Introduction to Computational Linguistics

Section

Olga Zamaraeva

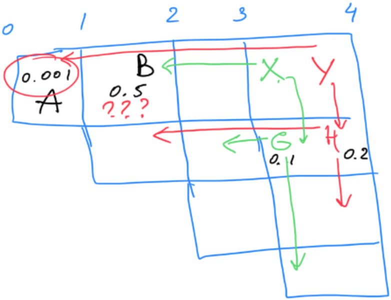
University of Washington

May 22, 2020

The greediness of CKY

- “Greedy” algorithms pick the highest score at every step
    - e.g. highest probability edge
- Not all greedy algorithms always return the correct result
- Why?
- Does CKY always return a correct result? Why?

CKY monotonicity



Grammar:

$P(X) \rightarrow B\ G = P(Y) \rightarrow A\ H$

$P(B) \gg P(A)$ ¿is it possible??

$P(G) < P(H)$

Example

Grammar	Lexicon
S → *NP VP*	*Det* → *that* | *this* | *the* | *a*
S → *Aux NP VP*	*Noun* → *book* | *flight* | *meal* | *money*
S → *VP*	*Verb* → *book* | *include* | *prefer*
NP → *Pronoun*	*Pronoun* → *I* | *she* | *me*
NP → *Proper-Noun*	*Proper-Noun* → *Houston* | *NWA*
NP → *Det Nominal*	*Aux* → *does*
Nominal → *Noun*	*Preposition* → *from* | *to* | *on* | *near* | *through*
Nominal → *Nominal Noun*	
Nominal → *Nominal PP*	
VP → *Verb*	
VP → *Verb NP*	
VP → *Verb NP PP*	
VP → *Verb PP*	
VP → *VP PP*	
PP → *Preposition NP*	

Figure 12.1 The $\mathscr{L}_1$ miniature English grammar and lexicon.

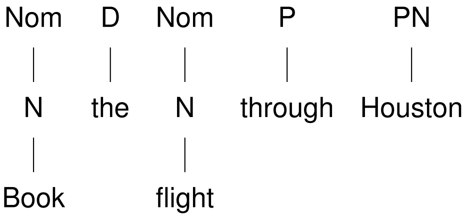
Bottom-up parsing

Book the flight through Houston

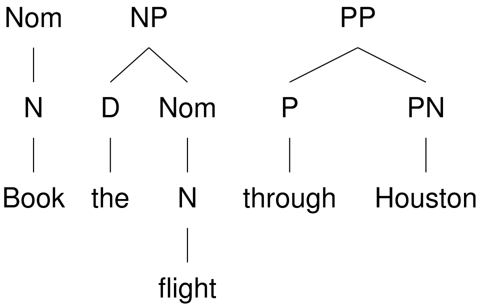
Bottom-up parsing

N	D	N	P	PN
Book	the	flight	through	Houston

Bottom-up parsing

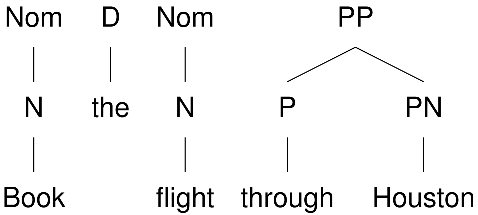


Bottom-up parsing

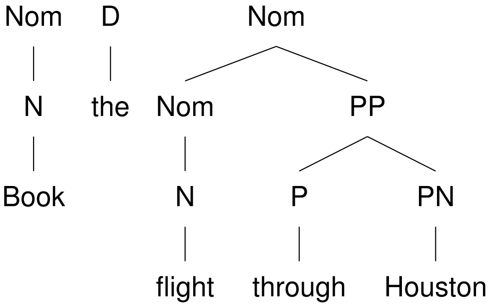


No more possibilities! Backtrack...

