[j]$ element i cannot be included in an increasing subsequence where every element is at most $A[j]$. So taking the largest among the first $i-1$ suffices.

If $A[i] \leq A[j]$, then if we include i, we may include elements to the left only if they are less than $A[i]$ (since $A[i]$ will now be the last, and therefore largest, of elements $1 \ldots i$. If we don't include i we want the maximum increasing subsequence among $1 \ldots i-1$.

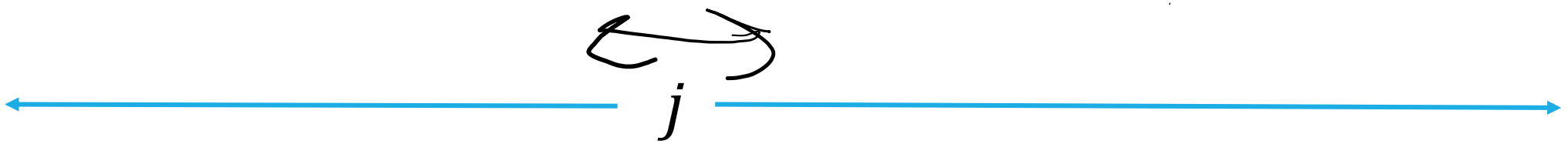
Longest Increasing Subsequence

$$LIS(i,j) = \begin{cases} 0 & \text{if } i < 0 \\ \mathbb{I}[A[i] \le A[j]] & \text{if } i = 0 \\ LIS(i-1,j) & \text{if } A[i] > A[j] \\ \max\{1 + LIS(i-1,i), LIS(i-1,j)\} & \text{otherwise} \end{cases}$$

Memoization structure? $n \times n$ array.

Filling order?

LIS



	0, 5	1, −6	2, 3	3, 6	4, −5	5, 2	6, 8	7, 10
0, 5								
1, −6								
2, 3								
3, 6			(3,2)		(3,4)			
4, −5			(4,2)					
5, 2								
6, 8								
7, 10								

i

LIS

$LIS(1,0)$ $A[1] < A[0]$ not allowed:
Take $LIS(0,0)$

j →

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

i ↓

LIS

$LIS(1,1)$ $A[1] \geq A[1]$ can add, $1 + LIS(0,1)$ or $LIS(0,1)$

j →

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

i ↓

LIS

$LIS(1,2)$ $A[1] \leq A[2]$ allowed to add:
$1 + LIS(0,1)$ or $LIS(0,2)$

j →

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

i ↓

LIS

$LIS(2,0)$ $A[2] \leq A[0]$ allowed to add
$1 + LIS(1,2)$ or $LIS(1,0)$

j →

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

i ↓

LIS

j

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

i

pseudocode

```
//real code snippet that actually generated the table on the last slide
for(int j=0; j < n; j++){
     vals[0][j] = (A[0] <= A[j]) ? 1 : 0;
}
for(int i = 1; i < 8; i++){
     for(int j = 0; j < n; j++){
          if(A[i] > A[j])
               vals[i][j] = vals[i-1][j];
          else{
               vals[i][j] = Math.max(1+vals[i-1][i], vals[i-1][j]);
          }
     }
}
```

Longest Increasing Subsequence

$$LIS(i,j) = \begin{cases} 0 & \text{if } i < 0 \\ \mathbb{I}[A[i] \leq A[j]] & \text{if } i = 0 \\ LIS(i-1,j) & \text{if } A[i] > A[j] \\ \max\{1 + LIS(i-1,i), LIS(i-1,j)\} & \text{otherwise} \end{cases}$$

