ANSWERS

# A5.2 Big O notation

Algorithms and complexity · A level · OCR H446 2.3.1, AQA 7517 4.4.4.1, Eduqas A500QS 1.3 · about 25 min

## What this lesson is about

Dominant terms, the orders of complexity from O(1) to O(2^n), deriving complexity from code, best, average and worst case, and space complexity.

- Keep the dominant term, drop the constants
- The orders you need
- Deriving the complexity of code
- Exponential growth in code
- Best, average and worst case
- Space complexity

## Questions

**6 marks in all**

**1.** An algorithm makes $4n^2 + 30n + 500$ operations. What is its time complexity? [1 mark]

- ☐ **A** $O(n^2)$
- ☐ **B** $O(4n^2)$
- ☐ **C** $O(n^2 + n)$
- ☐ **D** $O(500)$

> **Answer:** A. Keep only the dominant term, $n^2$, and drop its constant factor.

**2.** Put these orders of complexity from slowest-growing to fastest-growing. [1 mark]

Number the lines 1 to 6 to put them in the right order.

___ `O(log n)`
___ `O(n log n)`
___ `O(n²)`
___ `O(n)`
___ `O(2ⁿ)`
___ `O(1)`

> **Answer:**
>
> ```
> O(1)
> O(log n)
> O(n)
> O(n log n)
> O(n²)
> O(2ⁿ)
> ```
>
> Constant, logarithmic, linear, linearithmic, polynomial, exponential.

**3.** A loop runs n times, and inside it a second loop also runs n times. What is the time complexity? [1 mark]

- ☐ **A** $O(n^2)$
- ☐ **B** O(2n)
- ☐ **C** O(n)
- ☐ **D** O(log n)

> **Answer:** A. Nested loops multiply: n × n.

**4.** A loop halves a variable that starts at n until it reaches 1. What is its time complexity? [1 mark]

- ☐ **A** O(log n)
- ☐ **B** O(n)
- ☐ **C** O(n/2)
- ☐ **D** O(1)

> **Answer:** A. The number of halvings from n down to 1 is about $\log_2 n$.

**5.** How many times does the marked line run? The program prints the count. [1 mark]

```
count = 0
n = 1000
while n > 1:
    n = n // 2
    count = count + 1    # the marked line
print(count)
```

> **Answer:**
>
> ```
> 9
> ```
>
> 1000 halves to 500, 250, 125, 62, 31, 15, 7, 3, 1: nine halvings, which is $\lfloor \log_2 1000 \rfloor$.

**6.** When a single Big O is quoted for an algorithm without saying which case, which case does it normally describe? [1 mark]

- ☐ **A** The worst case
- ☐ **B** The best case
- ☐ **C** The average case
- ☐ **D** The case with the smallest n

> **Answer:** A. The worst case is a guarantee: the algorithm is never slower than this.

## The task: count the steps

Each function in the starter has one line marked `# the step`. Change each function so that, instead of its answer, it **returns the number of times its step ran**. Then, for n = 8, 16 and 32, call each function on `list(range(n))` (a list of n different numbers; `halvings` takes the number n itself) and print one line per function in exactly this form, with the three counts and then the order of growth: - `first: <count for 8> <count for 16> <count for 32> O(1)` - `total: ... O(n)` - `has_duplicate: ... O(n^2)` - `halvings: ... O(log n)` Count the step in `first` as running once. Do not type the counts in: the program must work them out.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

def first(items):
    return items[0]                                # the step

def total(items):
    s = 0
    for x in items:
        s = s + x                                  # the step
    return s

def has_duplicate(items):
    for i in range(len(items)):
        for j in range(i + 1, len(items)):
            if items[i] == items[j]:               # the step
                return True
    return False

def halvings(n):
    count = 0
    while n > 1:
        n = n // 2                                 # the step
        count = count + 1
    return count
```

**The hint students can ask for:** Give each function a counter that starts at 0 and goes up by one every time the marked line runs, and return the counter at the end. has_duplicate never finds a repeat in these lists, so every pair is compared. Then loop over the three sizes for each function and join the counts into one line.

### A solution

```
from bugbot import *
connect()

def first(items):
    x = items[0]
    return 1

def total(items):
    s, count = 0, 0
    for x in items:
        s = s + x
        count = count + 1
    return count
```

```
def has_duplicate(items):
    count = 0
    for i in range(len(items)):
        for j in range(i + 1, len(items)):
            count = count + 1
            if items[i] == items[j]:
                return count
    return count

def halvings(n):
    count = 0
    while n > 1:
        n = n // 2
        count = count + 1
    return count

sizes = [8, 16, 32]
print("first:", *[first(list(range(n))) for n in sizes], "O(1)")
print("total:", *[total(list(range(n))) for n in sizes], "O(n)")
print("has_duplicate:", *[has_duplicate(list(range(n))) for n in sizes], "O(n^2)")
print("halvings:", *[halvings(n) for n in sizes], "O(log n)")
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.