Complexity Analysis: revision

The running time of algorithms
Big-Oh notation for function growth
Standard analysis functions

Count distinct items in non-decreasing list

Count distinct items in non-decreasing list

First attempt:

```
for each item in turn
  if no item after it is the same
    count 1
```

```
def unique_check1(inputlist):
    count = 0
    for i in range(len(inputlist)):
        unique = True
        for j in inputlist[i+1:]:
            if inputlist[i] == j:
                unique = False
        if unique:
            count = count + 1
    return count
```

Count distinct items in non-decreasing list

Second attempt:

```
for each item in turn
   if the next item is different
      count 1
```

```
def unique_check2(inputlist):
    count = 1
    for i in range(len(inputlist)-1):
        if inputlist[i] != inputlist[i+1]:
            count = count + 1
    return count
```

```
>>> listExamples.perf_check_random(50)
List length:  50
Count 1 time     : 0.0058307576691731 36
Count 2 time     : 0.0049839922576211 39 36
>>> listExamples.perf_check_random(500)
List length:  500
Count 1 time     : 0.030033696792088 45 381
Count 2 time     : 0.0061229826533235 61 381
>>> listExamples.perf_check_random(5000)
List length:  5000
Count 1 time     : 2.237111685331911 3803
Count 2 time     : 0.00756309006828814 74 3803
>>> listExamples.perf_check_random(50000)
List length:  50000
Count 1 time     : 231.77267158718314 37429
Count 2 time     : 0.025658405560534447 37429
```

difference in running time for small lists doesn't matter

... but for big lists?

For lists of length 50000, the 2nd method is 10000 times faster.

Initial analysis

Both algorithms are *correct*

At high-level pseudocode, they look similar

```
for each item in turn
  if no item after it is the same
    count 1
```

```
for each item in turn
  if the next item is different
    count 1
```

but (when implemented in Python) their runtimes are significantly different on large lists.

Experimental analysis tells us that algorithm 2 is probably more efficient

Experimental comparison is not enough

1. Comparison requires full implementation of each algorithm (and development of a test framework)
2. Performance depends on the implementation in a particular language
    - underlying language constructs may introduce hidden complexity
3. Comparison requires testing in same software and hardware environment
4. Results are dependent on the test cases selected

It would be better if we can do an initial analysis on the algorithm pseudocode, and instead of checking time, we could check "work done” …

Counting basic steps

We will consider the following to be *basic steps*:

- reading the value of a variable
- assigning a value to a variable
- simple arithmetic operations (e.g. adding two numbers)
- comparing the values of two variables (basic types)
- calling a function (note: *calling* the function, not executing it)
- returning a value
- reading the value at a given index in a Python list
    - (we will show in later lectures why this can be counted as basic ...)

We expect each of these steps to take (approximately) a constant time to complete, regardless of the value

```
- e.g.  z = 1
        z = 1000000
```

takes the same time

```
>>> listExamples.perf_check_random(50)
List length:  50
1225  steps
Count 1 time     : 0.0058307576917318l 36
49  steps
Count 2 time     : 0.004983992257621139 36
>>> listExamples.perf_check_random(500)
List length:  500
124750  steps
Count 1 time     : 0.03003369679208845 381
499  steps
Count 2 time     : 0.006122982653323561 381
>>> listExamples.perf_check_random(5000)
List length:  5000
12497500  steps
Count 1 time     : 2.237111685331911 3803
4999  steps
Count 2 time     : 0.007563090068288l474 3803
>>> listExamples.perf_check_random(50000)
List length:  50000
1249975000  steps
Count 1 time     : 231.77267158718314 37429
49999  steps
Count 2 time     : 0.025658405560534447 37429
```

"step" is a comparison of two values

Worst-case upper bound

The number of steps for an algorithm will vary depending on the precise values in the input, but may grow in a pattern dependent on the *size* of the input.

Can we find a worst-case upper bound on the number of steps?

We don't care about small lists.

As the inputs grows past a certain size, we want a guarantee that the number of steps will be less than some function of the input size.

Big-oh notation

Consider two functions *f* and *g*, mapping positive integers to positive integers (so $f : \mathbb{N} \rightarrow \mathbb{N}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$)

We will say *f(x)* is *O(g(x))* if there are two constant values *k* and *C* so that whenever *x* is bigger than *k*, $f(x) \leq C^*g(x)$.

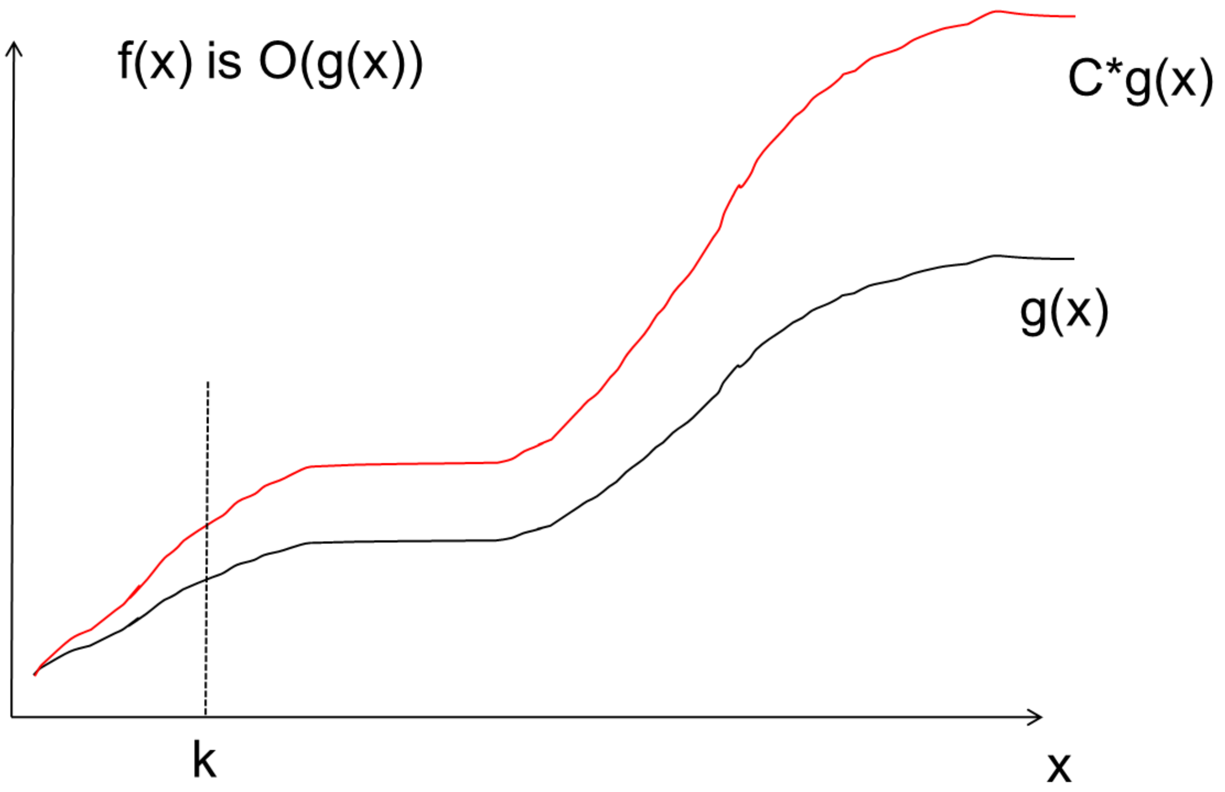
This means that when *x* is big enough, *f(x)* is never more than some constant multiple of *g(x)*, and so *f(x)* will not become drastically worse than *g(x)*.

We read this as "*f(x)* is big-oh of *g(x)*"

Formally,

f(x) is O(g(x)) if and only if $\exists k \, \exists C \, \forall x > k \;\; f(x) \leq C^*g(x)$

What does big-oh mean for function growth?



For all values of x > k, f(x) will be below the red line.

Big-Oh is an upper bound ...

We use the Big-Oh notation to restrict the functions we need to deal with.

Any polynomial with highest degree k is $O(n^k)$

Note: Big Oh gives an upper bound on the growth rate of the function

- any function that is $O(n^k)$ is also $O(n^{k+1})$

The standard functions

We will consider 7 standard functions to describe the worst-case upper bounds:

- constant
- logarithmic
- linear
- log linear
- quadratic
- cubic
- exponential

Constant

If f(.) specifies the runtime of a function on input of size n, and

f(n) = c, for some fixed constant value c

then the runtime is *independent* of the size of the input.

For example: a function which returns the value in the first position in a list – doesn't matter how long the list is.

Note: we say such a function is O(1).

Logarithmic

$f(n) = \log_b n$, for some fixed constant value b

The *log* of a number is the power to which the base must be raised to give the number.

$\log_b n = x$ if and only if $b^x = n$

$\log_{10} 1000$ is the x which gives $10^x = 1000$ so $x=3$

$\log_{10} 10$ is the x which gives $10^x = 10$ so $x=1$

$\log_{10} 1$ is the x which gives $10^x = 1$ so $x=0$

$\log_{10} 0.1$ is the x which gives $10^x = 0.1$ so $x= -1$

$\log_{10} 0.001$ is the x which gives $10^x = 0.001$ so $x= -3$

$\log_{10} 0$ is the x which gives $10^x = 0$ so $x=-\infty$

$\log_{10} 8$ is the x which gives $10^x = 8$ so $x\sim1$

$\log_{10} 117$ is the x which gives $10^x = 117$ so $x\sim2$

$\log_2 8$ is the x which gives $2^x = 8$ so $x=3$

$\log_2 64$ is the x which gives $2^x = 64$ so $x=6$

$\log_2 128$ is the x which gives $2^x = 128$ so $x=7$

Binary Search

Binary Search

```
def binary_search(my_list, target):
    if len(my_list) < 1:
        return False

    found = False
    low = 0
    upp = len(my_list)-1
    while low <= upp and not found:
        mid = (low + upp) //2
        if my_list[mid] == target:
            found = True
        else:
            if target < my_list[mid]:
                upp = mid-1
            else:
                low = mid+1
    return found
```

```
def binary_search(my_list, target):
    if len(my_list) < 1:
        return False

    found = False
    low = 0
    upp = len(my_list)-1
    while low <= upp and not found:
        mid = (low + upp) //2
        if my_list[mid] == target:
            found = True
        else:
            if target < my_list[mid]:
                upp = mid-1
            else:
                low = mid+1
    return found
```

Worst case analysis:

each time round the loop, we cut the list in half.

How many times can you divide by 2 until you reach a value of 1 or less?

$\boxed{\log_2 n}$

$\underbrace{n/2\ /2\ /2\ \ldots\ /2}_{\textit{what is smallest number of 2s to make this true?}} \leq 1$?

$n \leq \underbrace{2*2*2\ *\ldots*2}_{\textit{what is smallest number of 2s to make this true?}}$

$n \leq 2^k$

what is the smallest value of k?

We will do this operation (repeatedly dividing by 2) so often, that we will stop writing the base = 2

From now on, in this module, unless told otherwise,

$$\boxed{\log n \quad \text{means } \log_2 n}$$

Linear

f(n) = n

Typically, this comes from an algorithm with a single loop that iterates over each element of an input list.

E.g.

```
unique_check2(my_list)
```

Count unique elements in non-decreasing list

Second attempt:

```
for each element in turn
   if the next element is different
      count 1
```

```
def unique_check2(inputlist):
    count = 1
    for i in range(len(inputlist)-1):
        if inputlist[i] != inputlist[i+1]:
            count = count + 1
    return count
```

First attempt:

```
for each item in turn
    if no item after it is the same
        count 1
```

This pseudocode has a single loop that iterates over each item of the list.

So why is it not linear?

Log Linear

$f(n) = n \log n$

We will see later a number of sorting algorithms with complexity $O(n \log n)$

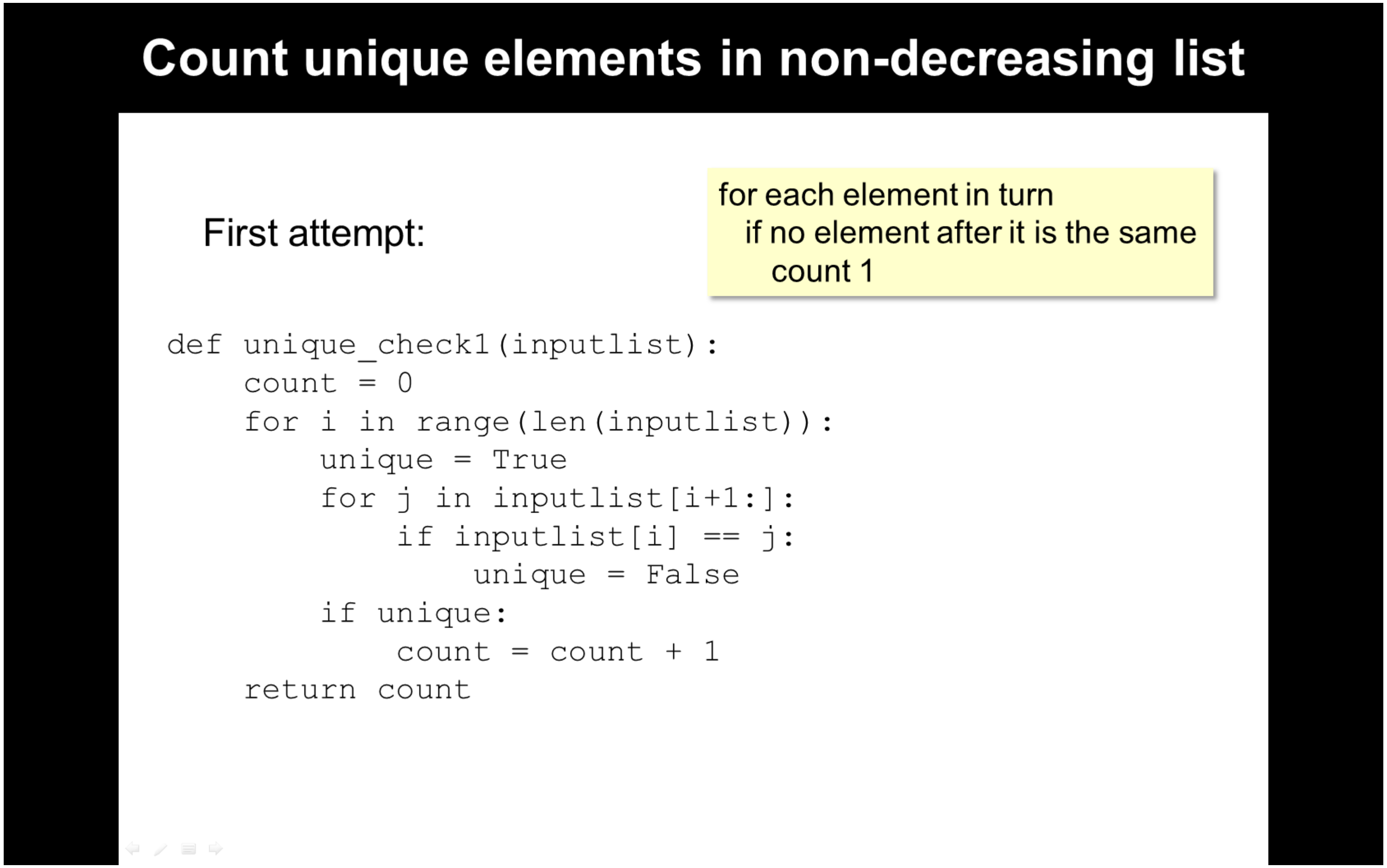
Typically, the algorithm is doing something like binary search, but repeating it once for each position in the list.

Quadratic

$f(n) = n^2$

Typically this comes from algorithms with a nested loop where we are iterating over the same structure

E.g. `unique_check1(my_list)`



Count unique elements in non-decreasing list

First attempt:

```
for each element in turn
   if no element after it is the same
      count 1
```

```
def unique_check1(inputlist):
    count = 0
    for i in range(len(inputlist)):
        unique = True
        for j in inputlist[i+1:]:
            if inputlist[i] == j:
                unique = False
        if unique:
            count = count + 1
    return count
```

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + (n-1) + n = 0.5 * n * (n+1)$$

Cubic

$f(n) = n^3$

Typically this comes from algorithms with a doubly nested loop where we are iterating over the same structure

We can go higher, to any arbitrary degree.

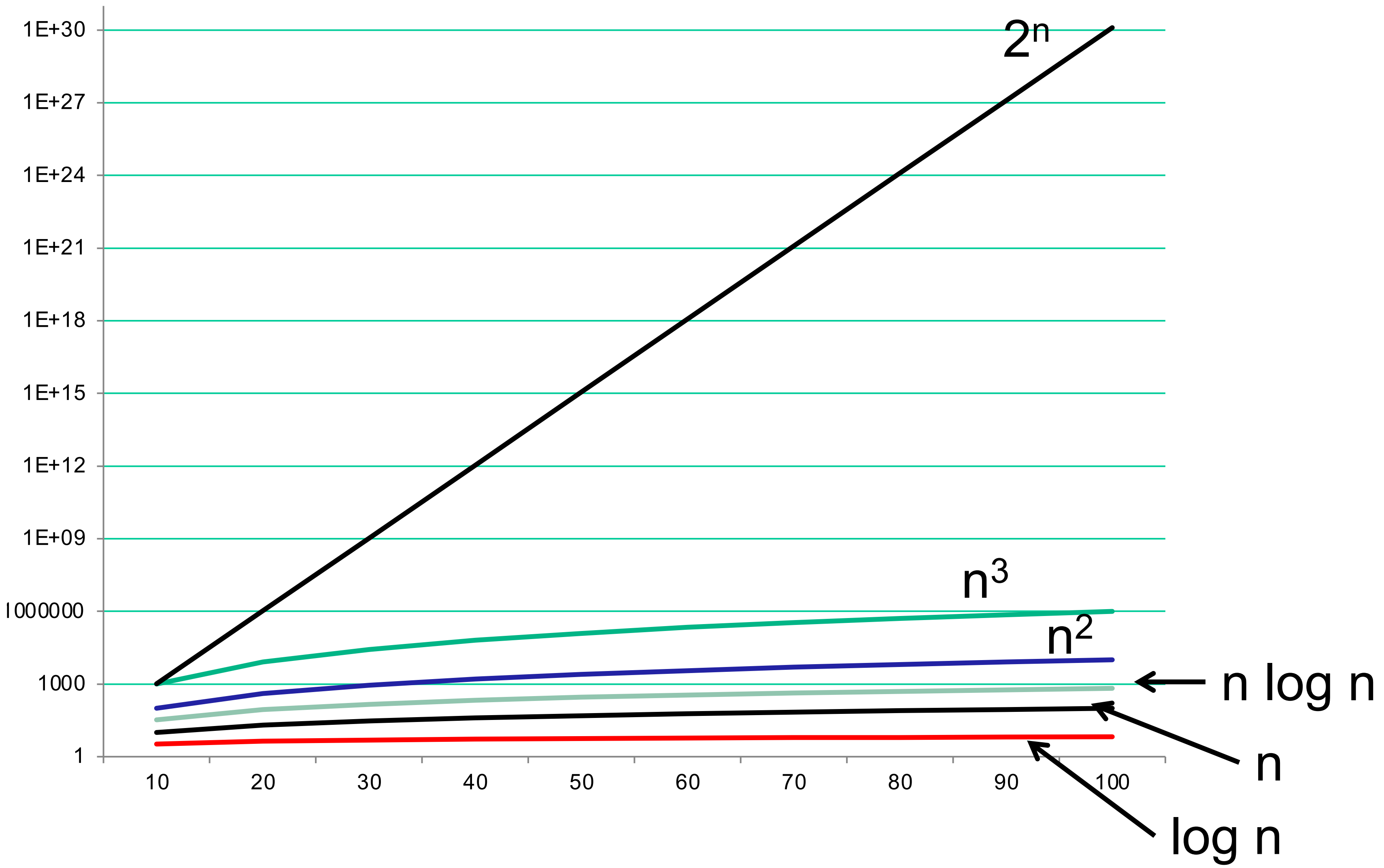
Very few of the algorithms we will consider in this module have run time approaching this complexity

Exponential

$f(n) = c^n$, for some constant c

Typically this comes from algorithms with a loop where the number of operations increases by a factor of c each time round the loop

Algorithms with exponential running time are considered inefficient, although we will see some practical algorithms of this sort later in the degree program.


1E+30
1E+27
1E+24
1E+21
1E+18
1E+15
1E+12
1E+09
1000000
1000
1
10
20
30
40
50
60
70
80
90
100
2^n
n^3
n^2
n log n
n
log n

Next week ...

NO LECTURES and NO LABS

But use your time to prepare and practice:

- If your 1st year programming was weak, revise 1st year python
- Complete the lab exercise on classes and objects for cards
- Revise this lecture and CS1113

Next lecture is on Wednesday 27th September