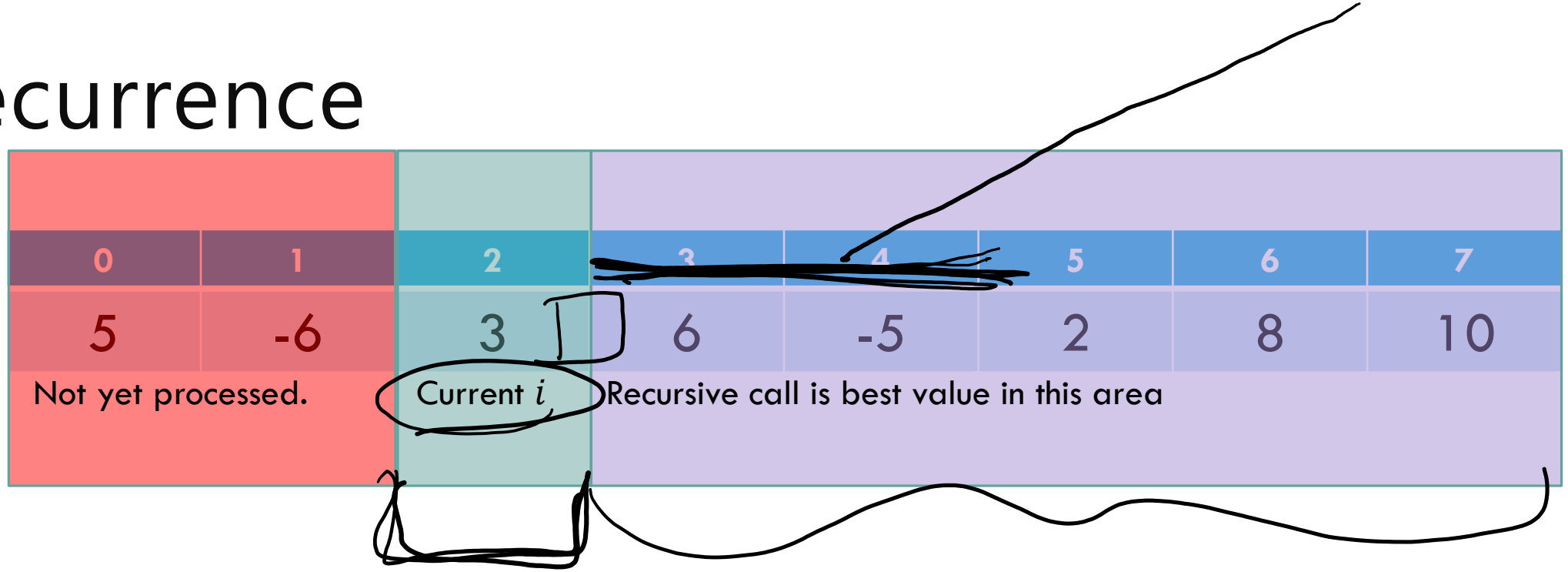
Memoization structure? $n \times n$ array.

Filling order?

