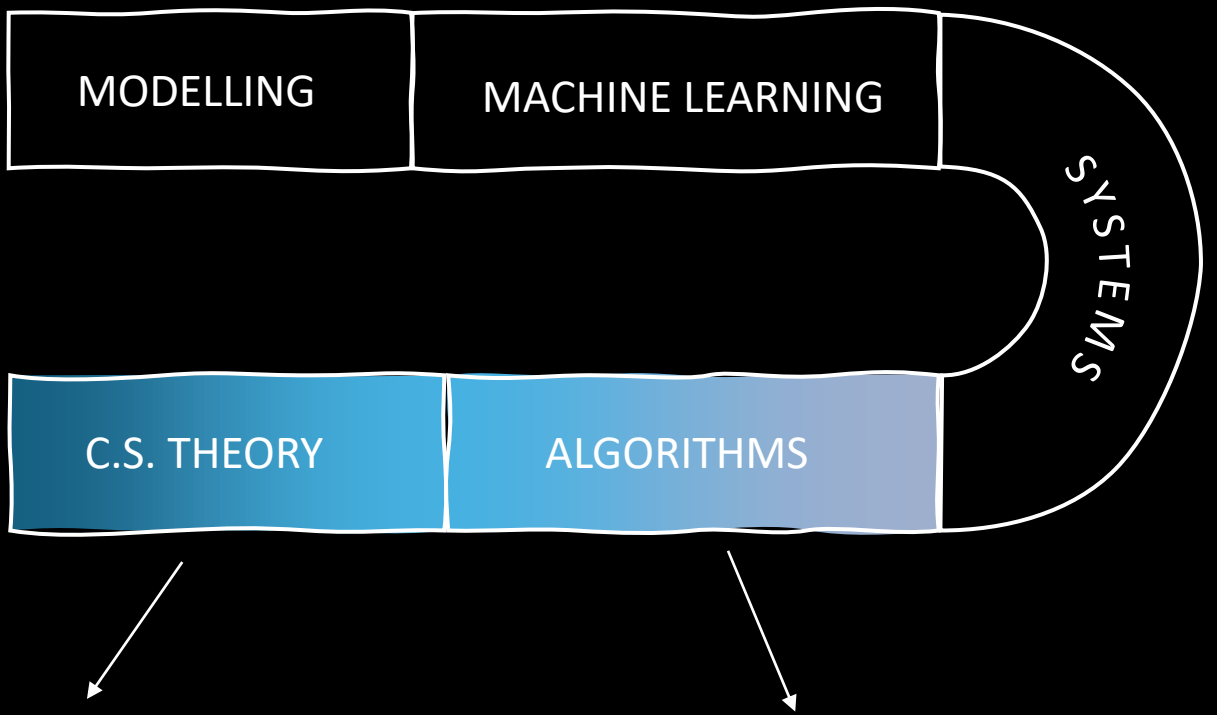
IA Algorithms 1 and 2

Dr Damon Wischik

~~What is an algorithm?~~

What sort of *activity* is working on algorithms?



