CS 331, Fall 2024

Lecture 5 (9/11)

Today:
- Scheduling
- Longest increasing subsequence
- Subset sum

Scheduling (Part III, Section 3.1)

Input: L is n tuples in $\mathbb{R}^2$

$[l_i, r_i]$ with $l_i < r_i$ $\forall i \in [n]$

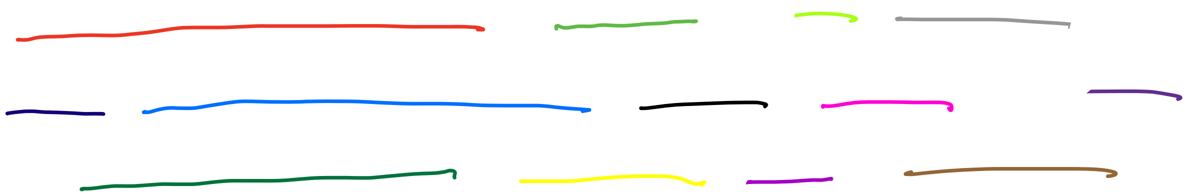
endpoints of i^{th} interval

Output: Maximum $|S|$ for nonoverlapping $S \subseteq [n]$

i.e. $\forall i \neq j \in [n]$, $[l_i, r_i] \cap [l_j, r_j] = \emptyset$

Not OK OK

$\boxed{\text{Example}}$



Q: How many can we schedule?

It is not so easy.

Naive algo: "try everything"

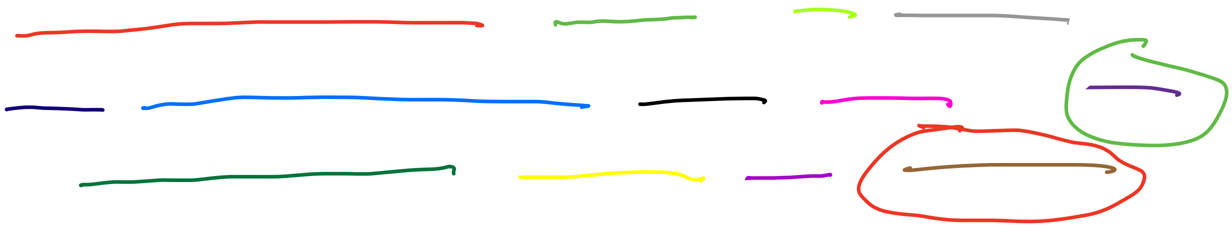
Problem: there's 2^n possible $S \subseteq [n]$...

Idea: DP?

Best(j) = largest subset taking interval j?

Issue 1: what order? (not sorted)

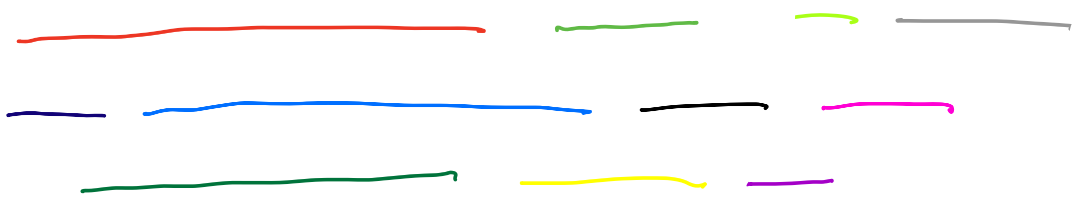
2: how to recurse?



Intuition: let's get rid of "last" interval

It means we can't take (overlapping)

Skip ahead to:



Idea: 1) Sort L

2) Define special problems Best(j)

3) ...

4) Profit?

