Algorithms - Spring 2025

Dynamic Programming

Recap

- HW2 - due Monday
  - Readings posted through then

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]
```

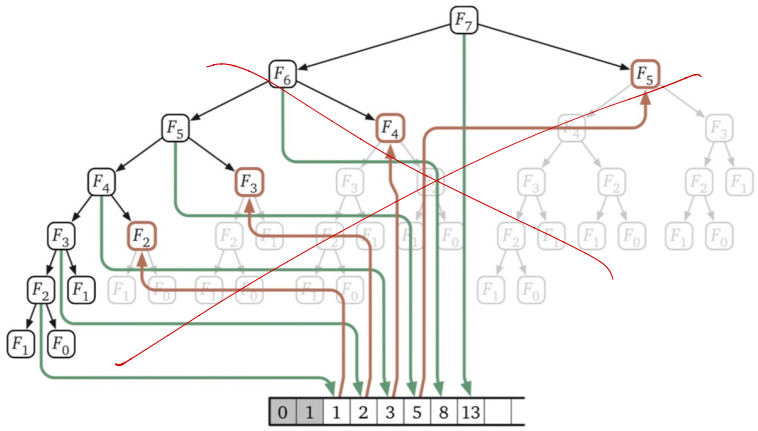


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

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

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

So - 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
```

Either way:

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
```

But wait — F_n is exponential!

Specifically,

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

$$\phi =$$

$$\hat{\phi} =$$

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?

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!

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 cases.
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.

Text segmentation : from Ch 2!

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

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

```
⟨⟨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
```

Can we try the same trick?

Memoization

Think about our recursion: calling splittable(i,j) quite a bit.

After first time it's computed, store the answer.

Then, later calls just look it up!

How many calls?

How to store?

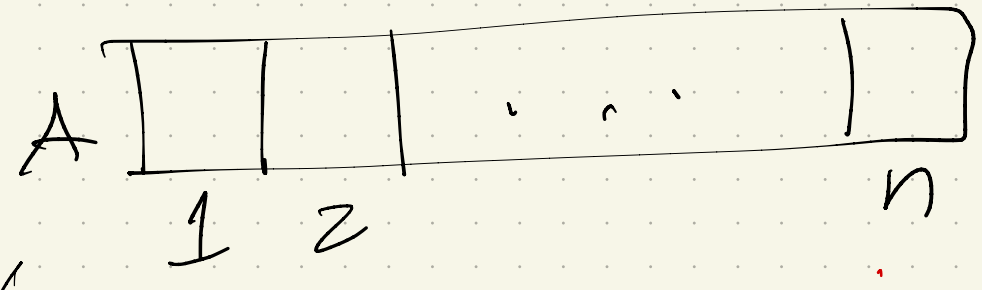
```
⟨⟨Is the suffix A[i..n] Splittable?⟩⟩
SPLITTABLE(i):
  if i > n
      return TRUE
  for j ← i to n
      if ISWORD(i, j)
          if SPLITTABLE(j + 1)
              return TRUE
  return FALSE
```

Run time / space:

Recap: Longest Increasing Subsequence

Why "jump to the middle"?
Need a recursion!

First: how many subsequences?



↳ could use or skip each #,
⇒ 2^n worst case

Backtracking approach:

At index i:

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 $A[i]$.

Store last "taken" index i.

Consider including A[j]:

- if $A[i] \geq A[j]$
  - ↳ must skip!
- if A[i] is less:
  - try both options

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: (helper function)

```
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}
```

Problem – what did we want??

LIS(A[1..n]).

So: don't forget our "main":

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

Problem:

Next: memoize?

What sort of calls are we making often? Can we save them, & avoid recomputing over and over?

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

```
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}
```

Here:

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

This is a recursion, but think for a moment of it as a function.

After computing, store values!

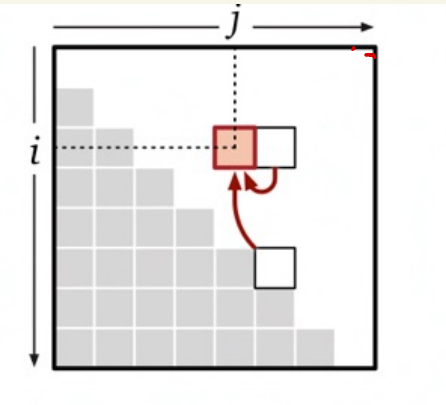
How many values to store?

How long to compute each?

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

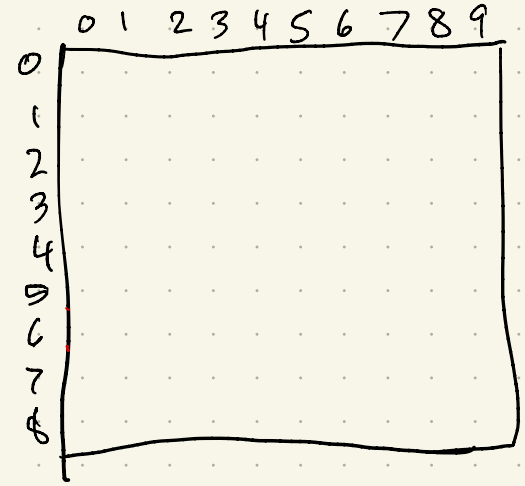
Rethink:

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



So, go in that order!

Ex: A = [10 2 4 1 6 11 7 9]



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]
```

Picture: