CS 331, Fall 2024

Lecture 4 (9/9)

Today:
- Fibonacci
- Word RAM model
- Largest jump
- Largest subsequence sum

## Fibonacci (Part III, Section 1)

Classic interview question / "gotcha"

Input: $n \in \mathbb{N}$

Output: $F_n$, $n^{th}$ Fibonacci number

Recursive definition:

$$F_0 = 1, \; F_1 = 2, \; F_2 = 3, \; F_3 = 5, \; F_4 = 8$$

$$\ldots \; F_n = F_{n-1} + F_{n-2}$$

Observation: $F_n = \exp(\Theta(n))$

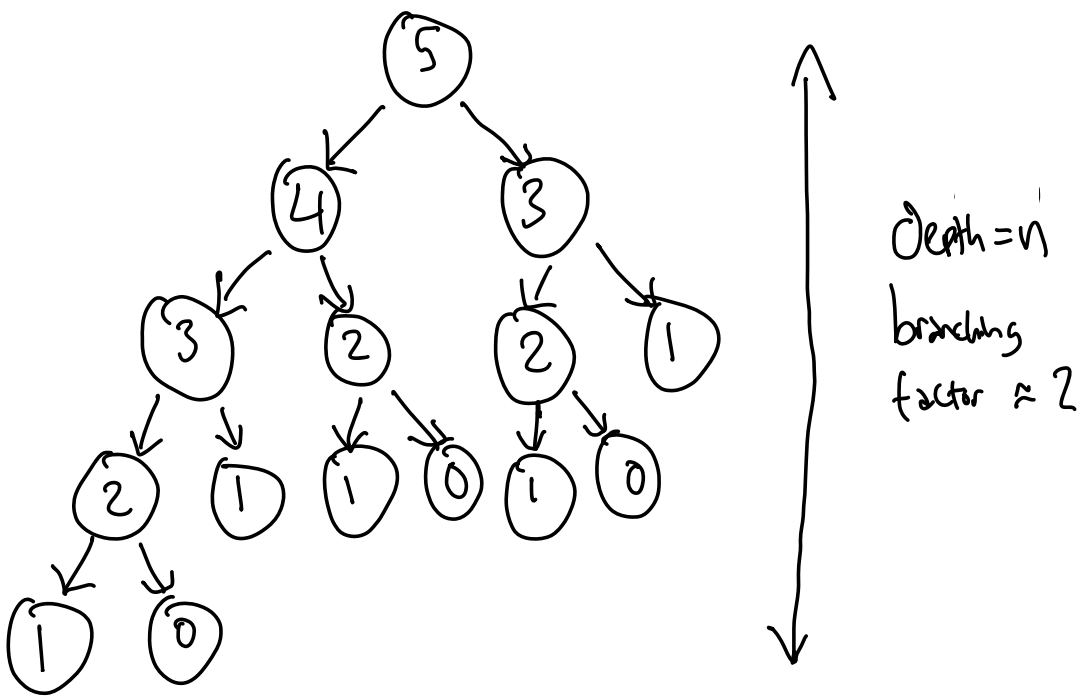
Aside: In fact, $F_n \approx \varphi^n$, where $\varphi \approx 1.618$ is "golden ratio"

First try:

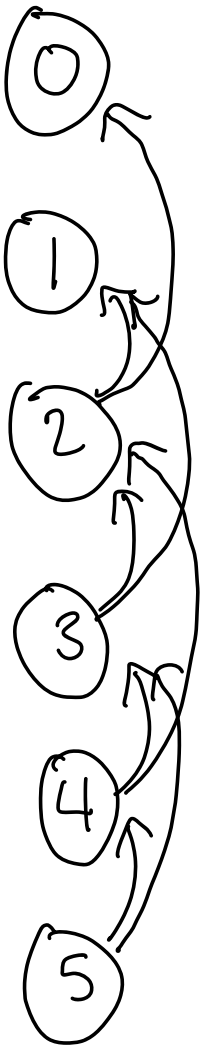
```
FibNaive(n):
    If n == 0: Return 1
    If n == 1: Return 2
    Return FibNaive(n-1) + FibNaive(n-2)
```

✗ Warning: this will take exponential time!

e.g. FibNaive(5):



