

The CKY Parser

Computational Linguistics

Alexander Koller

21 November 2017

Context-free grammars

$T = \{\text{John, ate, sandwich, a}\}$

$N = \{S, NP, VP, V, N, Det\}$; start symbol: S

Production rules:

$S \rightarrow NP \ VP$

$V \rightarrow \text{ate}$

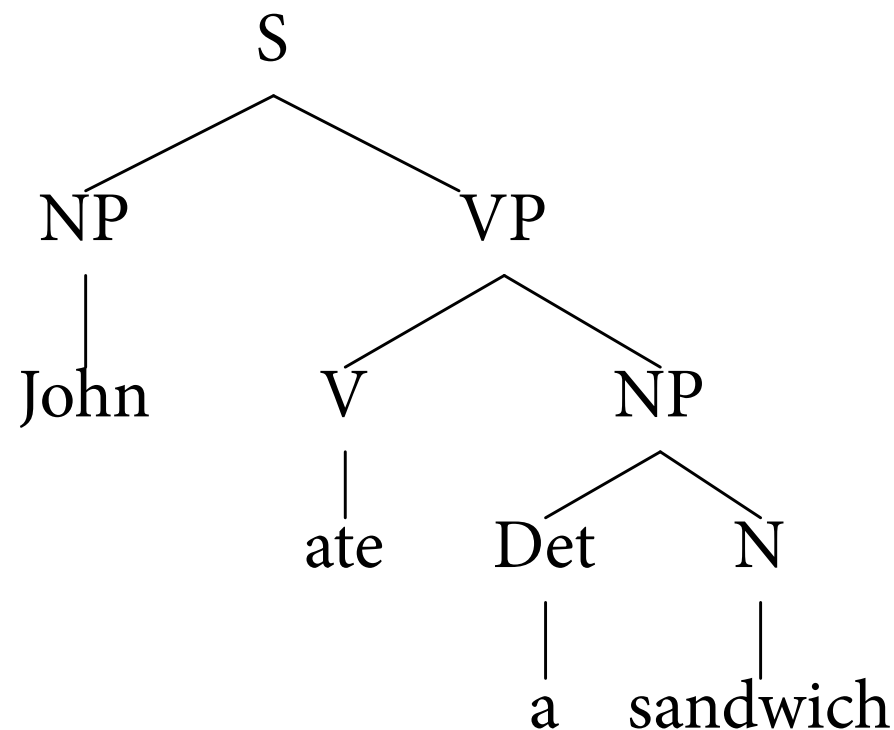
$Det \rightarrow a$

$NP \rightarrow Det \ N$

$NP \rightarrow \text{John}$

$N \rightarrow \text{sandwich}$

$VP \rightarrow V \ NP$



Shift-Reduce Parsing

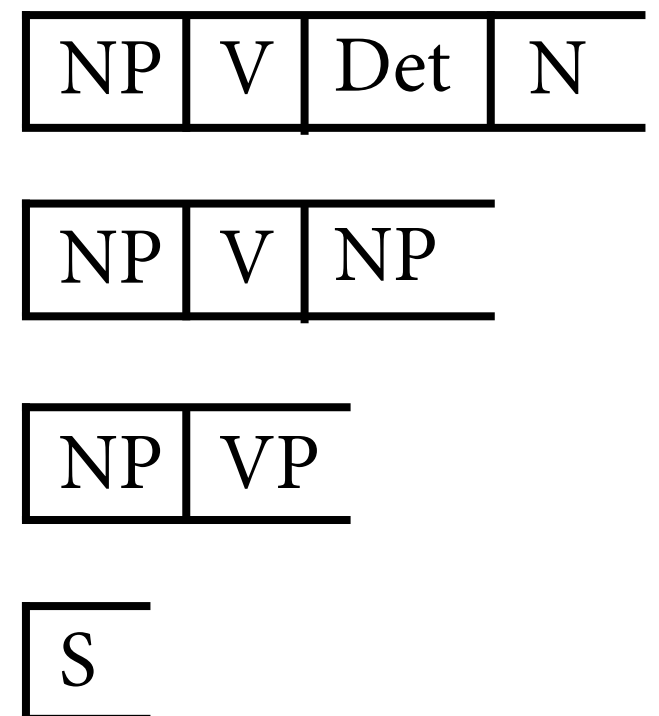
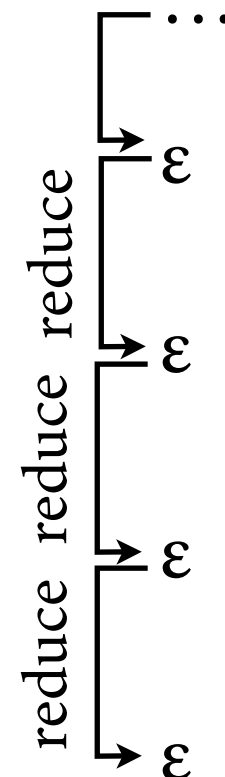
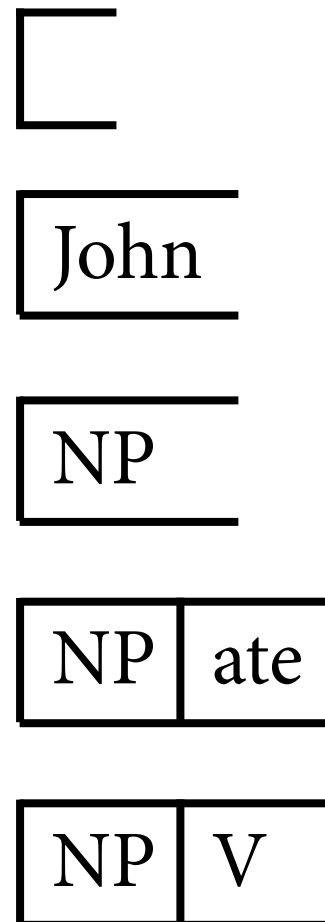
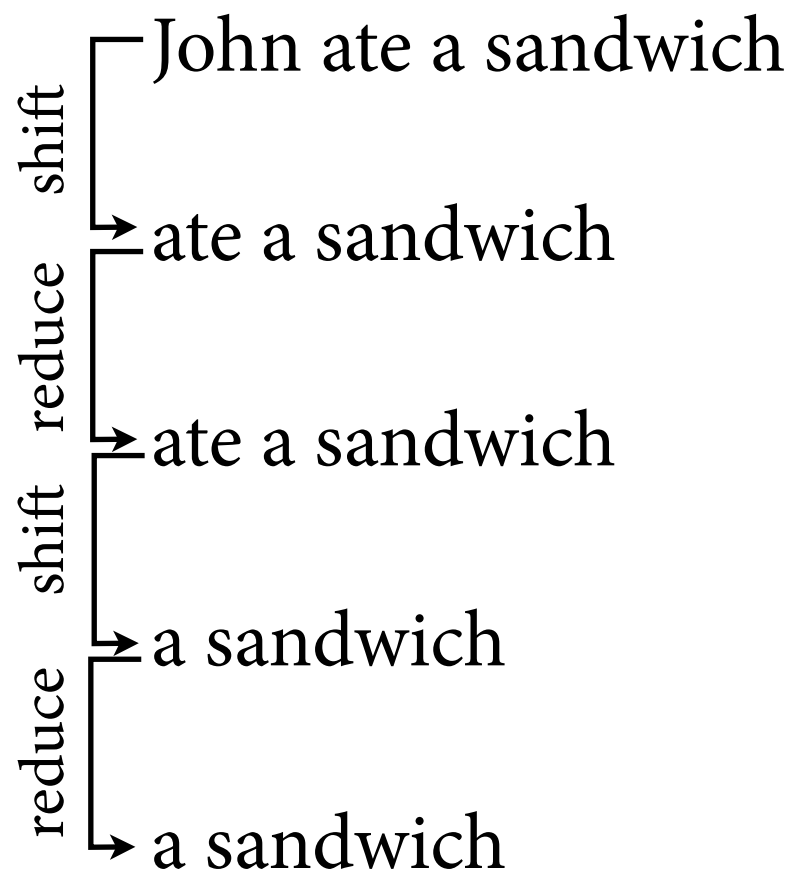
$$T = \{\text{John, ate, sandwich, a}\}$$

$N = \{S, NP, VP, V, N, Det\}$; start symbol: S

Production rules:

$$S \rightarrow NP \quad VP$$
$$VP \rightarrow V \ NP$$
$$V \rightarrow \text{ate}$$
$$\text{Det} \rightarrow \mathbf{a}$$
$$\text{NP} \rightarrow \text{Det N}$$
NP \rightarrow John

N → sandwich



Schema for shift-reduce

- Items are of the form (s, w') where w' is a suffix of the input string w , and s is the stack.

