Algorithms + Complexity, Spring '26

Dynamic Programming

Recap

- HW1 posted, due next Thursday
  ↳ Again, written HW, groups of ≤ 3
- Reading on Thursday,
  + next week's will be up by then

Text Segmentation

↳ ~~In~~ Backtracking + Dynamic Programming

Fix a "language", so can recognize "words".

Ex: - English text

- Genetic data

S: $\boxed{}$

⋮

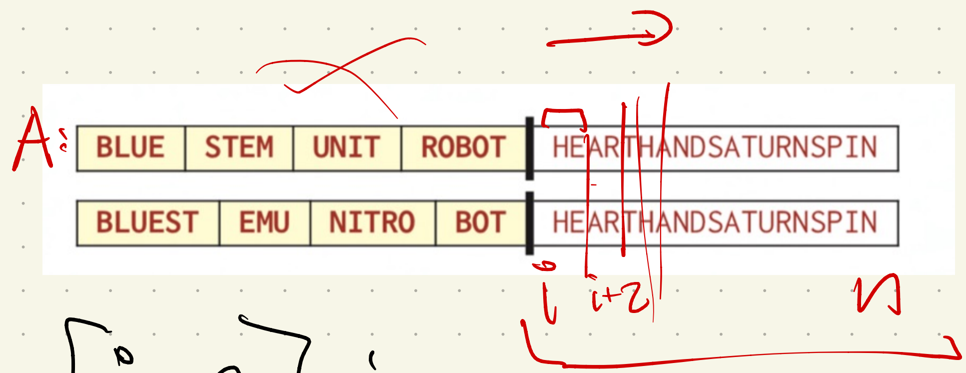
So: IsWord(s) is given, + O(1) time.

Aside: reasonable?

Usually hashed dictionary.

Backtracking:

Fix suffix
to decide on.

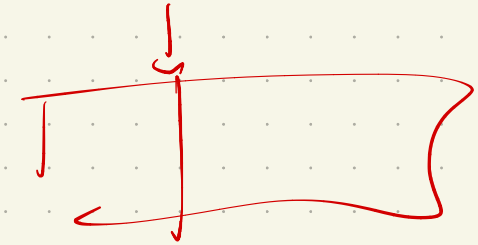


To solve Splittable[i..n]:

→ For every $j \in [i+1, n]$
check isWord[A[i..j]]
if it is,
check Splittable[j+1..n]

Code:

```
SPLITTABLE(A[1..n]):
    if n = 0
        return TRUE
    for i ← 1 to n
        if ISWORD(A[1..i])
            if SPLITTABLE(A[i+1..n])
                return TRUE
    return FALSE
```



Runtime:

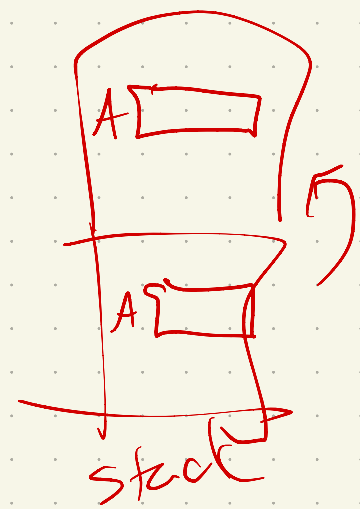
$$S(n) \le \sum_{i=1}^{n} S(n-i) + O(n)$$

exponential

Issue w/ passing arrays:

don't do it

assume array is global & pass indices



Passing by index / ptr / global / etc

Given an index i, find a segmentation of the suffix $A[i\,..\,n]$.

Formalize an (ugly?) recursion:

and

$$\boxed{Splittable(i)} = \begin{cases} \text{TRUE} & \text{if } i > n \\ \bigvee_{j=i}^{n} \big(\text{IsWord}(i,j) \wedge Splittable(j+1)\big) & \text{otherwise} \end{cases}$$

OR

And then translate to code:

```
⟨⟨Is the suffix A[i..n] Splittable?⟩⟩
SPLITTABLE(i):
  if i > n
      return TRUE
  for j ← i to n
      if ISWORD(i, j)
          if SPLITTABLE(j + 1)
              return TRUE
  return FALSE
```

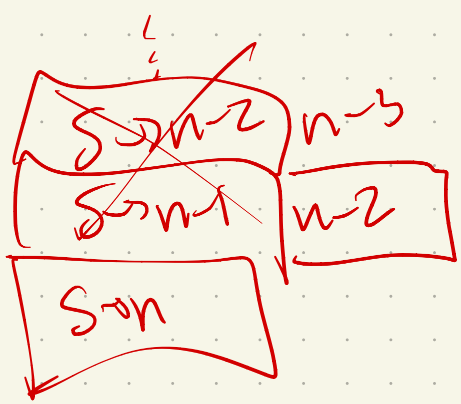
Why??

It's already exponential anyway, right?

Observation:

```
⟨⟨Is the suffix A[i..n] Splittable?⟩⟩
SPLITTABLE(i):
    if i > n
        return TRUE
    for j ← i to n
        if ISWORD(i, j)
            if SPLITTABLE(j + 1)
                return TRUE
    return FALSE
```

Consider stack point of view, + all of these function calls:



So: For any $k \in [1..n]$, might be calling Splittable(k) many times!

Question: Can its value change?
(ie is it a pure function?)

↳ one whose return doesn't ever change

Shouldn't compute the same thing twice!

Potential Improvement

Once you calculate Splittable(k) once, store it.

Then, can just look it up in a data structure! S[1..n]

↳ array of booleans

Here:

```
⟨⟨Is the suffix A[i..n] Splittable?⟩⟩
SPLITTABLE(i):
  if i > n
      return TRUE
  for j ← i to n
      if ISWORD(i, j)
          if SPLITTABLE(j + 1)
              return TRUE
  return FALSE
```

Change: check if already computed & look it up if so

↳ otherwise, do recursion

Then:

$\boxed{A[n]}$

$\boxed{A[2..n]}$

$\boxed{A[1..n]}$

Better yet:

- Splittable(n) is trivial ← save T/F
- Splittable(n-1) ← T/F only needs Splittable(n)
- Splittable(n-2) only needs n-1 & n-2

S[1 .. n]

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN
BLUEST	EMU	NITRO	BOT	HEARTHANDSATURNSPIN

So: memoize: how to store data?

for i ← n down to 1
  calculate Splittable[i] (based on later values)

$\sum_{i=1}^{n} (n-i) = \sum_{j=1}^{n} j = O(n^2)$

return Splittable[1]

for $i \leftarrow n$ down to 1
    $S[i] \leftarrow$ false
    for $j \leftarrow i$ to n
        if IsWord(i,j) and $S[j+1]$
            $S[i] \leftarrow$ true
    // at end of for loop, $S[i]$ is true only if $A[i..n]$ is splittable
return $S[1]$

Aside: Fibonacci Computations

```
MEMFIBO(n):
  if (n < 2)
      return n
  else
      if F[n] is undefined
          F[n] ← MEMFIBO(n − 1) + MEMFIBO(n − 2)
      return F[n]
```

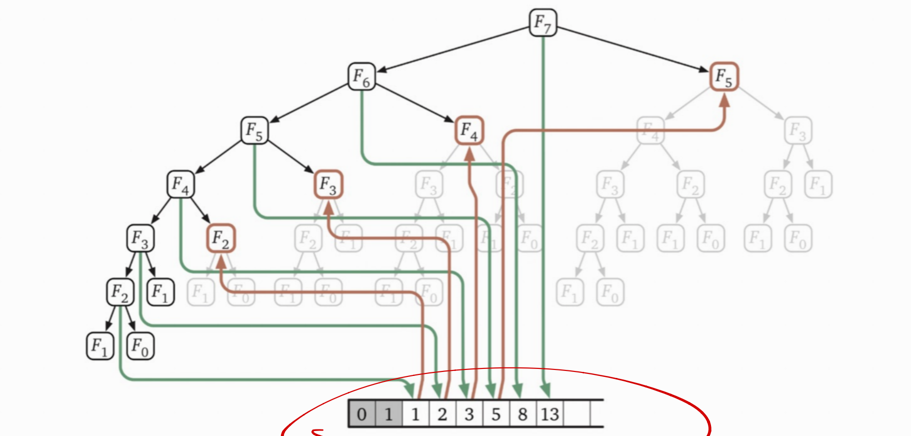
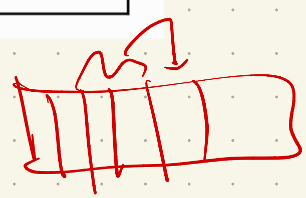


