5008CEM Programming For Developers

Big O Notation

Mini-Lecture 1

Beate Grawemeyer | John Halloran

Overview / Learning Outcomes

- This is one of three mini-lectures on Big O Notation
- We cover:
  - Orders of complexity
  - Constant complexity
  - Logarithmic complexity
  - Linear complexity
- At the end of this lecture you should:
  - Understand the principles of these three types of complexity
  - Be aware of example algorithms which have these types of complexity

Why do we have to be concerned with complexity?

- We need to know the time a program will take to execute, given the data that’s input
  - This needs to be as short as possible
  - This is **time complexity**
- We need to know the space (memory) a program will take to execute, given the data that’s input
  - This needs to be as little as possible
  - This is **space complexity**
- We will focus mainly on time complexity

Isn’t it all a bit theoretical / mathy?

- … and isn’t the Computing degree supposed to be practical / applied?
- Yes, but any software developer must be able to estimate the complexity of their code
- Which is why we’re doing it
- We’ll introduce relatively easy ways to estimate Big O

Haven’t we seen all this before?

- Yes, to an extent
  - 4000CEM and 4003CEM in particular
- Some topics (especially tough ones) come around again on your degree
- The idea is to revise, consolidate and extend over a number of modules
- The idea of covering Big O again here is:
  - You get the chance to improve your understanding
  - You get new opportunities to apply that understanding
  - You get new examples and exercises / challenges
  - You get better at doing Big O – including more challenging stuff e.g. Big O for recursions, and comparative Big O

Orders of time complexity

Best

Worst

Mathematical term	Big O Notation
Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
Log Linear	$O(n \log n)$
Polynomial *	$O(n^k)$
Exponential	$O(2^n)$
Factorial	$O(n!)$

