### FoCS Lecture 2

Recursion and Complexity 14th October 2019 Anil Madhavapeddy & Amanda Prorok

 $E_0 \to E_1 \to \dots \to E_n \to v$ 

 $E_0 \to E_1 \to \ldots \to E_n \to v$ 

Focus on *expressions;* ignore *side-effects* for now.

This discipline of separating expression from effects is often known as *functional programming* 

We will return to side effects later in the course to make useful programs!

 $E_0 \to E_1 \to \dots \to E_n \to v$ 

```
# let rec power x n =
 if n = 1 then x
 else if even n then
     power (x *. x) (n / 2)
 else
     x *. power (x *. x) (n / 2)
```

 $E_0 \to E_1 \to \dots \to E_n \to v$ 

<pre># let rec power x n =</pre>	
if $n = 1$ then x	
else if even n then	
power (x *. x) (n /	2)
else	
x *. power (x *. x)	(n / 2)

power(2, 12)  $\Rightarrow$ power(4, 6)  $\Rightarrow$ power(16, 3)  $\Rightarrow$ 16 × power(256, 1)  $\Rightarrow$ 16 × 256  $\Rightarrow$ 4096

## Summing first n integers

<pre># let rec nsum n =</pre>
if $n = 0$ then
0
else
n + nsum (n - 1)

 $nsum 3 \Rightarrow 3 + (nsum 2)$  $\Rightarrow 3 + (2 + (nsum 1))$  $\Rightarrow 3 + (2 + (1 + nsum 0))$  $\Rightarrow 3 + (2 + (1 + 0))$ 

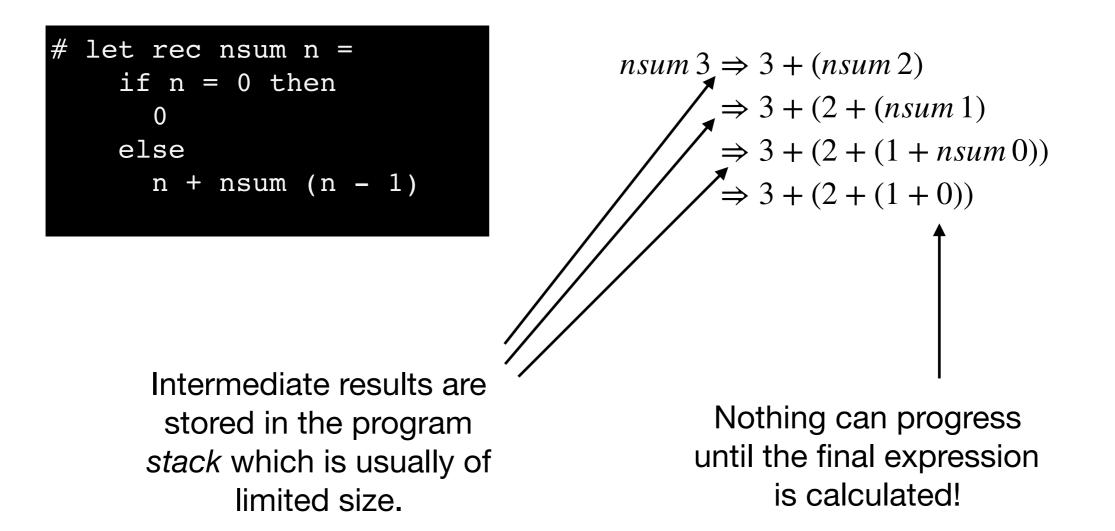
## Summing first n integers

<pre># let rec nsum n =</pre>
if $n = 0$ then
0
else
n + nsum (n - 1)

 $nsum 3 \Rightarrow 3 + (nsum 2)$   $\Rightarrow 3 + (2 + (nsum 1))$   $\Rightarrow 3 + (2 + (1 + nsum 0))$  $\Rightarrow 3 + (2 + (1 + 0))$ 

> Nothing can progress until the final expression is calculated!