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

Illustrates same pipeline (~~data~~ data structure!)

```
ITERFIBO(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

for loop version



Plus, space: O(n)

```
ITERFIBO2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

less space

His ♡ section: Can actually do better! (Fancy math tricks)

$$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}=\begin{array}{c}\begin{bmatrix}0\\1\end{bmatrix}\end{array}\begin{array}{l}F_0\\F_1\end{array}$$

$$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}=\begin{bmatrix}1\\1\end{bmatrix}\begin{array}{l}F_1\\F_2\end{array}$$

$$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}=\begin{bmatrix}1\\2\end{bmatrix}\begin{array}{l}F_2\\F_3\end{array}$$

$$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix}=\begin{bmatrix}2\\3\end{bmatrix}\begin{array}{l}F_3\\F_4\end{array}$$

$$\Rightarrow \begin{bmatrix}0&1\\1&1\end{bmatrix}^n\begin{bmatrix}0\\1\end{bmatrix}=\begin{bmatrix}F_{n-1}\\F_n\end{bmatrix}$$

Proof: induction

Base case: $n=1$ ✓

IH: assume $\begin{bmatrix}0&1\\1&1\end{bmatrix}^{n-1}\begin{bmatrix}0\\1\end{bmatrix}=\begin{bmatrix}F_{n-2}\\F_{n-1}\end{bmatrix}$

IS: $\begin{bmatrix}0&1\\1&1\end{bmatrix}^{n}\begin{bmatrix}0\\1\end{bmatrix}=\begin{bmatrix}0&1\\1&1\end{bmatrix}\left(\begin{bmatrix}0&1\\1&1\end{bmatrix}^{n-1}\cdot\begin{bmatrix}0\\1\end{bmatrix}\right)\overset{\text{by IH}}{=}\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}F_{n-2}\\F_{n-1}\end{bmatrix}$ ▭

Runtime: time to compute $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n$

$\hookrightarrow$ back to chapter 1!

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a & \text{otherwise} \end{cases}$$

```
PIṄGALAPOWER(a, n):
  if n = 1
      return a
  else
      x ← PIṄGALAPOWER(a, ⌊n/2⌋)
      if n is even
          return x · x
      else
          return x · x · a
```

or

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^2)^{n/2} & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^2)^{\lfloor n/2 \rfloor} \cdot a & \text{otherwise} \end{cases}$$

```
PEASANTPOWER(a, n):
  if n = 1
      return a
  else if n is even
      return PEASANTPOWER(a², n/2)
  else
      return PEASANTPOWER(a², ⌊n/2⌋) · a
```

Either way: $M(n) = M\left(\frac{n}{2}\right) + O(1)$ multiplications

$= O(\log n)$

But wait – F_n is exponential! Specifically,

$$F_n = \frac{1}{\sqrt{5}} \phi^n - \frac{1}{\sqrt{5}} (\hat{\phi})^n, \quad \phi = \frac{1+\sqrt{5}}{2} > 1, \quad \hat{\phi} = \frac{1-\sqrt{5}}{2}$$

So... how many bits to write it down?

number $n \rightarrow \log_2 n$

$16 \rightarrow 10000$

2^n n bits

Clarification:

Our earlier algorithms use $O(n)$ additions or subtractions

If a # $\leq$ 64-bits – sure!

But larger?

Let $M(n)$ = time to multiply 2 n-digit #s

Here: $T(n) = T(\frac{n}{2}) + M(n)$

Best known $M(n)$: $O(n \log n)$ (citing 2019 result)

so $T(n) = O(n \log n)$

[we'll still usually assume $O(1)$ time to add/multiply]

Fibonacci Recap:
good / bad

- "Simple" yet interesting example
- Illustrates how powerful this concept can be.
  ↳ saving both time & space

Downside:

Not always so obvious how to convert the recursion into an iterative structure!