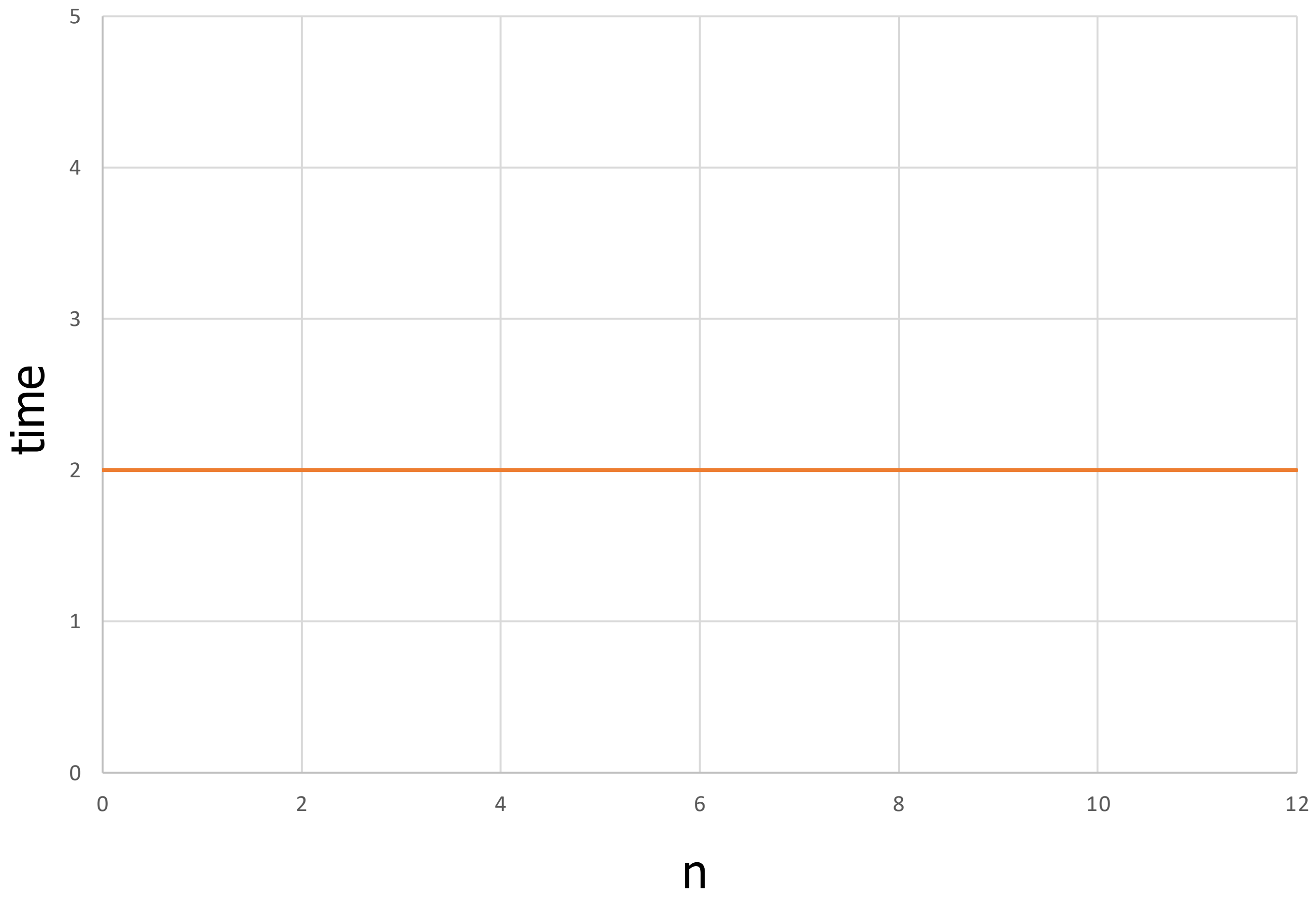
* Includes Quadratic: $O(n^2)$

Constant: O(1)

- Constant complexity
  - n (the size of the data) does not matter
  - Always takes the same time
  - e.g. getting the first element in an array

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 42]
b = [ 10, 11, 13]