▶ Claim of this item: Underlying cfg allows the derivation
 $s w' \Rightarrow^* w$

- Start item: (ε, w) ; goal item: (S, ε)

- Parsing rules:

$$\frac{(s, a \cdot w')}{(s \cdot a, w')} \text{ (shift)}$$

$$\frac{(s \cdot s', w') \quad A \rightarrow s' \text{ in } P}{(s \cdot A, w')} \text{ (reduce)}$$

Runtime of algorithms

- It is not enough to find an algorithm that is sound and complete. It should also be *efficient*.
- Runtime of an algorithm is measured:
 - ▶ as a function of input size n
 - ▶ for the worst case (= inputs of that size on which the algorithm runs longest)
 - ▶ asymptotically (= ignore constant factors)

A simple example

- Problem: test whether list of numbers is sorted.
 - ▶ given list L of ints of length n :
 - ▶ are there indices $1 \leq i < j \leq n$ s.t. $L_i > L_j$?
- Let's look at two algorithms for this problem.

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

Runtime

len(L)	quadratic	linear
100		
1000		
10000		
100.000		
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	
1000		
10000		
100.000		
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000		
100.000		
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000	4.5 sec	
100.000		
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

```
def linear_issorted(L):  
    for i in range(len(L)-1):  
        if L[i] > L[i+1]:  
            return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

```
def linear_issorted(L):  
    for i in range(len(L)-1):  
        if L[i] > L[i+1]:  
            return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

```
def linear_issorted(L):  
    for i in range(len(L)-1):  
        if L[i] > L[i+1]:  
            return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

```
def linear_issorted(L):  
    for i in range(len(L)-1):  
        if L[i] > L[i+1]:  
            return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

```
def linear_issorted(L):  
    for i in range(len(L)-1):  
        if L[i] > L[i+1]:  
            return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

```
def linear_issorted(L):  
    for i in range(len(L)-1):  
        if L[i] > L[i+1]:  
            return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		179 ms

Runtime comparison


```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

```
def linear_issorted(L):  
    for i in range(len(L)-1):  
        if L[i] > L[i+1]:  
            return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		179 ms

$\approx n \cdot 120 \text{ ns}$



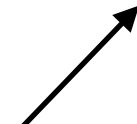
Runtime comparison


```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

```
def linear_issorted(L):  
    for i in range(len(L)-1):  
        if L[i] > L[i+1]:  
            return False  
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		179 ms


$$\approx n^2 \cdot 45 \text{ ns}$$


$$\approx n \cdot 120 \text{ ns}$$

Analysis

- Important parameters:
 - ▶ input size $n = \text{len}(L)$, i.e. length of list
 - ▶ worst case = L is sorted; every loop iterated n times
 - ▶ don't really care about time per iteration, linear is always faster if n grows large enough
- We can get a good sense of the algorithm's runtime by saying it grows *linearly* or *quadratically* with n .
 - ▶ abstraction over implementation details and hardware
 - ▶ *asymptotic* comparison of runtime classes

O Notation

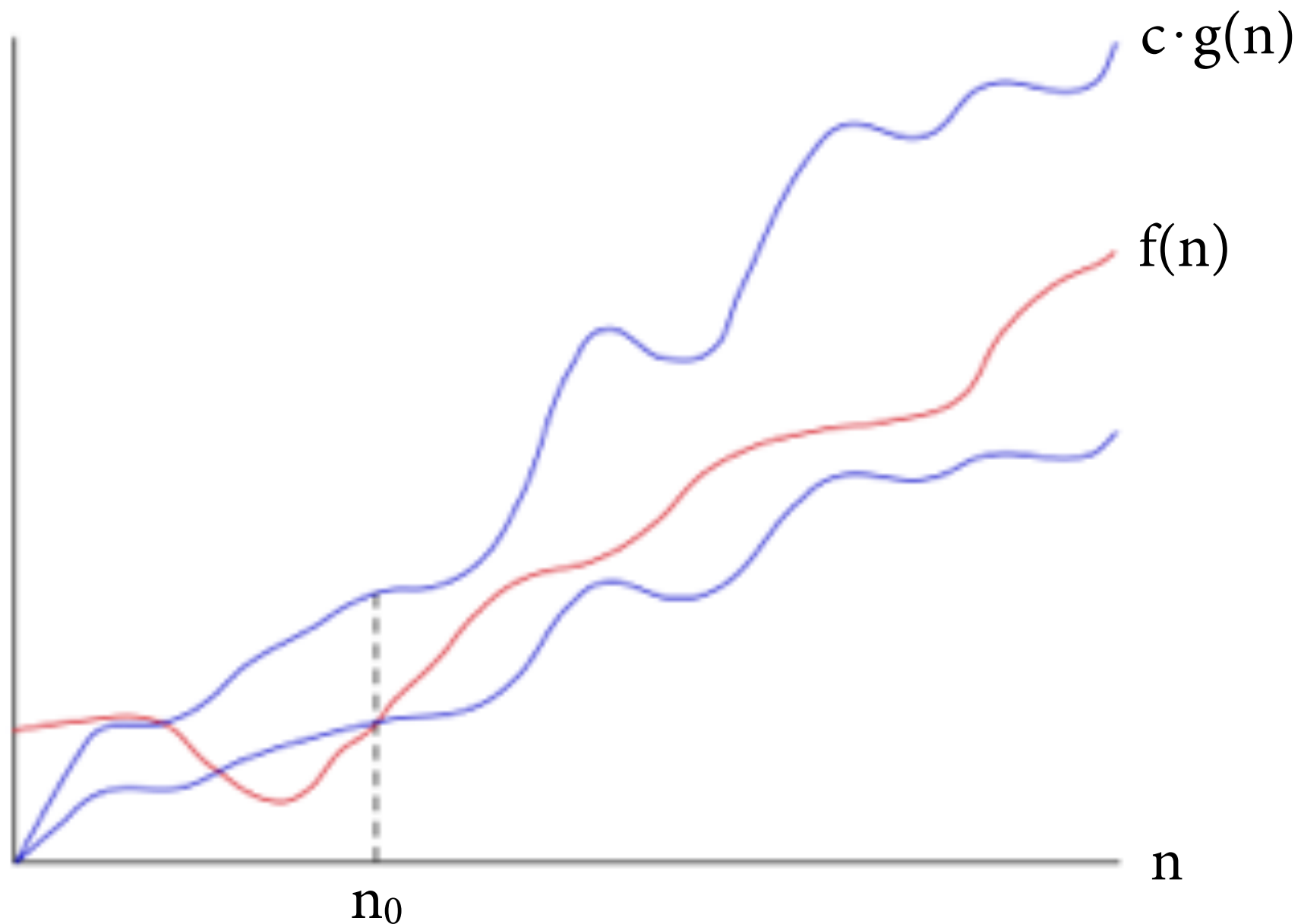
- Let f, g be functions. Then we define:

$$f = O(g) \text{ iff} \\ \text{exist } c, n_0 \text{ s.t. } f(n) \leq c \cdot g(n) \text{ f.a. } n \geq n_0$$

- Read “ f is O of g ”; “ $=$ ” denotes membership in a runtime class, not equality.
- Usually take the smallest g such that $f = O(g)$.

Illustration

$f = O(g)$ iff
exist c, n_0 s.t. $f(n) \leq c \cdot g(n)$ f.a. $n \geq n_0$



Back to the example

$f = O(g)$ iff
exist c, n_0 s.t. $f(n) \leq c \cdot g(n)$ f.a. $n \geq n_0$

```
def quadratic_issorted(L):  
    for i in range(len(L)):  
        for j in range(i+1, len(L)):  
            if L[j] < L[i]:  
                return False  
    return True
```

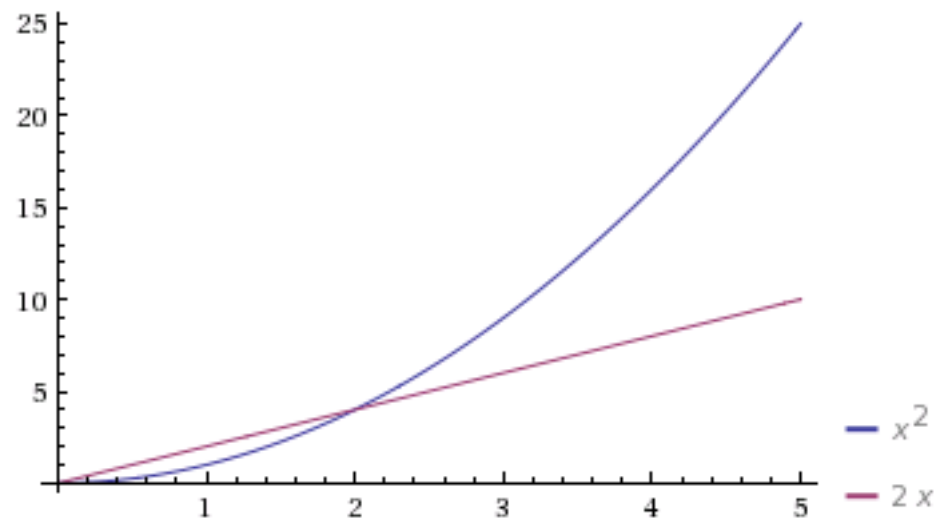
Runtime $f(n) \approx n^2 \cdot 45 \text{ ns} = O(n^2)$
“quadratic algorithm”

```
def linear_issorted(L):  
    for i in range(len(L)-1):  
        if L[i] > L[i+1]:  
            return False  
    return True
```

Runtime $f(n) \approx n \cdot 120 \text{ ns} = O(n)$
“linear algorithm”

Hierarchy of runtime classes

- For all c, c' , we have $c \cdot n \leq c' \cdot n^2$ after a certain point:



- For large n , low-rank polynomials are faster:
 - ▶ $O(n)$ linear $<$ $O(n^2)$ quadratic
(even for $n + 5$, $100 \cdot n - 27$ etc.)
 - ▶ $O(n^2)$ quadratic $<$ $O(n^3)$ cubic
 - ▶ etc.

Analyzing Shift-Reduce

$S \rightarrow B S$

$B \rightarrow b$

$S \rightarrow c$

$T \rightarrow C T$


$C \rightarrow b$

$T \rightarrow c$

b b b c



Analyzing Shift-Reduce

$S \rightarrow BS$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow CT$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	\rightarrow^* T 




Analyzing Shift-Reduce

$S \rightarrow BS$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow CT$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	\rightarrow^* T 
\rightarrow^*	C	C	B	T	

Analyzing Shift-Reduce

$S \rightarrow BS$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow CT$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	\rightarrow^* T 
\rightarrow^*	C	C	B	T	
\rightarrow^*	C	B	C	T	\rightarrow^* CBT 

Analyzing Shift-Reduce

$S \rightarrow BS$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow CT$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	$\rightarrow^* T$ ✗
\rightarrow^*	C	C	B	T	✗
\rightarrow^*	C	B	C	T	$\rightarrow^* CBT$ ✗
\rightarrow^*	C	B	B	T	✗

Analyzing Shift-Reduce

$S \rightarrow B S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C T$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	$\rightarrow^* T$ ✗
\rightarrow^*	C	C	B	T	✗
\rightarrow^*	C	B	C	T	$\rightarrow^* C B T$ ✗
\rightarrow^*	C	B	B	T	✗
...					
\rightarrow^*	B	B	B	S	

Analyzing Shift-Reduce

$S \rightarrow B S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C T$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	$\rightarrow^* T$ ✗
\rightarrow^*	C	C	B	T	✗
\rightarrow^*	C	B	C	T	$\rightarrow^* C B T$ ✗
\rightarrow^*	C	B	B	T	✗
...					
\rightarrow^*	B	B	B	S	$\rightarrow^* S$ ✓

Analyzing Shift-Reduce

$S \rightarrow BS$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow CT$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	$\rightarrow^* T$ ✗
\rightarrow^*	C	C	B	T	✗
\rightarrow^*	C	B	C	T	$\rightarrow^* CBT$ ✗
\rightarrow^*	C	B	B	T	✗
...					
\rightarrow^*	B	B	B	S	$\rightarrow^* S$ ✓


Analyzing Shift-Reduce

$S \rightarrow B S$	$B \rightarrow b$	
$T \rightarrow C T$	$C \rightarrow b$	$T \rightarrow c$

b b b c

Analyzing Shift-Reduce

$S \rightarrow B S$	$B \rightarrow b$	
$T \rightarrow C T$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	\rightarrow^* T 

Analyzing Shift-Reduce

$S \rightarrow B S$	$B \rightarrow b$	
$T \rightarrow C T$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	$\rightarrow^* T$ ✗
\rightarrow^*	C	C	B	T	✗

Analyzing Shift-Reduce

$S \rightarrow BS$	$B \rightarrow b$	
$T \rightarrow CT$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	$\rightarrow^* T$ ✗
\rightarrow^*	C	C	B	T	✗
\rightarrow^*	C	B	C	T	$\rightarrow^* CBT$ ✗

Analyzing Shift-Reduce

$S \rightarrow B S$	$B \rightarrow b$	
$T \rightarrow C T$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	$\rightarrow^* T$ ✗
\rightarrow^*	C	C	B	T	✗
\rightarrow^*	C	B	C	T	$\rightarrow^* C B T$ ✗
\rightarrow^*	C	B	B	T	✗

Analyzing Shift-Reduce

$S \rightarrow B S$	$B \rightarrow b$	
$T \rightarrow C T$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c	
\rightarrow^*	C	C	C	T	$\rightarrow^* T$ ✗
\rightarrow^*	C	C	B	T	✗
\rightarrow^*	C	B	C	T	$\rightarrow^* C B T$ ✗
\rightarrow^*	C	B	B	T	✗
...					
\rightarrow^*	B	B	B	T	✗

Analyzing Shift-Reduce

$S \rightarrow B S$	$B \rightarrow b$	
$T \rightarrow C T$	$C \rightarrow b$	$T \rightarrow c$

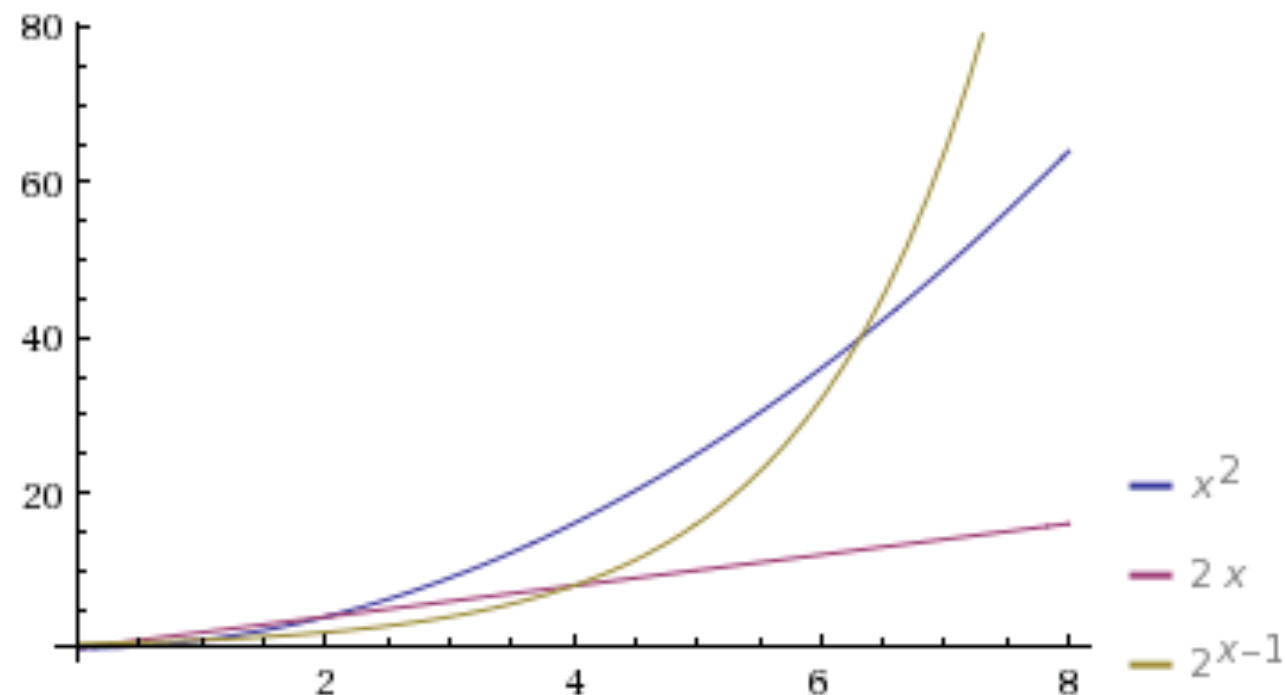
	b	b	b	c	
\rightarrow^*	C	C	C	T	$\rightarrow^* T$ ✗
\rightarrow^*	C	C	B	T	✗
\rightarrow^*	C	B	C	T	$\rightarrow^* C B T$ ✗
\rightarrow^*	C	B	B	T	✗
...					
\rightarrow^*	B	B	B	T	✗