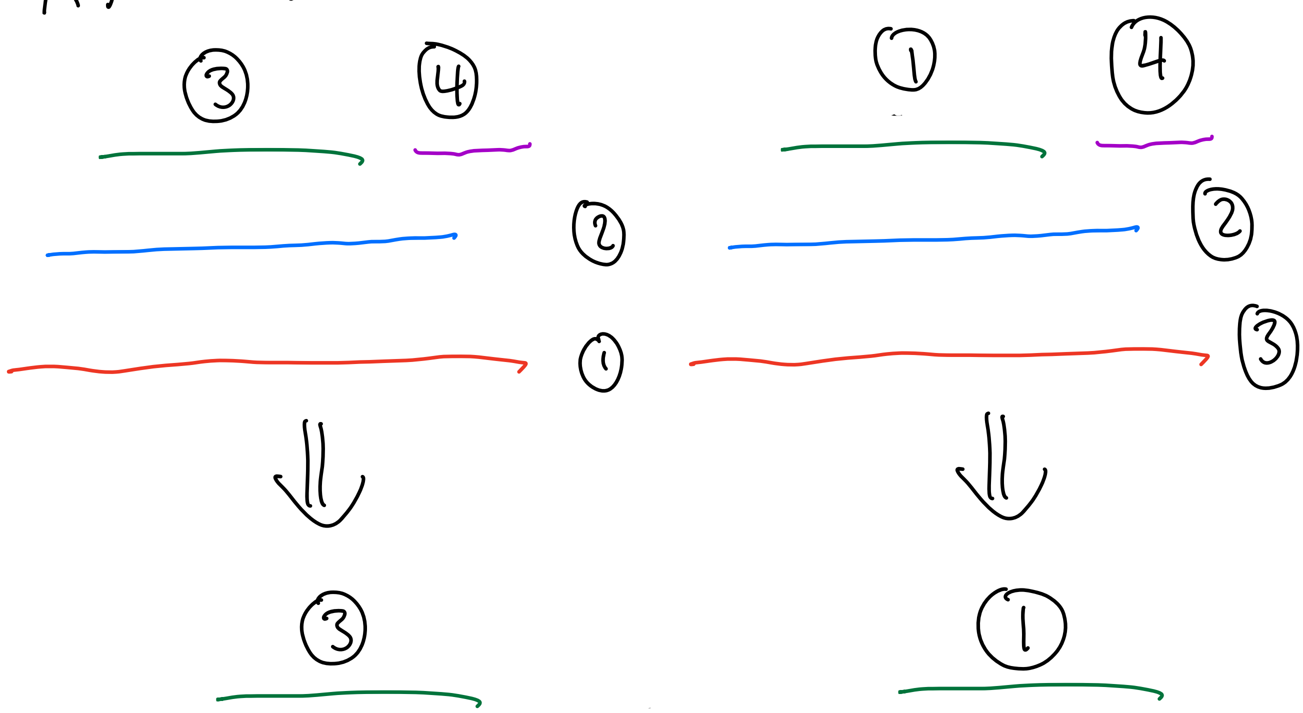
Step 2: Let

$\text{Best}(j)$ = largest nonoverlapping subset of $L[:j]$ (prefix of L)

Step 1: How to sort?

- left endpoint?
- right endpoint?

Ans: we should do whatever lets us recurse.