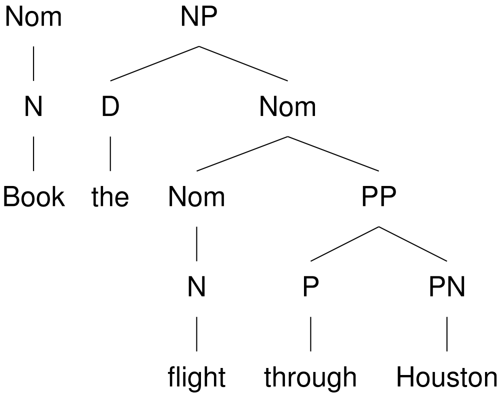
Bottom-up parsing



Bottom-up parsing



Bottom-up parsing



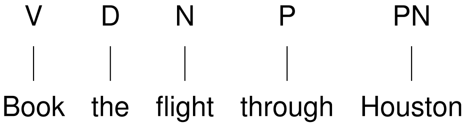
No more possibilities! Backtrack... Up to where?..

Bottom-up parsing

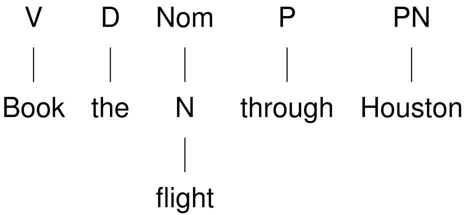
Backtrack to the very beginning, actually!

Book the flight through Houston

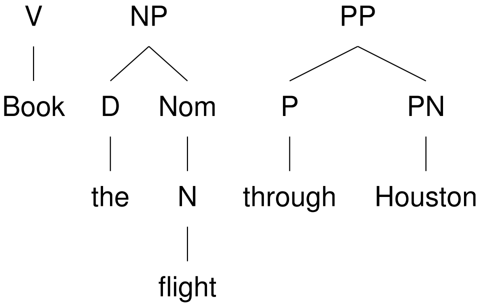
Bottom-up parsing



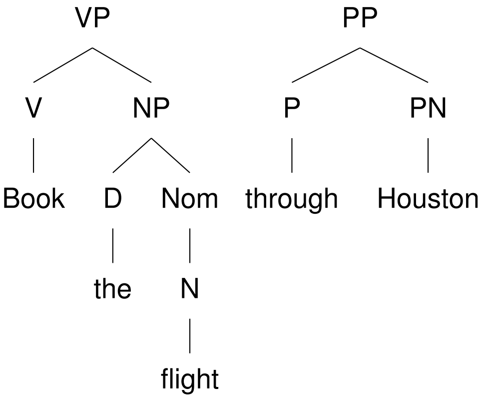
Bottom-up parsing



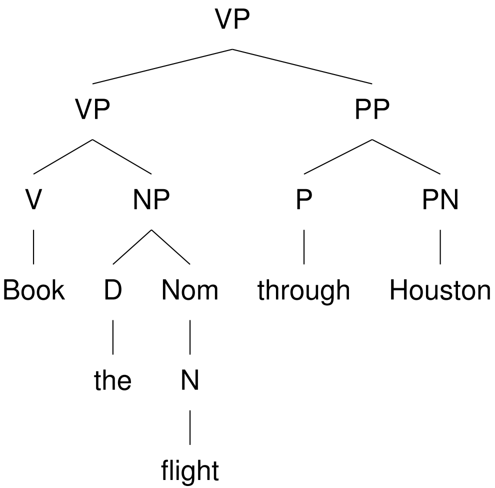
Bottom-up parsing



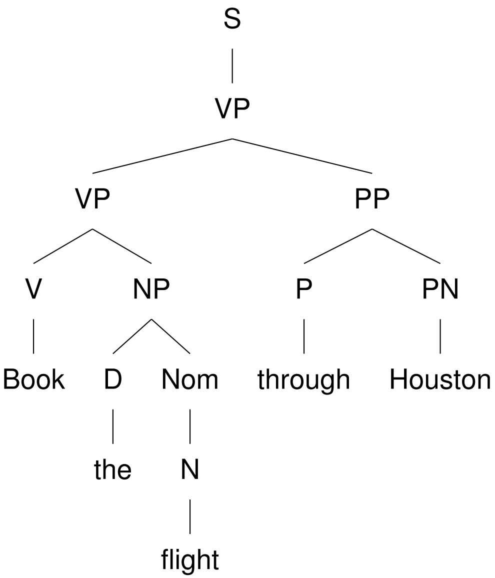
Bottom-up parsing



Bottom-up parsing

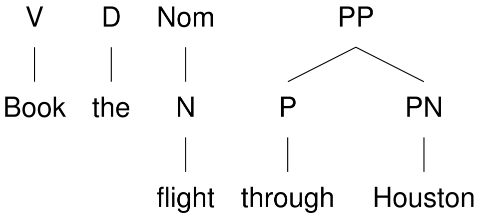


Bottom-up parsing

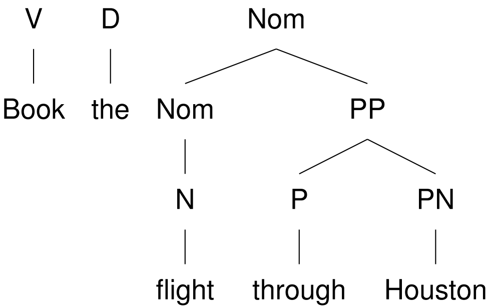


Bottom-up parsing

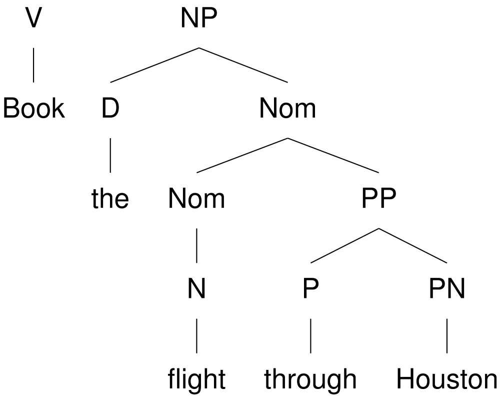
Or, we could have instead done:



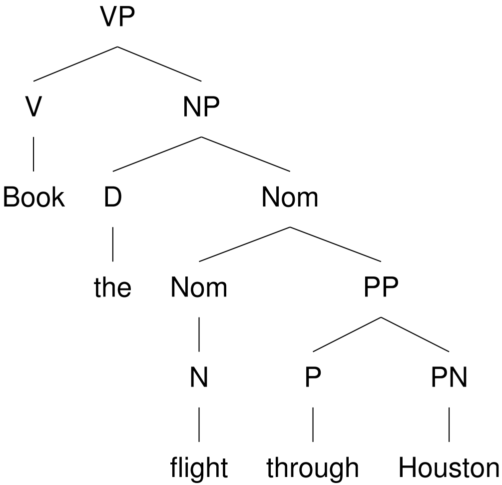
Bottom-up parsing



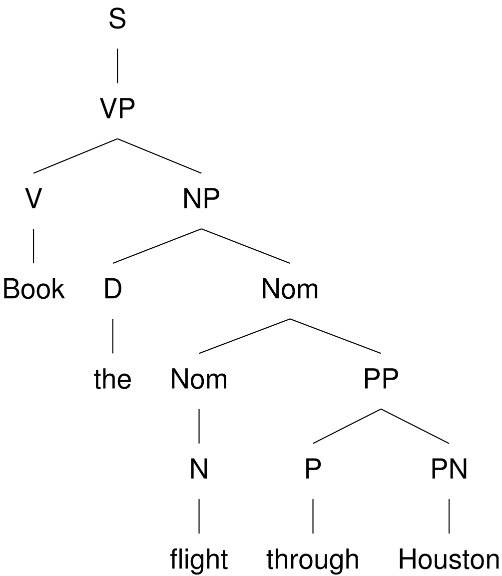
Bottom-up parsing



Bottom-up parsing



Bottom-up parsing



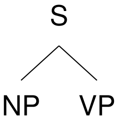
How do we make sure we get both trees?

- Go through **all** possibilities for productions

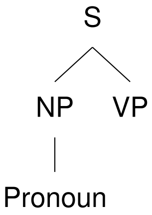
Top-Down Parsing

S

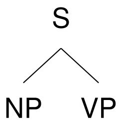
Top-Down Parsing



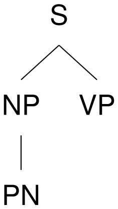
Top-Down Parsing



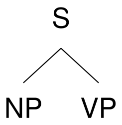
Top-Down Parsing



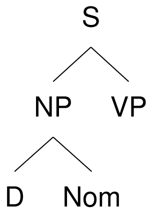
Top-Down Parsing



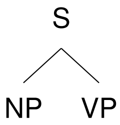
Top-Down Parsing



Top-Down Parsing

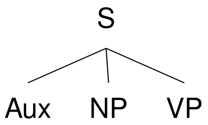


Top-Down Parsing



Top-Down Parsing

Top-Down Parsing

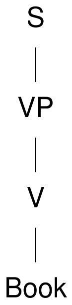


Top-Down Parsing

Top-Down Parsing

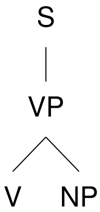
Top-Down Parsing

Top-Down Parsing

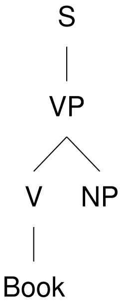


Yes, but we have more input still...

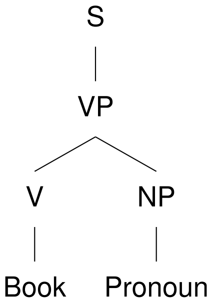
Top-Down Parsing



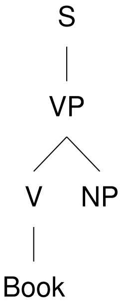
Top-Down Parsing



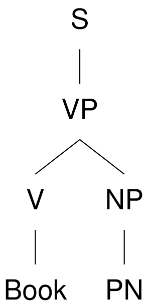
Top-Down Parsing



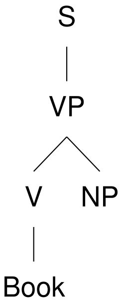
Top-Down Parsing



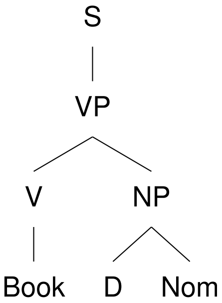
Top-Down Parsing



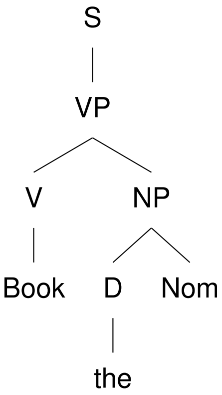
Top-Down Parsing



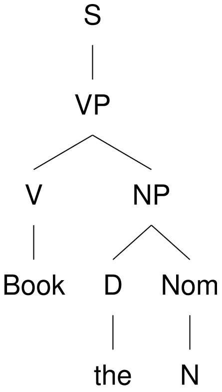
Top-Down Parsing



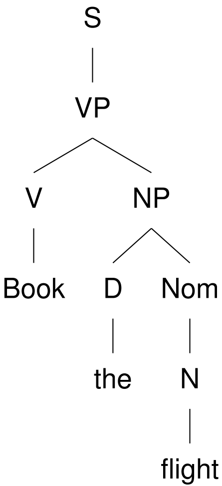
Top-Down Parsing



Top-Down Parsing

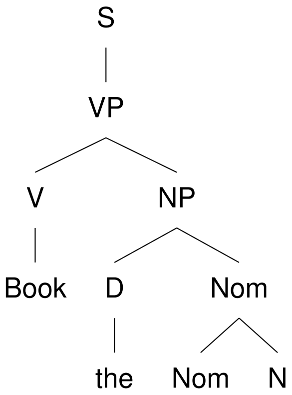


Top-Down Parsing

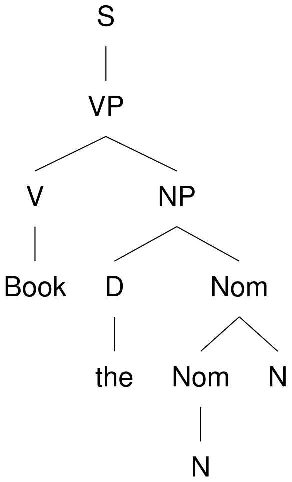


Yes, but we have more input still...

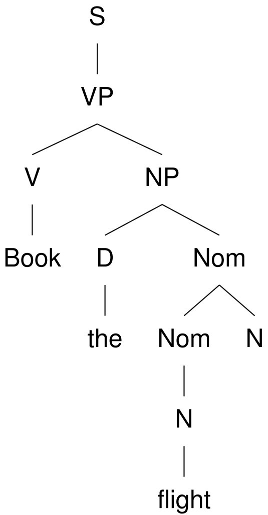
Top-Down Parsing



Top-Down Parsing

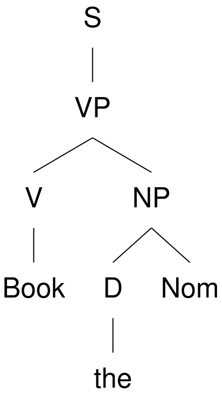


Top-Down Parsing

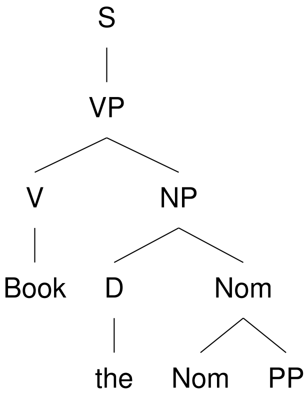


Nope... Backtrack again...

