Complexity

Martin Kay

Stanford University
and University of the Saarland

Complexity

Only really care about large values

What is the { least / most } amount of { time / space / ... } does it take for each

of the n { characters / words / nodes / ... } to compute f(n)

{ in the best case / in the worst case / on the average } as a function of n?

Complexity

What is the { least / most } amount of { time / space / ... } does it take for each

of the n { characters / words / nodes / ... } t

Can we afford it?
Is there a cheaper way?
Settle for an upper (lower) bound

{ in the best case / in the worst case / on the average } as a function of n?

Complexity

What is the { least / most } amount of { time / space / ... } does it take for each

How much better could we do, anyway?

of the n { characters / words / nodes / ... } to compute f(n)

{ in the best case / in the worst case / on the average } as a function of n?

Complexity

What is the {least / most} amount of {time / space / ...} does it take for each

of the n {characters / words / nodes / ...} to compute f(n)

There may be a trade off

{in the best case / in the worst case / on the average} as a function of n?

Complexity

What is the {least / most} amount of {time / space / ...} does it take for each

of the n {characters / words / nodes / ...} to compute f(n)

Not very interesting

{in the best case / in the worst case / on the average} as a function of n?

Complexity

What is the {least / most} amount of {time / space / ...} does it take for each

of the n {characters / words / nodes / ...} to compute f(n)

Generally hard

{in the best case / in the worst case / on the average} as a function of n?

Complexity

What is the {least / most} amount of {time / space / ...} does it take for each

of the n {characters / words / nodes / ...} to compute f(n)

The main concern

{in the best case / in the worst case / on the average} as a function of n?

Complexity

How fast does your algorithm run?

It depends on

- the programming language you write it in
- the computer you run it on
- how cleverly you code it up

Cannot give time or space. Instead, give complexity class.

Complexity

Program 1 will always do better than Program 2 for problems of size n if n is large enough

Asymptotic Complexity

In general, just the order of the asymptotic complexity is of interest, i.e. if it is a linear, a quadratic or an exponential function.

The order is denoted by a complexity class using the O-notation.

Complexity Classes: Big O

$O(1)$	constant
$O(\log n)$	logarithmic
$\log(\log(n))$	log log
$O(n)$	linear
$O(n^k)$	polynomial—quadratic
$O(m^n)$	exponential (intractable)

$g(n) \in O(f(n))$

$g(n) = O(f(n))$

Big O

Big Oh
Landau
Bachmann–Landau
asymptotic notation

} notation

All subsets

```
| ?- combination([1,2,3], C).
C = [1,2,3] ? ;
C = [1,2] ? ;
C = [1,3] ? ;
C = [1] ? ;
C = [2,3] ? ;
C = [2] ? ;
C = [3] ? ;
C = [] ? ;
no
```

```
combination([], []).
combination([H | T], [H | C]) :-
  combination(T, C).
combination([_ | T], C) :-
  combination(T, C).
```


All subsets

```
combination([], []).
combination([H | T], [H | C]) :-
  combination(T, C).
combination([_ | T], C) :-
  combination(T, C).
```

```
| ?- combination([1,2,3,4], C).
C = [1,2,3,4] ? ;
C = [1,2,3] ? ;
C = [1,2,4] ? ;
C = [1,2] ? ;
C = [1,3,4] ? ;
C = [1,3] ? ;
C = [1,4] ? ;
C = [1] ? ;
C = [2,3,4] ? ;
C = [2,3] ? ;
C = [2,4] ? ;
C = [2] ? ;
C = [3,4] ? ;
C = [3] ? ;
C = [4] ? ;
C = [] ? ;
no
```

2^n

Exponential
=
Intractable!

