LECTURE 4 (September 4th)

Dynamic Programming

Dynamic programming is "Recursion without Repetition" where we solve each distinct recursive sub-problem exactly once.

The algorithm does not look like a recursion but nested for loops but under the hood it is doing smart recursion.

Last time,

LONGEST INCREASING SUBSEQUENCE

Given a sequence of numbers, find a subsequence that is increasing and the longest

3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6

increasing sequence = empty
>x OR

[number y > x] + [increasing sequence > y]

This is my job ↑ ↑ Recursion fairy takes care of it

So, now for our input sequence, we need to identify for every item whether its in the longest increasing subsequence. Let's jump into the middle and assume a few of the first items have already been identified.

3 1 4 1 5 9 2 6 5 3 | 5 8 9 7 9 3 2 3 8 4 6 2 6

Decision already made

What about this ?
Is it in LIS ?
So, 5 is not in LIS

How about 8 ?
Not easy to tell !
What do we do ?

RECURSE in both cases, assuming 8 ∈ LIS and one assuming 8 ∉ LIS

Here, we need to remember for recursion which case we are in

3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6

This does not matter

Only need to remember the largest number in the subsequence so far

Consider the recursion :

LISbigger (i,j) = Length of LIS of A[j....n] all bigger than A[i]

↳ We already saw the recursive definition of this

We need to check if A[j] is in this subsequence.

What does this function look like ?

Last # we have decided is in in LIS

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\left\{ \begin{array}{c} LISbigger(i,j+1) \\ 1 + LISbigger(j,j+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

If j^{th} number is smaller then we start looking at j+1 but have the same largest # A[i]

In this we try both, whether

A[j] ∉ LIS OR A[j] ∈ LIS

You can also write it this recursion in pseudocode as well :

```
LISBIGGER (i,j) :

    if j>n : return 0

    else if A[i] ≥ A[j] : return LISBIGGER (i,j+1)

    else : skip ← LISBIGGER (i,j+1)
           take ← LISBIGGER (j, j+1) + 1
           return max {skip, take}
```

This gives an algorithm to find the LIS bigger than some value x.
How do we solve the original problem?

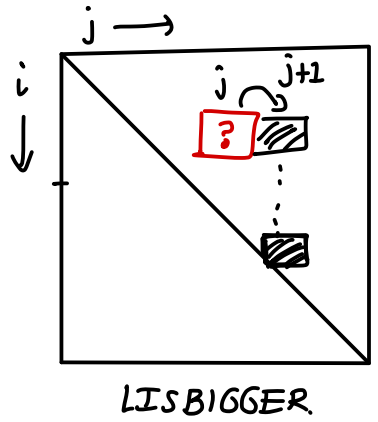
One way: add an artificial value for x, say $-\infty$

Another way: in a for loop, try all possible initial value for x

But wait, this might take exponential time: we are only reducing the problem size by one but potentially recursing on each sub-problem twice.

How do we memoize this function to get an efficient algorithm?

What sort of data structure to use? We need to pass 2 indices and return a number, so we use a 2D array.



LISBIGGER

So, let's see in what order the recursion fills up the array, so we can do it ourselves with for loops

Consider an entry [?] in the array

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \quad [1] \\ \max \begin{cases} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{cases} \begin{matrix} [2] \\ [3] \end{matrix} & \text{otherwise} \end{cases}$$

[1] This line looks up an entry in the same row but to the right of [?]

[2] This line does the same as above

[3] This line looks an entry in the same column but below [?] just above the diagonal

So, when we want to know [?], we need to fill [▨] and [▨]

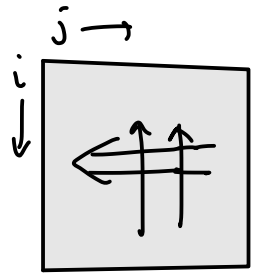
What order should we choose to fill the table?