Outer loop: increasing i

Inner loop: increasing j

Recurrence



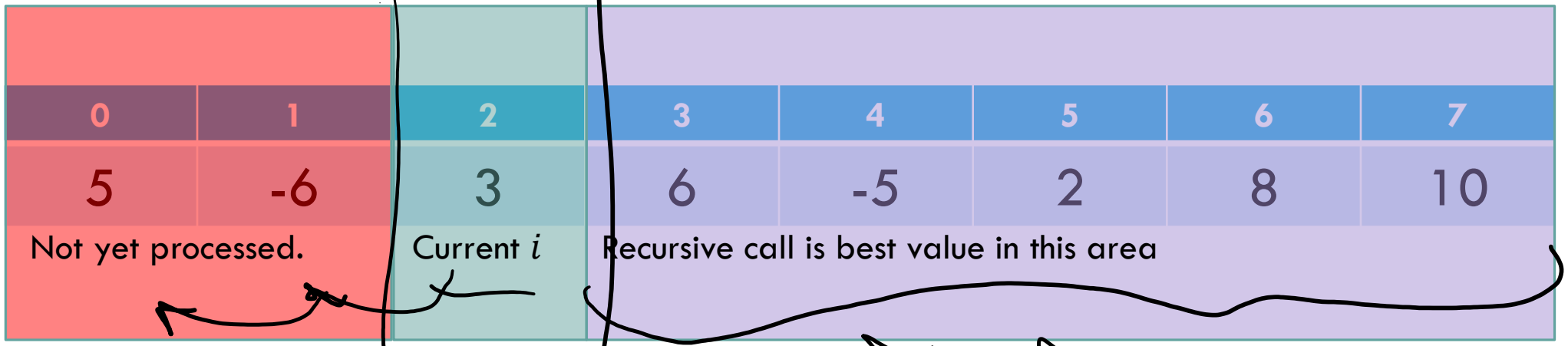
Need recursive answer to the right

Currently processing i

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

Will move i to the left in our iterative algorithm

Recurrence



Fill out the poll everywhere for Activity Credit!
Go to pollev.com/cse417 and login with your UW identity

Try to write a different recurrence for longest increasing subsequence.

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 $i, \ldots, n$ where smallest element of the sequence is $A[i]$"

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

Longest Increasing Subsequence

$LISAlt(i, j)$ is "Number of elements of the maximum increasing subsequence from $i, \ldots, n$ 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 0 to $n - 1$

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

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"

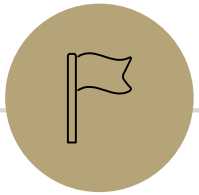
Get a different recurrence, but not a better running time.

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 $A[0], \ldots, A[i]$ can sum to t.

Write a recurrence

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

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 $i \ldots n$, 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

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

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.

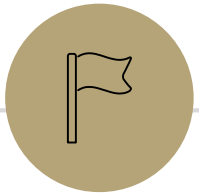
Checking n values for each new calculation, not $O(1)$

Still $O(n^2)$ time.

Be careful!

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

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



DP on Trees

DP on Trees

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.

DP on 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)

"vertices cover all the edges"

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.