SECTION 2

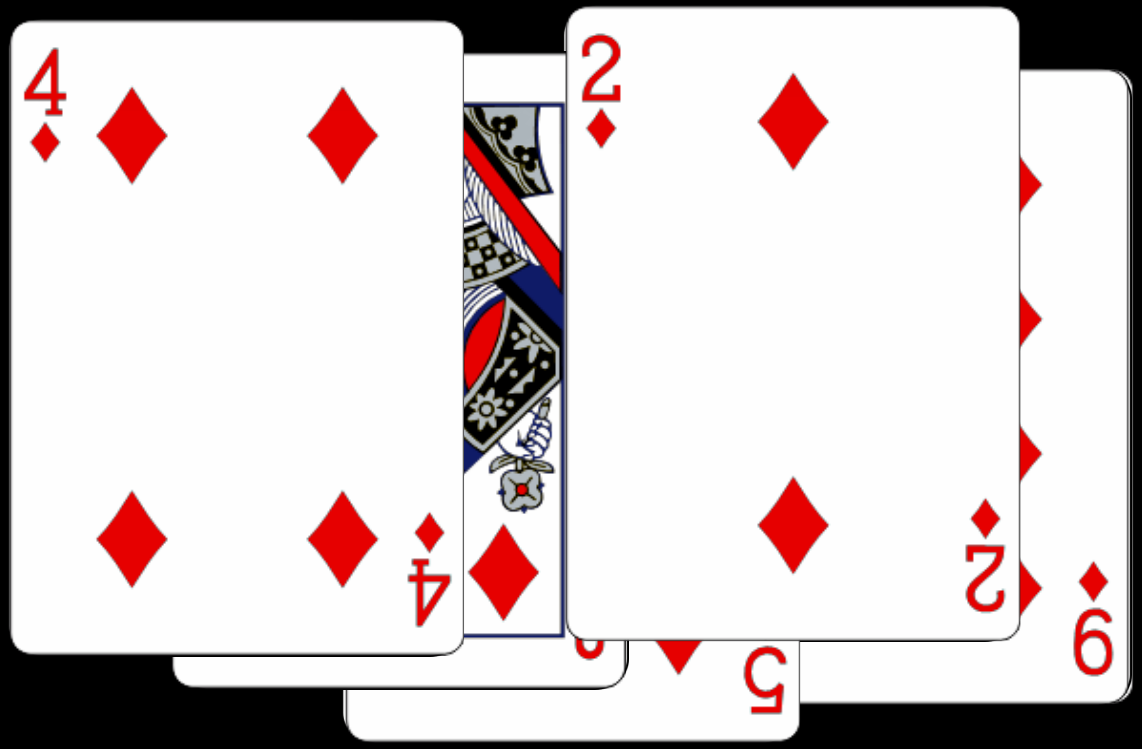
Sorting algorithms

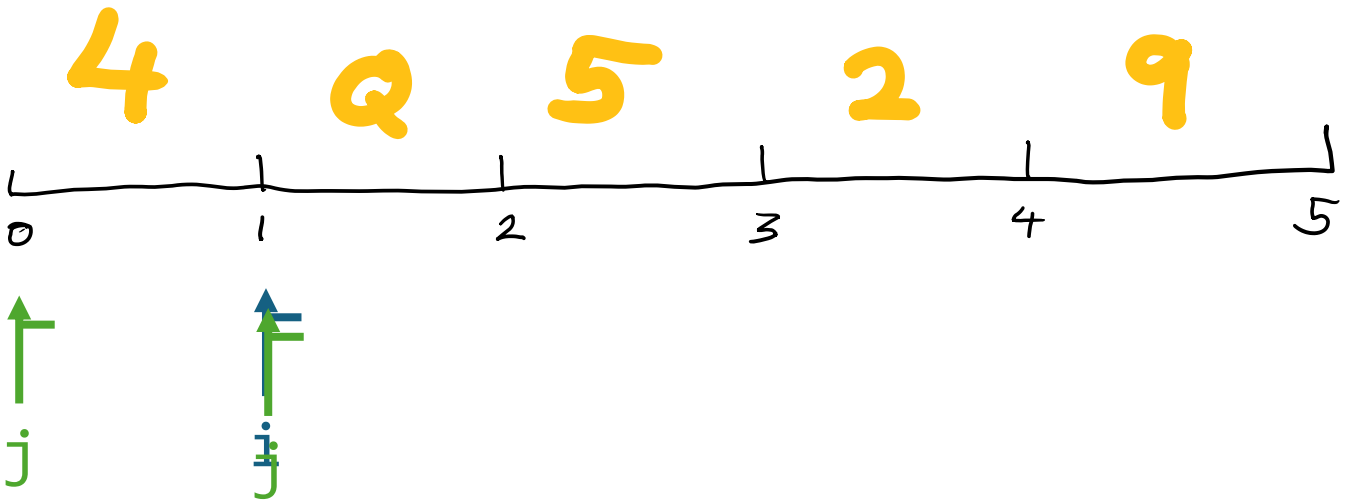
Insertion Sort

```
# let rec ins = function
    | x, [] -> [x]
    | x, y::ys ->
        if x <= y then
          x :: y :: ys
        else
          y :: ins (x, ys)

# let rec insort = function
    | [] -> []
    | x::xs -> ins (x, insort xs)
```

- Items from input are copied to the output
- Inserted in order, so the output is always sorted





NOTATION ALERT
Indexes refer to positions *between* cells.

x[i] = cell just to the right of i

x[i:j] = x[i] ... x[j-1]

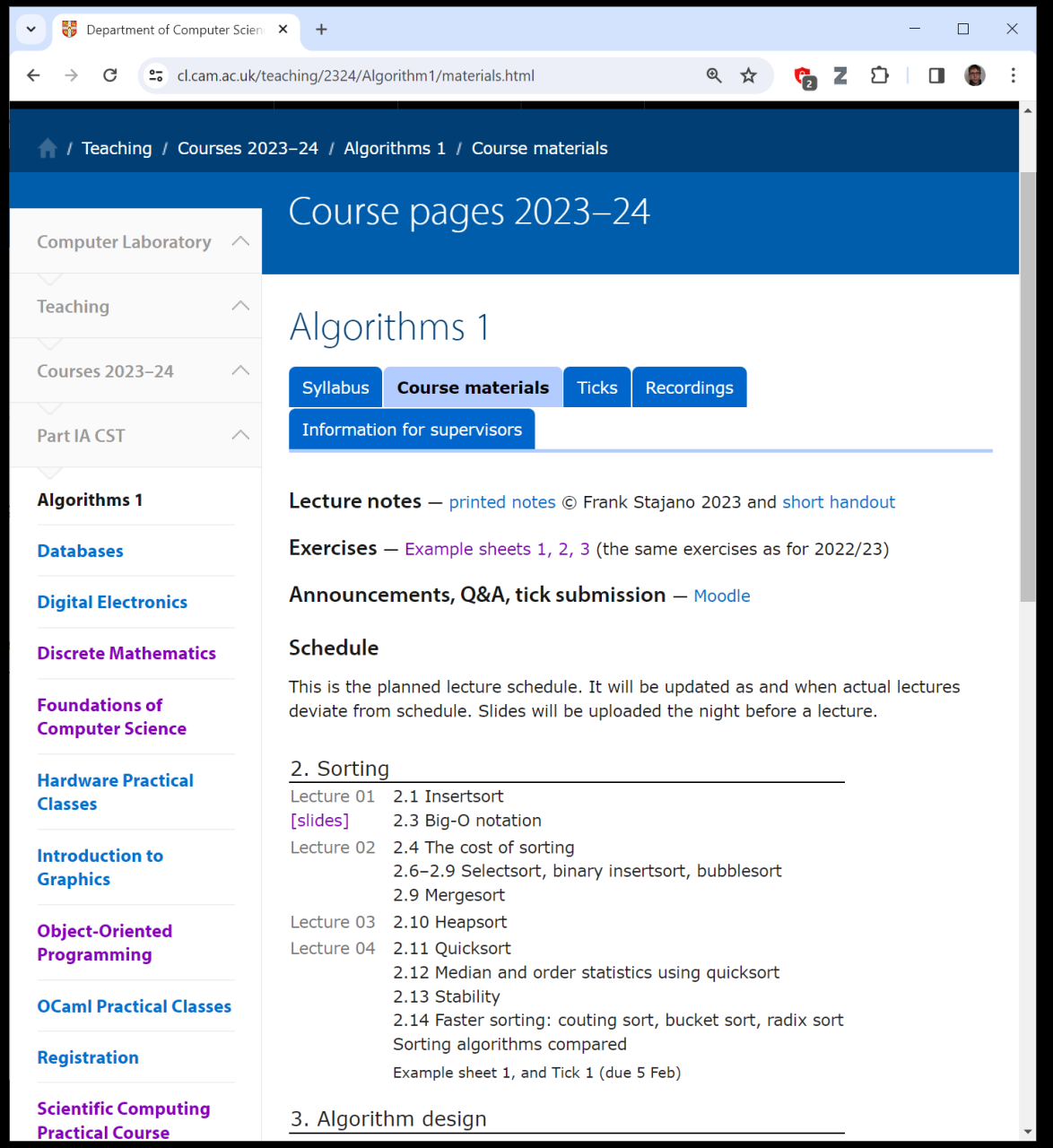
n = len(x)
x[n] ✗ out-of-bounds error
x[n-1] last element.

