Algorithms

Dynamic Programming (part 1)

Recap:

- #Scholarstrike today:
  we'll cover algorithmic fairness
  a bit later in course
  (Ch. 4)
  I'll be assigning an extra
  credit reading in Perusall
  (optional!)
- HW - due next week
  planning on oral grading,
  but stay tuned -
  sign-up details coming
  on Canvas
  (over backtracking only)
- Reading: every Sunday & Thursday!

Dynamic Programing

- a fancy term for smarter recursion :

memoization

- Developed by Richard Bellman in mid 1950s

("programming" here actually means planning or scheduling)

Key: When recursing, if many recursive calls to overlapping subcases, remember prior results and don't do extra work!

Simple example:

Fibonacci Numbers: familiar

$F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ $\forall n \geq 2$

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Directly get an algorithm:

assuming $n \geq 0$

```
FIB(n):
    if n < 2:
        return n                 ] base cases
    else:
        return FIB(n-1) + FIB(n-2)
```

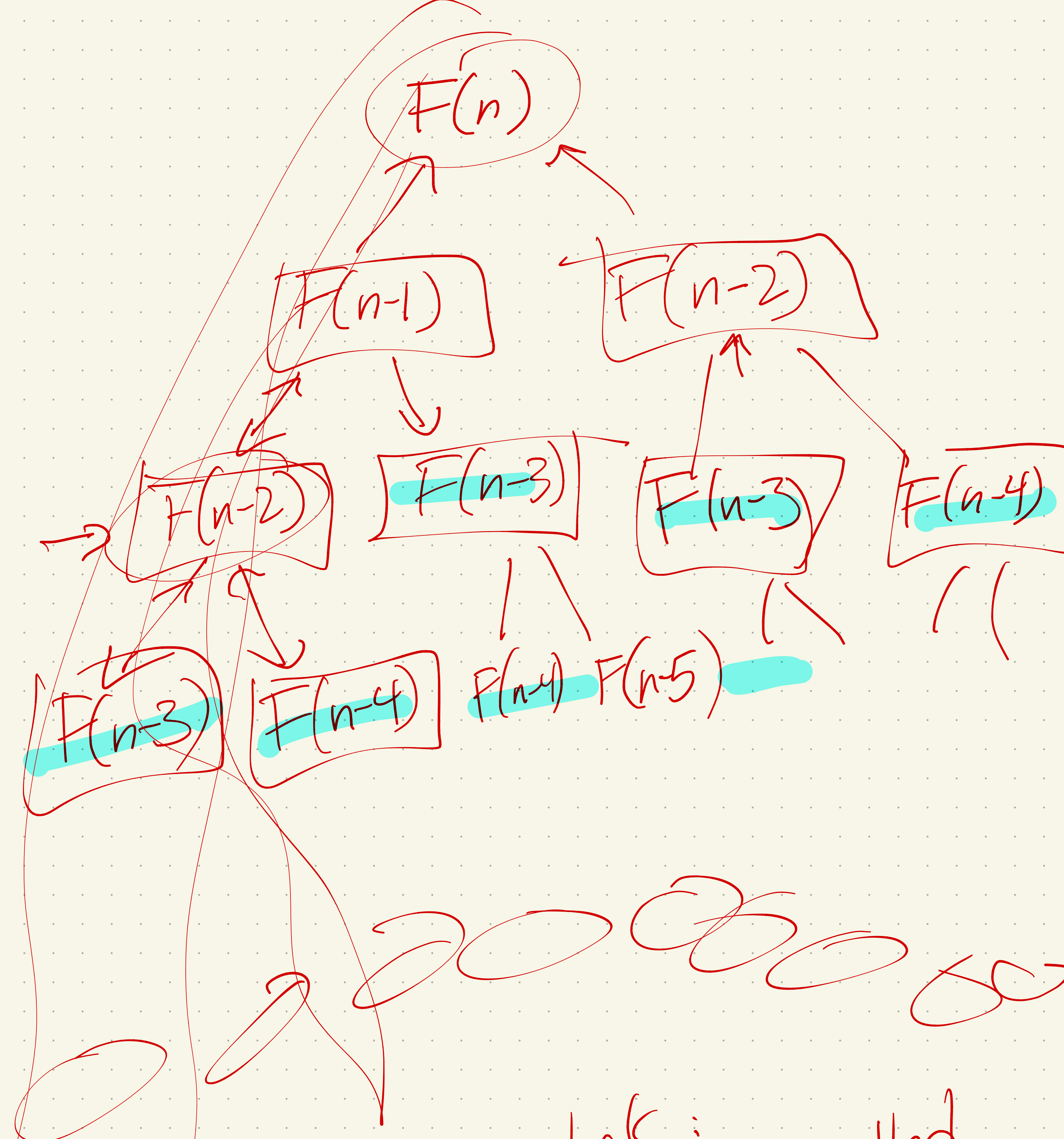
Runtime:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