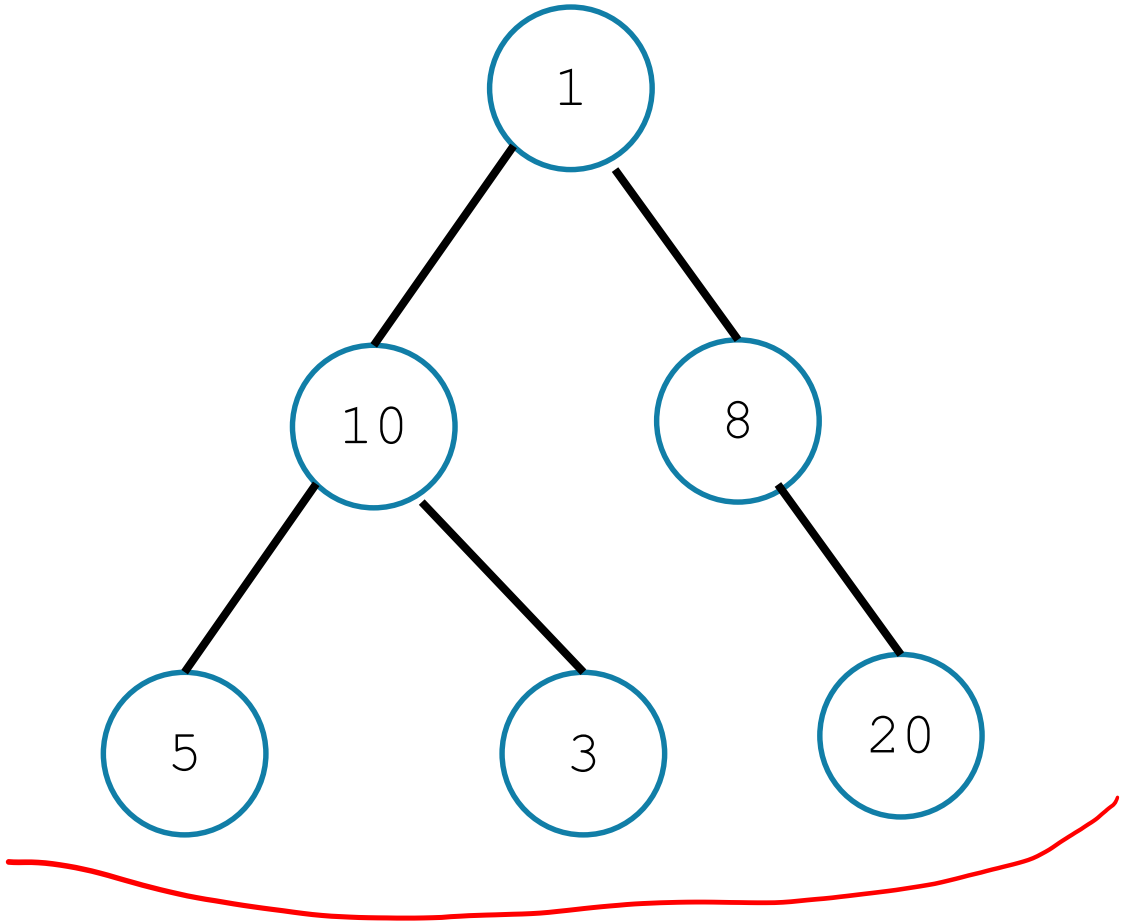
Vertex Cover

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



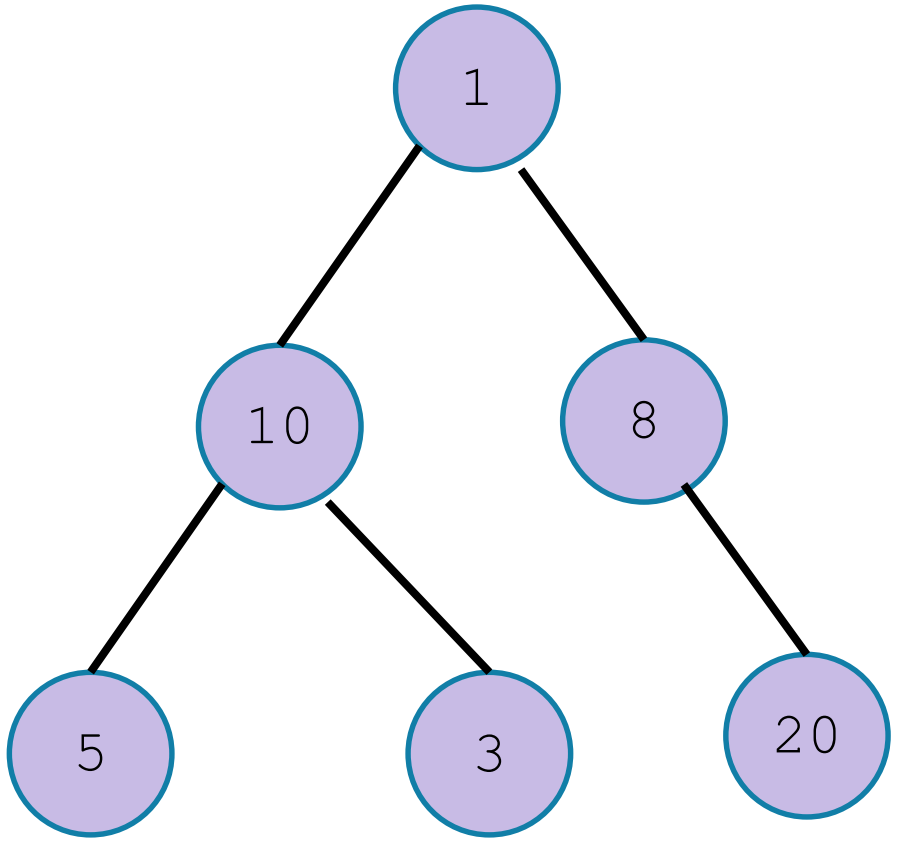
Vertex Cover