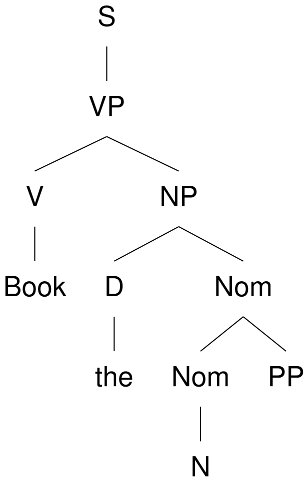
Top-Down Parsing



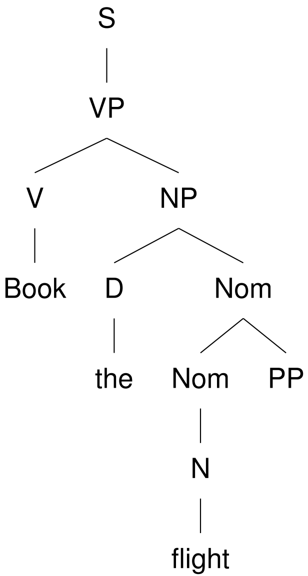
Top-Down Parsing



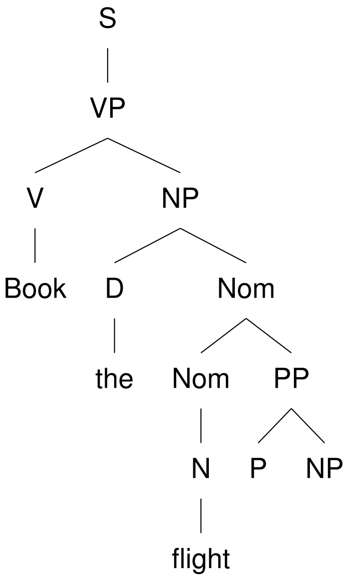
Top-Down Parsing



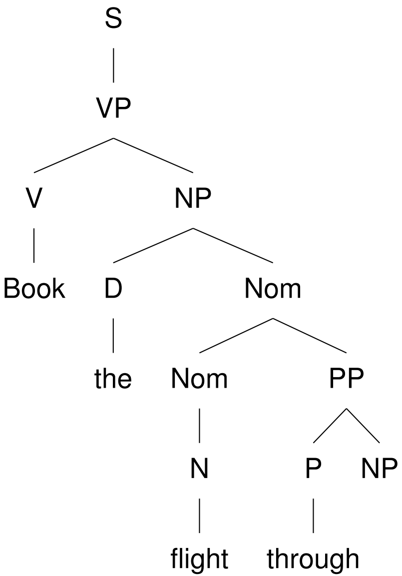
Top-Down Parsing



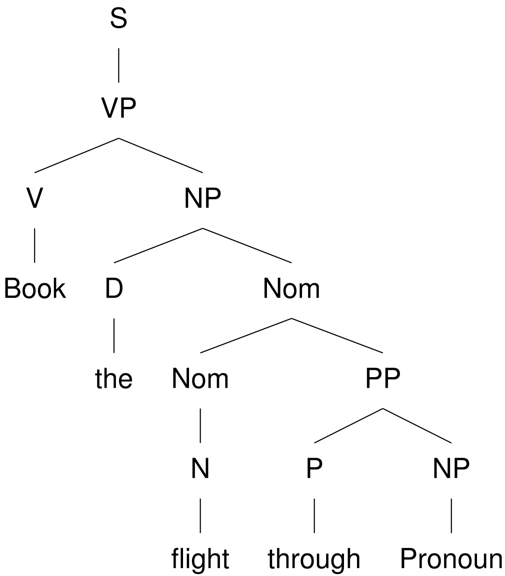
Top-Down Parsing



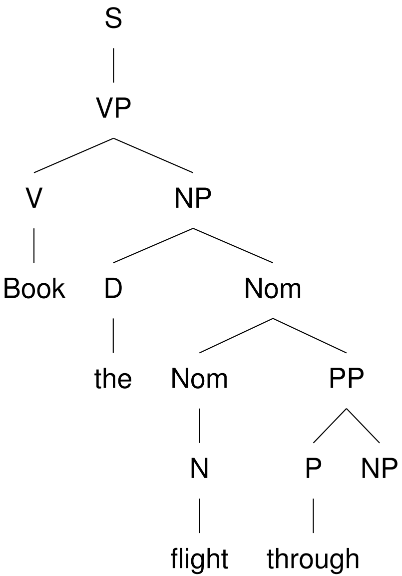
Top-Down Parsing



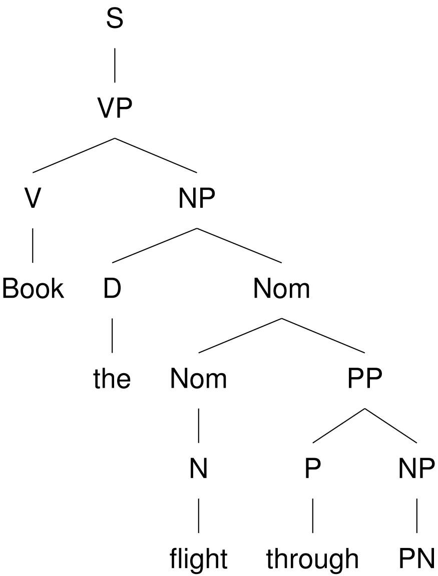
Top-Down Parsing



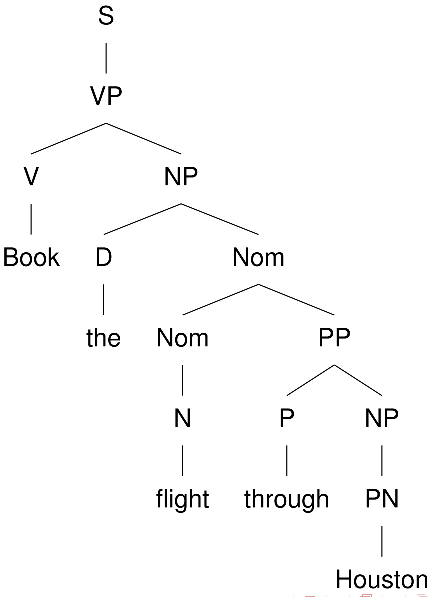
Top-Down Parsing



Top-Down Parsing



Top-Down Parsing



Top-Down Parsing

- Could we have gotten the second tree by top-down parsing?

Top-Down Parsing

- Could we have gotten the second tree by top-down parsing?
  - Yes; it is a matter of which rule happened to be on the top of the stack
  - We grabbed **VP** → **V NP**
  - But the option **VP** → **VP PP** is also on the stack somewhere
  - Thus the returned parse is subject to an arbitrary listing of rules in the grammar

Bottom Up vs. Top Down parsing

- Top-down parsers do not waste time exploring hypotheses not leading to S
  - ...but do waste time exploring hypotheses not matching the input
- Bottom-up parsers do not waste time exploring hypotheses not matching input
  - ...but do waste time exploring hypotheses not leading to S
- Both can take exponential time
  - (in the worst case, easier shown on abstract CFG)
  - Some recursive parsers are $O(n^4)$
- An answer to poor time complexity: **dynamic programming**
  - $O(n^3)$

Example: The Fibonacci numbers

Recursive definition: f(0) = 0; f(1) = 1 f(n) = f(n-1) + f(n-2)

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657...

f(100) = 218922995834555169026

The Fibonacci numbers: naive implementation

- Since we have a recursive definition, let's implement the Fibonacci numbers printer recursively!

```
def fibonacci(n):
    if n in [0,1]:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

What's the problem with this?

The Fibonacci numbers: better implementation


```
def fibonacci(n):
    return fibonacci_helper(n,{})

def fibonacci_helper(n,memo):
    if n in [0,1]:
        return n
    if not n in memo:
        memo[n] = fibonacci_helper(n-1,memo)
            + fibonacci_helper(n-2,memo)
    return memo[n]
```

Dynamic programming

- Fill in a table with solutions to subproblems
- Then can just look up momentarily the precomputed solution
- No need to perform the same computation many times

Fibonacci numbers

- f(0) = 0
- f(1) = 1
- f(2) = f(1) + f(0)
  - what's f(1)?
  - what's f(0)?

Fibonacci numbers

- f(3) = f(2) + f(1)
  - f(2) = f(1) + f(0)
  - f(1) = 1
  - f(0) = 0

Fibonacci numbers

- f(4) = f(3) + f(2)
    - f(3) = f(2) + f(1)
    - f(2) = f(1) + f(0)
    - f(1) = 1
    - f(0) = 0

Fibonacci numbers

- f(5) = f(4) + f(3)
  - f(4) = f(3) + f(2)
  - f(3) = f(2) + f(1)
  - f(2) = f(1) + f(0)
  - f(1) = 1
  - f(0) = 0
- etc... (deep recursion; slow; do the same computation again and again)

Fibonacci numbers

f(0) = ?

	0	1	2	3	4	5

Fibonacci numbers

f(0) = ?

Not in the table, so compute: f(0)=0 (or rather, return the base case)

fill in the cell

	0	1	2	3	4	5

Fibonacci numbers

f(1) = ?

Not in the table, so compute: f(1)=1 (or rather, return the base case)

fill in the cell

	0	1	2	3	4	5
	0					

Fibonacci numbers

f(2) = ?

Not in the table, so compute: f(2)=f(2-1) + f(2-2) = f(1) + f(0)

But both f(1) and f(0) are already in the table! No need to compute! Just look up!

fill in the cell

	0	1	2	3	4	5
	0	1				

Fibonacci numbers

f(3) = ?

Not in the table, so compute: f(3)=f(3-1) + f(3-2) = f(2) + f(1)

But both f(2) and f(1) are already in the table! No need to compute! Just look up!

fill in the cell

	0	1	2	3	4	5
	0	1	1			

Fibonacci numbers

f(4) = ?

Not in the table, so compute: f(4)=f(4-1) + f(4-2) = f(3) + f(2)

But both f(3) and f(2) are already in the table! No need to compute! Just look up!

fill in the cell

	0	1	2	3	4	5
	0	1	1	2		

Dynamic programming for parsing

Once the constituent has been discovered, store the information

- Example: The CKY algorithm (Cocke-Kazami-Younger)