↳ go check discrete math reference

exponential

$= O(\phi^n)$ $\phi = \frac{1+\sqrt{5}}{2} > 1$

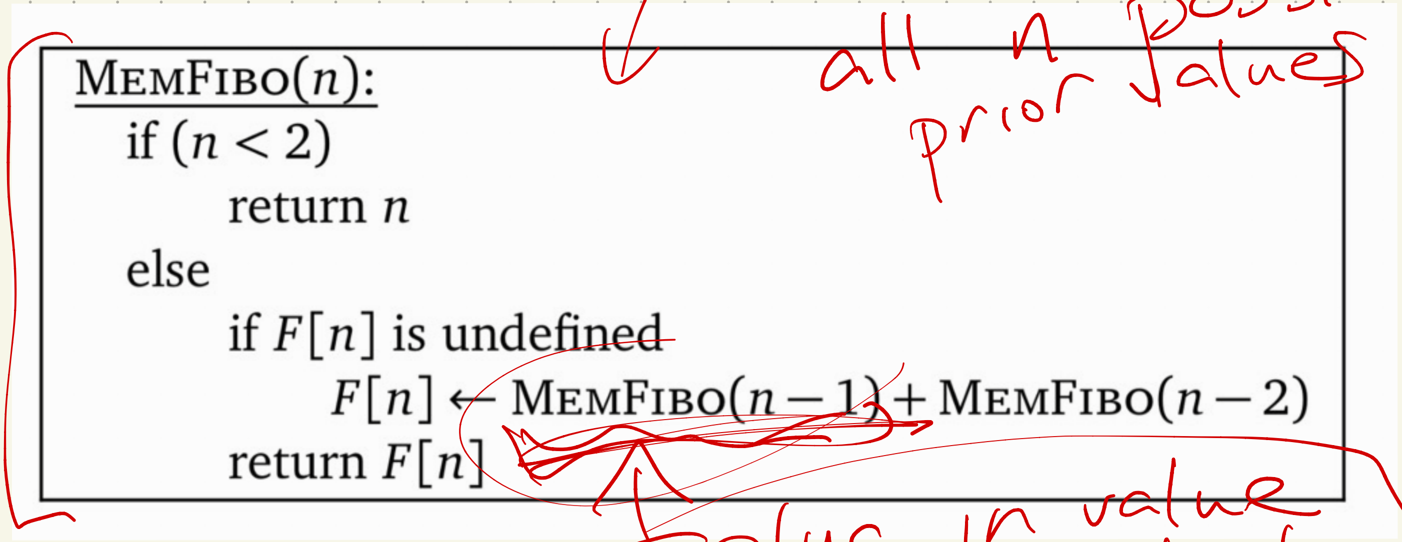


Program stack:

F(n-4) is called 4 times!

F(5) will happen a lot

Applying memoization:



```
MemFibo($n$):
    if ($n < 2$)
        return $n$
    else
        if $F[n]$ is undefined
            $F[n] \leftarrow$ MemFibo($n-1$) + MemFibo($n-2$)
        return $F[n]$
```

need to store all n possible prior values

plug in value if already called otherwise call function

Remember calculated values.