We need two nested for loops

1. Indices for each of the loop
2. Increasing / decreasing order for the loop
3. Which is in the inner / outer loop ?

For example, suppose we do this

```
for j ← n to 1
    for i ← n to j
```



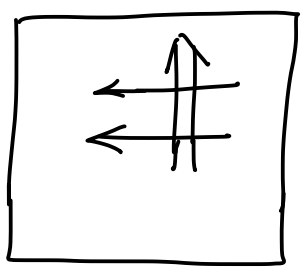
⇐ Outer loop & order

↑ Inner loop & order

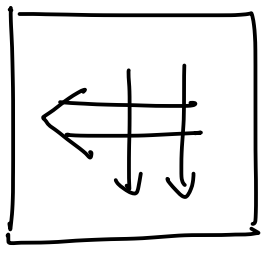
Column j+1 is evaluated before column j

This is not the only order that works.

We can also do the following



OR



One can check that in all of these orders fill ▨ before [?]

It takes $O(n^2)$ time to fill the table

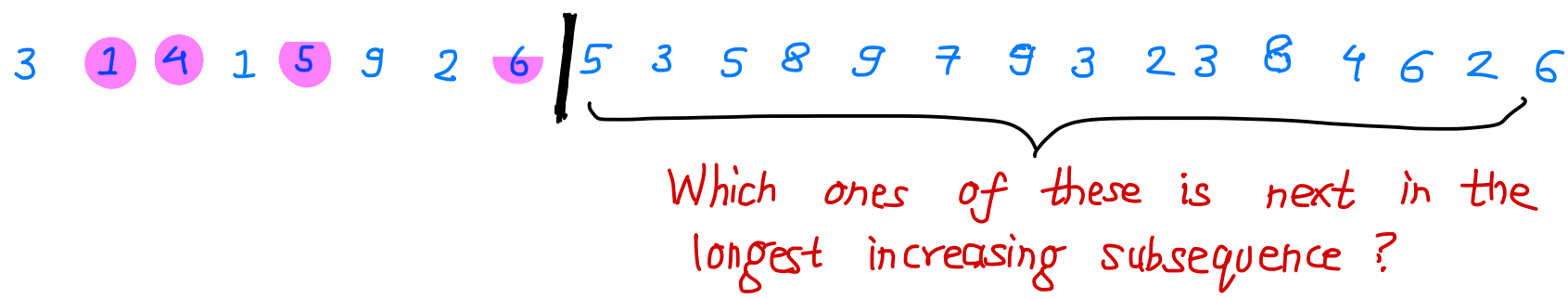
The final algorithm looks like the following

```
FastLIS(A[1..n]):
  A[0] ← −∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n                     ⟨⟨Base cases⟩⟩
      LISbigger[i, n+1] ← 0
  for j ← n down to 1
      for i ← 0 to j − 1             ⟨⟨...or whatever⟩⟩
          keep ← 1 + LISbigger[j, j+1]
          skip ← LISbigger[i, j+1]
          if A[i] ≥ A[j]
              LISbigger[i, j] ← skip
          else
              LISbigger[i, j] ← max{keep, skip}
  return LISbigger[0, 1]
```

Pseudocode is not required for HW & exams for dynamic programming problems

This is not the only way to solve this LIS problem

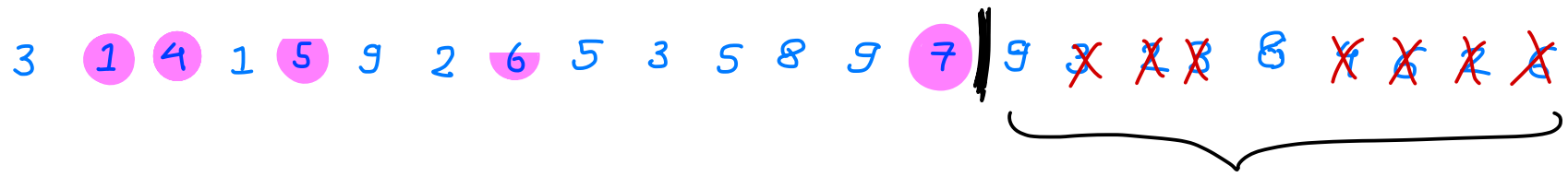
Instead of asking a simple yes/no question about each element of the input sequence, we could directly try to ask about the elements of the output sequence



Here we are relying more on the recursive structure of the output more than the recursive structure of the input

There are some things we know aren't going to be next – Anything less than 6

For others, just try all possible options. Suppose, we decide that $7 \in LIS$



Consider the recursion :

$LISbigger(i,j)$ = LIS of $A[j \ldots n]$ & all $> A[i]$

But now notice that j is always $i+1$!