Concatenation (append)

```
| ?- concat([1,2,3],[4,5,6], X).
X = [1,2,3,4,5,6] ?
yes
```

m n

```
concat([], X, X).
concat([H | T], X, [H | Y]) :-
        concat(T, X, Y).
```

Linear in m
Constant for n

(Naive) Reverse

```
| ?- reverse([1,2,3,4], R).
R = [4,3,2,1] ?
yes
| ?-
```

```
concat([], X, X).
concat([H | T], X, [H | Y]) :-
        concat(T, X, Y).

reverse([], []).
reverse([H | T], R) :-
        reverse(T, RT),
        concat(RT, [H], R).
```



```
| ?- reverse( [ 1 ] , R ).
R = [1] ?   2 Rev, 1 Conc

| ?- reverse([1,2], R).
R = [2,1] ?   3 Rev, 3 Conc

| ?- reverse([1,2,3], R).
R = [3,2,1] ?   4 Rev, 6 Conc

| ?- reverse([1,2,3,4], R).
R = [4,3,2,1] ?   5 Rev, 10 Conc

| ?- reverse([1,2,3,4,5], R).
R = [5,4,3,2,1] ?   6 Rev, 15 Conc
```

Quadratic

```
concat([], X, X).
concat([H | T], X, [H | Y]) :-
        concat(T, X, Y).

reverse([], []).
reverse([H | T], R) :-
        reverse(T, RT),
        concat(RT, [H], R).
```

reverse([...], R) — n

n+1 Rev

$\binom{n+1}{2}$ Conc


```
| ?- reverse( [ 1 ] , R ).
R = [1] ?   2 Rev, 1 Conc

| ?- reverse([1,2], R).
R = [2,1] ?   3 Rev, 3 Conc

| ?- reverse([1,2,3], R).
R = [3,2,1] ?   4 Rev, 6 Conc

| ?- reverse([1,2,3,4], R).
R = [4,3,2,1] ?   5 Rev, 10 Conc

| ?- reverse([1,2,3,4,5], R).
R = [5,4,3,2,1] ?   6 Rev, 15 Conc
```


Quadratic


```
concat([], X, X).
concat([H | T], X, [H | Y]) :-
        concat(T, X, Y).

reverse([], []).
reverse([H | T], R) :-
        reverse(T, RT),
        concat(RT, [H], R).
```

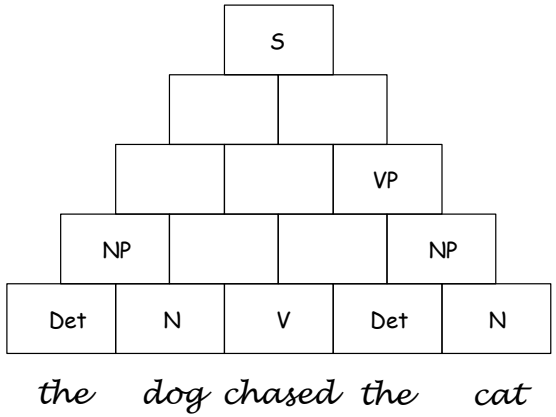

[]	1
[4]	2
[4, 3]	3
[4, 3, 2]	4
[4, 3, 2, 1]	5
	15 Concats.

Reverse

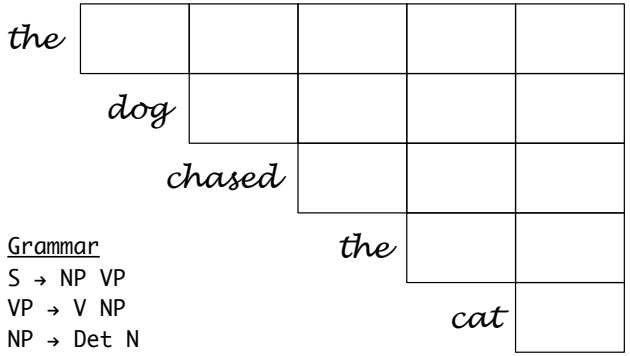
```
| ?- reverse([1,2,3,4], R).
R = [4,3,2,1] ?
yes
| ?-
```

Linear

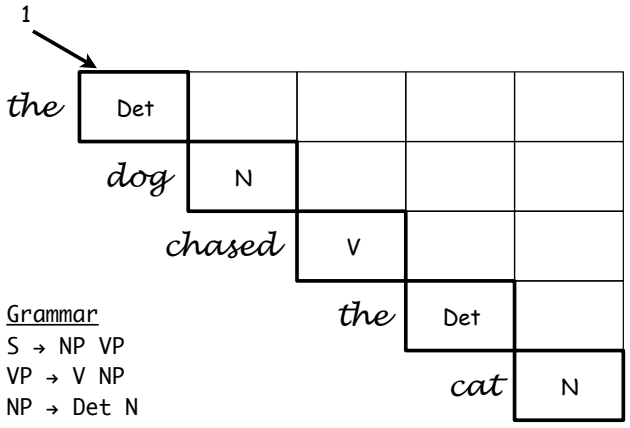
```
reverse1([H | T], X, Y) :-
        reverse1(T, [H|X], Y).
reverse1([], X, X).

reverse(X, Y) :-
        reverse1(X, [], Y).
```