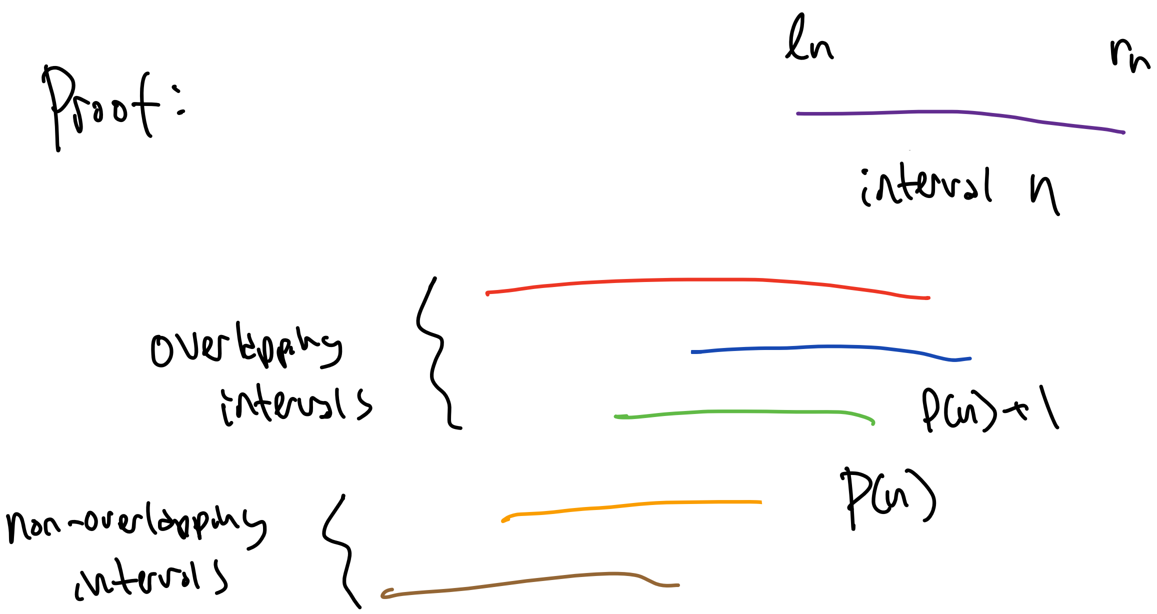


Left sort: not a prefix

Right sort: a prefix (memoized)

Claim: Suppose L sorted by right endpoint. Removing overlaps w/ last interval gives a prefix of L.

Proof:



Let $P(n)$ be last interval with $r_{p(n)} < \ell_n$

Overlapping: $p(n)+1, p(n)+2, \ldots n$ $\qquad r_i \geq r_{p(n)+1} \geq \ell_n$

Non-overlapping: $\underbrace{1, 2, \ldots P(n)}_{\text{prefix}}$ $\qquad r_i \leq r_{p(n)} < \ell_n$

1) Sort L by right endpoint

2) Define special subproblems:

$\text{Best}(j)$ = largest nonoverlapping subset of $L[:j]$ (prefix of L)

3) recursion

$$\text{Best}[j] = \max\Big(\underbrace{\text{Best}(j-1)}_{\text{don't include interval } j},\ 1 + \underbrace{\text{Best}[P(j)]}_{\text{include interval } j}\Big)$$

Here, $P[j]$ is last acceptable interval:

$$\underset{\text{(can take)}}{r_{P(j)}} < \ell_j \le \underset{\text{(can't take)}}{r_{P(j)+1}}$$

Runtime analysis:

1) $O(n\log(n))$

2) N/A

3) $\underbrace{O(n)}_{\text{\# subproblems}} \times \underbrace{O(\log(n))}_{\text{compute } P[j] \text{ via binary search}}$

4) $O(n\log(n))$ time

(before, $\exp(n)$)

☺

$\boxed{\text{Extension}}$ Recover the subset?

Best(1)	Best(2)	...	Best(P(n))	...	Best(n-1)	Best(n)

We memoized everything, can infer the path.

Work backwards!

$S \leftarrow S \cup \{n\}$

$\boxed{\text{Extension}}$ Weighted scheduling

Same idea, but interval i has weight $W(i)$

Goal: maximize $\displaystyle\sum_{i \in S} W(i)$ for non-overlapping S

Motivation: not all intervals created equal

e.g. $W(i) = r_i - l_i$ (by length)

$W(i) = 1$ (vanilla scheduling)

Strategy: $\text{Best}(j)$ = max weight non-overlapping subset of first j intervals.

$$\text{Best}(j) = \max\left(\text{Best}(j-1),\ \text{Best}(p(j)) + W(j)\right)$$

Longest increasing subsequence (Part III, Section 3.2)

Input: L is a list of n elements in $\mathbb{R}$

Output: Longest increasing subsequence of L

$$S \subseteq [n], \quad L(i) \leq L(j) \quad \forall\, i<j \in S$$

$\boxed{\text{Example}}$

$$5, 10, 7, 1, 8, 3, 2, 6, 12, 4, 9, 11$$

(random permutation of $[12]$)

Again, not so easy! "try everything" $\Rightarrow$ 2^n tries

Best(j) = longest increasing subsequence ending on $L(j)$

(prefix also OK: just not as clean)

How to recurse?

if b is small,
rather not take

w a x y z b L[n]

w a x y z b L[n]

less options
to continue growing
the sequence

takeaway: there are tradeoffs...