Analyzing Shift-Reduce

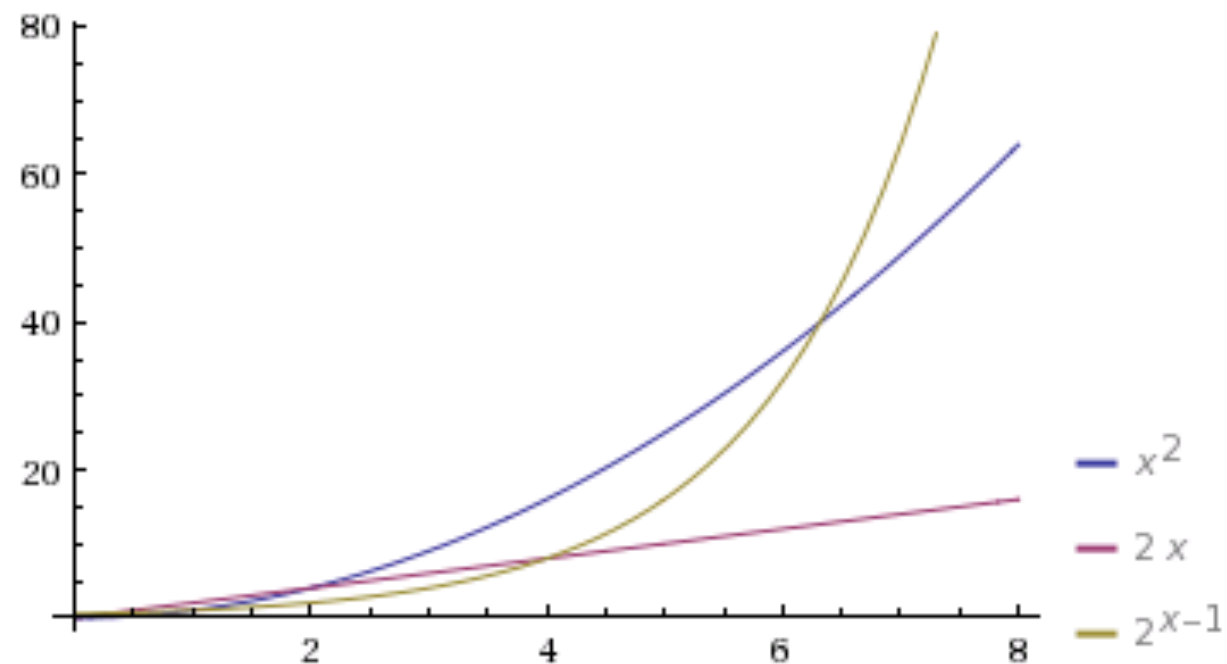
- If string has length n and grammar has k nonterminals, then there are $O(k^n)$ ways of assigning strings of nonterminals to words.
- These can all be explored, especially when the string is *not* in the language.
- In addition, SR recognizer may try out analyses for substrings that cannot be used top-down. Let's ignore this part today.

Exponential runtime

- Worst case runtime of shift-reduce: roughly k^n computation steps.
- Exponential functions grow faster than every polynomial: if $k > 1$, then there is no m such that $k^n = O(n^m)$.

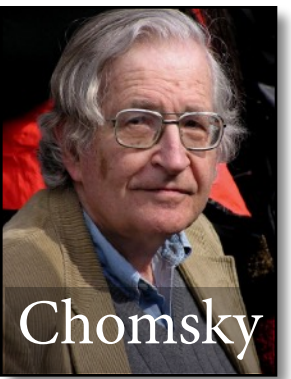


Polynomial vs. exponential



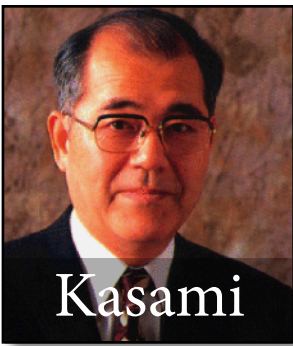
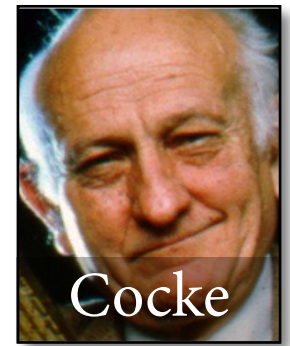
- We often distinguish between *polynomial* and *exponential* runtime. Rule of thumb: exponential = too slow for practical use.
- Is there a polynomial algorithm for the word problem?

Chomsky Normal Form



- A cfg is *in Chomsky normal form (CNF)* if each of its production rules has one of these two forms:
 - ▶ $A \rightarrow BC$: right-hand side is exactly two nonterminals
 - ▶ $A \rightarrow c$: right-hand side is exactly one terminal
- For every cfg G , there is a weakly equivalent cfg G' which is in CNF.
 - ▶ that is, $L(G) = L(G')$

The CKY Algorithm



- Simplest and most-used chart parser for cfgs in CNF.
- Developed independently in the 1960s by John Cocke, Daniel Younger, and Tadao Kasami.
 - ▶ sometimes also called CYK algorithm
- Bottom-up algorithm for discovering statements of the form “ $A \Rightarrow^* w_i \dots w_{k-1} ?$ ”

The CKY Recognizer

$S \rightarrow NP \ VP$

$V \rightarrow \text{ate}$

$\text{Det} \rightarrow \text{a}$

$NP \rightarrow \text{Det} \ N$

$NP \rightarrow \text{John}$

$N \rightarrow \text{sandwich}$

$VP \rightarrow V \ NP$

Chart

	$i = 1$	2	3	4
5				
4				... a sandwich
3			... ate a	
$k = 2$... John ate		
	John			

Cell at column i , row k :
 $\{ A \mid A \Rightarrow^* w_i \dots w_{k-1} \}$

The CKY Recognizer

$S \rightarrow NP \ VP$

$V \rightarrow \text{ate}$

$\text{Det} \rightarrow \text{a}$

$NP \rightarrow \text{Det} \ N$

$NP \rightarrow \text{John}$

$N \rightarrow \text{sandwich}$

$VP \rightarrow V \ NP$

Chart

	$i = 1$	2	3	4	
5					... sandwich
4				... a	sandwich
3			... ate	a	
$k = 2$	NP	... John	ate		
	John				

Cell at column i , row k :
 $\{ A \mid A \Rightarrow^* w_i \dots w_{k-1} \}$

The CKY Recognizer

$S \rightarrow NP \ VP$
 $NP \rightarrow Det \ N$
 $VP \rightarrow V \ NP$

$V \rightarrow ate$
 $NP \rightarrow John$

$Det \rightarrow a$
 $N \rightarrow sandwich$

Chart

	$i = 1$	2	3	4
5				
4				...
3		V	...	a
$k = 2$	NP	...	ate	sandwich
	John			

Cell at column i , row k :
 $\{ A \mid A \Rightarrow^* w_i \dots w_{k-1} \}$

The CKY Recognizer

$S \rightarrow NP \ VP$
 $NP \rightarrow Det \ N$
 $VP \rightarrow V \ NP$

$V \rightarrow ate$
 $NP \rightarrow John$

$Det \rightarrow a$
 $N \rightarrow sandwich$

Chart

	$i = 1$	2	3	4
5				
4			Det	...
3		V	...	a
$k = 2$	NP	...	ate	sandwich
	John			

Cell at column i , row k :
 $\{ A \mid A \Rightarrow^* w_i \dots w_{k-1} \}$

The CKY Recognizer

$S \rightarrow NP \ VP$
 $NP \rightarrow Det \ N$
 $VP \rightarrow V \ NP$

$V \rightarrow ate$
 $NP \rightarrow John$

$Det \rightarrow a$
 $N \rightarrow sandwich$

Chart

	$i = 1$	2	3	4	
5				N	... sandwich
4			Det	a	sandwich
3		V	ate	a	
$k = 2$	NP	John	ate		
	John				

Cell at column i , row k :
 $\{ A \mid A \Rightarrow^* w_i \dots w_{k-1} \}$

The CKY Recognizer

$S \rightarrow NP \ VP$
 $NP \rightarrow Det \ N$
 $VP \rightarrow V \ NP$

$V \rightarrow ate$
 $NP \rightarrow John$

$Det \rightarrow a$
 $N \rightarrow sandwich$

Chart

	$i = 1$	2	3	4
5			NP	N
4			Det	...
3		V	...	a
$k = 2$	NP	...	ate	sandwich
	John			

Cell at column i , row k :
 $\{ A \mid A \Rightarrow^* w_i \dots w_{k-1} \}$

The CKY Recognizer

$S \rightarrow NP \ VP$
 $NP \rightarrow Det \ N$
 $VP \rightarrow V \ NP$

$V \rightarrow ate$
 $NP \rightarrow John$

$Det \rightarrow a$
 $N \rightarrow sandwich$

Chart

	$i = 1$	2	3	4
5		VP	NP	N
4			Det	...
3		V	...	
$k = 2$	NP	...		
	John	ate	ate	a

... sandwich

Cell at column i , row k :

$$\{ A \mid A \Rightarrow^* w_i \dots w_{k-1} \}$$

The CKY Recognizer

$S \rightarrow NP \ VP$
 $NP \rightarrow Det \ N$
 $VP \rightarrow V \ NP$

$V \rightarrow ate$
 $NP \rightarrow John$

$Det \rightarrow a$
 $N \rightarrow sandwich$

Chart

	$i = 1$	2	3	4
5	S	VP	NP	N
4			Det	...
3		V	...	
$k = 2$	NP	...		

John

ate

...

ate

a

...

sandwich

...

sandwich

Cell at column i , row k :
 $\{ A \mid A \Rightarrow^* w_i \dots w_{k-1} \}$

The CKY Recognizer

$S \rightarrow NP \ VP$
 $NP \rightarrow Det \ N$
 $VP \rightarrow V \ NP$

$V \rightarrow ate$
 $NP \rightarrow John$

$Det \rightarrow a$
 $N \rightarrow sandwich$

Chart

$S \Rightarrow^* w$

	$i = 1$	2	3	4
5	S	VP	NP	N
4			Det	...
3		V	...	
$k = 2$	NP	...		

John

ate

ate

a

sandwich

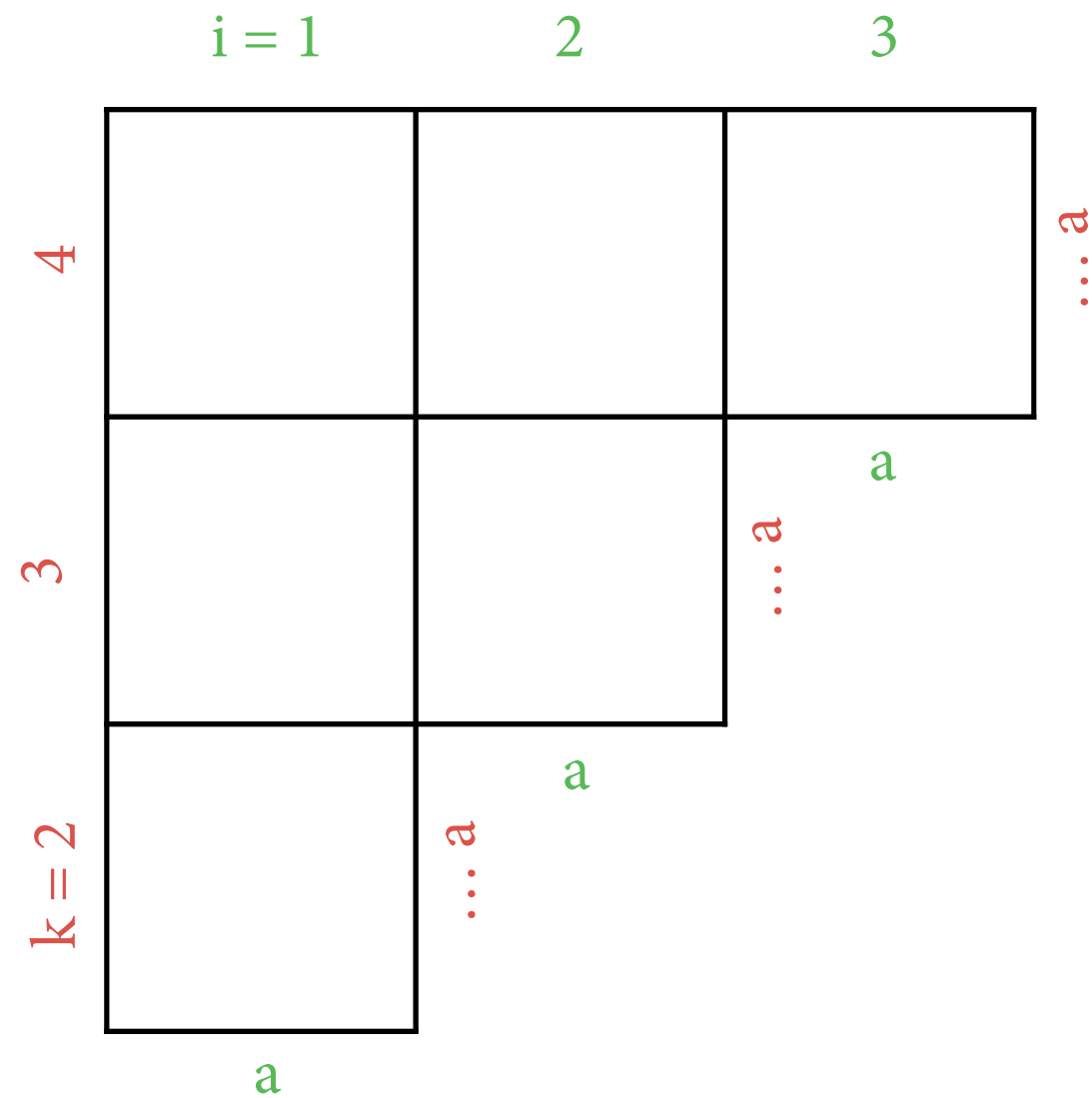
... sandwich

Cell at column i , row k :
 $\{ A \mid A \Rightarrow^* w_i \dots w_{k-1} \}$

The CKY Recognizer

$S \rightarrow S S$

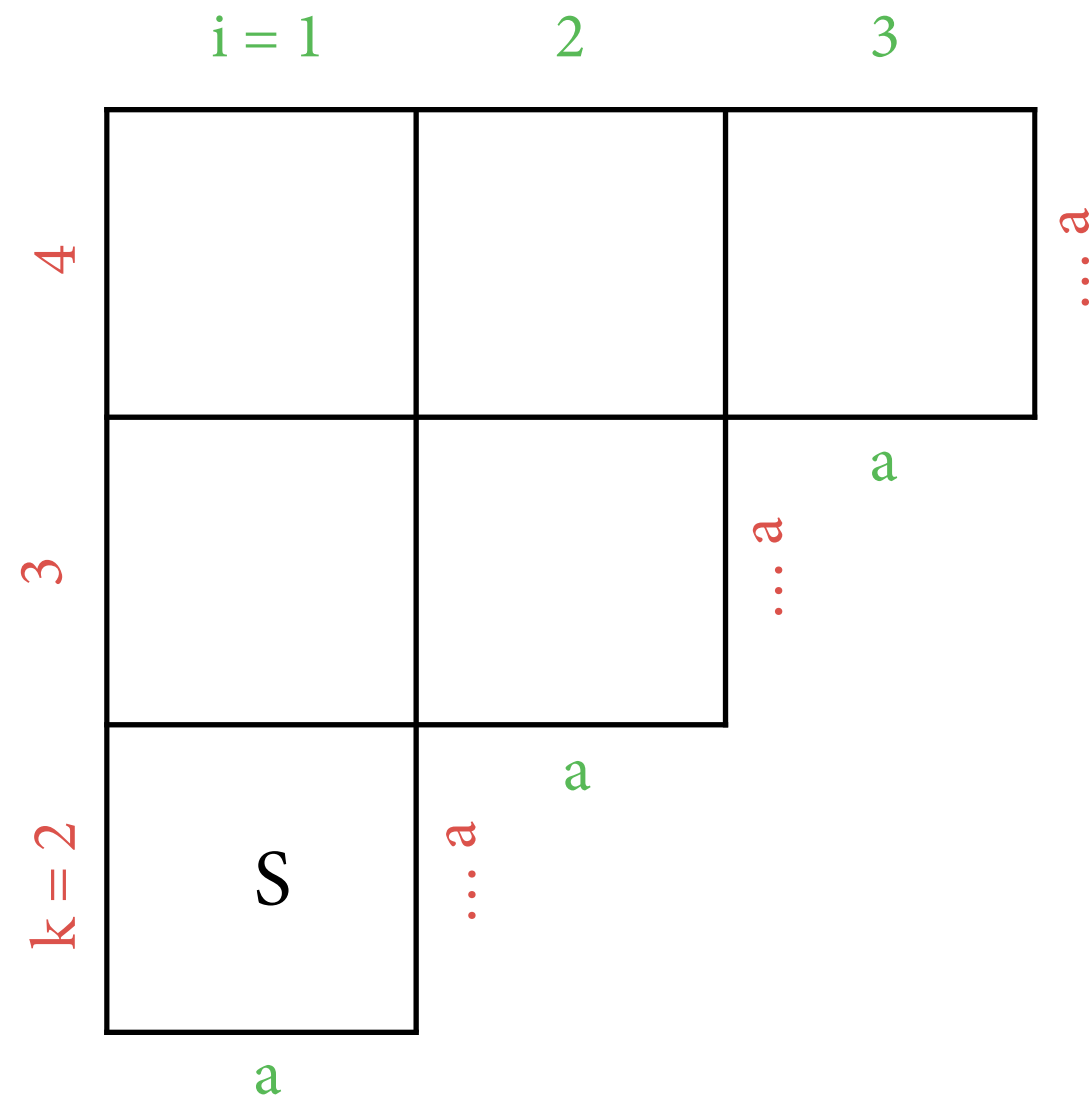
$S \rightarrow a$



The CKY Recognizer

$S \rightarrow S S$

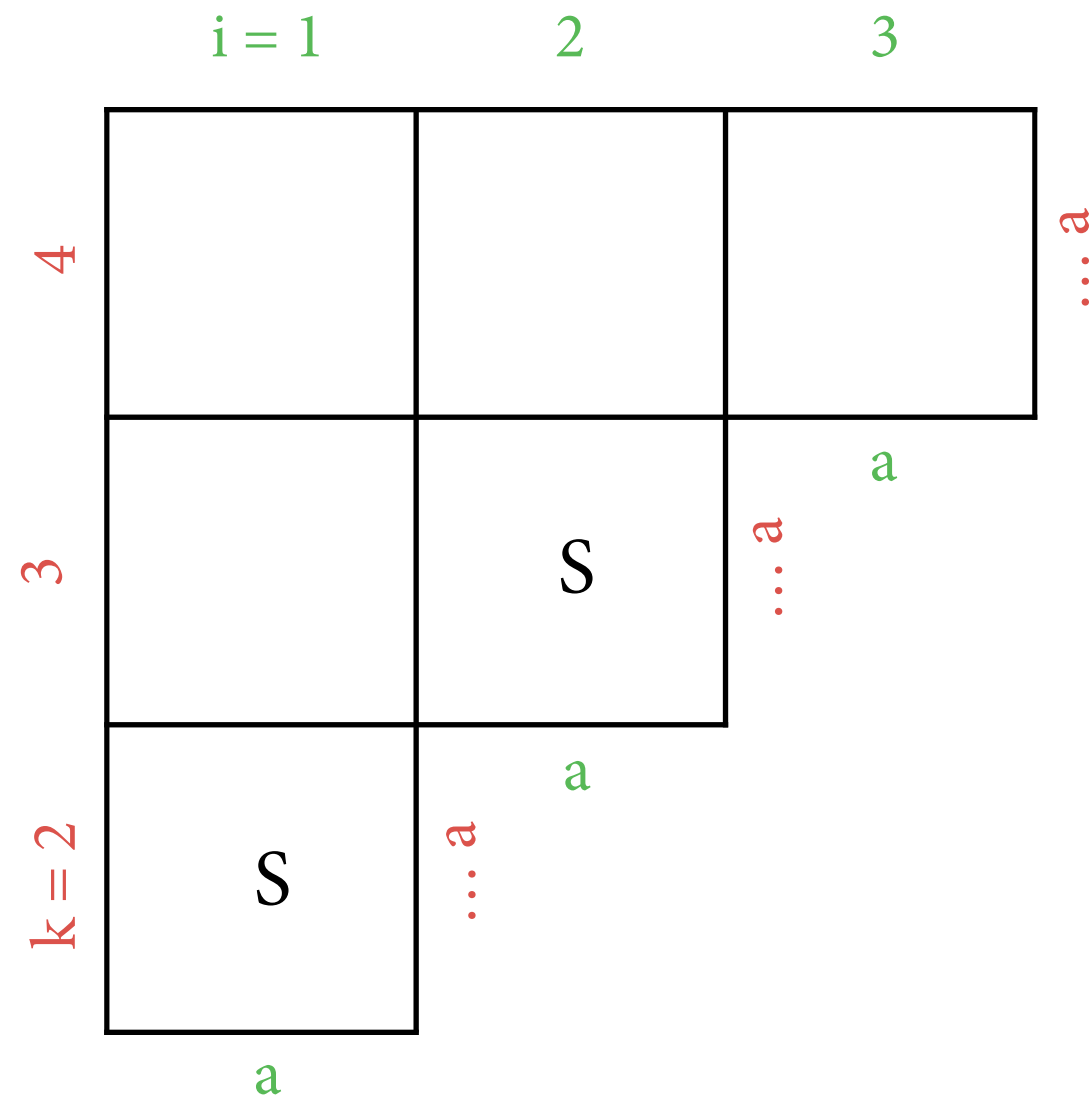
$S \rightarrow a$



The CKY Recognizer

$S \rightarrow S S$

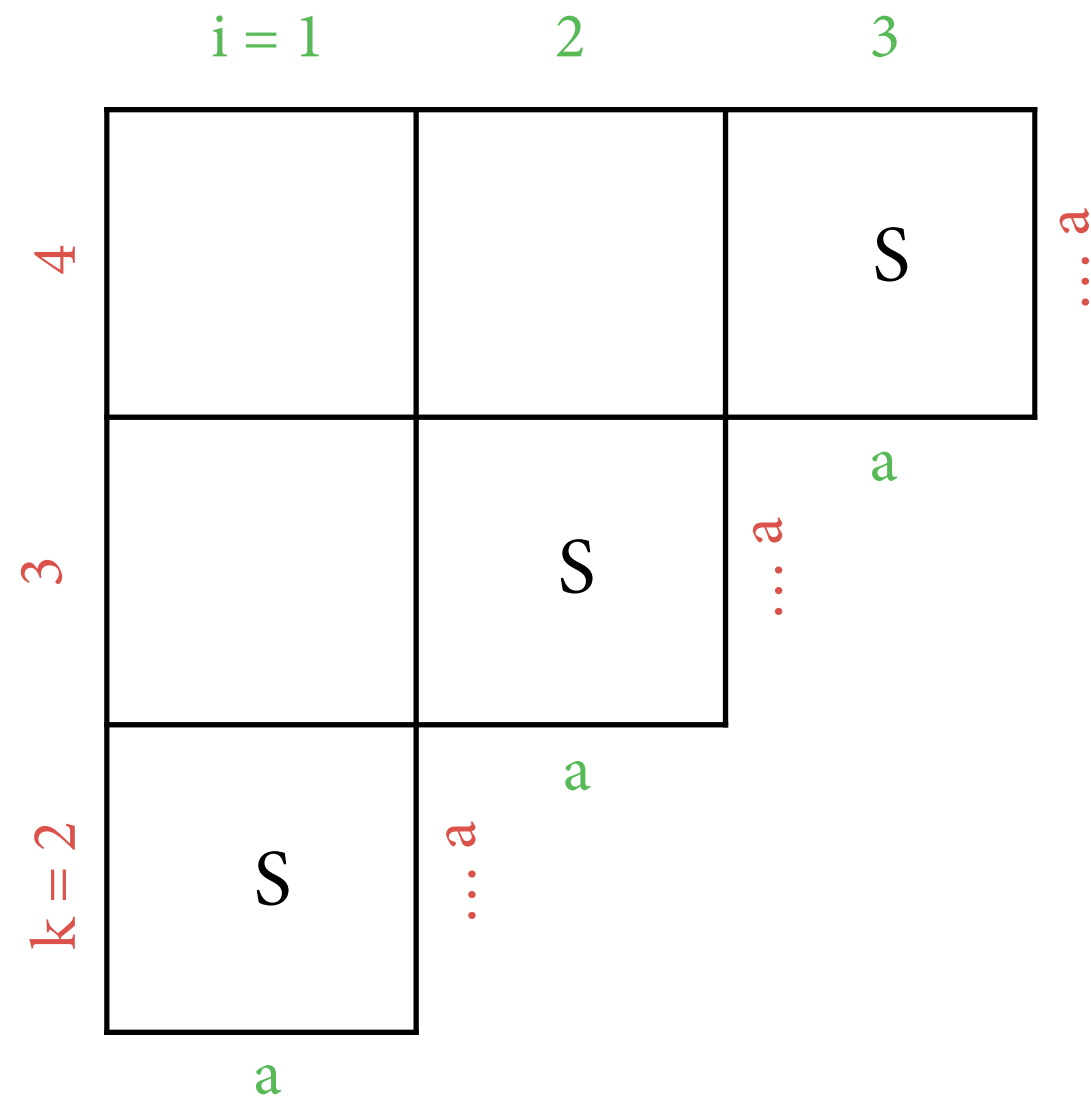
$S \rightarrow a$



The CKY Recognizer

$S \rightarrow S S$

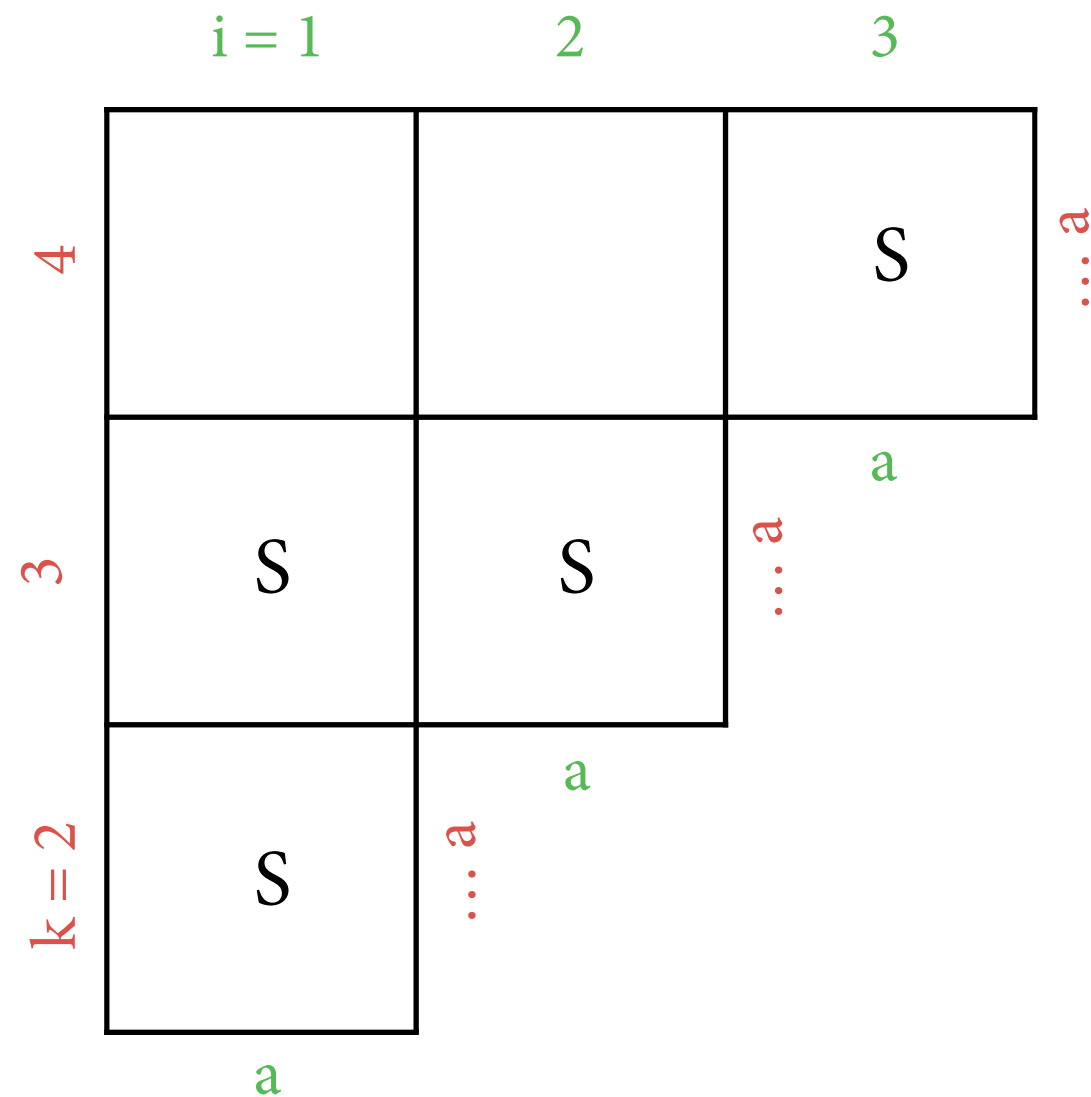
$S \rightarrow a$



The CKY Recognizer

$S \rightarrow S S$

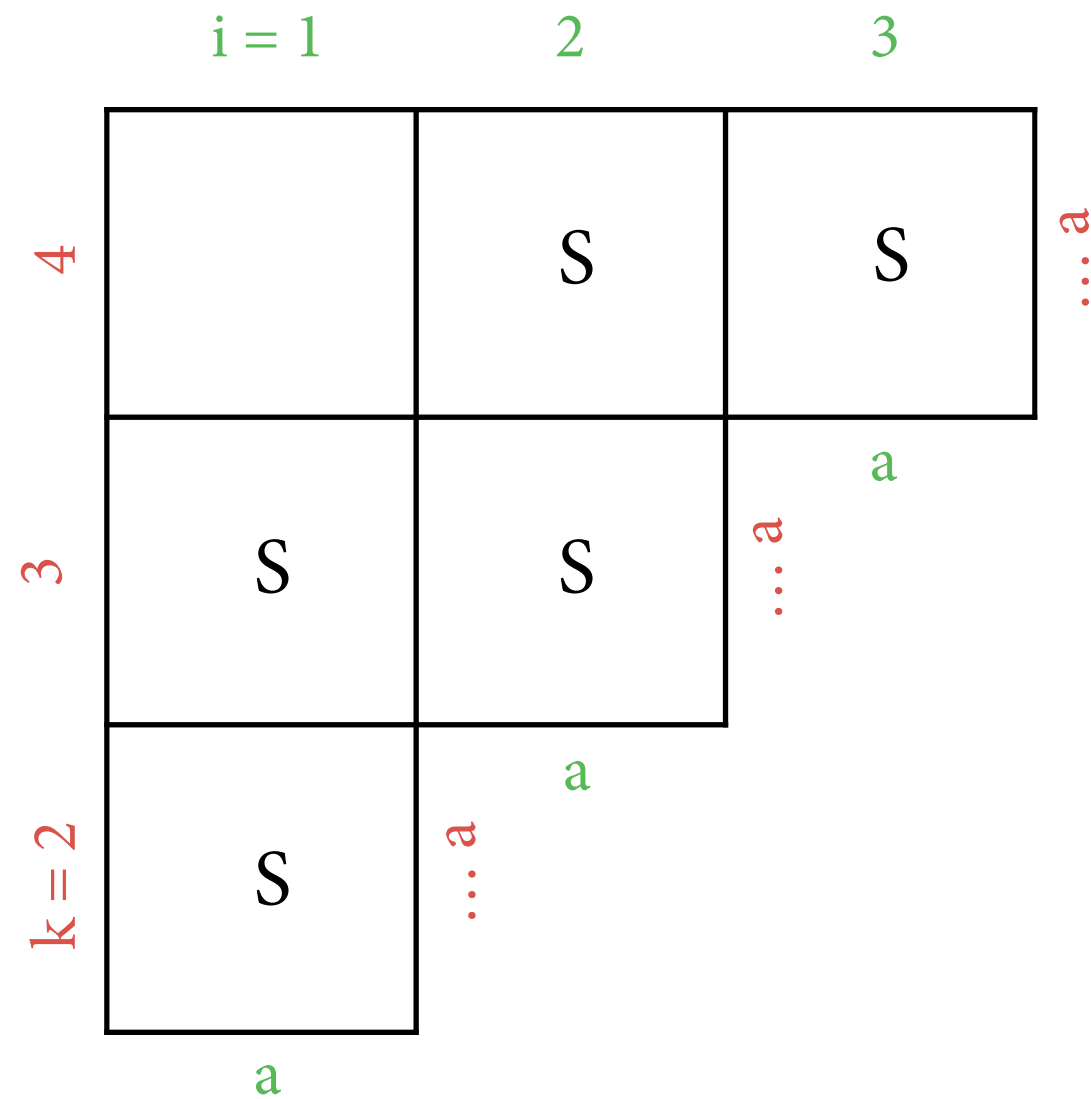
$S \rightarrow a$



The CKY Recognizer

$S \rightarrow S S$

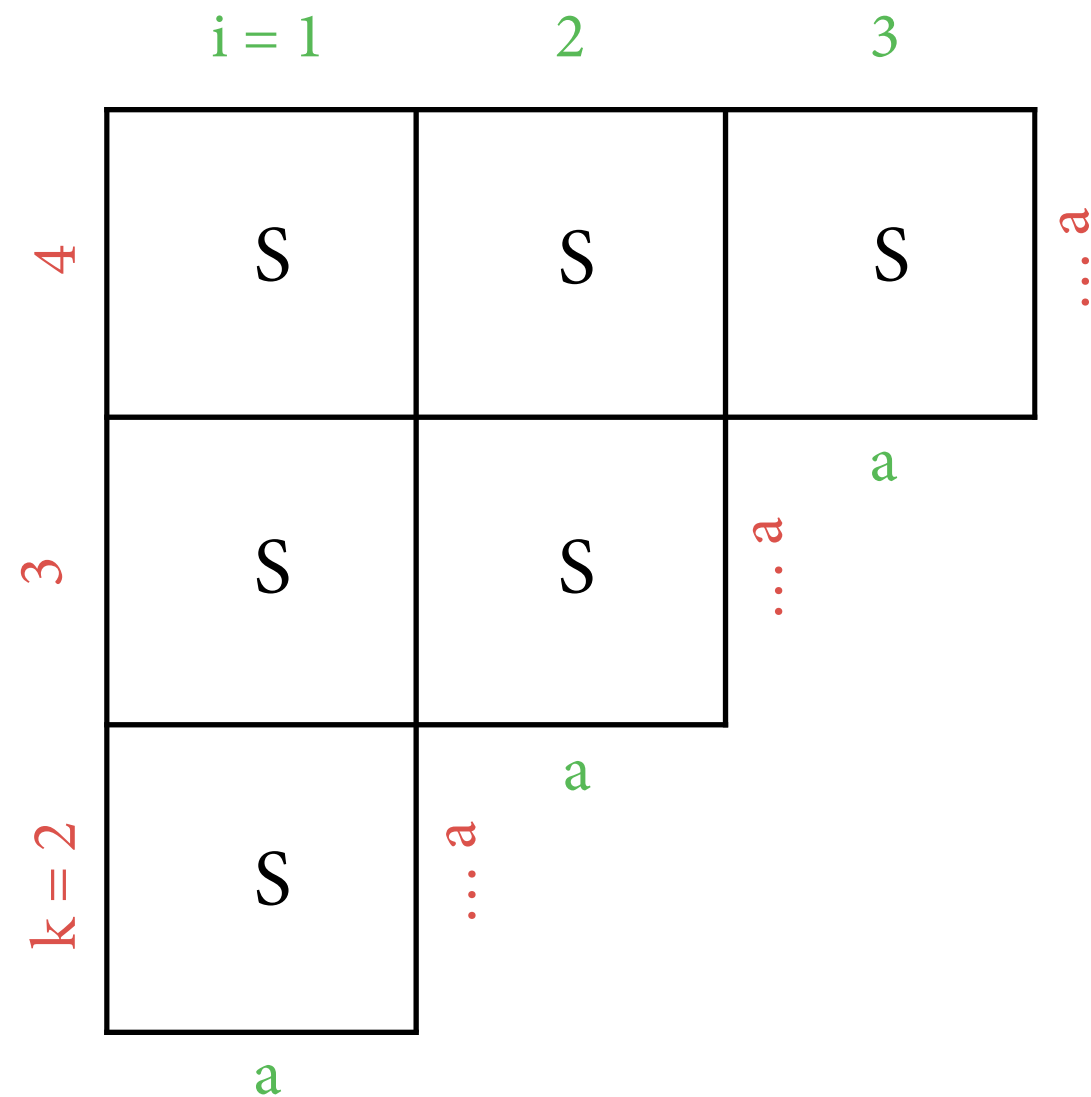
$S \rightarrow a$



The CKY Recognizer

$S \rightarrow S S$

$S \rightarrow a$



CKY recognizer: pseudocode

Data structure: $\text{Ch}(i,k)$ eventually contains $\{A \mid A \Rightarrow^* w_i \dots w_{k-1}\}$
(initially all empty).

for each i from 1 to n :

 for each production rule $A \rightarrow w_i$:

 add A to $\text{Ch}(i, i+1)$

