CITY
UNIVERSITY OF LONDON
EST 1894

STEM Digital Academy

School of Science & Technology

www.city.ac.uk

Algorithms II

Programming and Algorithms

Lecture by

Dr Daniil Osudin

```
n = 3
for i in range(1,n+1):
    print("Hello World!")
```

```
Hello World!
Hello World!
Hello World!
```

What will we Cover?

- More on the algorithm efficiency and big-O notation
- Examples of analysing algorithm complexity

Properties of big-O I

Scaling by a Constant

When analysing algorithm complexity, the constants can be removed, as their impact on overall runtime is negligible when comparing algorithm efficiency.

Examples

- $O(n/2+1) = O(n)$
- $O(2n^2 + n^2 + 3) = O(3n^2 + 3) = O(n^2)$

Properties of big-O II

Finding a Dominant Factor

Each subsequent order of complexity will overshadow the impact on algorithm efficiency of any factors of lower orders

Examples

- $O(n^2 + n) = O(n^2)$
- $O(\log n + n^4 + 2^n) = O(2^n)$

Properties of big-O III

Multiplication

When two orders of complexity depend on each other, then overall complexity is taken as both will impact the algorithms efficiency.

Examples

- $O(n * n) = O(n^2)$
- $O(n * \log n + 5n + 1) = O(n \log n + n) = O(n \log n)$

Constant Order of Complexity Example

```
Function foo(x):
y = 5x
y = y + 5
Return y
```

Executed only once

Complexity:
O(1 + 1) = O(1)

Linear Order of Complexity Example

```
Function foo2(list):
bar = 0
i = 1
while i <= len(list)
    bar = bar + 2*array[i]
    i = i + 1
Return bar
```

Executed only once

Executed once per element in the list

Complexity:
O(2 + 3n) = O(n),
if n = len(list)

Quadratic Order of Complexity Example

```
Function foo3(list):
sum = 0
c = 1
while c <= len(list)
    c1 = 1
    while c1 <= len(list)
        sum = sum + array[c] * array[c1]
        c1 = c1 + 1
        c = c + 1
Return sum
```

Executed only once

Executed once per element in the list

Executed once per element in the list

Complexity:
$O(2 + n(2 + 4n)) = O(2n + 4n*n) = O(n^2)$, if n = len(list)

Logarithmic Order of Complexity Example

```
Function foo4(n):
product = 1
c = n
while c > 0
    product = product * c
    c = c//2
Return product
```

Executed only once

Executed less than n times. After each iteration of the loop, the value of c is halved. This is logarithmic complexity

If n = 100
product = 100*50*25*12*6*3*1

Complexity:
$O(2 + 3 \log n) = O(\log n)$