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".

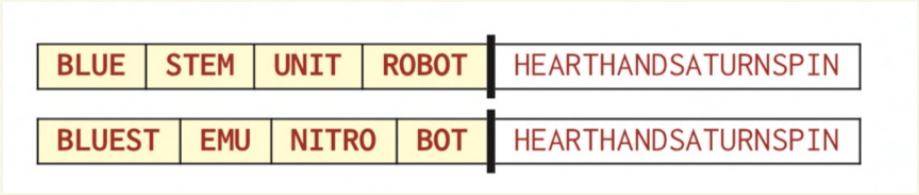
Ex: - English text
- Genetic data
⋮

So: Isword(s) is given, & O(1) time.

{Aside: reasonable?

Backtracking:

Fix suffix to decide on.



To solve Splittable[i..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:

Issue w/ passing arrays:

Passing by index/ptr/global/etc

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

Formalize an (ugly?) recurson:

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

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?)

Potential Improvement

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

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

Here:

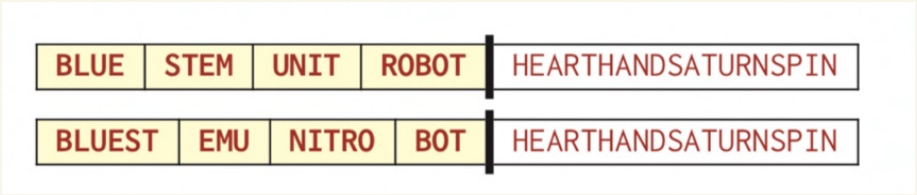
Then:

```
⟨⟨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:

Better yet:

- Splittable(n) is trivial
- Splittable(n-1) only needs Splittable(n)
- Splittable(n-2) only needs n-1 & n-2



So: memoize how to store data?

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]
```

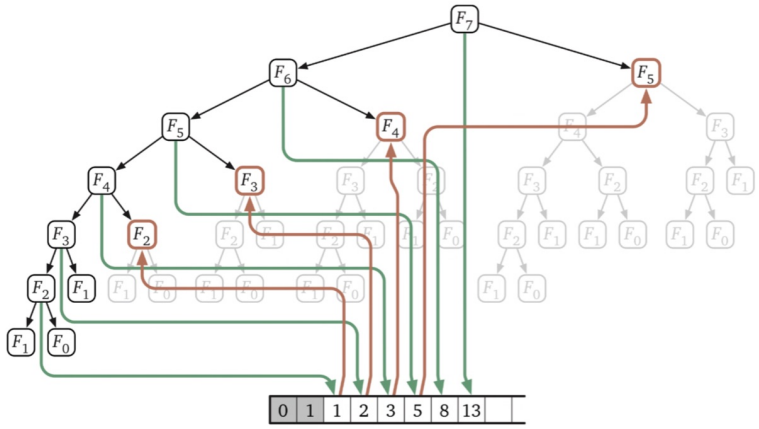


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

Illustrates same pipeline: data 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]
```

Plus, space:

```
IterFibo2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

This ♡ section: Can actually do better!

(Fancy math tricks)

$$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}=$$

$$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}\ \\ \ \end{bmatrix}=$$

$$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}\ \\ \ \end{bmatrix}=$$

$$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}\ \\ \ \end{bmatrix}=$$

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

Proof: induction

Base case:

IH:

IS:

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

↳ back to chapter 1!

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

```
PiñgalaPower(a, n):
  if n = 1
      return a
  else
      x ← PiñgalaPower(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(a^2, n/2)
  else
      return PeasantPower(a^2, ⌊n/2⌋) · a
```

Either way:

But wait – F_n is exponential! Specifically,

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

$$\hat{\phi} =$$

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

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: $T(n) =$

Best known $M(n)$:

so $T(n) =$

Fibonacci Recap: good / bad

- "Simple" yet interesting example
- Illustrates how powerful this concept can be.

Downside:

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

Aduce

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.

