Algorithms - Spring 2025

Dynamic Programming

Recap

- HW 2 - due Monday
- Readings posted through then
- For emails:
  If I don't reply w/in 24 hours, please ping me again!

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]
```

"memoize" repeated calls

stack

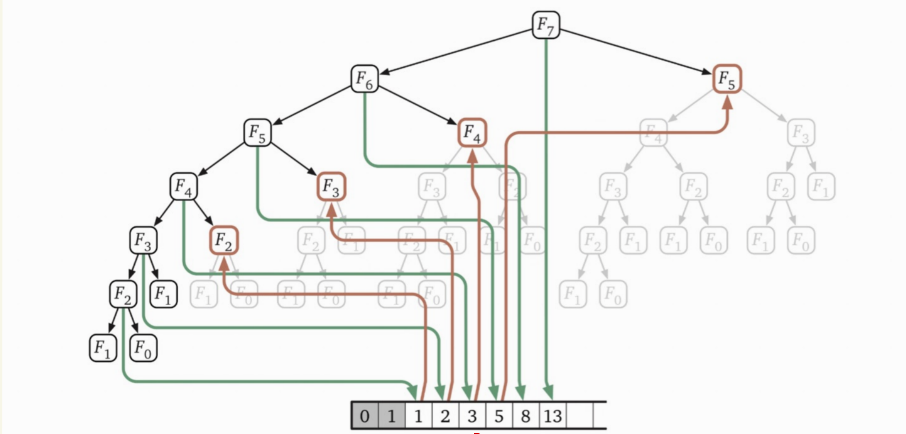


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

Reset our view:

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

becomes a loop

Cost: space $O(n)$

$O(1)$ space

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

H's ♡ section: Can actually do better!

Fancy math tricks

$$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}=\begin{bmatrix}0\\1\end{bmatrix}\begin{matrix}F_0\\F_1\end{matrix}$$

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

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

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

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

<u>Proof</u>: 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: Consider $\begin{bmatrix}0&1\\1&1\end{bmatrix}^{n}\begin{bmatrix}0\\1\end{bmatrix}=$

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

$$=\begin{bmatrix}0\cdot F_{n-2}+1\cdot F_{n-1}\\1\cdot F_{n-2}+1\cdot F_{n-1}\end{bmatrix}=\begin{bmatrix}F_{n-1}\\F_n\end{bmatrix}\ \blacksquare$$

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:

$O(\log n)$??

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,

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

$$\phi = \frac{1+\sqrt{5}}{2}$$

$$\hat{\phi} = \frac{1-\sqrt{5}}{2}$$

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

$\left(\frac{1+\sqrt{5}}{2}\right)^n \longrightarrow \log_\phi n$ bits to write down

Why? k digit number: $O(2^k)$

$\underbrace{\_ \ \_ \ \_ \ \_ \ \_}_{0 \text{ or } 1 \text{ in each}}$

Clarification:

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

If a # $\le$ 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: $n \log n$

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

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!

Key: data 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 <u>this</u> in
a loop + data structure
format.

<u>Finally</u>: analyze <u>both</u>
space & time

Rant about greed:

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

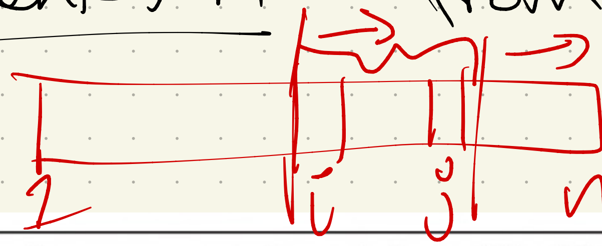
But- often such intuitive strategies fail.

Dynamic programming & backtracking will always work.

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

Text segmentation: from Ch 2!

Text!



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

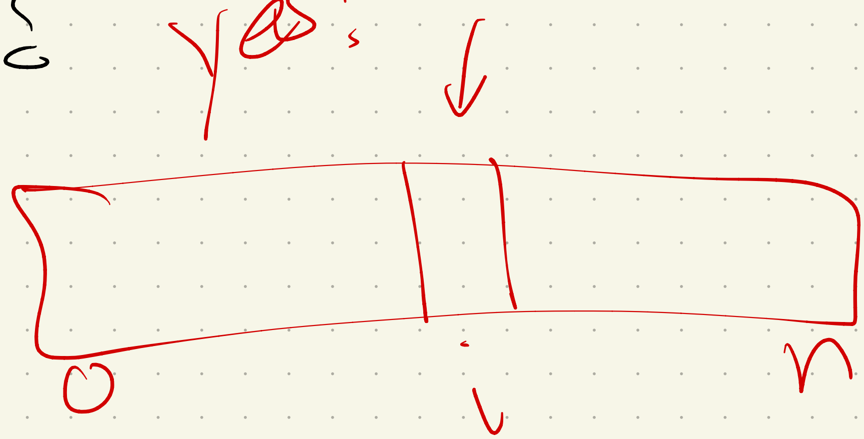
$$
Splittable(i) = \begin{cases} \text{TRUE} & \text{if } i > n \\ \displaystyle\bigvee_{j=i}^{n} \big(\text{IsWord}(i,j) \wedge Splittable(j+1)\big) & \text{otherwise} \end{cases}
$$

OR AND

```
⟨⟨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? yes!

Text:



Memoization

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

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

Then, later calls just look
it up!

How many
calls?

once!

Yes/No

How to store?

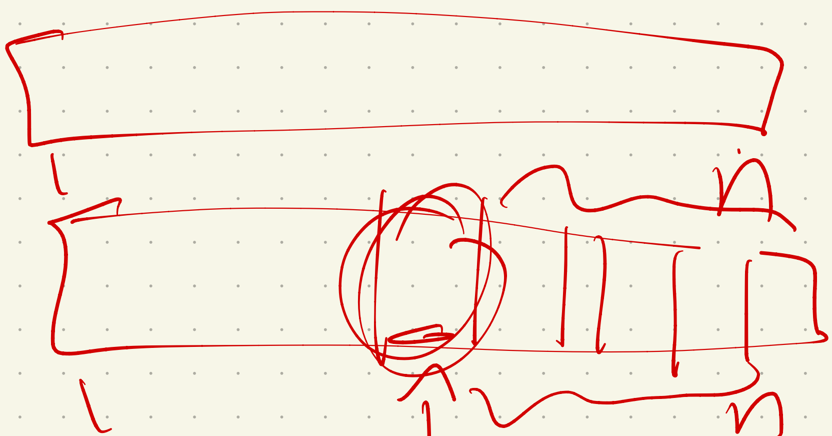
array!

```
⟨⟨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
```

Splittable[i]

Text

Splittable



for $i \leftarrow n$ down to 1

fill Splittable(i) → for $j \leftarrow i$ to n

Run time / space:

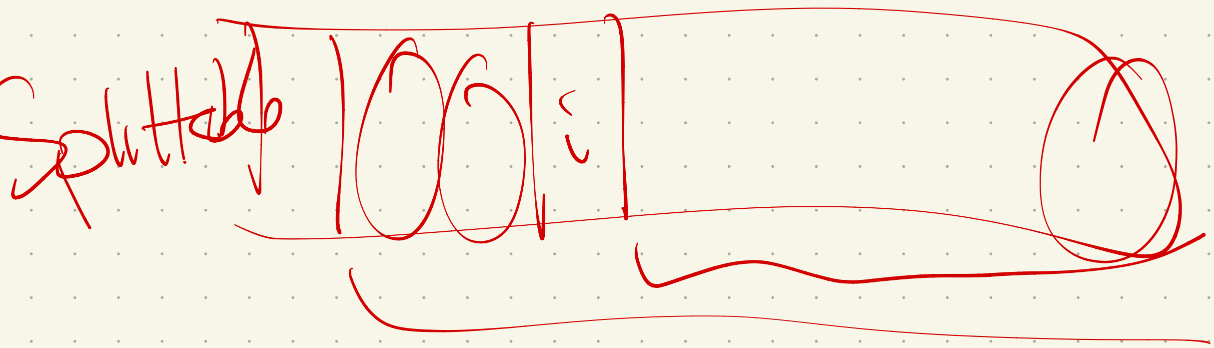
Space: adding $O(n)$ for extra bool. array

$$\sum_{i=1}^{n} \left(\sum_{j=i}^{n} (n-i) \right)$$

lowering each time

$$= \sum_{i=1}^{n} \left(\sum_{k=1}^{n-i} k \right)$$

$O(n)$ space



$$(n-1)+(n-2)+(n-3)+\cdots+1$$

$$= \Theta(n^2)$$

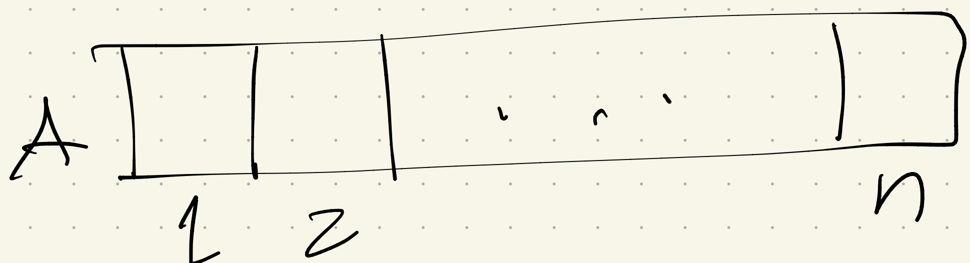
compare with backtracking!

Recap: Longest Increasing Subsequence

Why "jump to the middle"?

Need a recursion!

First: how many subsequences?



↳ could use or skip each #, so 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:

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

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?

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

```
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:

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

This is a recursion, but think for a moment of it as a <u>function</u>.

After computing, store values!

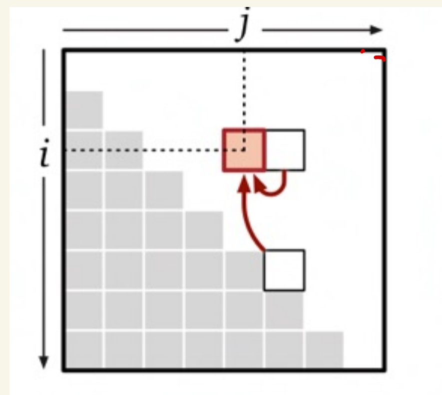
How many values to store?

How long to compute each?

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

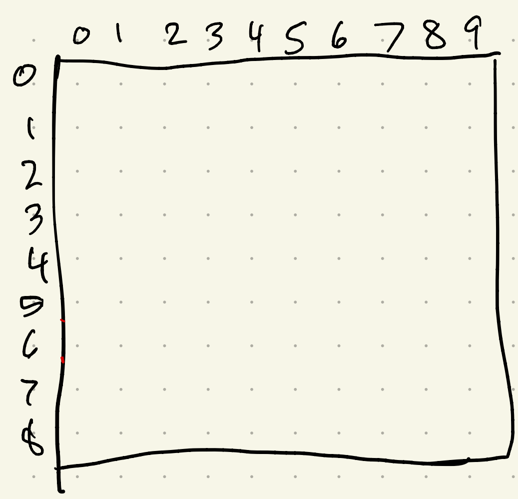
Rethink:

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: