

# Foundations of Computer Science: Lecture 2

**Recursion and Complexity**

10th October 2022

Anil Madhavapeddy

# The Practical Classes

<https://www.cl.cam.ac.uk/teaching/2223/OCaml/>

- Executed online in the [hub.cl.cam.ac.uk](http://hub.cl.cam.ac.uk) server
- There are 5 ticks, each of which have a deadline for submission 10 days after they are issued (except last tick, which goes into Lent term).

Tick 1: released 2022-10-07 due 2021-10-17

Tick 2: released 2022-10-14 due 2021-10-24

Tick 3: released 2022-10-21 due 2021-10-31

Tick 4: released 2022-10-28 due 2021-11-07

Tick 5: released 2022-11-04 due 2022-01-20

# Expression Evaluation

$$E_0 \rightarrow E_1 \rightarrow \dots \rightarrow E_n \rightarrow v$$

# Expression Evaluation

$$E_0 \rightarrow E_1 \rightarrow \dots \rightarrow E_n \rightarrow v$$

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!

# Expression Evaluation

$$E_0 \rightarrow E_1 \rightarrow \dots \rightarrow E_n \rightarrow v$$

```
# 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)
```

# Expression Evaluation

$$E_0 \rightarrow E_1 \rightarrow \dots \rightarrow E_n \rightarrow v$$

```
# 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)
```

power(2, 12)  $\Rightarrow$

power(4, 6)  $\Rightarrow$

power(16, 3)  $\Rightarrow$

16  $\times$  power(256, 1)  $\Rightarrow$

16  $\times$  256  $\Rightarrow$

4096

# Summing first $n$ integers

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

$$\begin{aligned}nsum\ 3 &\Rightarrow 3 + (nsum\ 2) \\ &\Rightarrow 3 + (2 + (nsum\ 1)) \\ &\Rightarrow 3 + (2 + (1 + nsum\ 0)) \\ &\Rightarrow 3 + (2 + (1 + 0))\end{aligned}$$

# Summing first $n$ integers

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

$$\begin{aligned}nsum\ 3 &\Rightarrow 3 + (nsum\ 2) \\ &\Rightarrow 3 + (2 + (nsum\ 1)) \\ &\Rightarrow 3 + (2 + (1 + nsum\ 0)) \\ &\Rightarrow 3 + (2 + (1 + 0))\end{aligned}$$



Nothing can progress  
until the final expression  
is calculated!

# Summing first $n$ integers

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

Intermediate results are stored in the program *stack* which is usually of limited size.

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

Nothing can progress until the final expression is calculated!

# Iteratively summing

```
# let rec summing n total =  
  if n = 0 then  
    total  
  else  
    summing (n - 1) (n + total)
```

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

# Iteratively summing

```
# let rec summing n total =  
  if n = 0 then  
    total  
  else  
    summing (n - 1) (n + total)
```

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

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

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

# Iteratively summing

```
# let rec summing n total =  
  if n = 0 then  
    total  
  else  
    summing (n - 1) (n + total)
```

*summing 3 0* ⇒ *summing 2 3*  
⇒ *summing 1 5*  
⇒ *summing 0 6*  
⇒ 6

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

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

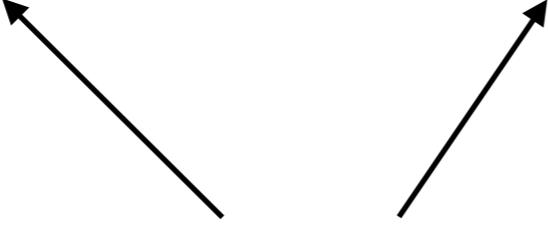
# Recursion vs iteration

- 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

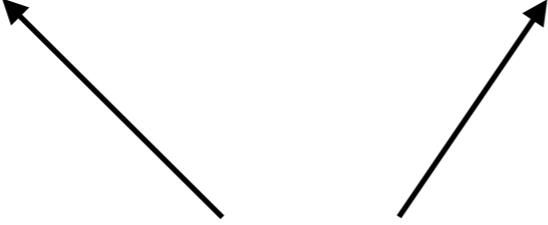
```
# let rec sillySum n =  
  if n = 0 then  
    0  
  else  
    n + (sillySum (n-1) + sillySum (n-1)) / 2
```



