



A6.8 Computable problems and the Halting problem

Theory of computation · A level · AQA 7517 4.4.4.6 · about 20 min

What this lesson is about

Non-computable problems, the Halting problem and why it cannot be solved, and what step limits can and cannot tell you.

- Computable and non-computable
- The Halting problem
- Why no halting checker can exist
- What it means in practice

Questions

5 marks in all

1. What is the Halting problem?

[1 mark]

- ☐ A Whether a program has a syntax error
- ☐ B Whether a general program can decide, for any program and any input, whether it will halt, without running it
- ☐ C How to stop a program that has crashed
- ☐ D Whether a program finishes within a time limit

Answer: B. Turing proved no such general program can exist: the Halting problem is non-computable.

2. What is the significance of the Halting problem?

[1 mark]

- ☐ A It shows that some problems cannot be solved by any computer
- ☐ B It shows that all programs eventually halt
- ☐ C It shows that faster computers solve more problems
- ☐ D It shows that recursion is unsafe

Answer: A. It was the first proof that there are problems no algorithm can solve, however much time or speed is available.

3. Which statements are true?

[1 mark]

Tick every answer that is true.

- ☐ A A non-computable problem has no algorithm that solves every case
- ☐ B An intractable problem has an algorithm, but it is too slow for large inputs
- ☐ C A faster computer can turn a non-computable problem into a computable one
- ☐ D Running a program for a long time can prove that it halts, if it does

Answer: A, B, D. Speed never makes a non-computable problem computable. Running a program can show it halts, but never that it runs forever.

4. Put the steps of the Halting problem proof by contradiction in order.

[1 mark]

Number the lines 1 to 5 to put them in the right order.

- ___ Write contrary(p): if halts(p, p) is True, loop forever; otherwise stop
- ___ Whatever halts answers about contrary(contrary), contrary does the opposite
- ___ Suppose a function halts(p, d) always correctly says whether p halts on d
- ___ Run contrary with its own code as input
- ___ So halts cannot exist

Answer:

Suppose a function halts(p, d) always correctly says whether p halts on d
Write contrary(p): if halts(p, p) is True, loop forever; otherwise stop
Run contrary with its own code as input
Whatever halts answers about contrary(contrary), contrary does the opposite
So halts cannot exist

Assume the checker exists, build a program that does the opposite of its prediction, and feed it itself.

5. What does this program print?

[1 mark]

```
def watch(x, limit):  
    for steps in range(limit):  
        if x == 0:  
            return "halted after " + str(steps)  
        x = x - 3  
    return "no answer"  
  
print(watch(12, 100))  
print(watch(10, 100))
```

Answer:

halted after 4
no answer

From 12, x reaches 0 after 4 steps. From 10, x goes 7, 4, 1, -2, ... and never equals 0, so the limit runs out.

The task: the watchdog

Write a watchdog that runs simple programs with a step limit and reports honestly on each. - Each program in `programs` is a tuple `(name, x, step, done)`: `name` is a string, `x` is the starting whole number, `step(x)` returns the next value of `x`, and `done(x)` returns `True` when the program has halted. - Write `watch(name, x, step, done, limit)`. Before each step, check `done(x)`. If it is `True`, print `<name>: halted after <n> steps`, where `n` is the number of steps taken so far, and stop watching. Otherwise take a step. If `limit` steps have been taken and `done(x)` is still not `True`, print `<name>: no answer after <limit> steps`. - Call `watch` for each program in order, with `limit` set to 1000. Three lines in all, such as `p1: halted after 3 steps`. Work the counts out by running the programs, not by hand.

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

programs = [
    ("p1", 10, lambda x: x - 3, lambda x: x == 1),
    ("p2", 10, lambda x: x - 4, lambda x: x == 0),
    ("p3", 27, lambda x: x // 2 if x % 2 == 0 else 3 * x + 1, lambda x: x == 1),
]

def watch(name, x, step, done, limit):
    pass
```

The hint students can ask for: Keep a count of steps taken. Loop while the count is below the limit: if the program is done, report the count and return straight away; otherwise take a step and add one. If the loop ends without returning, the limit was reached. Remember to check `done` once more after the last step.

A solution

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

programs = [
    ("p1", 10, lambda x: x - 3, lambda x: x == 1),
    ("p2", 10, lambda x: x - 4, lambda x: x == 0),
    ("p3", 27, lambda x: x // 2 if x % 2 == 0 else 3 * x + 1, lambda x: x == 1),
]

def watch(name, x, step, done, limit):
    steps = 0
    while steps <= limit:
        if done(x):
            print(f"{name}: halted after {steps} steps")
            return
        if steps == limit:
            break
        x = step(x)
        steps = steps + 1
    print(f"{name}: no answer after {limit} steps")

for name, x, step, done in programs:
    watch(name, x, step, done, 1000)
```

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