So, let's define

$LISFirst(i)$ = LIS $A[i \ldots n]$ starting with $A[i]$

So, when we decide to recurse from

3 1 4 1 5 9 2 6 | 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6

there are many possible recursive calls

3 1 4 1 5 9 2 6 5 3 5 8 | 9 7 9 3 2 3 8 4 6 2 6

3 1 4 1 5 9 2 6 5 3 5 8 9 | 7 9 3 2 3 8 4 6 2 6

3 1 4 1 5 9 2 6 5 3 5 8 9 7 | 9 3 2 3 8 4 6 2 6

3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 | 3 2 3 8 4 6 2 6

3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 | 4 6 2 6

LIS First (i) = LIS A[i....n] starting with A[i]

$$LISfirst(i) = 1 + \max\{LISfirst(j) \mid j > i \text{ and } A[j] > A[i]\}$$

```
LISFIRST(i):
  best ← 0
  for j ← i + 1 to n
      if A[j] > A[i]
          best ← max{best, LISFIRST(j)}
  return 1 + best
```

What's the base case of the recurrence?

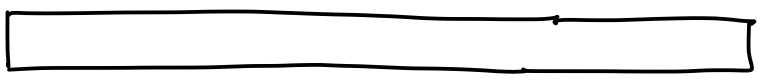
When $\{LISfirst(j) \mid j > i \text{ and } A[j] > A[i]\}$ is the empty set.

In this case, we can just define $\max \emptyset = 0$ and we are done with the base case as well.

To compute the final answer, we call

```
LIS(A[1..n]):
  A[0] ← −∞
  return LISFIRST(0) − 1
```

What's the right data structure to memoize ? An array because now there is just one index i !



What order to evaluate ? Right to left because recursive call for i only makes calls to j where $j \geq i$

What's the running time ? To evaluate each entry is $O(n)$ time and $O(n)$ entries in the array, so time is $O(n^2)$

So, these are two $O(n^2)$ time algorithm for LIS

There is a third algorithm as well, which runs in time $O(n \cdot \log n)$ called patience sorting