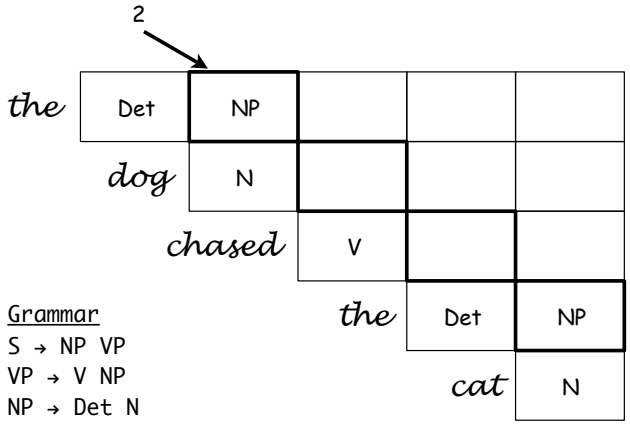
The Cocke-Kasami-Younger Algorithm (1960)



Grammar
S → NP VP
VP → V NP
NP → Det N

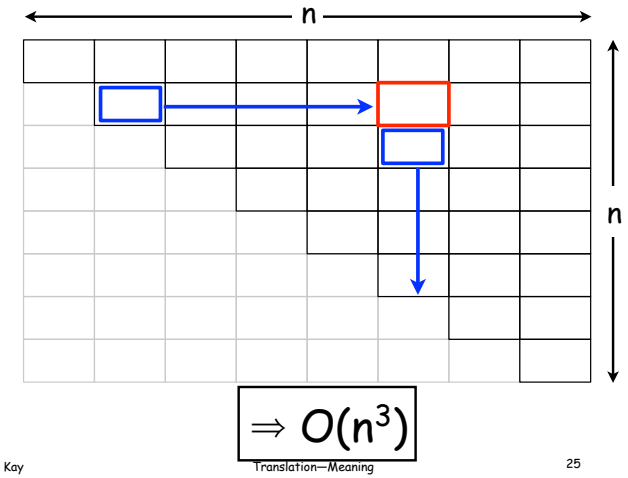


Grammar
S → NP VP
VP → V NP
NP → Det N



Grammar
S → NP VP
VP → V NP
NP → Det N

$\frac{n(n+1)}{2}$ = $O(n^2)$ cells to fill n ways of filling them



$\Rightarrow O(n^3)$

The CKY Algorithm

```
for len in (2..words.length)
  for pos in (0...words.length-len+1)
    for first_len in (1...len)
      for first_cat in chart[pos][first_len-1]
        for second_cat in chart[pos+first_len][len-first_len-1]
          if c = @grammar[[first_cat.cat, second_cat.cat]]
            add_edge(chart[pos][len-1], c, first_len, left, right)
          end
        end
      end
    end
  end
end
```

n^3

C^2

Big O

$f(x) \in O(g(x))$ as $x \to \infty$

iff

$|f(x)| \leq M|g(x)|$ for all $x > x_0$

i.e. for sufficiently large values of x has an upper bound which is a constant times g(x). Generally, we take the "as $x \to \infty$" part for granted, and write simply

$f(x) \in O(g(x))$ or $f(x) = O(g(x))$

Big O

- If x is a sum of several terms, we keep the one with the largest growth rate, and all others are omitted.
- If x is a product of several factors, any constants (terms in the product that do not depend on x) are omitted.

Complexity

- As n grows large, the n^2 will come to dominate, so that other terms can be neglected. For example, when n=500, $4n^2$ is 1000 times as large as 2^n.
- Coefficients become irrelevant when compared with, for example, an expression containing n^3 or n^2.
- Numbers of steps depends on the machine model, but differences among machines rarely contribute more than a constant factor.