Advice

Start with the recursion!

↳ Use it to prove correctness.

Then, for code:

Start at base cases. Save them!

Build up "next" level:

the recursions that call base case(s).

Try to formalize **this** in a loop + data structure format.

Finally: analyze **both** space + time

Rant about greed:

When they work, "greedy" strategies are very fast & effective!

But - often such intuitive strategies fail.

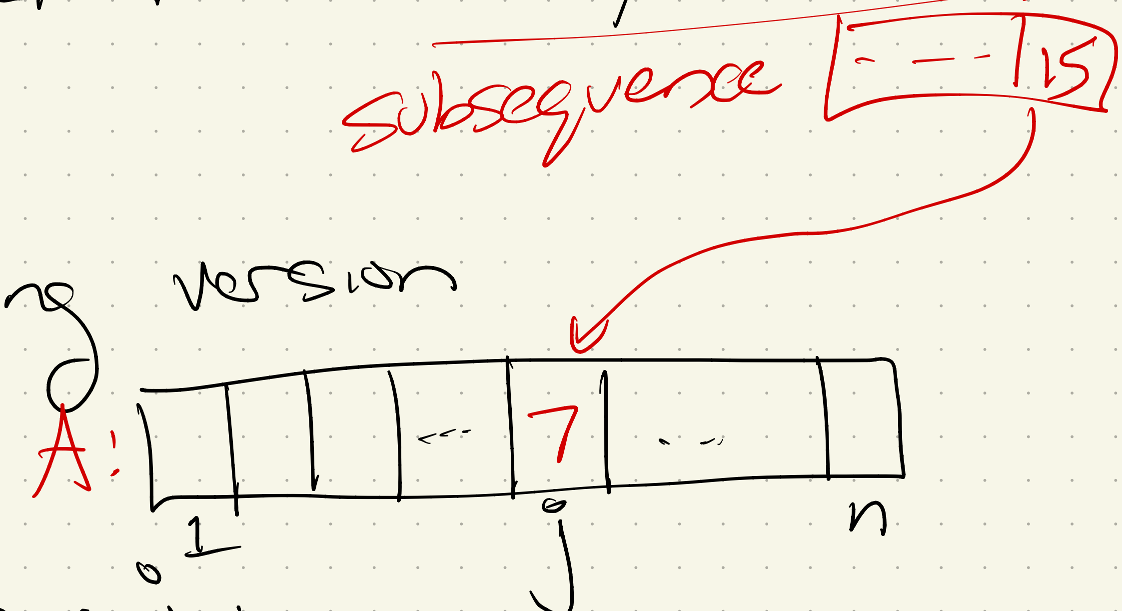
Dynamic programming & backtracking will always work.

We'll study both, but better to start here.

Next reading: Longest increasing subsequence (again)

(or, why he did all those crazy recurrence versions)

Recap: Backtracking version



Recursion:

At each index j:

- could include $A[j]$ in subsequence (if it is larger than last element we included)

need to know last element!

- could skip & not include $A[j]$ in subsequence

Result:

> Given two indices i and j, where $i < j$, find the longest increasing subsequence of $A[j..n]$ in which every element is larger than $A[i]$.

Need 2 things in recursion!

Store last "taken" index i.

$A[\ \ 11\ \ 10\ \ ]$
$i \quad j$

Consider including $A[j]$:

- if $A[i] \geq A[j]$: can't add $A[j]$
  $\hookrightarrow$ must skip j
- if $A[i]$ is less: could include $A[j]$ or not

Recursion:

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

can't b/c $A[j]$ is too small

include

skip

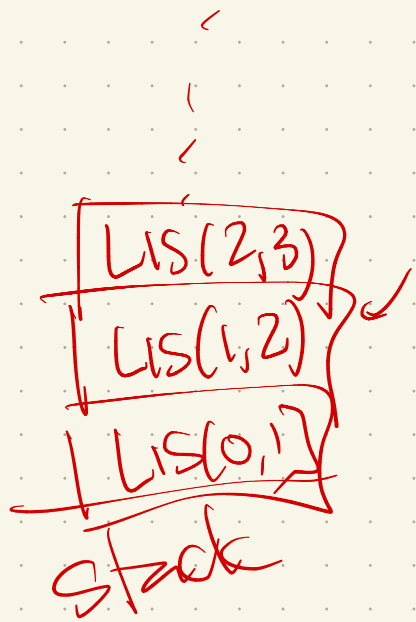
Code version: don't pass arrays! Why?

```
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}
```

must skip

plus the "main":

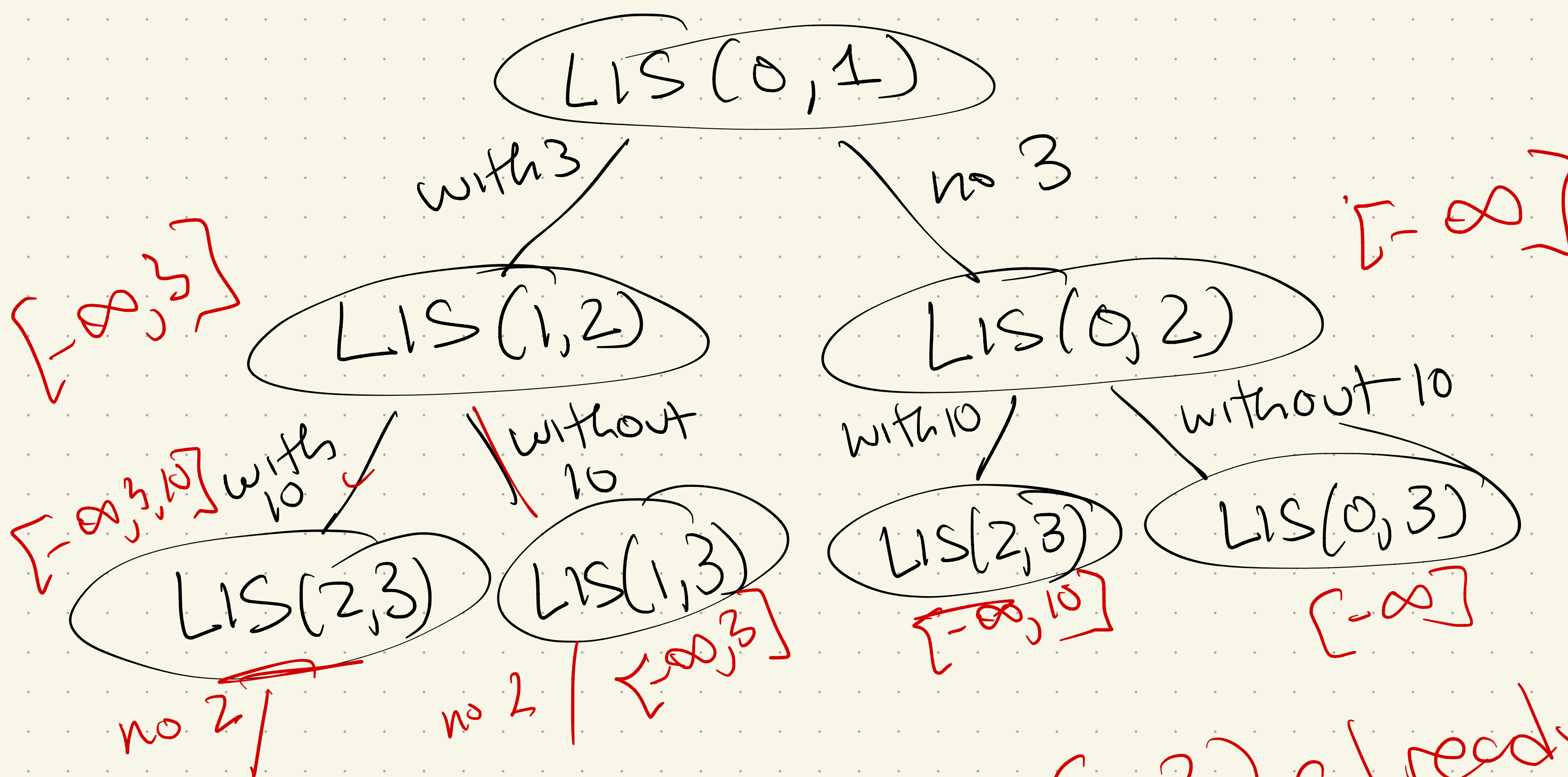
```
LIS(A[1..n]):
  A[0] ← −∞
  return LISBIGGER(0, 1)
```



Example:

$$A: \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \end{array}$$
$$A: [3, 10, 2, 11, 5, 7]$$

$$\begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array}$$
$$\hookrightarrow [-\infty, 3, 10, 2, 11, 5, 7]$$



LIS(2,3) already computed twice!...

Question: Is this function pure? does answer change?

Yes

Memoize: What are we recomputing?

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

← correct if $k > i$ & $l > j$, $L[k,l]$ is correct

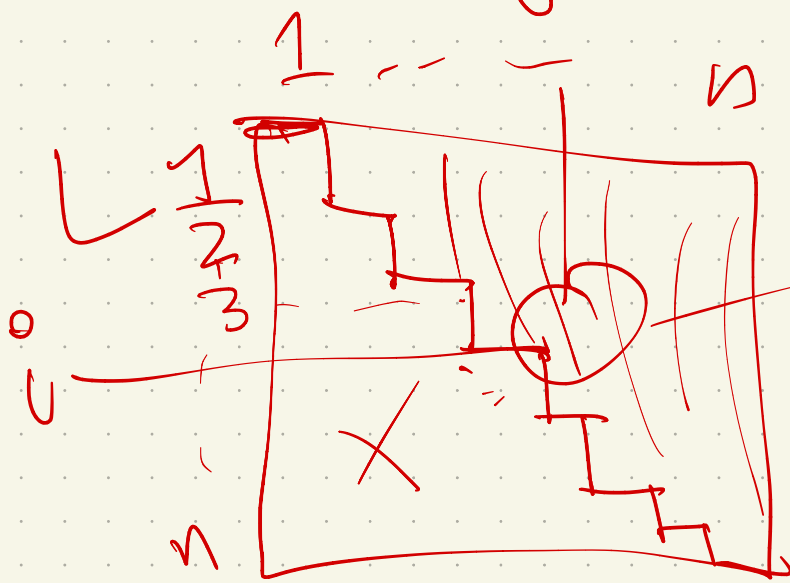
```
LISBIGGER(i, j):
  if j > n
      return 0
  else if A[i] ≥ A[j]
      return LISBIGGER(i, j + 1)
  else
      skip ← LISBIGGER(i, j + 1)
      take ← LISBIGGER(j, j + 1) + 1
      return max{skip, take}
```

How should we store?

values $1 \leq i < j \leq n$

↳ L

$n \times n$ array of integers $\leq n$



$L[i,j]$ will store length of longest inc. sub from $j..n$, if i was included

Now, can we do the same trick as Fibonacci memoization, & convert to something loop-based?

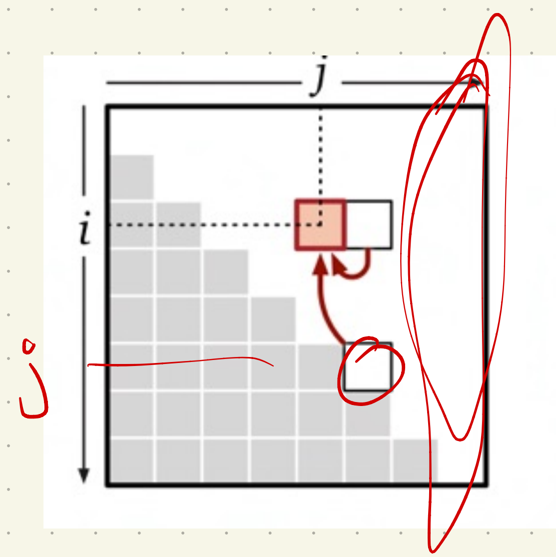
<u>Aside</u>: Why should we? (memory!)

for $j \leftarrow n$ down to 1
  for $i \leftarrow 1$ to n
    fill in cell

<u>Rethink</u>:

To fill in $L[i][j]$, what do I need?