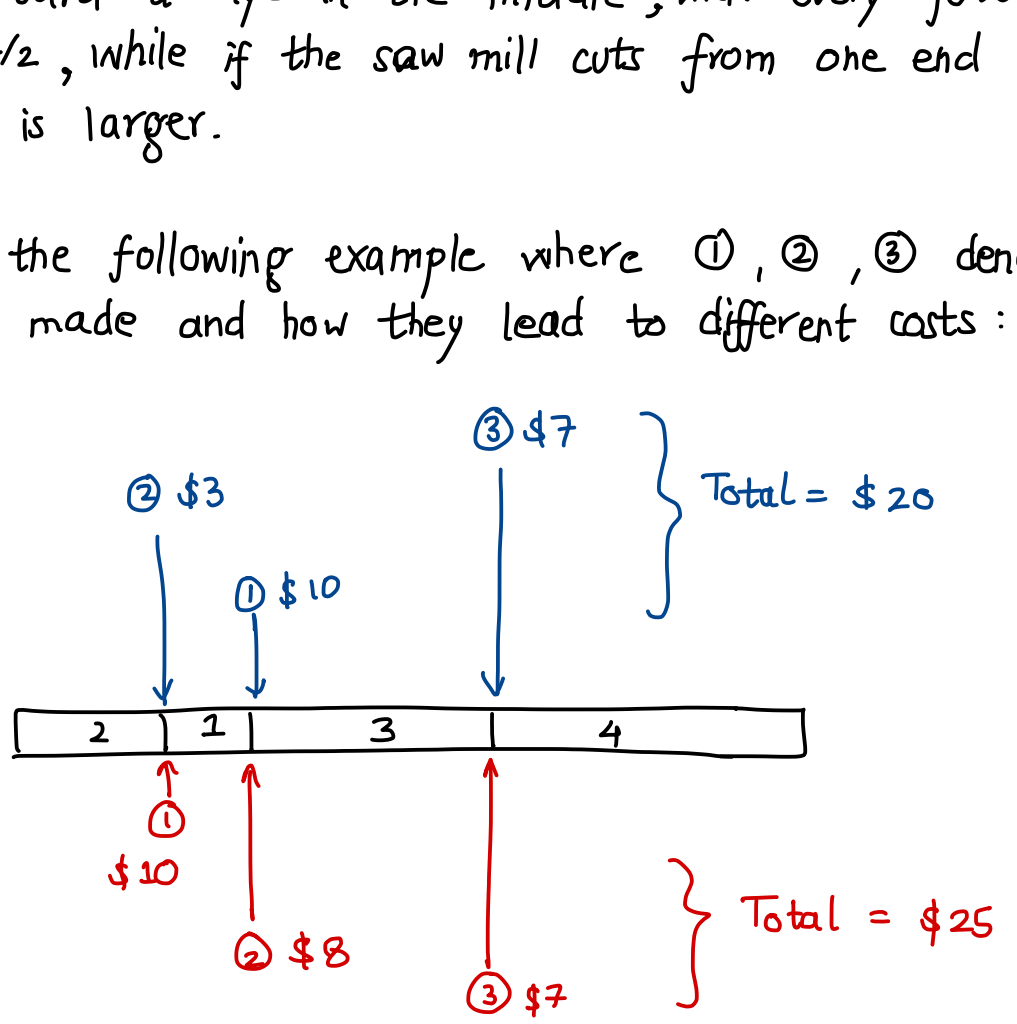
WOODCUTTERS PROBLEM

You have a long woden board into fixed positions

You need to take it to the sawmill, which can cut any L-foot board for $2

Depending on where the sawmill cuts the board first, you will have to pay different amounts. For example, if the cuts are evenly spaced, if the saw mill cuts the board always in the middle, then every future cut is going to cost at most L/2, while if the saw mill cuts from one end to the other, the cost of each cut is larger.

Consider the following example where ①, ②, ③ denotes the order in which cuts are made and how they lead to different costs :



So, the problem is to figure out the right ordering to make the cost as small as possible given the position of the cuts

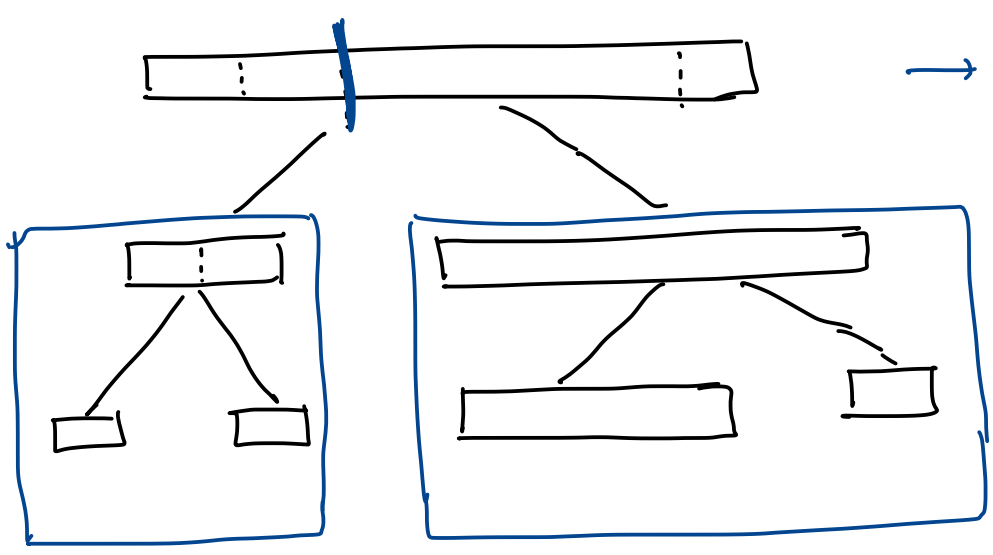
How to solve this via dynamic programming?