print(a[0])
print[b[0])
```

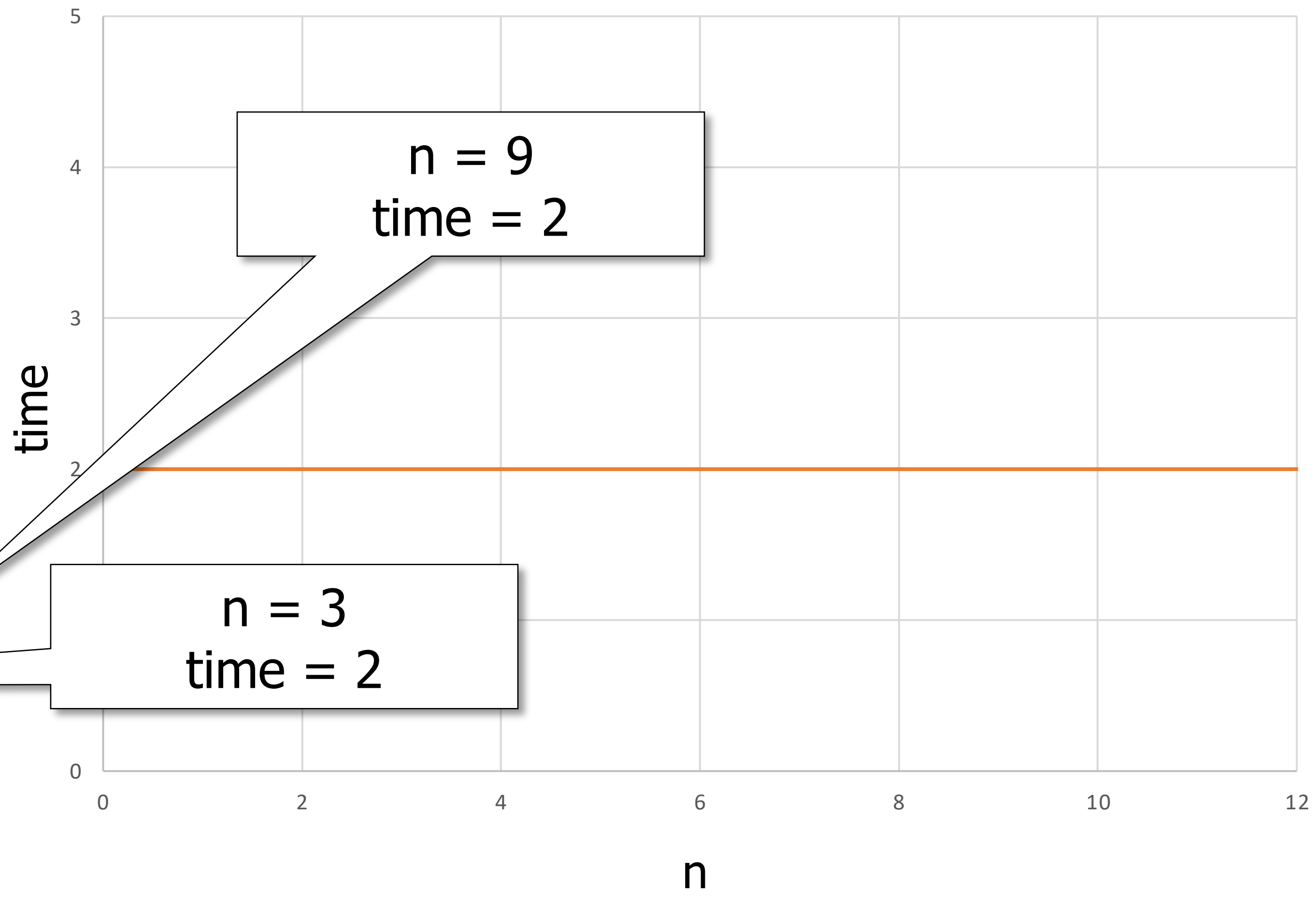


Constant: O(1)

- Constant complexity
  - n (the size of the data) does not matter
  - Always takes the same time