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

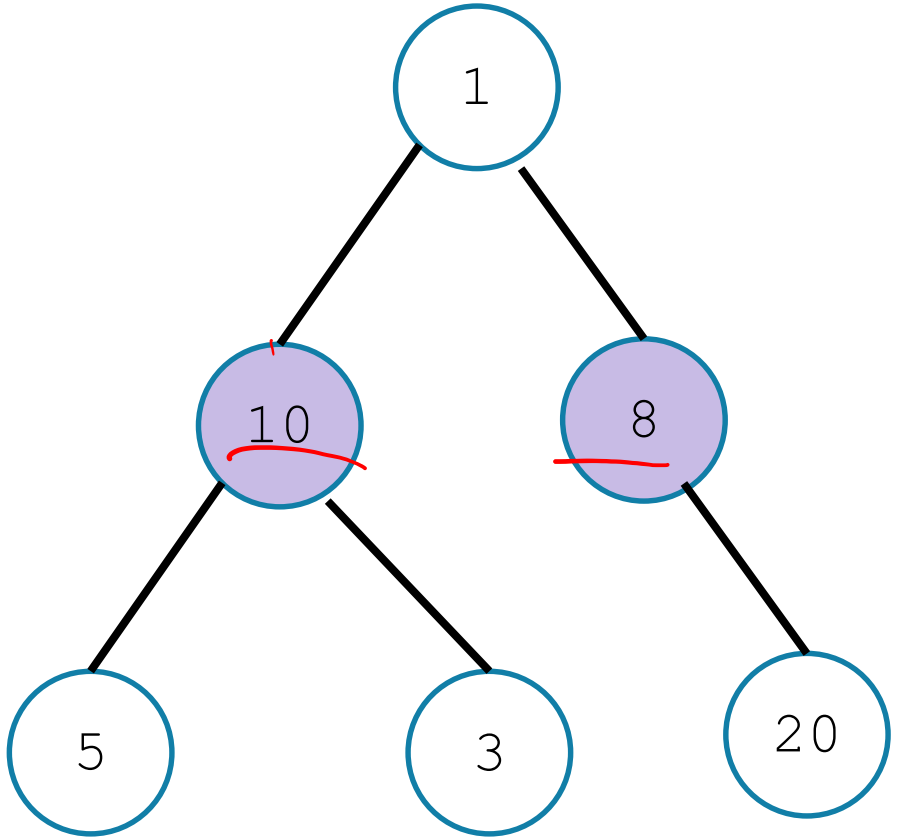
Vertex Cover

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

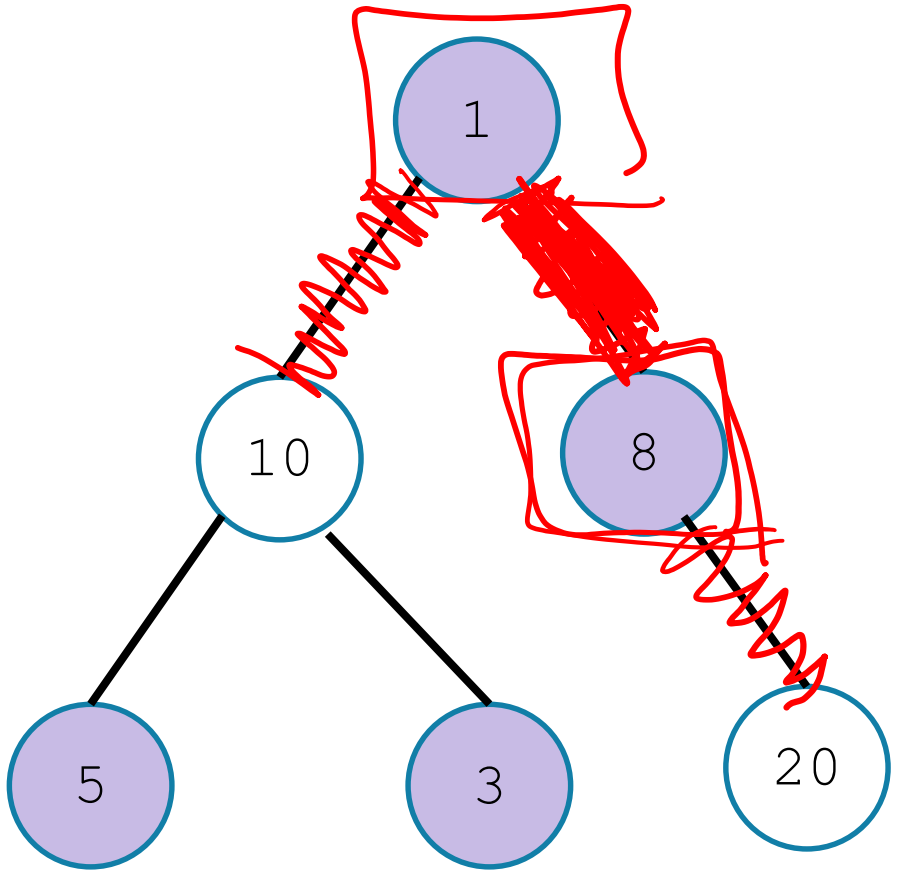
Vertex Cover

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

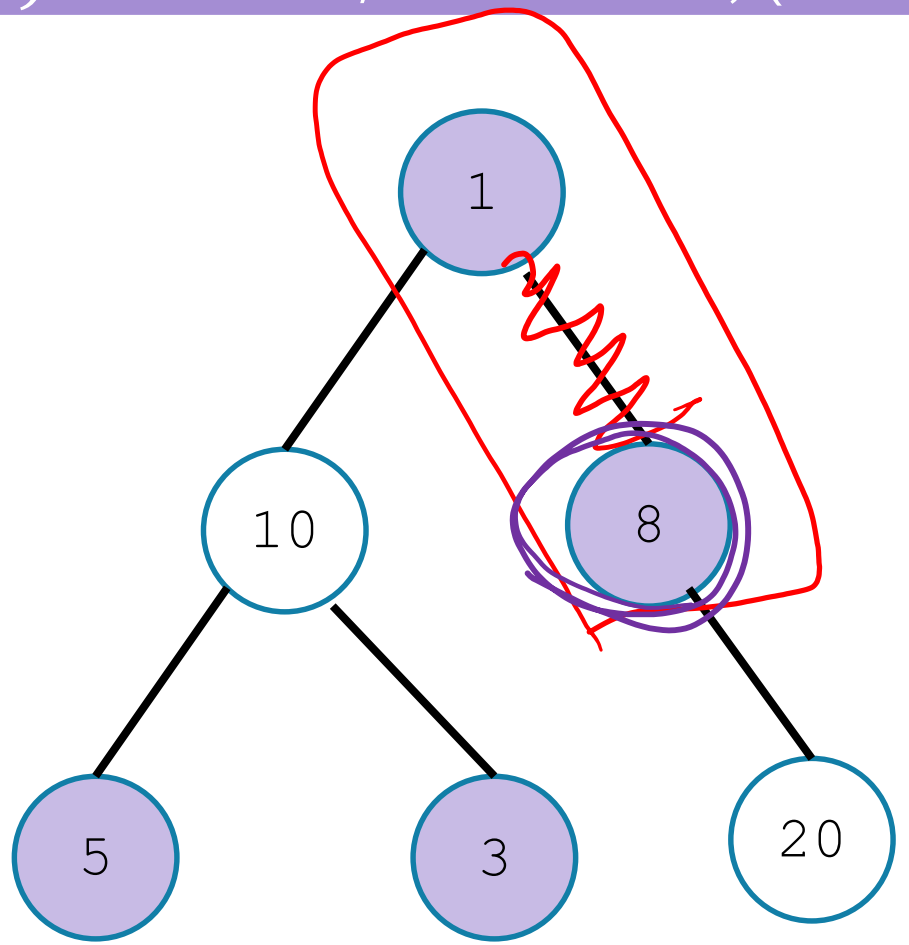
Vertex Cover

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.