(stored by computer)

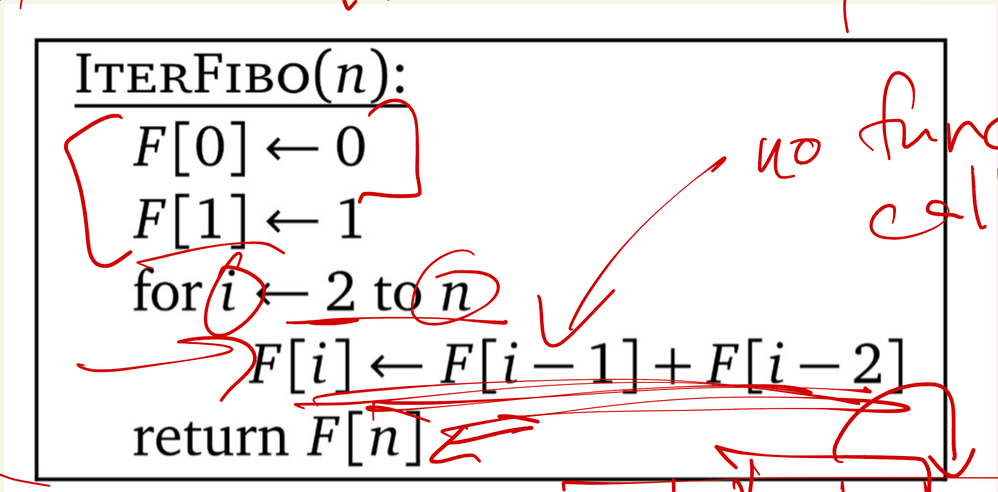
"memo function"

(lookup cost)

applying some data structure

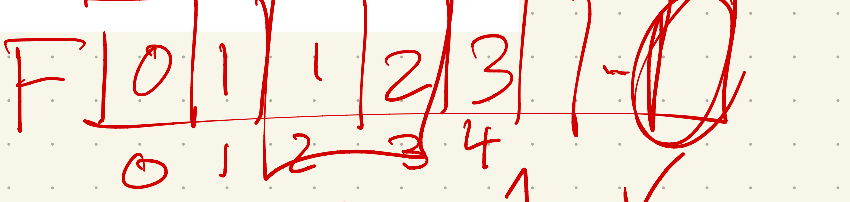
Better yet: Build DS specifically

Here: array to remember values F



```
ITERFIBO(n):
    $F[0] \leftarrow 0$
    $F[1] \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        $F[i] \leftarrow F[i-1] + F[i-2]$
    return $F[n]$
```

no function calls!



Correctness:

induction! BC: n=0 or 1 ✓

Ind Hyp: works for $i < n$

IS: consider n: correct formula!

Run time & space:

easier to analyze!

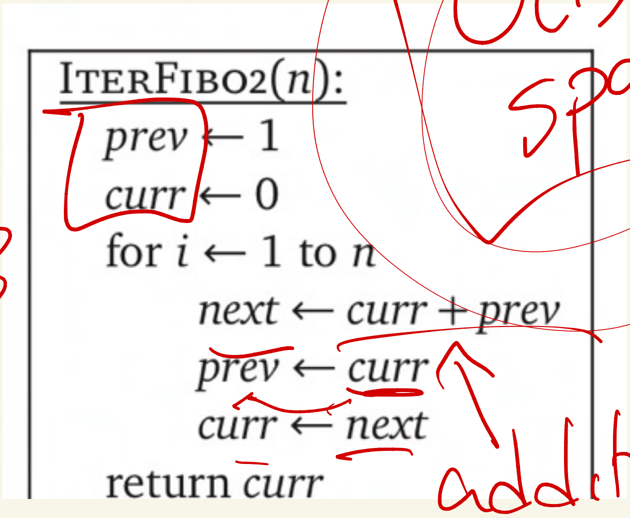
for loop! 1 addition + one store per iteration

⇒ O(n) time

O(n) space to write F[1..n]

Even better: Do we need that entire array?

No! Why?



