ANSWERS

A6.7 Limits of computation: tractable and intractable problems

Theory of computation · A level · OCR H446 2.3.1, AQA 7517 4.4.4.4 · about 25 min

What this lesson is about

How complexity and hardware limit computation, tractable and intractable problems, and a heuristic route for the robot.

- Two things limit computation
- How fast things grow
- Tractable and intractable
- The travelling robot
- Heuristics

Questions

6 marks in all

1. What makes a problem intractable? [1 mark]

- ☐ **A** It cannot be solved by any algorithm
- ☐ **B** It has no known polynomial-time solution
- ☐ **C** It needs more than one processor
- ☐ **D** Its algorithm uses recursion

> **Answer:** B. An intractable problem can be solved, but only in exponential or worse time. A problem no algorithm can solve is non-computable.

2. Which time complexities count as tractable? [1 mark]

Tick every answer that is true.

- ☐ **A** $O(n^2)$
- ☐ **B** $O(2^n)$
- ☐ **C** $O(n \log n)$
- ☐ **D** $O(n!)$
- ☐ **E** $O(1)$

> **Answer:** A, C, E. Polynomial time or better is tractable. $O(2^n)$ and $O(n!)$ grow faster than any polynomial.

3. What is a heuristic? [1 mark]

- ☐ **A** An algorithm guaranteed to find the best answer
- ☐ **B** A rule of thumb that finds a good enough answer in reasonable time, without a guarantee it is the best
- ☐ **C** A way of proving a problem is non-computable
- ☐ **D** A faster processor

> **Answer:** B. Heuristics trade the guarantee of the best answer for speed, which is why they are used on intractable problems.

4. A brute-force route planner tries every order of visiting n places, n! orders. How many orders does it try for 6 places? [1 mark]

> **Answer:** 720. 6! = 6 x 5 x 4 x 3 x 2 x 1 = 720.

5. An $O(2^n)$ algorithm can handle n = 40 in an hour. A new computer is twice as fast. What is the largest n it can handle in an hour? [1 mark]

- ☐ **A** 41
- ☐ **B** 80
- ☐ **C** 60
- ☐ **D** 1600

> **Answer:** A. 2^{41} is only twice 2^{40}, so doubling the speed adds one to n. Hardware barely helps an exponential algorithm.

6. What does this program print? [1 mark]

```
import math
for n in [4, 8, 12]:
    print(n, n ** 2, 2 ** n, math.factorial(n))
```

> **Answer:**
>
> ```
> 4 16 16 24
> 8 64 256 40320
> 12 144 4096 479001600
> ```
>
> n^2 grows slowly, 2^n doubles with each step, and n! grows faster still.

The task: the delivery route

Plan and drive a route to all five drop points, using the heuristic, and compare it with the best route. - The drop points are in `drops`: a dictionary from a letter to an `(x, y)` position on the mat in cm. The robot starts at `START`. Distances are straight lines: use `math.dist`. - `drive_to(x, y)` is written for you: it slides the robot to that mat position. - Write `tour_length(order)`: `order` is a sequence of letters; return the total distance from `START` through each point in that order, without coming back. - Write `nearest_neighbour()`: return a list of the five letters in the order the nearest neighbour heuristic visits them, starting from `START`. - Print `nearest neighbour:` then the letters separated by single spaces, a space, then the length to 1 decimal place and `cm`. For example `nearest neighbour: A B C D E 123.4 cm`. - Find the shortest order by brute force over every order (`itertools.permutations`) and print it the same way, starting `best:`. - Then drive the **nearest neighbour** route, visiting the points in that order. Do not type any orders or lengths in; work them out.

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

import math
from itertools import permutations

START = (50, 10)
drops = {"A": (80, 45), "B": (65, 70), "C": (85, 85), "D": (20, 20), "E": (55, 35)}

def wrapped(h):
    return (h + 180) % 360 - 180

def drive_to(x, y, speed=70):
    # slide to the mat point (x, y), holding heading 0, until within 3 cm
    while True:
        px, py = position()
        dx, dy = x - START[0] - px, y - START[1] - py
        if math.hypot(dx, dy) < 3:
            break
        a = math.radians(wrapped(math.degrees(math.atan2(dx, dy)) - heading()))
        drive(speed * math.cos(a), speed * math.sin(a), wrapped(0 - heading()) * 3)
        wait(0.1)
    stop()
```

The hint students can ask for: For the heuristic, keep a list of the points not yet visited and where you are now; each step, pick the unvisited point closest to where you are, move there and cross it off. For brute force, find the order whose tour length is smallest. Only drive once both have been printed.

A solution

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

import math
from itertools import permutations

START = (50, 10)
drops = {"A": (80, 45), "B": (65, 70), "C": (85, 85), "D": (20, 20), "E": (55, 35)}
```

```
def wrapped(h):
    return (h + 180) % 360 - 180

def drive_to(x, y, speed=70):
    # slide to the mat point (x, y), holding heading 0, until within 3 cm
    while True:
        px, py = position()
        dx, dy = x - START[0] - px, y - START[1] - py
        if math.hypot(dx, dy) < 3:
            break
        a = math.radians(wrapped(math.degrees(math.atan2(dx, dy)) - heading()))
        drive(speed * math.cos(a), speed * math.sin(a), wrapped(0 - heading()) * 3)
        wait(0.1)
    stop()

def tour_length(order):
    total = 0
    here = START
    for name in order:
        total = total + math.dist(here, drops[name])
        here = drops[name]
    return total

def nearest_neighbour():
    order = []
    here = START
    left = sorted(drops)
    while left:
        nearest = left[0]
        for name in left:
            if math.dist(here, drops[name]) < math.dist(here, drops[nearest]):
                nearest = name
        order.append(nearest)
        left.remove(nearest)
        here = drops[nearest]
    return order

route = nearest_neighbour()
print("nearest neighbour:", " ".join(route), f"{tour_length(route):.1f} cm")
best = min(permutations(drops), key=tour_length)
print("best:", " ".join(best), f"{tour_length(best):.1f} cm")
for name in route:
    drive_to(*drops[name])
```

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