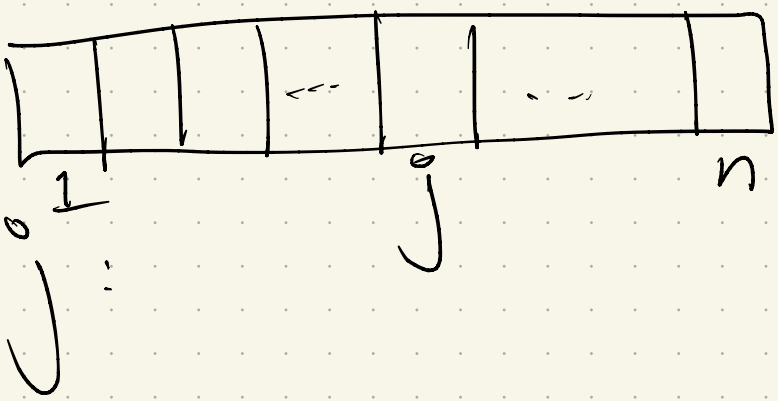
Next reading: Longest increasing subsequence (again)

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

Recap: Backtracking version

Recursion:

At each index j:



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.

Consider including A[j]:

- if A[i] ≥ A[j]:

- if A[i] is less:

Recursion:

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

Code version: don't pass arrays! Why?

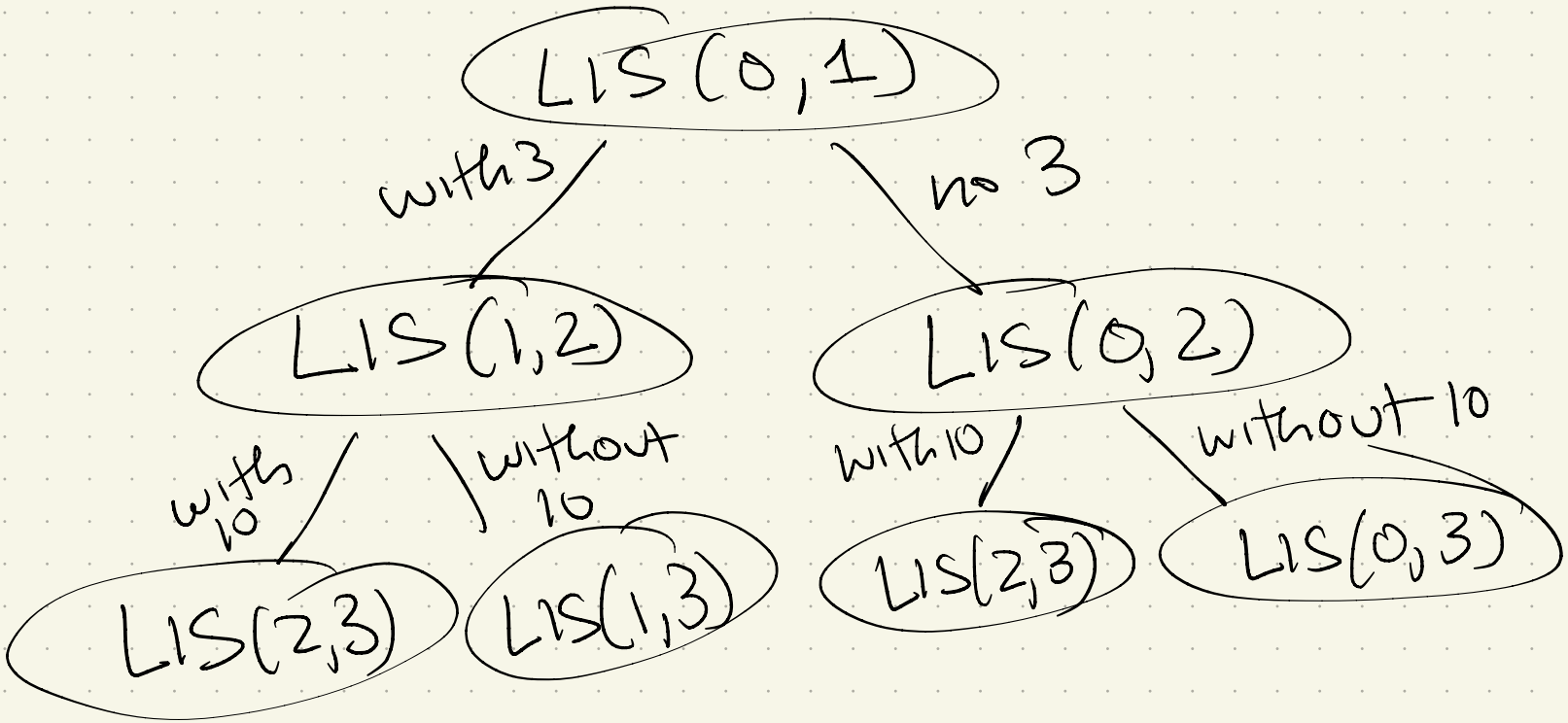
plus the "main":

```
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}
```