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

O(1) space

addition

i = ~~1~~ 2

prev: ~~1~~ ~~0~~ ~~1~~ ~~1~~ 2

curr: ~~0~~ ~~1~~ ~~1~~ ~~2~~ 3

next: ~~1~~ ~~1~~ ~~2~~ ~~3~~ 5

His ♡ section: Can actually do better!

Fancy math tricks:

$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}=\begin{bmatrix}\overset{0}{\cancel{0\cdot1+1\cdot0}}\\1\end{bmatrix}$

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

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

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

$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}2\\3\end{bmatrix}=\begin{bmatrix}3\\5\end{bmatrix}$ ...

$\begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}F_i\\F_{i+1}\end{bmatrix}=\begin{bmatrix}F_{i+1}\\F_{i+2}\end{bmatrix}$

mult!

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

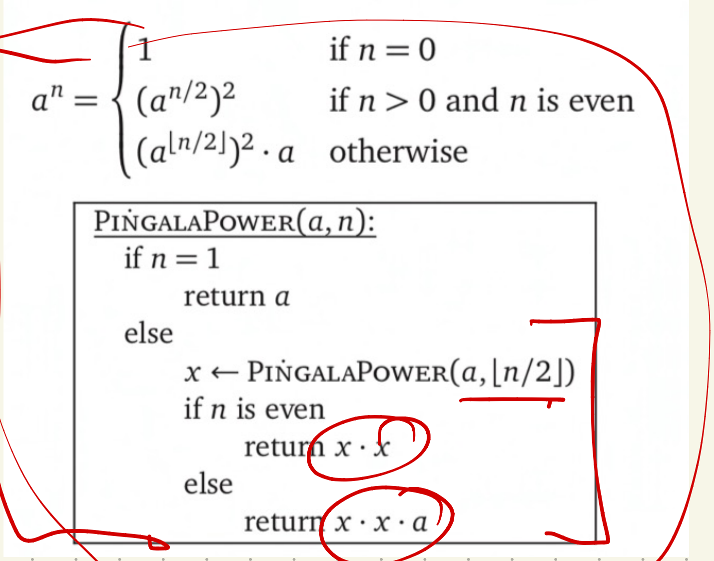
Why? doesn't seem helpful.

Pf: by induction!

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

So - back to chapter 1!

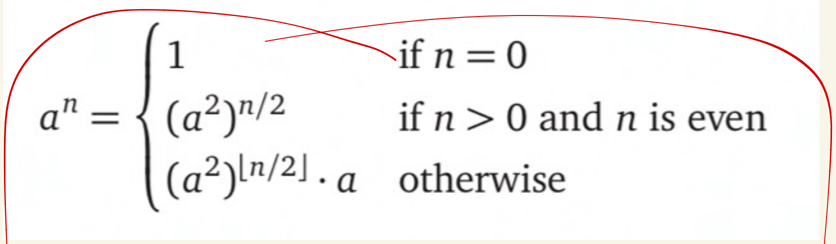
$a = \begin{bmatrix}0&1\\1&1\end{bmatrix}$