Idea: do things bottom-up to avoid recomputation

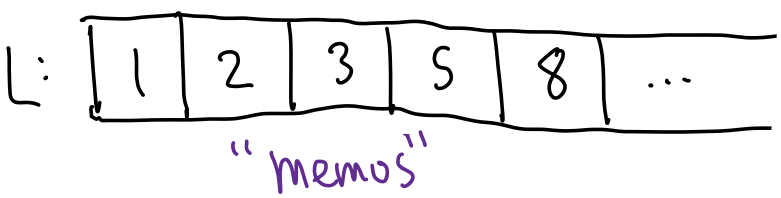



Intuition: induction is typically proven from the base case.

```
Fib(n):
    L ← Array.Init(n+1)      // L[i] = F_{i+1}
    L[1] ← 1
    L[2] ← 2
    For 3 ≤ i ≤ n+1: L[i] ← L[i-1] + L[i-2]
    Return L[n+1]
```

Key takeaways:

1) order matters! (top-down vs. bottom-up)
2) save your work




"memos"

Dynamic programming:

> Smarter and more flexible recursion

(up to now, focus is "divide-and-conquer")

Applications:

- Coding interviews (seriously.)
- Reinforcement learning (founder: Bellman)
- Game theory / economics

Basic idea: 1) define subproblems you want to save / "memoize"

2) define order to solve them

It can be hard to understand at first.

Payoffs enormous: exponential $\Rightarrow$ near-linear time!

Next 4 lectures: many examples as a field guide

# Word RAM model (Part I, Section 7)

Quiz: how fast is Fib?

```
... // base cases
For 3 ≤ i ≤ n+1:  L[i] ← L[i-1] + L[i-2]
Return L[n+1]
```

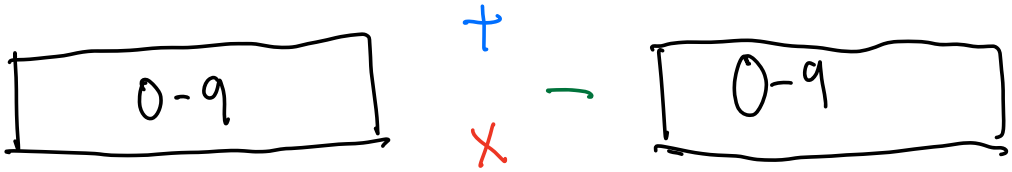
$O(n)$? No!

Remember, $L[i] = \exp(\Theta(n))$ is a $\Theta(n)$-digit number!

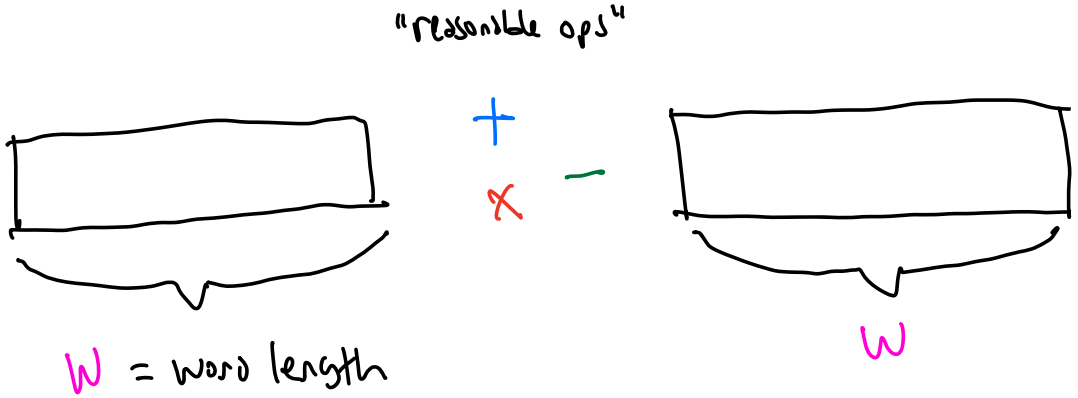
$O(n^2)$ = $n$ steps × $O(n)$ time per step

(add $n$-digit numbers)

Bit complexity model: bit/digit ops take $O(1)$ time.



Word RAM model:




Assume: takes $O(1)$ time.

Think: $W = 64$, maybe few 100's tops.

It's reasonable to assume

$$\log(n) \leq W.$$

Will do henceforth unless stated otherwise

Unreasonable to assume

$$n \leq W.$$

Keep an eye out!

**Examples**

- Incrementing a for loop counter $i \in [n]$ is $\log(n)$ digits, $O(1)$ time.
- Comparing two numbers in sorting if both $\leq 2^W$, $O(1)$ time.