1 .... len(x)-1 inclusive

```
1  def insert_sort(x):
2    for i in 1..(len(x)-1):
3      # assert x[0:i] is sorted
4      j = i - 1
5      while j >= 0 and x[j] > x[j+1]:
6        swap x[j] with x[j+1]
7        j = j - 1
8      # assert x[0:i+1] is sorted
```

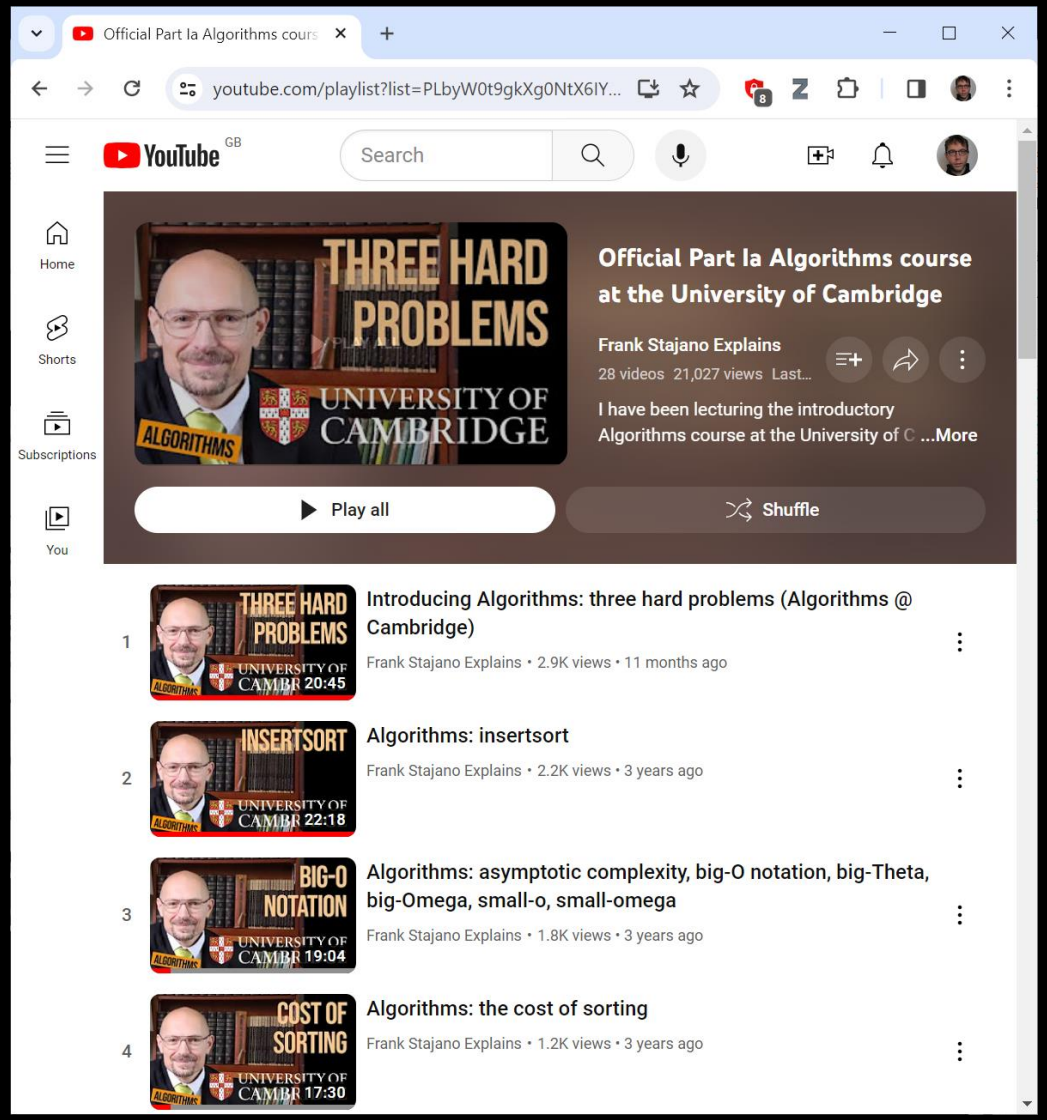
- Is this algorithm correct?
- What is its performance?
  Can we do better?

```
1  def insert_sort(x):
2    for i in 1..(len(x)-1):
3      # assert x[0:i] is sorted
4      j = i - 1
5      while j >= 0 and x[j] > x[j+1]:
6        swap x[j] with x[j+1]
7        j = j - 1
8      # assert x[0:i+1] is sorted
```



- lecture notes
- slides (uploaded the night before, and re-uploaded after the lecture)
- example sheets & ticks
- recordings

Pre-recorded videos from Prof Frank Stajano



Consent to recordings of live lectures

https://www.educationalpolicy.admin.cam.ac.uk/files/recording_policy.pdf

For any teaching session where your contribution is mandatory or expected, we must seek your consent to be recorded.

You are not obliged to give this consent, and you have the right to withdraw your consent after it has been given.