Solution: try everything.

$$\text{Best}[j] = \max_{\substack{i \in [j-1] \\ L[i] \le L[j]}} \text{Best}[i] + 1$$

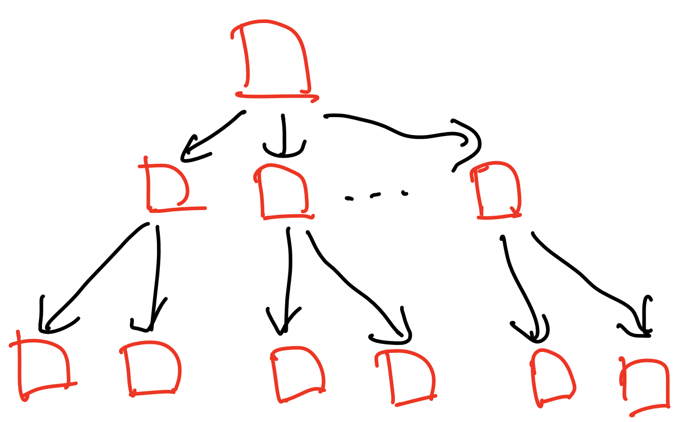
Best continuation.
We let it $= 0$
if there are no $L[i] \le L[j]$.

include $L[j]$

Takes $O(n)$ time per $j \in [n]$,
$O(n^2)$ time total.

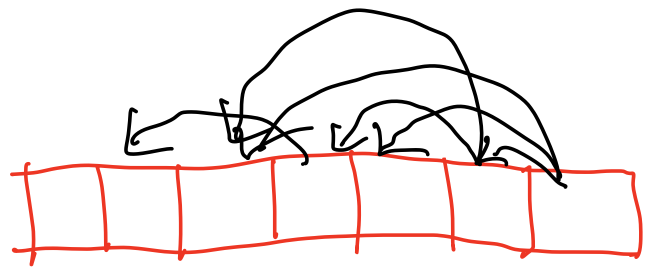
Intuition:

$\boxed{\text{Normal recursion}}$



Smart recursion

still can have
large branching
factor. just reusing
the same n problems



Subset Sum (Part III, Section 3.3)

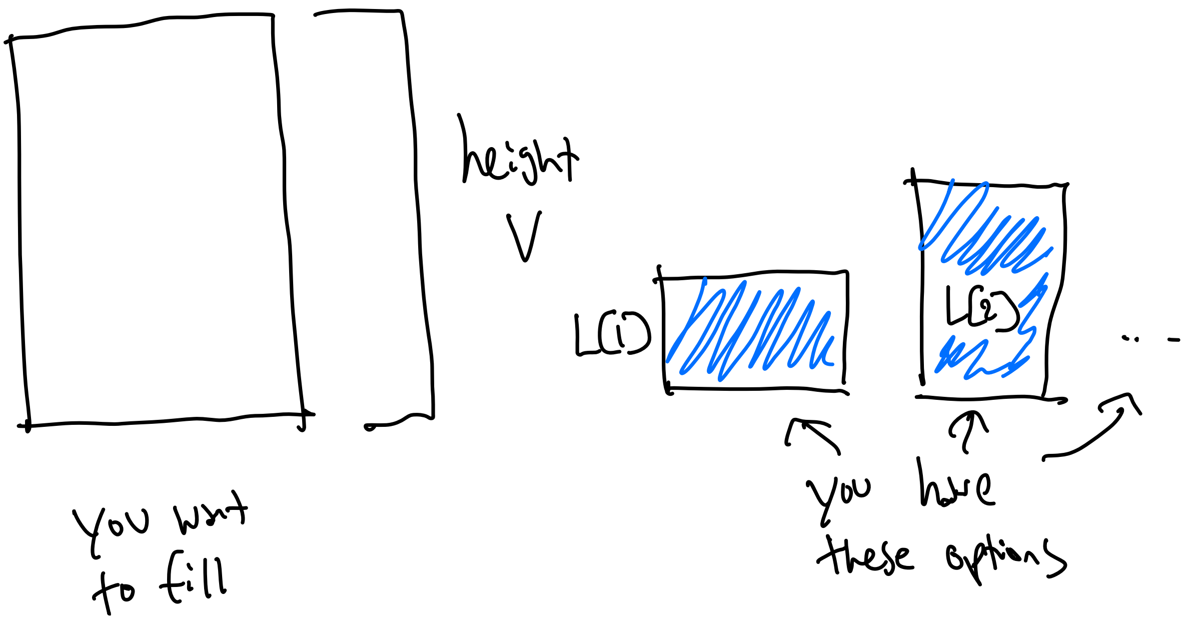
Input: L is a list of n natural #s
1, 2, 3, 4, . . .

$V \in \mathbb{N}$ is a target value

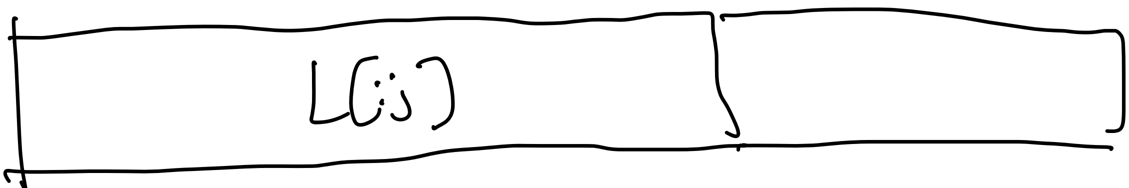
Output: True if $\exists S \subseteq [n], \sum_{i \in [n]} L[i] = V$

False else

Intuition:

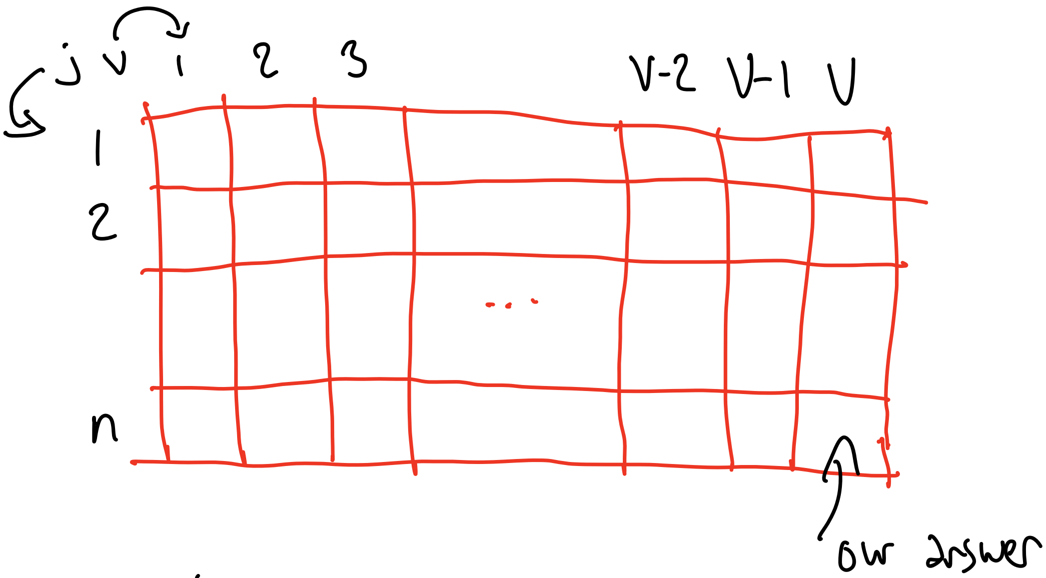


How to define subproblems?



if false, doesn't
yield much info later...
need to consider <u>values</u>

$S[j][v]$ = Can you hit target V using first j items?



Formula:

$$S[j][v] = S[j-1][v]$$
$$\text{OR } S[j-1][v-L[j]]$$

What order?

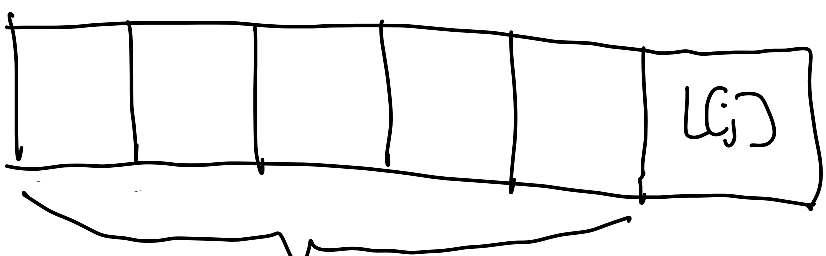
Row by row. Formula depends on prev row.

Time: $O(nV)$. (good if V small)

Bonus: Faster LIS

Solvable in $O(n\log(n))$ time.

Intuition: Can we make length-k using $L[j]$?

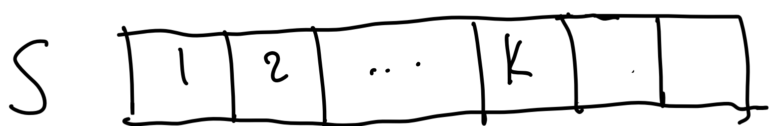


is there length $k-1$?

may as well store smallest end

Goal: Iterate thru $j \in [n]$

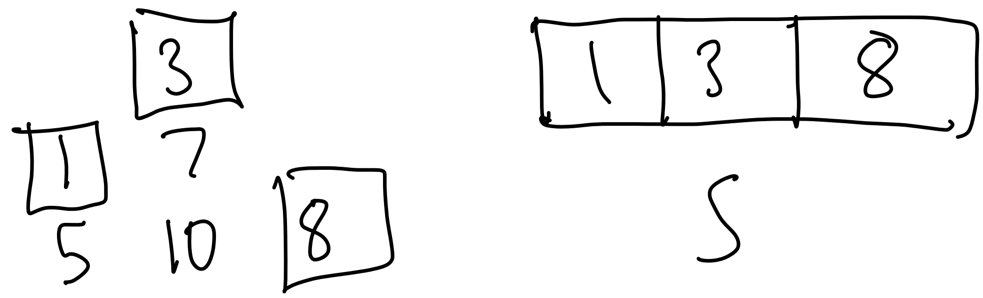
After time j:



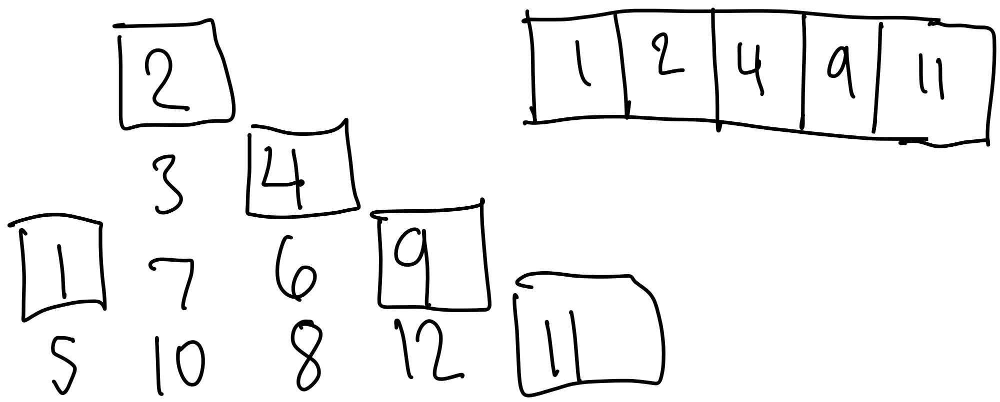
smallest end of length-k increasing subseq. of $L[:j]$

Example 5, 10, 7, 1, 8, 3, (j=6)
2, 6, 12, 4, 9, 11 (j=12)

j=6



j=12



Observations:

- S always sorted
- Incoming element unique stack
  - earlier? not smaller
  - later? can't keep building
- log(n) time per iter