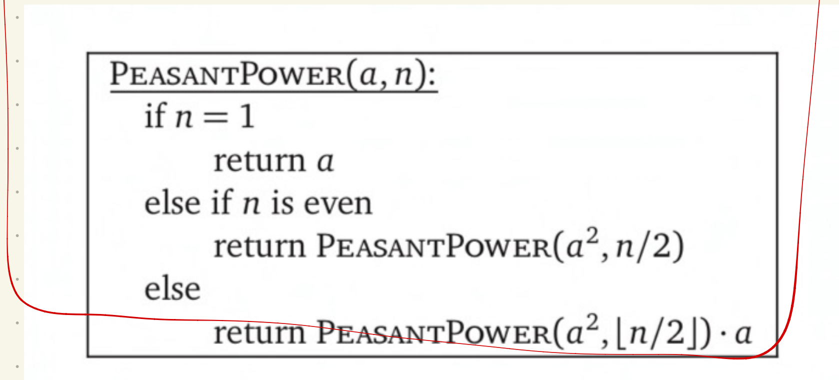
$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a & \text{otherwise} \end{cases}$$

```
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:

a^n: take $O(\log n)$ matrix multiplication

→ take ~~[illegible]~~ 8 normal multiplications

$\Rightarrow O(\log n)$

But wait – F_n is exponential!

Specifically,

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

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

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

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

give you 2^n how many bits to write it?

$\underbrace{\leftarrow \leftarrow \leftarrow \leftarrow \leftarrow \overset{0 \text{ or } 1}{}}_{k \text{ bits}}$: $< 2^{k+1}$

→ n bits.

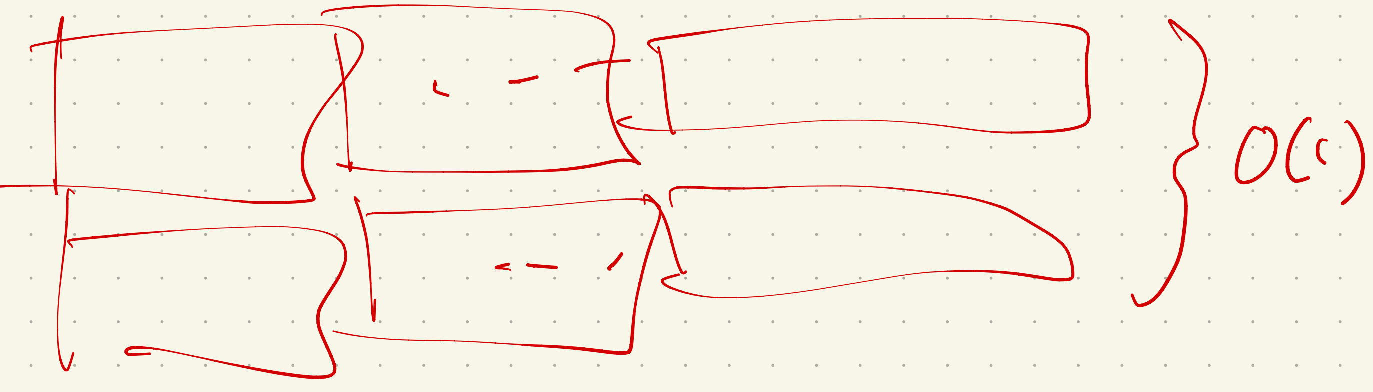
Clarification:

our earlier algorithms use $O(n)$ additions or subtractions. atomic, so $O(1)$

If a # ≤ 64-bits - sure!

But larger? 2^n # takes $O(n)$ time to mult, add, etc

↳ ϕ^n number, take n time to multiply



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!

deceptively simple.

↳ simple data structure, "obvious" loop to convert to

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.

Next problem: back to LIS

Recall:

Input: Array A[1..n] of #s

Output: length of the longest subsequence in A such that each element is ≤ next element

Subsequence:

take A, & delete any values

length 3

Ex: 10 2 4 1 6 11 7 9

1 5

Aside: will greed work? (No)

~~Ch1~~-recap

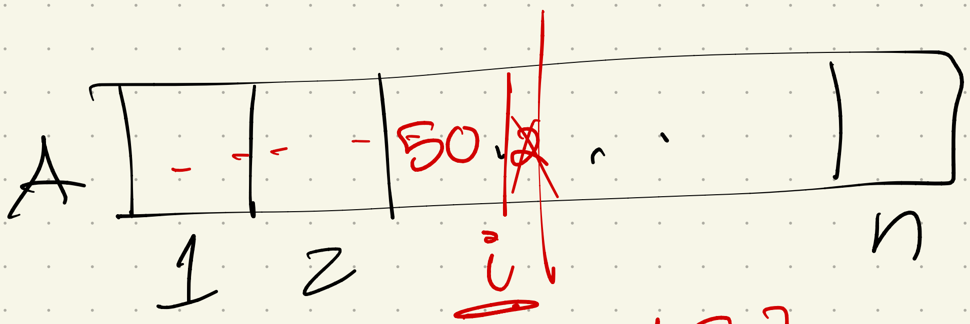
Longest Increasing Subsequence

copy of prior lecture

Why "jump to the middle"?

Need a recursion!

What is our decision?



Do I include A[i] or not?
- if A[i] is too small, skip
- if A[i] is "big enough", try both

Aside: How many subsequences are there?

| 2 | 2 | 2 | … | 2 | $= 2^n$

1 … n

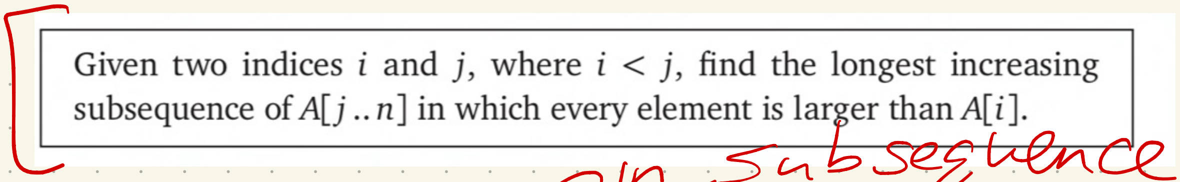
Backtracking approach:

At index i: need last element included

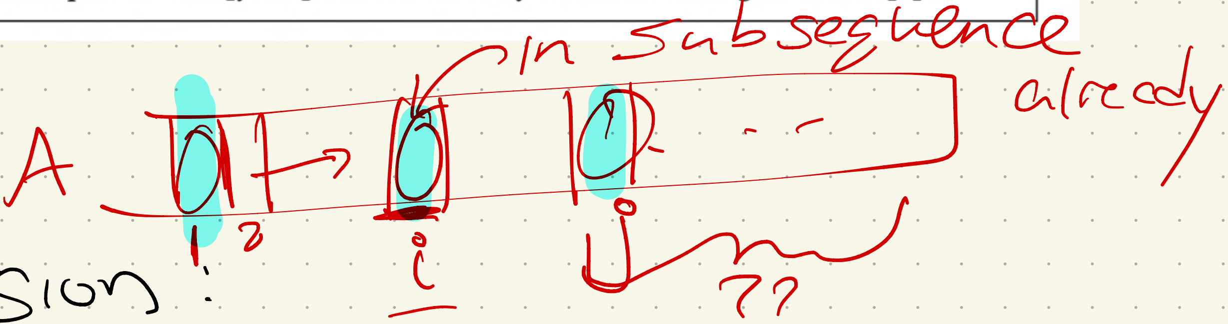
if A[i] < last element, skip

if A[i] > last, try both ways

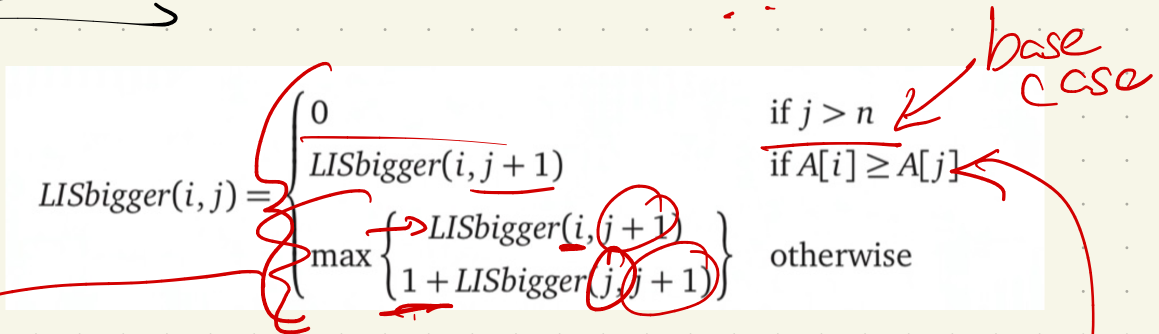
Result:



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 $A[i]$.



Recursion:



base case

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

$j \longmapsto j+1$

$\Rightarrow$ base case: $j = n+1$

A[j] is too small

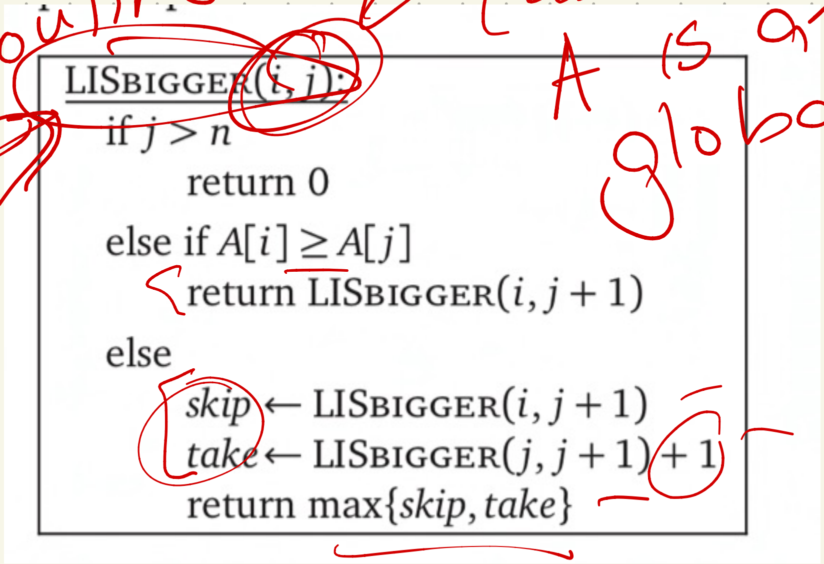
$\rightarrow$ A[j] is big enough, so try both ways

Code version:

subroutine:

2 #5

(assuming A is a global)