  - e.g. getting the first element in an array

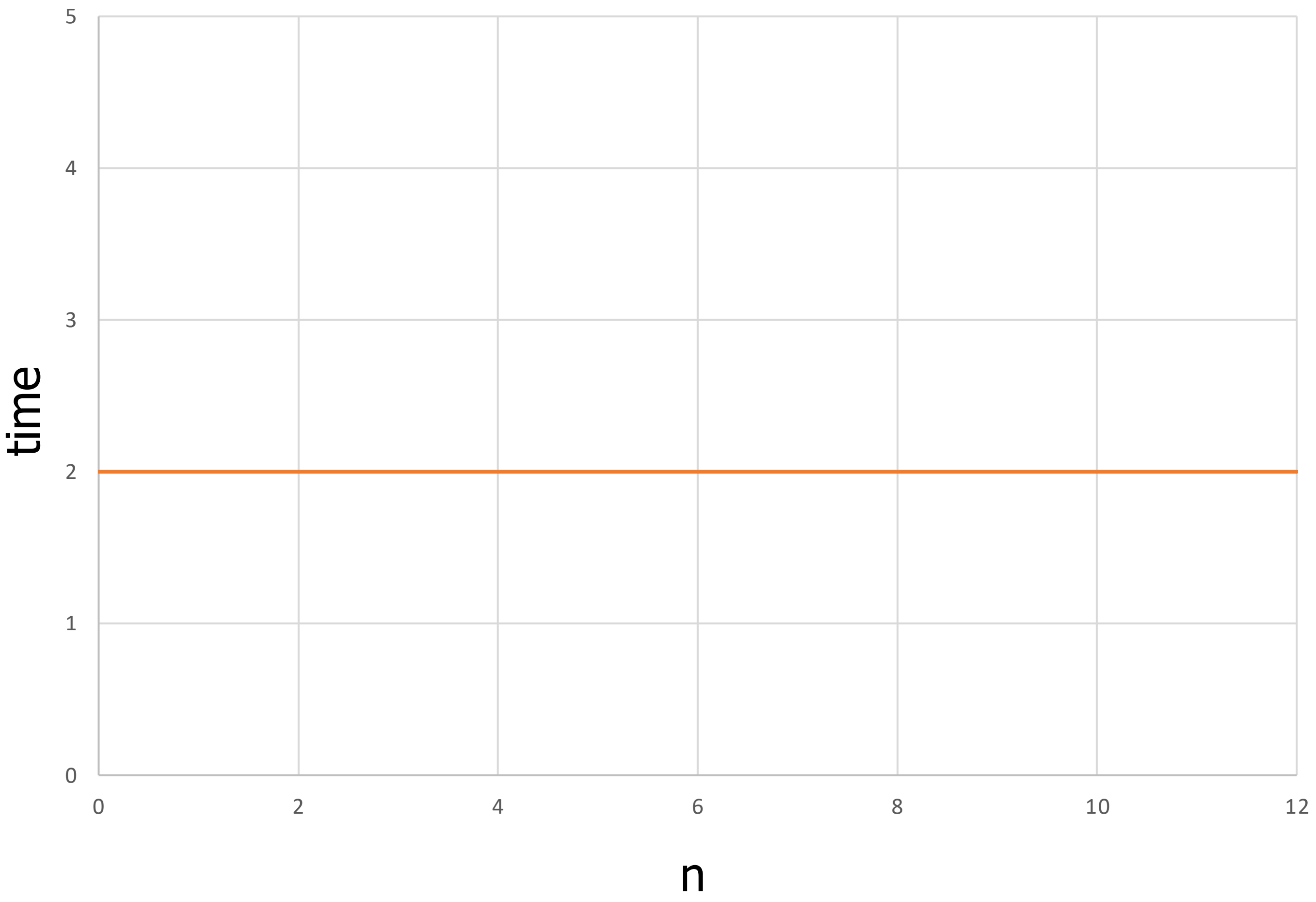
```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 42]
b = [ 10, 11, 13]


print(a[0])
print[b[0])
```



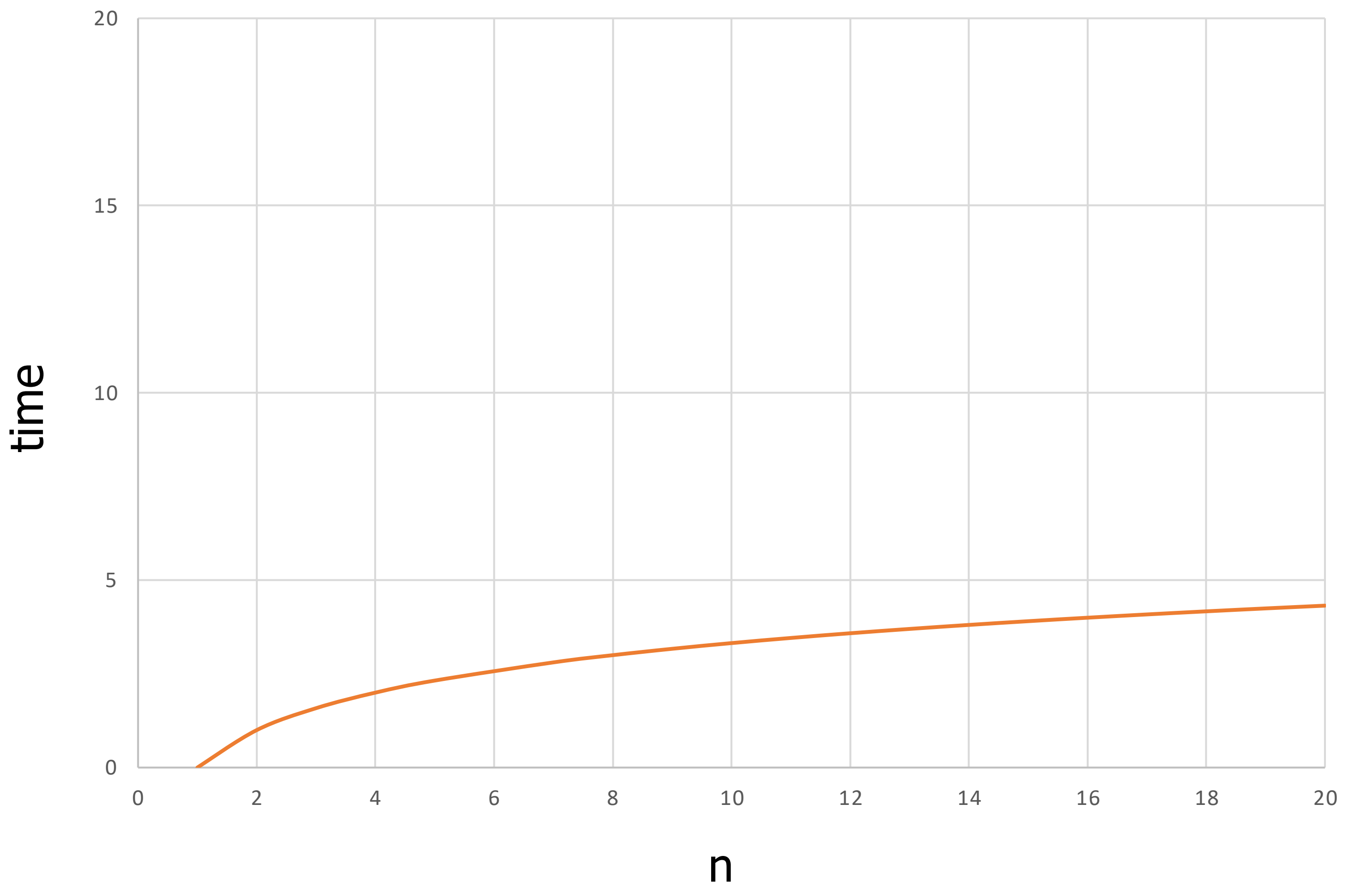
Constant complexity and Sorts

- Look back at the Week 2 Lecture on sort algorithms
- Can you find any algorithm of the 5 presented which has constant complexity O(1)?
- Why / why not?



Logarithmic: O(log n)

- Logarithmic complexity
- As n increases, the increase in time gets smaller



Example: Binary Search

- NB Binary search can be implemented in a range of slightly different ways
- This example retains or removes the middle value to preserve an even number when the sequence is split
- That will help us work through the maths
- So, sorted array of length 16:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	3	5	8	12	13	15	16	18	20	22	30	40	50	55	67

Example: Binary Search

- Target 12:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	3	5	8	12	13	15	16	18	20	22	30	40	50	55	67

- Middle = length / 2:

$$16 * \frac{1}{2} = 8$$

0	1	2	3	4	5	6	**7** *	8	9	10	11	12	13	14	15
1	3	5	8	12	13	15	**16**	18	20	22	30	40	50	55	67

* Because array indexes start at 0, our calculation result 8 is the value at index 7

Example: Binary Search

- Middle = length / 2:

0	1	2	**3**	4	5	6	7
1	3	5	**8**	12	13	15	16

$$8 * \frac{1}{2} = 4$$

- Target is right of the middle, so remove left half:

4	5	6	7
12	13	15	16

Example: Binary Search

- Middle = length / 2:

4	**5**	6	7
12	**13**	15	16