Also an indication that greedy probably won't work!



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)

Recurrence

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

What memoization structure should we use?

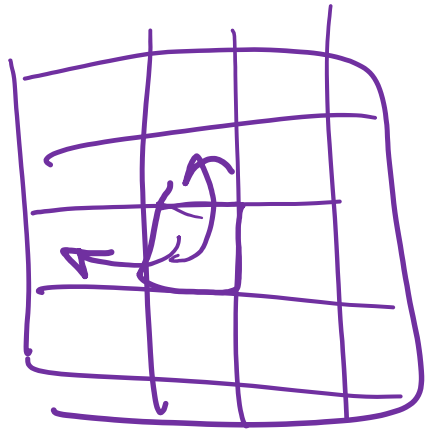
What code should we write?

What's the running time?

Vertex Cover Dynamic Program

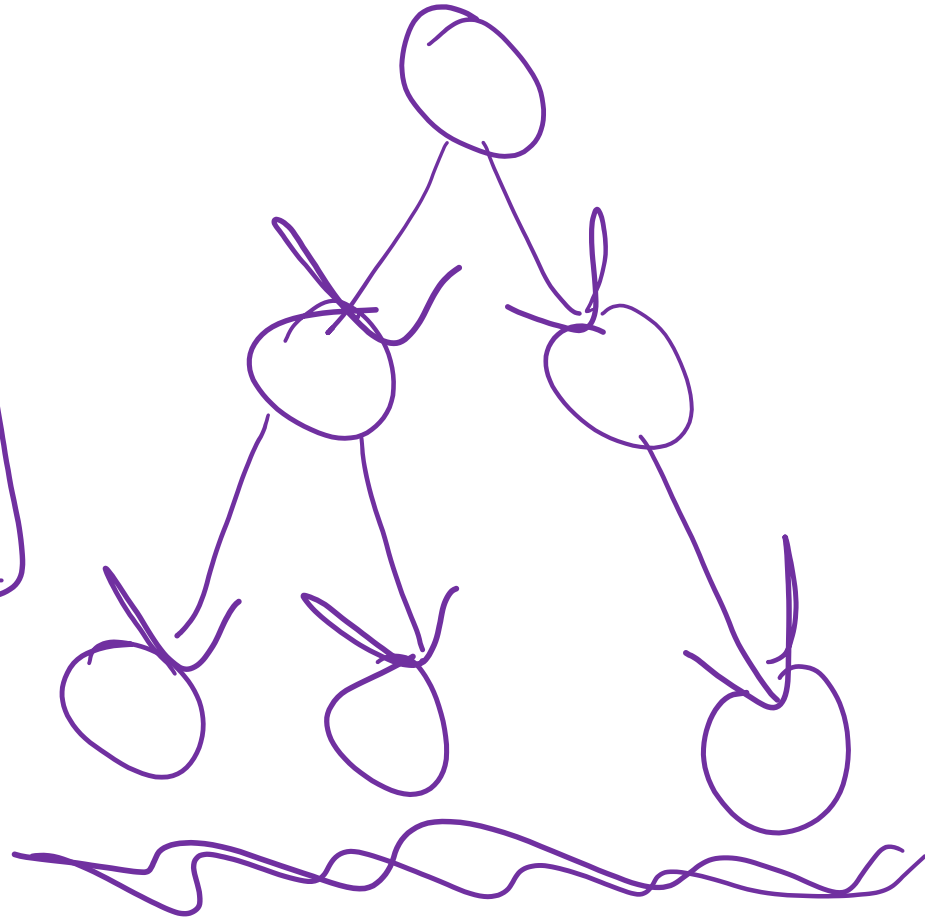
What memoization structure should we use?

the tree itself!



