WORKSHEET

A6.8 Computable problems and the Halting problem

BugBotLab

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

Name _____ Class _____ Date _____

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

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

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

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`

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))
```

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
```

Plan your program here, then type it in and press Run.

Do it on the robot

www.bugbotlab.com/learn/a6-8-computability-and-halting/

The simulator checks it and tells you when it passes. Nothing to install, no account.

Challenges

1. Program p2 never halts. Prove it by thinking about the values `x` takes, not by running it.
2. A program whose `x` can only take a few different values must either halt or repeat a value. Add a check to `watch` that prints `loops forever` when a value repeats. Why does this not solve the Halting problem?
3. Explain the difference between "intractable" and "non-computable" to someone who thinks a faster computer fixes both.