**Recursively** calls itself  
twice for every invocation

# Silly summing first $n$ integers

```
# let rec sillySum n =  
  if n = 0 then  
    0  
  else  
    n + (sillySum (n-1) + sillySum (n-1)) / 2
```



**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

<i>complexity</i>	<i>1 second</i>	<i>1 minute</i>	<i>1 hour</i>	<b>gain</b>
$n$	1000	60,000	3,600,000	$\times 60$
$n \lg n$	140	4,893	200,000	$\times 41$
$n^2$	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  $|f(n)| \leq c |g(n)|$

# Comparing Algorithms with $O(n)$

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

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

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

# Sample costs in O-notation

Function	Time	Space
npower, nsum	$O(n)$	$O(n)$
summing	$O(n)$	$O(1)$
$n(n + 1)/2$	$O(1)$	$O(1)$
power	$O(\log n)$	$O(\log n)$
sillySum	$O(2^n)$	$O(n)$

# Simple recurrence relations

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

Typical *base case*:  $T(1) = 1$

Some *recurrences*:

$$T(n + 1) = T(n) + 1 \quad O(n)$$

$$T(n + 1) = T(n) + n \quad O(n^2)$$

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

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

# Mapping this to OCaml

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

Given  $(n+1)$ , does a  
constant amount of  
work



Then calls itself  
with  $n$



# Mapping this to OCaml

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

Given (n+1), does a constant amount of work

Then calls itself with n

Therefore, recurrence relations are:

$$T(0) = 1$$

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

# Mapping this to OCaml

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

Given (n+1), does a constant amount of work

Then calls itself with n

Therefore, recurrence relations are:

$$\begin{aligned} T(0) &= 1 \\ T(n + 1) &= T(n) + 1 \end{aligned} \quad O(n)$$

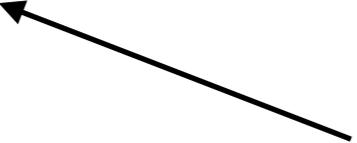
# Mapping this to OCaml

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

Calls itself  
recursively once



Calls nsum which  
takes  $O(n)$



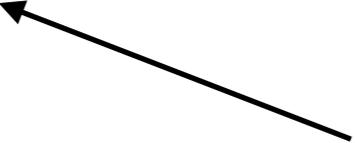
# Mapping this to OCaml

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

Calls itself  
recursively once



Calls nsum which  
takes  $O(n)$



Therefore, recurrence relations are:

$$T(0) = 1$$

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

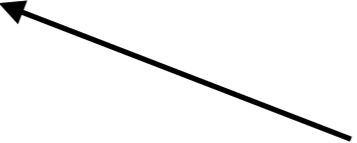
# Mapping this to OCaml

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

Calls itself  
recursively once



Calls nsum which  
takes  $O(n)$



Therefore, recurrence relations are:

$$\begin{aligned} T(0) &= 1 \\ T(n + 1) &= T(n) + n \end{aligned} \quad O(n^2)$$

# Mapping this to OCaml

```
# 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)
```

← Calls itself  
recursively once

← Always divides  
iteration count by 2

# Mapping this to OCaml

```
# 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)
```

← Calls itself  
recursively once

← Always divides  
iteration count by 2

Therefore, recurrence relations are:

$$T(0) = 1$$

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

# Mapping this to OCaml

```
# 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)
```

← Calls itself  
recursively once

← Always divides  
iteration count by 2

Therefore, recurrence relations are:

$$\begin{aligned} T(0) &= 1 \\ T(n) &= T(n/2) + 1 \end{aligned} \qquad O(\log n)$$