$$4 * \frac{1}{2} = 2$$

- Target is to the left of the middle, so remove right half:

4
12

Example: Binary Search

- Middle = length / 2:

4

12

$$2 * \frac{1}{2} = 1$$

- Target is the middle.

Example: Binary Search

- Middle = length / 2:

4

$\boxed{12}$

$$2 * \frac{1}{2} = 1$$

- To get to this result we had to divide the array 4 times
- Mathematically, this is:

$$16 * \left(\frac{1}{2}\right)^4 = 1$$

Example: Binary Search

- This formula $16 * \left(\frac{1}{2}\right)^4 = 1$ can be rewritten as: $n * \left(\frac{1}{2}\right)^k = 1$
- In the new formula, n means the length of the array, and k is the number of divisions to reach 1. So n and k are variables.
- We can change the formula like this:

$$n * \frac{1}{2^k} = 1$$

Example: Binary Search

- So if n is 16 and k is 4, n times 1 divided by 2 to the power of 4 equals 1.
- Let’s try it:

  16 * (1 / 2^4) = 1

$$n * \frac{1}{2^k} = 1$$

2 ^ 4 = 16

Example: Binary Search

- The formula can be rewritten as: $2^k * \dfrac{n}{2^k} = 2^k$
- Let's try it. Remember, n = 16; k = 4:

  16 * (16 / 16) = 16

Example: Binary Search

- The formula can be rewritten as: $2^k * \frac{n}{2^k} = 2^k$
- Let's try it. Remember, n = 16; k = 4:

  16 * (16 / 16) = 16
- This can be represented as:

  $n = 2^k$

Example: Binary Search

- The formula can be rewritten as: $2^k * \frac{n}{2^k} = 2^k$
- Let's try it. Remember, n = 16; k = 4:

  16 * (16 / 16) = 16
- This can be represented as:

  $n = 2^k$
- The definition of a logarithm is:

  A quantity representing the power to which a fixed number (the base) must be raised to produce a given number

Example: Binary Search

- The formula can be rewritten as: $2^k * \frac{n}{2^k} = 2^k$
- Let's try it. Remember, n = 16; k = 4:

  16 * (16 / 16) = 16
- This can be represented as:

  $n = 2^k$
- The definition of a logarithm is:

  A quantity representing the power to which a fixed number (the base) must be raised to produce a given number
- Which gives us:

  $\log_2 n = k$

Example: Binary Search

- We've reached this formula:

  $\log_2 n = k$

- In our example, n is 16 and k is 4
- The logarithm of 16 is 4 (k)
- Therefore, the complexity of Binary Search is:

  $\log n$

And sorts

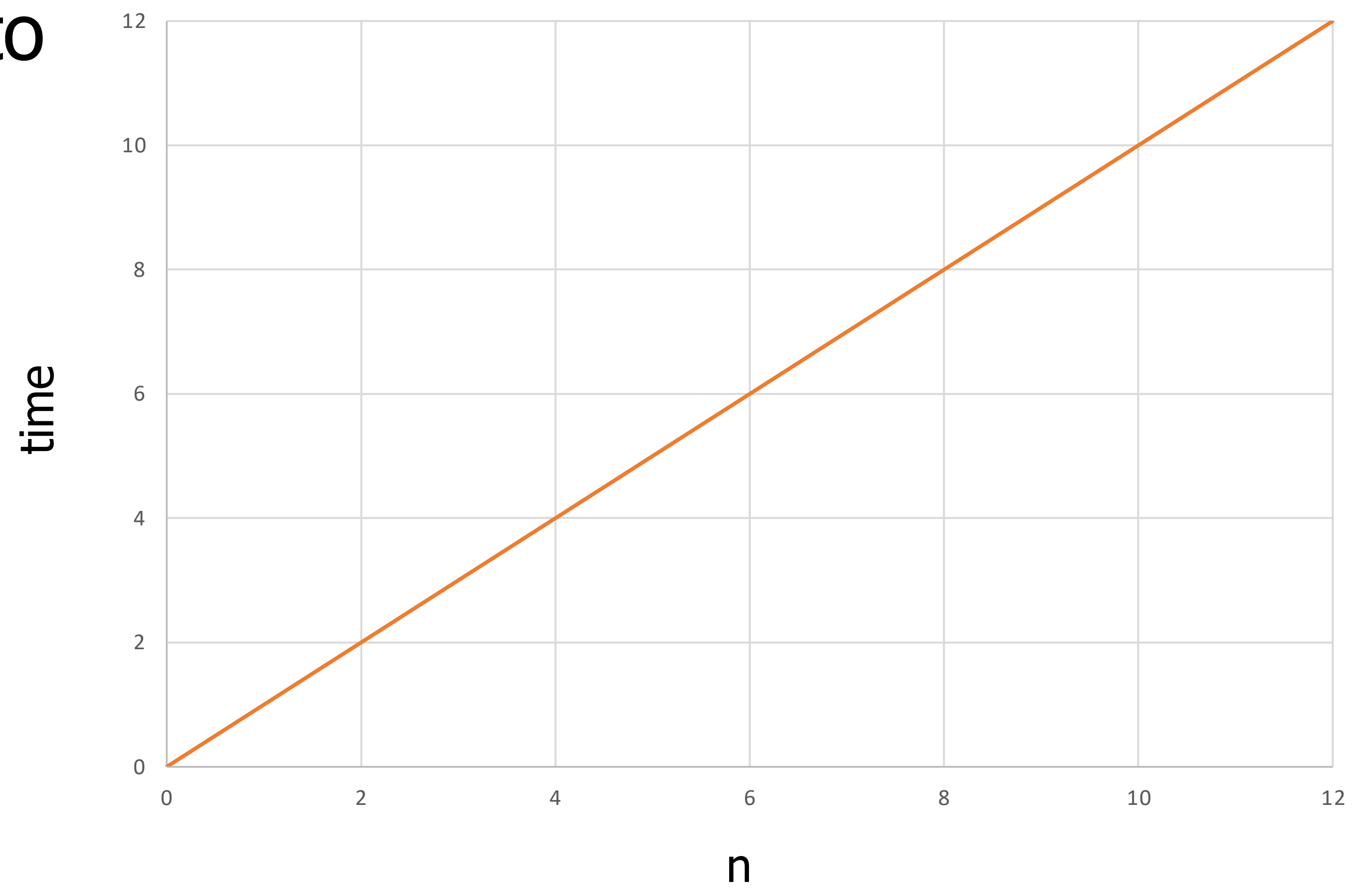
- Are any of the 5 sorts presented in Week 2 O(log n)?
- If so, why?
- If not, why not?

Linear Complexity O(n)

- n directly proportional to time
- If n doubles then time doubles
- E.g. linear/sequential search

Linear Complexity O(n)

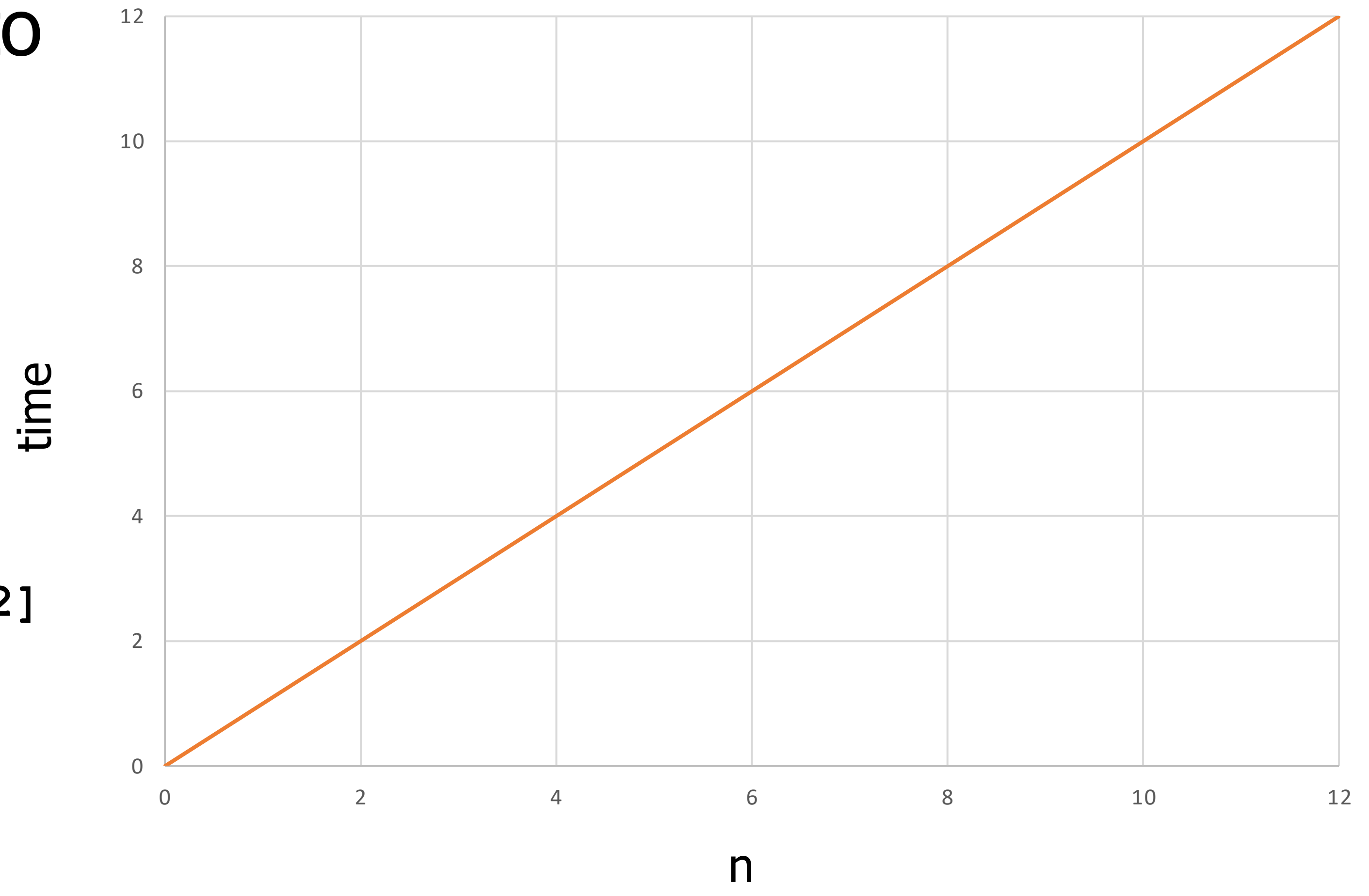
- n directly proportional to time
- If n doubles then time doubles
- E.g. linear/sequential search



Linear Complexity O(n)

- n directly proportional to time
- If n doubles then time doubles
- E.g. linear/sequential search

```
a = [ 0, 2, 3, 4, 5, 7, 8, 9, 42]
for i in a:
      if i == 42:
            print ('Found it')
            break
```

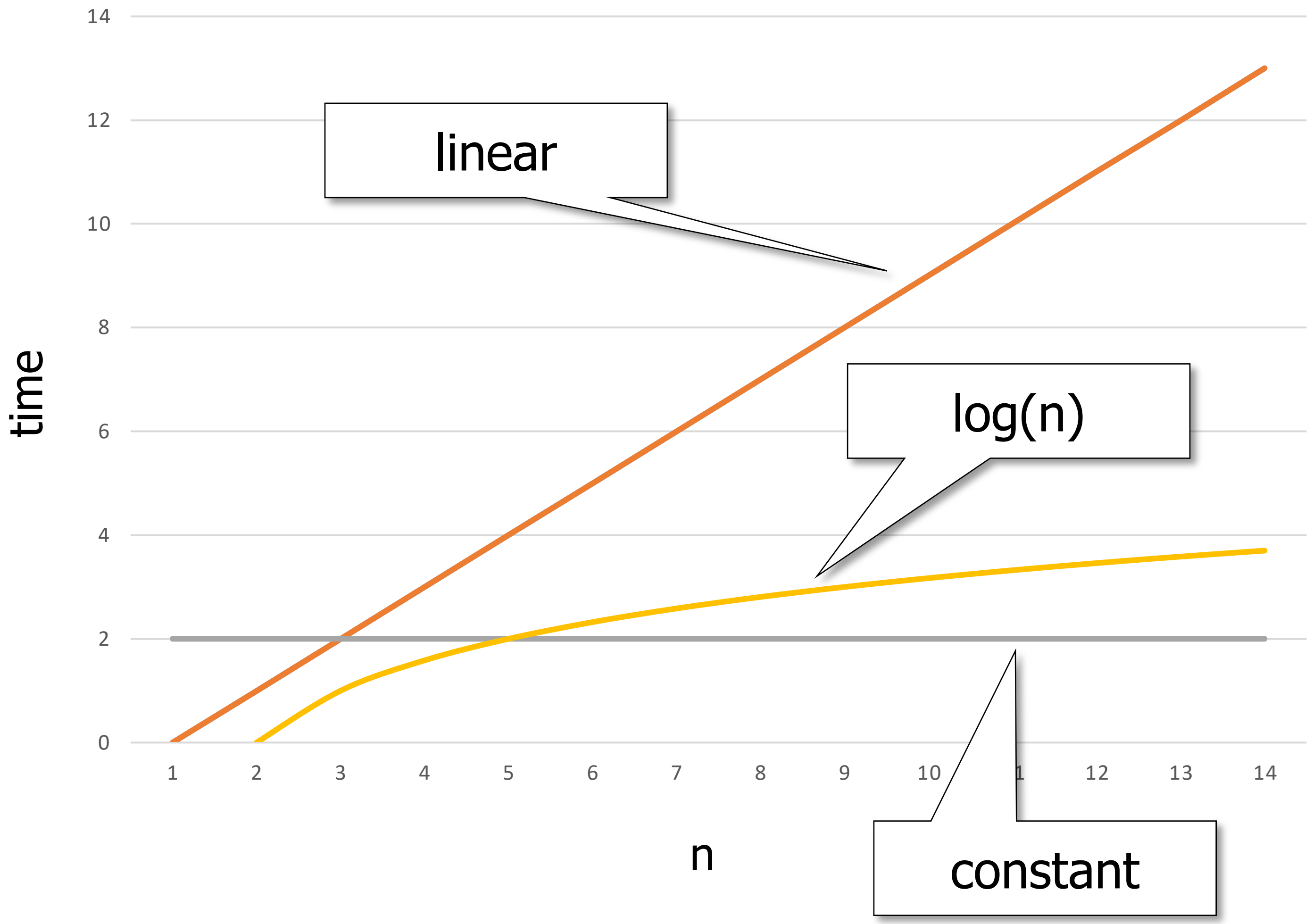


And sorts

- Are any of the 5 sorts presented in Week 2 linear, i.e. O(n)?
- If so, why?
- If not, why not?

Comparison

- So which is best?



Summary

- We looked at 3 types of complexity:
  - Constant complexity e.g. print a[0]
  - Logarithmic complexity e.g. binary search
  - Linear complexity e.g. linear search
- There will be two more Big O mini-lectures, in Weeks 4 and 5. These will be on:
  - Big O mini-lecture 2: log-linear and polynomial complexity
  - Big O mini-lecture 3: exponential and factorial complexity
- After that, we'll move onto code profiling