ANSWERS

# A2.7 Thinking ahead and thinking logically

Recursion and computational thinking · A level · OCR H446 2.1.2, AQA 7517 4.4.1.1 · about 25 min

## What this lesson is about

Inputs, outputs and preconditions, caching and reuse, decisions and conditions, and solving logic problems.

- Inputs, outputs and preconditions
- Caching
- Reusable components
- Thinking logically
- Following and writing algorithms

## Questions

**6 marks in all**

**1.** What is a precondition? [1 mark]

- ☐ **A** Something that must be true before a subroutine or algorithm runs for it to work correctly
- ☐ **B** The first line of a program
- ☐ **C** A condition tested at the end of a loop
- ☐ **D** The output an algorithm must produce

> **Answer:** A. For example, binary search has the precondition that the list is sorted.

**2.** Which are drawbacks of caching? [1 mark]

Tick every answer that is true.

- ☐ **A** The cached copy can be out of date (stale)
- ☐ **B** It uses extra memory or storage
- ☐ **C** Deciding what to keep and when to discard it adds complexity
- ☐ **D** Repeated requests are always slower

> **Answer:** A, B, C. Repeated requests are the ones caching speeds up.

**3.** What does this program print? [1 mark]

```
cache = {}
calls = 0

def fib(n):
    global calls
    calls = calls + 1
    if n in cache:
        return cache[n]
    if n < 2:
        answer = n
    else:
        answer = fib(n - 1) + fib(n - 2)
    cache[n] = answer
    return answer

print(fib(10), calls)
```

**Answer:**

```
55 19
```

Each value from 0 to 10 is worked out once, and the second recursive call at each level is answered from the cache: 19 calls instead of 177.

**4.** Why use reusable program components? [1 mark]

Tick every answer that is true.

- ☐ **A** They save development time
- ☐ **B** Code that has been used and tested many times is more reliable
- ☐ **C** A fault is fixed in one place for every program that uses it
- ☐ **D** They are always faster than code written for one job

**Answer:** A, B, C. A general component can be slower or larger than code written for one task; that is a trade-off, not a benefit.

**5.** What does this program print? [1 mark]

```
for culprit in ["Ada", "Bolt", "Cog"]:
    ada = culprit == "Bolt"
    bolt = culprit != "Bolt"
    cog = culprit != "Cog"
    if [ada, bolt, cog].count(True) == 1:
        print(culprit)
```

**Answer:**

```
Cog
```

Only when Cog is the culprit is exactly one statement true (Bolt's), so the puzzle has one solution.

**6.** Which constructs are enough to write any algorithm? [1 mark]

Tick every answer that is true.

- ☐ **A** Sequence
- ☐ **B** Assignment
- ☐ **C** Selection
- ☐ **D** Iteration
- ☐ **E** Recursion

> **Answer:** A, B, C, D. Sequence, assignment, selection and iteration are the standard constructs. Recursion can always be replaced by iteration with a stack.

## The task: routes with a cache

The robot moves on a grid, only ever one cell right or one cell up. `routes(right_steps, up_steps)` is the number of different routes from one corner to a cell that many steps right and up. Both parameters are whole numbers. If either is 0 there is exactly 1 route (a straight line); otherwise every route arrives either from the cell to the left or from the cell below, so `routes(r, u) = routes(r - 1, u) + routes(r, u - 1)`. - Write `routes` recursively, with a dictionary `cache` so no pair is ever worked out twice. - Count every call of `routes` in a global `calls`, including calls answered from the cache. - **Precondition**: both parameters must be 0 or more. At the start of `routes`, if either is negative, `raise ValueError` with a message of your choice. Then print, in this order: 1. `routes(3, 2) = <answer>`; 2. `routes(16, 16) = <answer>`, with `calls` set back to 0 just before this call; 3. `calls: <n>`, the calls made by that one call (without a cache this would be over a billion); 4. `refused: <message>`, by calling `routes(-1, 4)` inside `try` and printing the error's message in `except ValueError`.

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

cache = {}
calls = 0

def routes(right_steps, up_steps):
    if right_steps == 0 or up_steps == 0:
        return 1
    return routes(right_steps - 1, up_steps) + routes(right_steps, up_steps - 1)

print("routes(3, 2) =", routes(3, 2))
```

**The hint students can ask for:** Before any calculation, look the pair up in the cache and hand back the stored answer if it is there. After calculating, store the answer before returning it. The precondition check belongs at the very top, before the cache is touched. Reset the call counter just before the big call so it only counts that one.

## A solution

```
from bugbot import *
connect()

cache = {}
calls = 0

def routes(right_steps, up_steps):
    global calls
    calls = calls + 1
    if right_steps < 0 or up_steps < 0:
        raise ValueError("steps cannot be negative")
    if (right_steps, up_steps) in cache:
        return cache[(right_steps, up_steps)]
    if right_steps == 0 or up_steps == 0:
        answer = 1
    else:
        answer = routes(right_steps - 1, up_steps) + routes(right_steps, up_steps - 1)
    cache[(right_steps, up_steps)] = answer
    return answer

print("routes(3, 2) =", routes(3, 2))
```

```
cache = {}
calls = 0
print("routes(16, 16) =", routes(16, 16))
print("calls:", calls)
try:
    routes(-1, 4)
except ValueError as error:
    print("refused:", error)
```

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