## Summing first n integers



## Iteratively summing

#	let	rec	sumn	ning	j n	t	ota.	_ =	=
	i	f n =	= 0 t	her	ì				
		tota	al						
	e	lse							
		sum	ning	( n		1)	( n	+	total

# let rec nsum n =
 if n = 0 then
 0
 else
 n + nsum (n - 1)

## Iteratively summing

# ]	let	rec	summ	ling	n	tc	ota]		=
	if	n =	= 0 t	hen					
		tota	al						
	el	se							
		sum	ning	( n		1)	( n	+	total)

# let rec nsum n =
 if n = 0 then
 0
 else
 n + nsum (n - 1)

summing  $3\ 0 \Rightarrow$  summing  $2\ 3$  $\Rightarrow$  summing  $1\ 5$  $\Rightarrow$  summing  $0\ 6$  $\Rightarrow 6$   $nsum 3 \Rightarrow 3 + (nsum 2)$  $\Rightarrow 3 + (2 + (nsum 1))$  $\Rightarrow 3 + (2 + (1 + nsum 0))$  $\Rightarrow 3 + (2 + (1 + 0))$ 

## Iteratively summing

<pre># let rec summing n total =</pre>
if $n = 0$ then
total
else
summing (n - 1) (n + total)

Extra argument total acts as the *accumulator* to keep track explicitly instead of using the stack

summing 3 0  $\Rightarrow$  summing 2 3  $\Rightarrow$  summing 1 5  $\Rightarrow$  summing 0 6  $\Rightarrow$  6

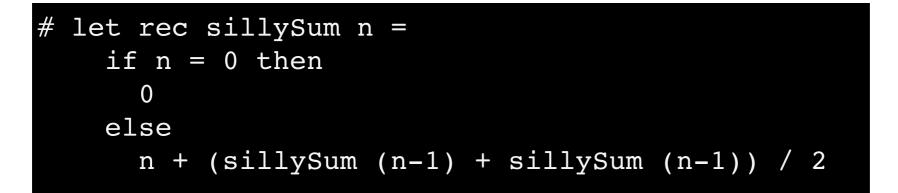
Algorithms like this are known as *iterative* or *tail recursive* 

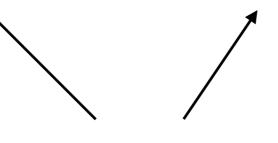
## Recursion vs iteration

- Why two terms *iterative* and *tail recursive*?
  - "Iterative" normally refers to a loop: e.g. coded using while.
  - "Tail-recursion" involves the recursive function call being the last thing that expression does.
- Tail-recursion is efficient only if the compiler detects it.
  - Mainly it saves space, though iterative code can run faster.
- Do not make programs iterative unless you determine the gain is significant.

How can we analyse our programs for efficiency?

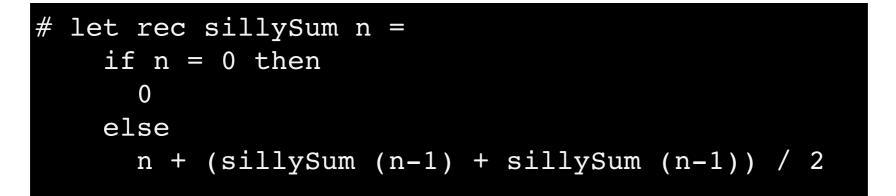
#### Silly summing first n integers

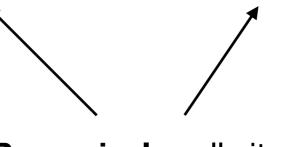




**Recursively** calls itself twice for every invocation

#### Silly summing first n integers





**Recursively** calls itself twice for every invocation

Should **assign** the result to a local variable to prevent evaluating it twice

```
# let x = 2.0 in
let y = Float.pow x 20.0 in
y *. (x /. y)
```

# Asymptotic complexity refers to how program costs grow with increasing inputs

#### Usually space or time, with the latter usually being larger than the former.

**Question:** if we double our processing power, how much does our computation capability increase?

## **Time Complexity**

complexity	1 second	1 minute	1 hour	gain
n	1000	60,000	3,600,000	×60
n lg n	140	4,893	200,000	×41
$n^2$	31	244	1,897	$\times 8$
$n^3$	10	39	153	$\times 4$
$2^n$	9	15	21	+6

complexity = milliseconds of runtime given an input of size n

#### Comparing Algorithms with O(n)

Formally, define f(n) = O(g(n))provided that  $|f(n)| \le c |g(n)|$ 

#### Comparing Algorithms with O(n)

Formally, define 
$$f(n) = O(g(n))$$
  
provided that  $|f(n)| \le c |g(n)|$ 

Intuitively, consider the most significant term and ignore constant or smaller factors