Bonus algo: Fib(n) in $O(n \log(n))$ time

(See Part II, Section 5.3)

## Largest jump (Part III, Section 2.1)

Input: L is a list of n elements in $\mathbb{R}$

Output: $(i, j)$ with $1 \le i \le j \le n$

maximizing $L[j] - L[i]$

Example

Day

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| GME stocks | 8 | 12 | 4 | 9 | 17 | 1 | 13 |

How to maximize profit?

| Buy ↓ / Sell → | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 4 | -4 | 1 | 9 | -7 | 5 |
| 2 | | 0 | -8 | -3 | 5 | -11 | 1 |
| 3 | | | 0 | 5 | 13 | -3 | 9 |
| 4 | | | | 0 | 8 | -8 | 4 |
| 5 | | | | | 0 | -16 | -4 |
| 6 | | | | | | 0 | 12 |
| 7 | | | | | | | 0 |
| Best: | 0 | 4 | 0 | 5 | 13 | 0 | 12 |

Time: $O(n^2)$ (for each $i, j$, compute $L[j] - L[i]$)

How to improve?

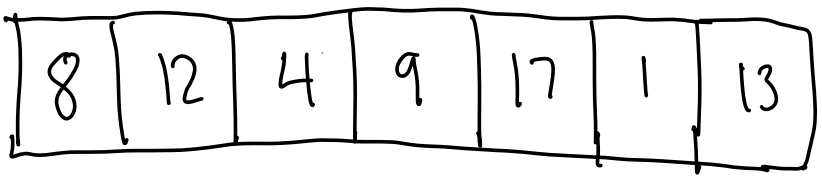
Idea: What do we need to know for $Best[j]$?

$$Best[j] = \max_{i \in [j]} L[j] - L[i] = L[j] - \underbrace{\min_{i \in [j]} L[i]}_{\text{maintain}}$$

Faster algo, take 1:

① Pass over list, maintain running min.

e.g.




MinUpTo 8 8 4 4 4 1 1

$$MinUpTo[j] = \min_{i \in [j]} L(i)$$

② Compute all $Best[j] = L[j] - MinUpTo[j]$

Both steps $O(n)$ time. Why?

$$MinUpTo[j] = \min\left(\underbrace{MinUpTo[j-1]}_{\text{memoized}}, L[j]\right)$$

Faster algo, take 2:

All subproblems: $(i, j)$, $n + \binom{n}{2} = \Theta(n^2)$ of them.

Special subproblems: $(i^*, j)$ where $i^* = \text{MinUpTo}[j]$

$\text{Best}(j) = L[j] - L(i^*)$.

only $O(n)$ of them.

Are Special subproblems harder?

Not with memoization!

$$\text{Best}(j) = \max\left(0, \text{Best}(j-1) + \underbrace{L[j] - L[j-1]}_{\text{difference in sell price}}\right)$$

$0$: buy on day $j$, $i^* = j$

$\text{Best}(j-1)$: buy on day $< j$, keep $i^*$ the same

No need for MinUpTo. $O(1)$ time per $\text{Best}(j)$
$= O(n)$ total.

General strategy: 1) Design any correct algo.

2) Repeated structure? Memoize!

3) Go back to step 1). Simplify.

(the hardest step)

## Largest subsequence sum (Part II, Section 2.2)

Input: $L$ is a list of $n$ elements in $\mathbb{R}$

Output: $(i, j)$ with $1 \le i \le j \le n$

maximizing $\underbrace{L[i] + L[i+1] + \ldots + L[j-1] + L[j]}_{\text{subsequence sum}}$

Example

| 25 | -60 | 7 | -13 | 30 |
|---|---|---|---|---|

First try:

| start \ end → | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 25 | -35 | -28 | -41 | -11 |
| 2 | | -60 | -53 | -66 | -36 |
| 3 | | | 7 | -6 | 24 |
| 4 | | | | -13 | 17 |
| 5 | | | | | 30 |

Naive: $O(n^2) \times O(n) = O(n^3)$

$O(n^2)$: # subproblems; $O(n)$: time / subproblem

Better: $O(n^2) \times O(1) = O(n^2)$

proceed row by row, left-to-right.

$$\sum_{k=i}^{j} L(k) = \underbrace{\sum_{k=i}^{j-1} L[k]}_{\text{memoized}} + L[j]$$

Time to simplify.

Can we define $O(n)$ special subproblems?

$Best(j)$ = largest subsequence sum ending on $j$

Recursive formula:

$$Best(j) = \max\left(L[j],\ Best(j-1) + L[j]\right)$$

$L[j]$: don't include $L[j-1]$

$Best(j-1) + L[j]$: include $L[j-1]$. may as well continue in best possible way

Again, $O(1)$ per subproblem using memoization

$\Rightarrow$ LSS in $O(n)$ time. ☺

(Kadane, at a CMU seminar)