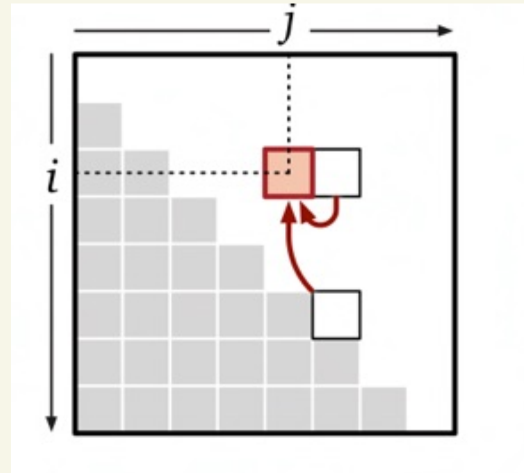
$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max \left\{ \begin{array}{c} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

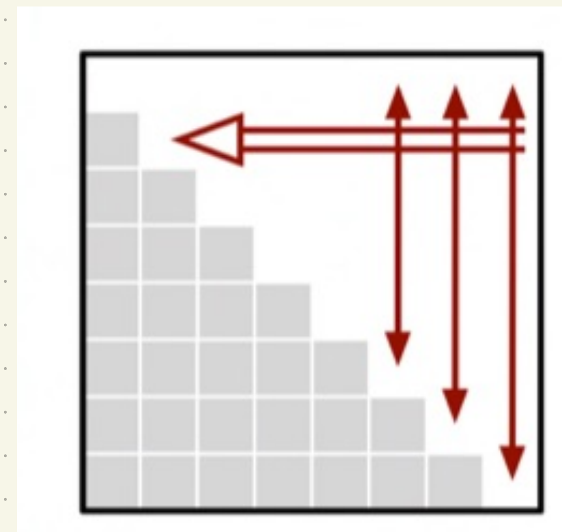


elements one column to right

Result:

```
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]
```





Next time: Edit distance

<u>HUGE</u> in bioinformatics!

One of the basic tools in sequence alignment -

(I have a book with an entire chapter on how to optimize.)

Also: spell checkers, word prediction, etc.

From backtracking mindset: how to think recursively?

Consider 2 last characters:

ALGORITM

ALTRUISTIC

Options:

Example: TGCATAT to ATCCGAT

TGCATAT
↓ delete last T
TGCATA
↓ delete last A
TGCAT
↓ insert A at the front
ATGCAT
↓ substitute C for G in the third position
ATCCAT
↓ insert a G before the last A
ATCCGAT

TGCATAT
↓ insert A at the front
ATGCATAT
↓ delete T in the sixth position
ATGCAAT
↓ substitute G for A in the fifth position
ATGCGAT
↓ substitute C for G in the third position
ATCCGAT

recursively

-	T	G	C	-	A	T	A	T
A	T	C	C	G	A	T	-	-

recursively

-	T	G	C	A	T	A	T
A	T	C	C	G	-	A	T

Input: $A[1..m]$ & $B[1..n]$

$\text{Edit}(\quad,\quad)$

$= \min \left\{ \phantom{\begin{matrix} \\ \\ \\ \end{matrix}} \right.$

<u>So</u>:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

What do we store in?
& how can we adapt loop?

Final code:

```
EditDistance(A[1..m], B[1..n]):
    for j ← 0 to n
        Edit[0, j] ← j

    for i ← 1 to m
        Edit[i, 0] ← i
        for j ← 1 to n
            ins ← Edit[i, j − 1] + 1
            del ← Edit[i − 1, j] + 1
            if A[i] = B[j]
                rep ← Edit[i − 1, j − 1]
            else
                rep ← Edit[i − 1, j − 1] + 1
            Edit[i, j] ← min {ins, del, rep}

    return Edit[m, n]
```