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

Example: A: [3, 10, 2, 11, 5, 7]

↳ [-∞, 3, 10, 2, 11, 5, 7]



Question: Is this function pure?

Memoize: What are we recomputing?

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

How should we store?

```
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}
```

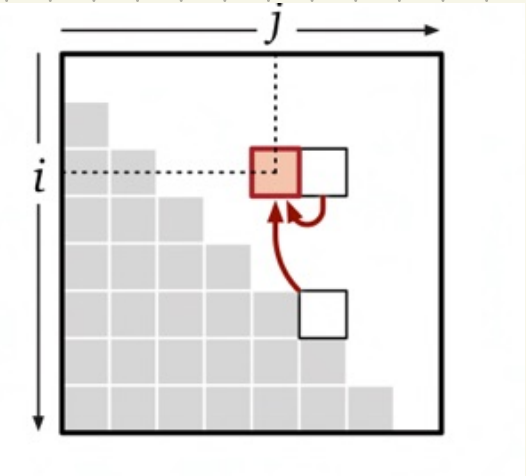
Now, can we do the same trick as Fibonacci memoization, & convert to something loop-based?

Aside: Why should we?

Rethink:

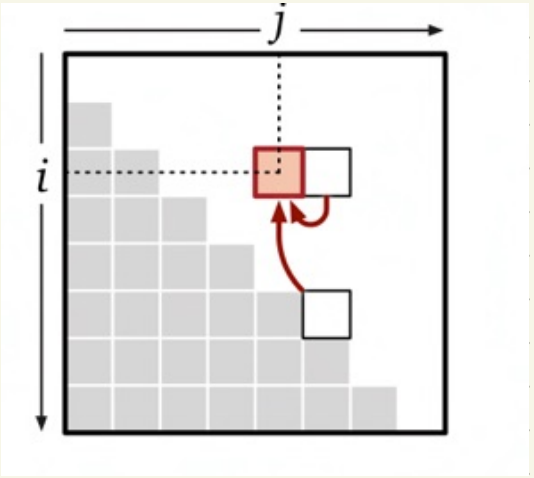
To fill in L[i][j], what do I need?

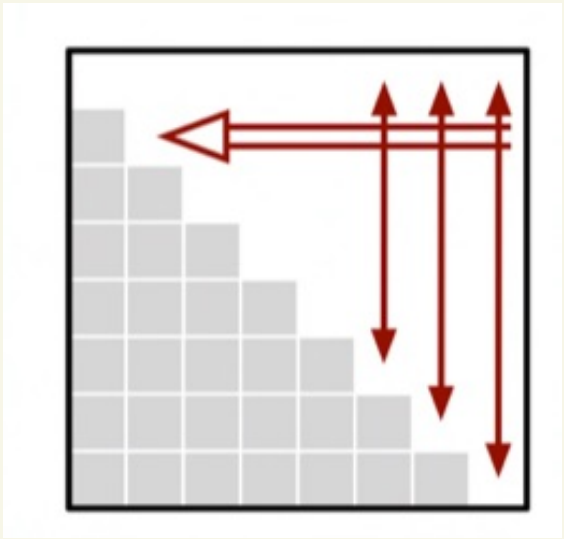
$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max \begin{Bmatrix} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$



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

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

Edit(,)

$= \min$ {