```
LISBIGGER(i, j):
    if $j > n$
        return 0
    else if $A[i] \geq A[j]$
        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??

input: A[1..n]

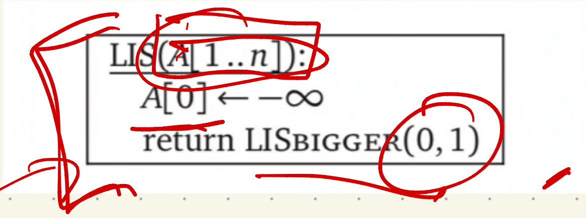
output: length of LIS

what are i + j?

Could call LIS(1, 2) – what would happen?

would include A[1] always – bad!

So:



```
LIS(A[1..n]):
    A[0] ← $-\infty$
    return LISBIGGER(0, 1)
```

wrapper

$-\infty$, 10, 1, 2, 3, 4

should get 4

Runtime:

$$L(n) \leq \underline{2}\,\underline{L(n-1)} + O(1)$$

$$L(0) = 1$$

Not Master Thm friendly

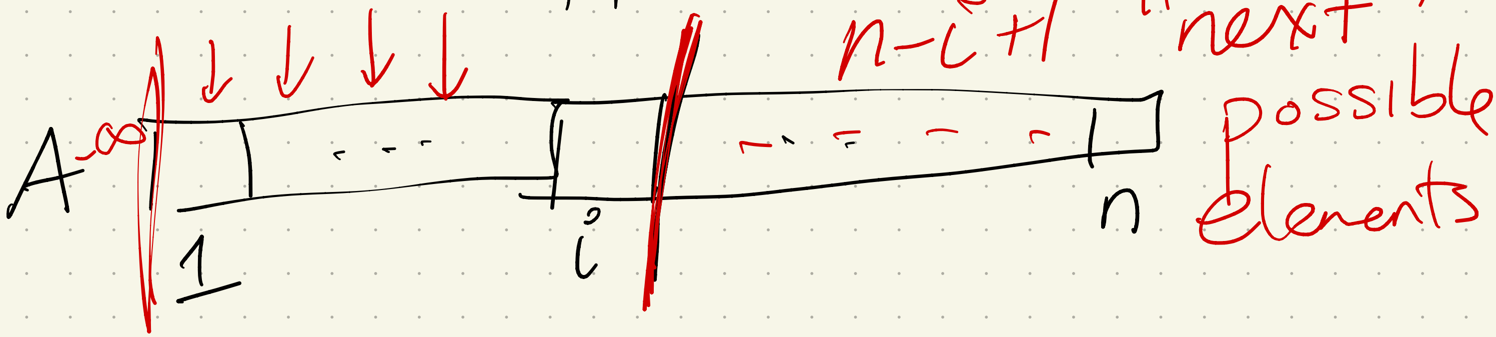
no $\frac{n}{c}$ in recursion

Looks like Towers of Hanoi:

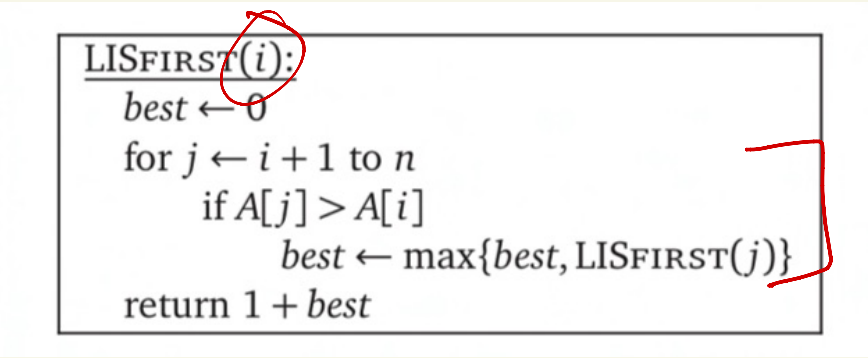
$$O(2^n)$$

(can solve, seen in book)

Alternative approach:



At index i, choose next element in the sequence. (means n calls, not 2!)



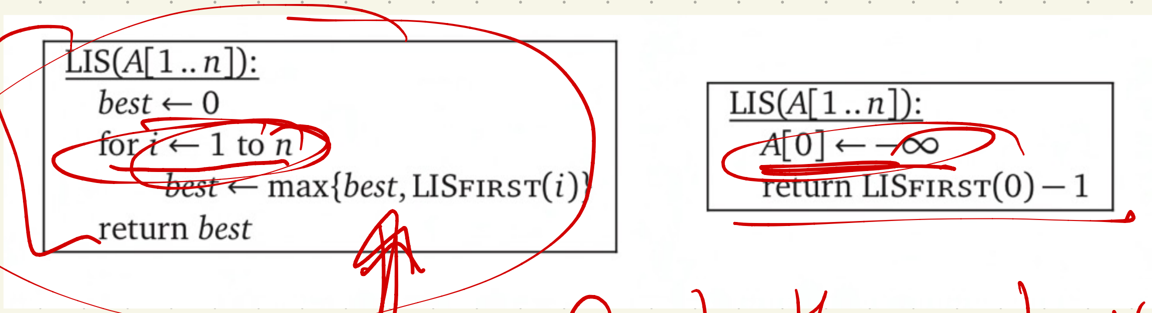
```
LISFIRST($i$):
    $best \leftarrow 0$
    for $j \leftarrow i+1$ to $n$
        if $A[j] > A[i]$
            $best \leftarrow \max\{best, \text{LISFIRST}(j)\}$
    return $1 + best$
```

↑ subroutine

Issue - what was our goal again??

Find LIS of A[1..n], LISFIRST(1). ✗

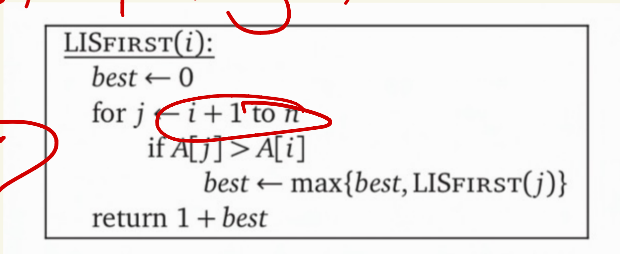
Final version:



```
LIS(A[1..n]):
    best ← 0
    for i ← 1 to n
        best ← max{best, LISFirst(i)}
    return best
```

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

choosing first thing to include



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

helper

Runtime:

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

(BAD)