Q. Do you consent?

How to learn effectively

PASSIVE LEARNING	ACTIVE LEARNING	REFLECTIVE LEARNING
▪ attend lectures ▪ read code snippets, watch animations, see examples ▪ read notes, watch videos	▪ copy out any hand-writing from lectures ▪ annotate the code snippets in the handout	▪ example sheets and ticks ▪ execute algorithms by hand, and invent test cases to achieve full code coverage ▪ skeptical reading

§2.2 Proving correctness

Right from the beginning, and all through the course, we stress that the programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification.

Edsger Dijkstra, 1930—2002

What is an algorithm?

ChatGPT
An algorithm is a set of well-defined instructions or a step-by-step procedure designed to perform a specific task or solve a particular problem.

DW **You**
Are you an algorithm?

ChatGPT
Yes, I am a product of complex algorithms. Specifically, I'm powered by a type of algorithm known as a machine learning model.

DW **You**
If you are an algorithm, then what is the particular problem you solve?

ChatGPT
The overarching problem I solve is bridging the gap between human communication and machine understanding, making information and interaction more accessible through natural language processing.

```
1 def insert_sort(x):
2   for i in 1..(len(x)-1):
3     # assert x[0:i] is sorted
4     j = i - 1
5     while j >= 0 and x[j] > x[j+1]:
6       swap x[j] with x[j+1]
7       j = j - 1
8     # assert x[0:i+1] is sorted
```

Problem statement:

Given an array x, sort it in-place.

outline of proof of correctness of insert_sort:

let $P(i)$ be the predicate "The assertion on line 3 didn't fail at i"
$Q(i)$ ——— 8 ———

Lemma. $P(1)$ is true. ✓

Lemma. $P(i) \Rightarrow Q(i)$. for $i \le len(x)-1$ — see Ex. Sheet 2.

Lemma. $Q(i) \Rightarrow P(i+1)$ for $i \le len(x)-1$ ✓ trivial.

By induction $Q(n-1)$ is true, ie when $i = n-1$ then $x[0:i+1]$ is sorted (at line 8)
$x[0:n]$ ———
x ———

This is the last line of code that's run, so x can't change subsequently.

Thus, when the alg. terminates x is sorted.

§2.3 Computational complexity

Asymptotic complexity refers to how program costs grow with increasing inputs

Usually **space** or **time**, with t
larger than the

Question: if we double our p
much does our computatio

Insertion Sort

```
# let rec ins = function
    | x, [] -> [x]
    | x, y::ys ->
        if x <= y then
          x :: y :: ys
        else
          y :: ins (x, ys)

# let rec insort = function
    | [] -> []
    | x::xs -> ins (x, insort xs)
```

- Items from input are copied to the output
- Inserted in order, so the output is always sorted

Complexity is $O(n^2)$ comparisons
vs the theoretical best case of $O(n \log n)$

We'll analyze our algorithm's running time on large problems, and pretend we're running on an idealized machine:

- we can create arrays as large as we'll need, in time proportional to array size
- any array element $x[i]$ can be accessed in constant time
- all numerical operations take constant time

```
1  def insert_sort(x):
2    for i in 1..(len(x)-1):
4      j = i - 1
5      while j >= 0 and x[j] > x[j+1]:
6        swap x[j] with x[j+1]
7        j = j - 1
```

let $n = \text{len}(x)$.

outer loop runs $n-1$ times

inner loop runs $\leq i$ times.

cost of inner loop $\leq k_1 i$ k_1 const.

full cost $\leq \sum_{i=1}^{n-1} (k_1 i + k_2)$ k_2 const

k_2: extra cost per outer loop

$= \frac{1}{2} k_1 n(n-1) + k_2(n-1)$.

Why is this this style of complexity analysis OK?

- It's the conventional mathematical playground for this discipline
- It's often a good approximation to real performance, unless ...

Q. In what situations would this sort of analysis be a bad idea?

- Cache locality:
  it's faster if our code has a small working set,
  so that most memory accesses hit the CPU's cache.
- Large constants:
  complexity analysis just says "constant cost", it doesn't say what the constant is — and it may be huge.
- Small problems:
  if n is small then there's no point in asymptotic analysis,
  and we should just benchmark.
- Arbitrarily large problems:
  for an array of length n we need $\Theta(\log n)$ bits to even store a pointer,
  so memory access isn't really $O(1)$.