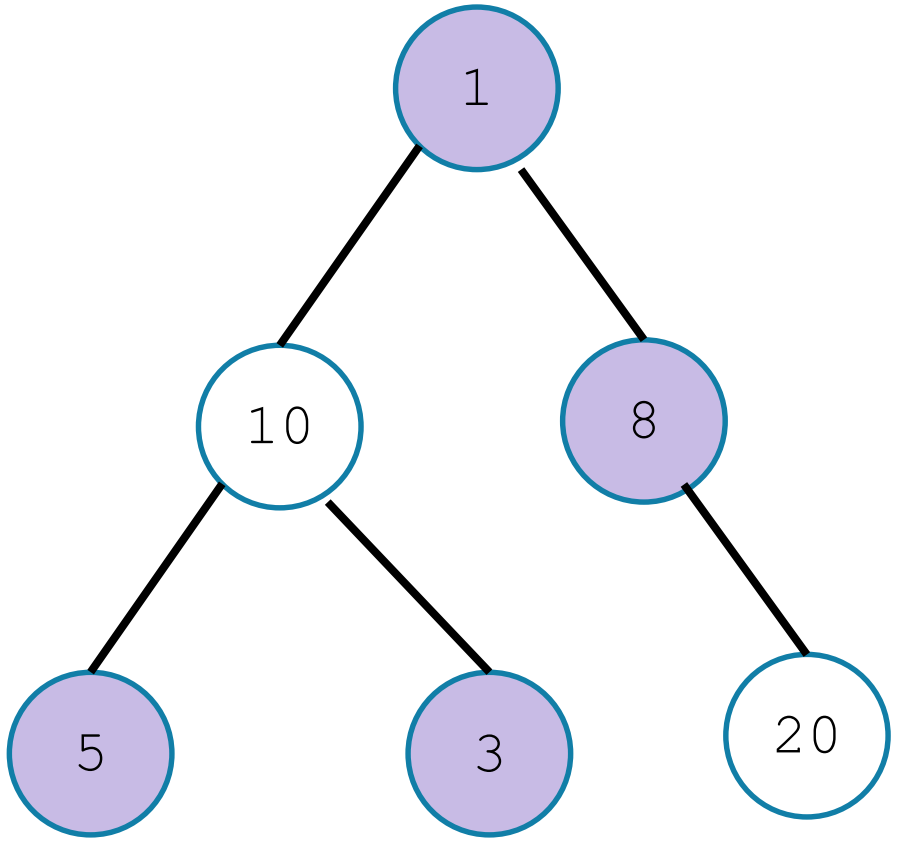
What code should we write?

What's the running time?



Vertex Cover

What order do we do the calculation?



Vertex Cover Dynamic Program

What memoization structure should we use?

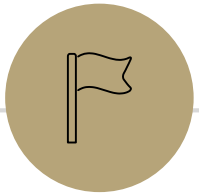
the tree itself!

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?

$\Theta(n)$



DP Context

DP Design Notes

We haven't done a single proof for DP…

We won't ask you to do one.

DP proofs are almost always just "the code does the recurrence"

But that just moves the correctness question – why is the recurrence correct?

And the proof of the recurrence being correct is almost always "I included all the cases"

I'd rather you focus on checking it than trying to explain it.

DP history

So…why is it called “dynamic programming?”

“programming” is an old-timey meaning of the word.

It means “scheduling”

Like a conference has a “program” of who speaks where when.
Or a television executive decides on the nightly programming (what show airs when).

DP history

So…*dynamic*?

The phrase “dynamic programming” was popularized by Richard Bellman (we’ll see one of his algorithms on Monday)

He was a researcher, funded by the U.S. military….

But the Secretary of Defense [as Bellman tells it] hated research. And hated math even more.

So Bellman needed a description of his research that everyone

DP history

Dynamic

Is actually an accurate adjective – what we think is the best option (include/exclude) can change over time.

Even better

“It’s impossible to use the word ‘dynamic’ in a pejorative sense”

“It was something not even a Congressman could object to.”

Next week

Dynamic Programming on General graphs

Linear Programming