1. Break recursively into sub-problems

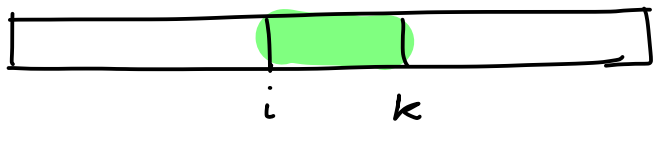
   How do we that?

   - We have to find the first cut which divides the board into two parts
   - After that, we recursively solve the left part and the right part.

So, we are building a binary tree



What are the subproblems here? Intervals → Each time we have a part of the board that starts at the i^{th} cut and ends at the k^{th} cut



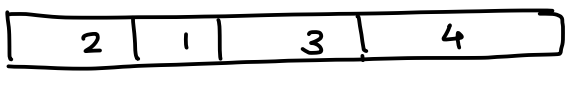
Our problem now is to identify where the left and right ends of the interval are We want to compute

OPTCOST (i, k) = min cost to cut up segment from cut i to cut k

In order to write the actual recurrence, we need to define how the input is represented

INPUT $L[0, \ldots n]$ = array of board length

$L[i]$ = length of board between cut i & cut $i+1$

e.g.  corresponds to $L = [2, 1, 3, 4]$

Let's develop our recurrence

$$\text{OPTCOST}(i,k) = \begin{cases} 0 & \text{if } k = i+1 \\ \sum_{j=i}^{k-1} L[j] + \min\{\text{OPTCOST}(i,j) + \text{OPTCOST}(j,k) \mid i<j<k\} & \end{cases}$$

if $k = i+1$ → base case (no cuts to be made)

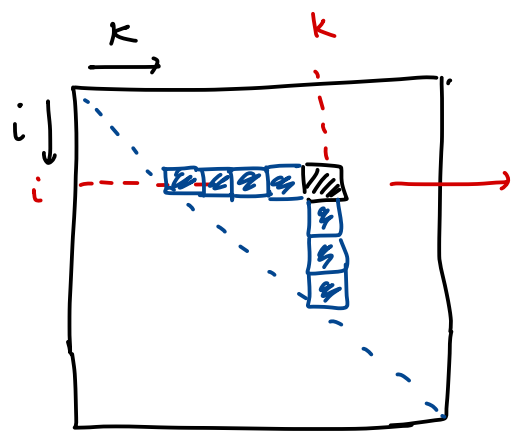
This is the cost we have to pay for the first cut no matter where we make it

What we don't know is where the first cut is and the future costs depends on the first cut. How do we find the first cut? Just try them all and take the minimum.

But computing this recurrence takes $O(3^n)$ time, so we need to make it efficient by memoizing so that we don't solve the same sub-problem multiple times.

What data structure should we use to store OPTCOST(i,k)?

2-D array

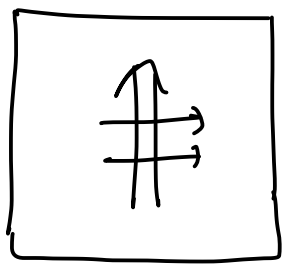


Which subproblem OPTCOST(i',k') does this depend on?

We need to compute OPTCOST(i,j) which is in the same row but to the left of k & right of i
These are the blue entries to the left of ▨. Similarly we also have to compute the entries below ▨ until the diagonal

In what order should we fill this array?

One valid order:



Outer for loop for i = n to 1
Inner for loop for k = 1 to n
compute OPTCOST(i,k) with the recurrence

Running Time is $O(n^3)$
two for loops over n indices and filling each entry takes O(n) time since we have to compute the min entry among O(n) things