for each *width* b from 2 to n :

 for each *start position* i from 1 to $n-b+1$:

 for each *left width* k from 1 to $b-1$:

 for each $B \in \text{Ch}(i, i+k)$ and $C \in \text{Ch}(i+k, i+b)$:

 for each production rule $A \rightarrow B C$:

 add A to $\text{Ch}(i, i+b)$

claim that $w \in L(G)$ iff $S \in \text{Ch}(1, n+1)$

Complexity

- *Time* complexity of CKY recognizer is $O(n^3)$, although number of parse trees grows exponentially.
- *Space* complexity of CKY recognizer is $O(n^2)$ (one cell for each substring).
- Efficiency depends crucially on CNF.
Naive generalization of CKY to rules $A \rightarrow B_1 \dots B_r$ raises time complexity to $O(n^{r+1})$.

Correctness

- Soundness: CKY *only* derives true statements.
 - ▶ If CKY puts A into $\text{Ch}(i,k)$, then there is rule $A \rightarrow BC$ and some j with $B \in \text{Ch}(i,j)$ and $C \in \text{Ch}(j,k)$.
 - ▶ Induction hypothesis: for shorter spans, have $B \Rightarrow^* w_i \dots w_{j-1}$.
Thus $A \Rightarrow B C \Rightarrow^* w_i \dots w_{j-1} C \Rightarrow^* w_i \dots w_{k-1}$
- Completeness: CKY derives *all* true statements.
 - ▶ Each derivation $A \Rightarrow^* w_i \dots w_{k-1}$ starts with a first step;
say $A \Rightarrow B C \Rightarrow^* w_i \dots w_{j-1} C \Rightarrow^* w_i \dots w_{k-1}$
 - ▶ Important: ensure that all nonterminals for shorter spans are known before filling $\text{Ch}(i,k)$.

Recognizer to Parser

- Parser: need to construct parse trees from chart.
- Do this by memorizing how each $A \in \text{Ch}(i,k)$ can be constructed from smaller parts.
 - ▶ built from $B \in \text{Ch}(i,j)$ and $C \in \text{Ch}(j,k)$ using $A \rightarrow B C$: store (B,C,j) in *backpointer* for A in $\text{Ch}(i,k)$.
 - ▶ analogous to backpointers in HMMs
- Once chart has been filled, enumerate trees recursively by following backpointers, starting at $S \in \text{Ch}(1,n+1)$.

Conclusion

- Context-free grammars: most popular grammar formalism in NLP.
 - ▶ there are also other, more expressive grammar formalisms
- CKY: most popular parser for cfgs.
 - ▶ very simple polynomial algorithm, works well in practice
 - ▶ there are also other, more complicated algorithms
- Next time: put parsing and statistics together.