E.g. simplify 
$$3n^2 + 34n + 433 \rightarrow n^2$$

## Facts about O notation

O(2g(n)) is the same as O(g(n))  $O(\log_{10} n)$  is the same as  $O(\ln n)$   $O(n^2 + 50n + 36)$  is the same as  $O(n^2)$   $O(n^2)$  is contained in  $O(n^3)$   $O(2^n)$  is contained in  $O(3^n)$  $O(\log n)$  is contained in  $O(\sqrt{n})$ 

# Common complexity classes

<i>O</i> (1)	constant
$O(\log n)$	logarithmic
O(n)	linear
$O(n \log n)$	quasi-linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(a^n)$	exponential (for fixed $a$ )

#### Sample costs in O-notation

function	time	space
npower,nsum	O(n)	O(n)
summing	O(n)	<b>O</b> (1)
n(n + 1)/2	<b>O</b> (1)	<b>O</b> (1)
power	$O(\log n)$	$O(\log n)$
stupSum	$O(2^n)$	O(n)

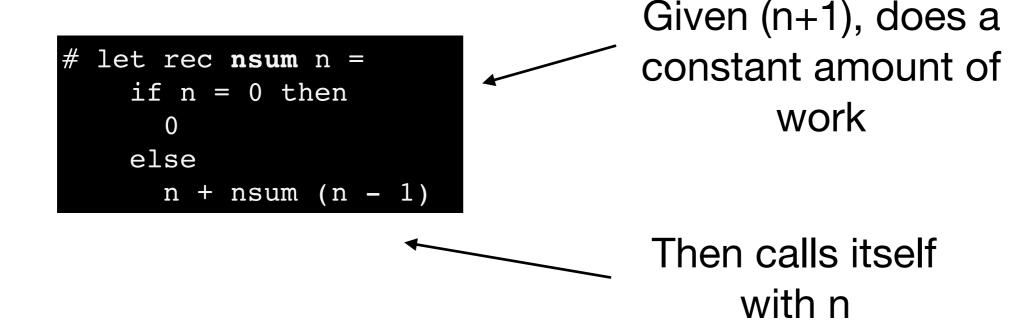
#### Simple recurrence relations

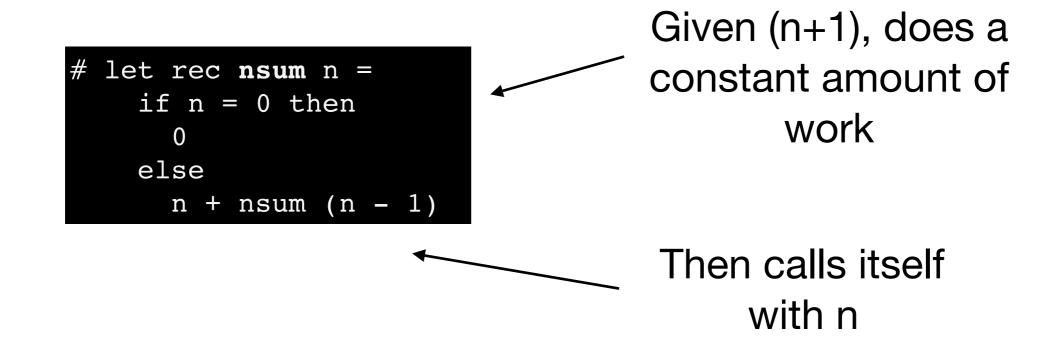
T(n): a cost we want to bound using O notation

Typical base case: T(1) = 1

Some recurrences:

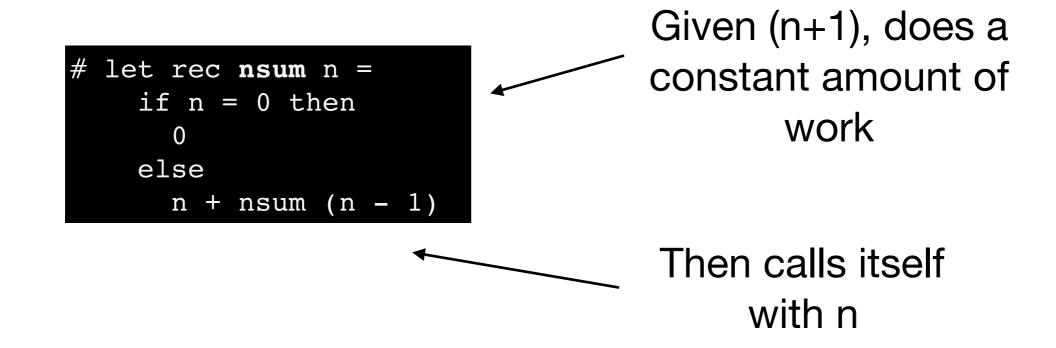
T(n + 1) = T(n) + 1 T(n + 1) = T(n) + n T(n) = T(n/2) + 1 T(n) = 2T(n/2) + n O(n) O(n)  $O(\log n)$  $O(n \log n)$ 





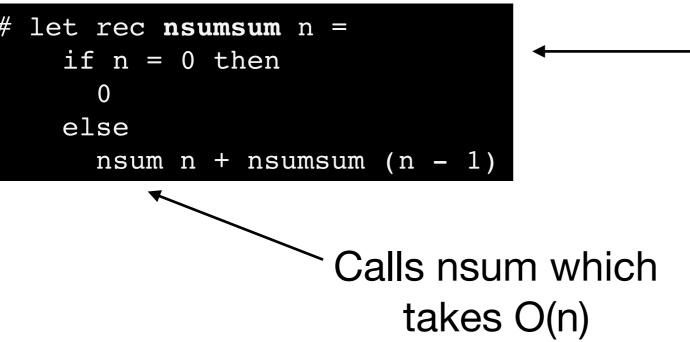
Therefore, recurrence relations are:

$$T(0) = 1$$
$$T(n+1) = T(n) + 1$$

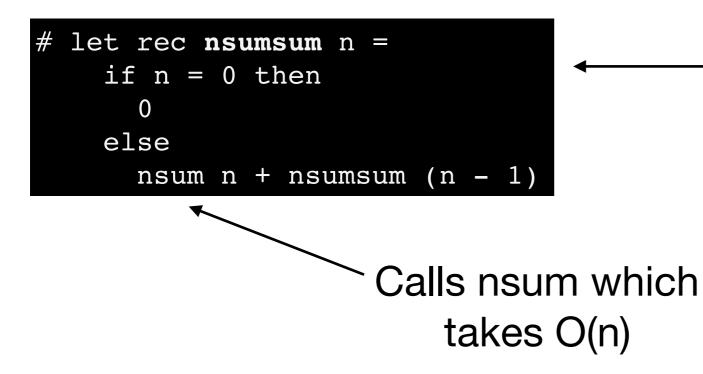


Therefore, recurrence relations are:

$$T(0) = 1$$
  
 $T(n+1) = T(n) + 1$   $O(n)$ 



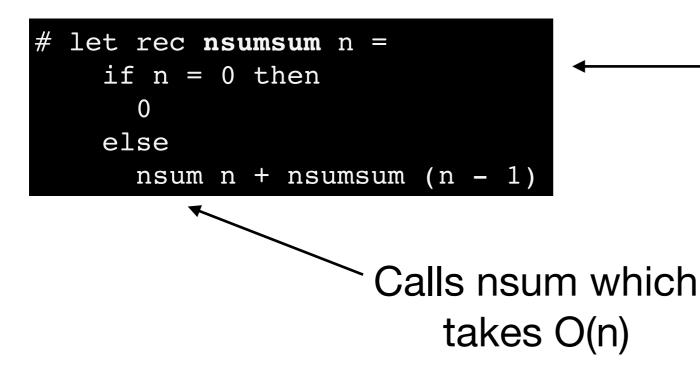
Calls itself recursively once



Therefore, recurrence relations are:

$$T(0) = 1$$
$$T(n+1) = T(n) + n$$

Calls itself recursively once

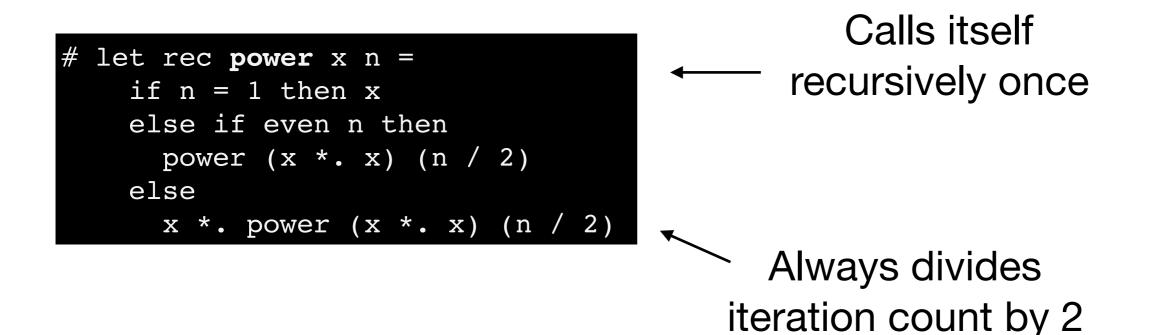


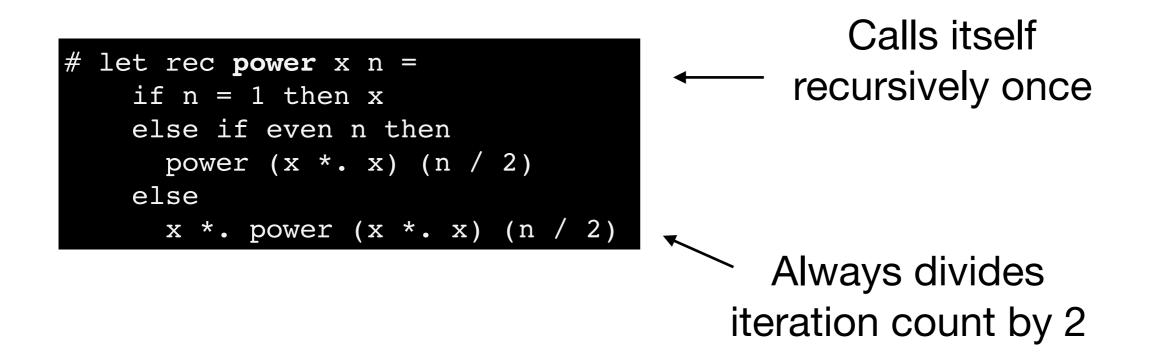
Calls itself – recursively once

Therefore, recurrence relations are:

$$T(0) = 1$$
$$T(n+1) = T(n) + n$$

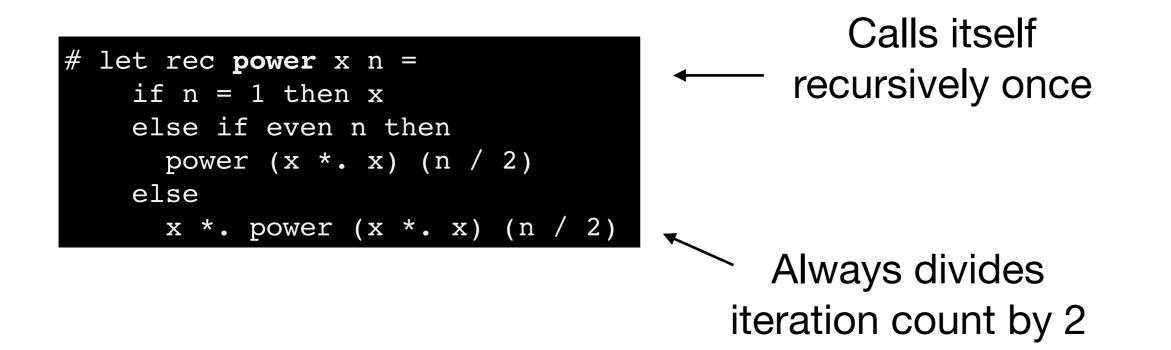
$$O(n^2)$$





Therefore, recurrence relations are:

$$T(0) = 1$$
$$T(n) = T(n/2) + 1$$



Therefore, recurrence relations are:

$$T(0) = 1$$
$$T(n) = T(n/2) + 1$$

 $O(\log n)$