ANSWERS

A15.5 Writing algorithms and code in the exam

Exam preparation · A level · OCR H446 2.2.1, AQA 7517 4.4.1.2, Eduqas A500QS 1.8 · about 50 min

What this lesson is about

Planning before writing, how code is marked, the standard algorithms to know by heart, and Dijkstra's algorithm driving the robot.

- How code is marked
- A method that works
- Algorithms to know by heart
- Dijkstra's algorithm, written to a plan

Questions

5 marks in all

1. Your code for a 5-mark question has a correct loop and comparison, but the final average is wrong. What is most likely? [1 mark]

- ☐ **A** You earn the marks for the features that are correct
- ☐ **B** You score 0 because it does not work
- ☐ **C** You lose a mark for every line
- ☐ **D** Only the return value is marked

> **Answer:** A. Code is marked by the features of a correct answer, so partial solutions earn partial marks.

2. What is the time complexity of merge sort? [1 mark]

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

> **Answer:** A. Merge sort halves the list log n times and does linear work to merge at each level.

3. Which algorithms require their input to meet a condition first? [1 mark]

Tick every answer that is true.

- ☐ **A** Binary search: the list must be sorted
- ☐ **B** Dijkstra's algorithm: no negative edge weights
- ☐ **C** Linear search: the list must be sorted
- ☐ **D** Bubble sort: the list must be sorted

> **Answer:** A, B. Linear search and bubble sort work on any list.

4. This is the start of Dijkstra's algorithm on a small graph. What does it print? [1 mark]

```
graph = {"A": {"B": 4, "C": 1}, "B": {"D": 1}, "C": {"B": 2, "D": 6}, "D": {}}
dist = {n: float("inf") for n in graph}
dist["A"] = 0
unvisited = set(graph)
while unvisited:
    current = min(sorted(unvisited), key=lambda n: dist[n])
    unvisited.remove(current)
    for nb, w in graph[current].items():
        dist[nb] = min(dist[nb], dist[current] + w)
print(dist)
```

Answer:

```
{'A': 0, 'B': 3, 'C': 1, 'D': 4}
```

C is reached for 1, which gives B 1 + 2 = 3 instead of 4, and D then costs 3 + 1 = 4 rather than 1 + 6.

5. What does A* search add to Dijkstra's algorithm? [1 mark]

- ☐ **A** A heuristic estimate of the distance still to go, to guide which node is visited next
- ☐ **B** Negative edge weights
- ☐ **C** A stack instead of a priority queue
- ☐ **D** A limit of one path per node

Answer: A. A* orders nodes by the cost so far plus the heuristic, so it usually visits fewer nodes.

The task: shortest route, then drive it

The robot stands at node A. `coords` gives each node's position as `(x, y)` in cm from A, where positive x is to the robot's right and positive y is straight ahead. `graph` is an adjacency list: `graph[node]` is a dictionary from each neighbour to the whole-number weight of the corridor. 1. Write `dijkstra(graph, start, goal)`, which returns a tuple `(path, cost)`: `path` is the list of node names from `start` to `goal` along the cheapest route, and `cost` is its total weight. 2. Call it for A to F and print `path: <nodes>`, with the nodes separated by single spaces, then `cost: <cost>`. 3. Drive the path, leg by leg. For each pair of consecutive nodes, work out `dx` and `dy` from `coords`. Move right `dx` cm if it is positive (left if negative), then forward `dy` cm if it is positive (backward if negative), at speed 50. 4. When the robot reaches the goal, turn the LED green. Keep off the carpet between A and B.

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

coords = {"A": (0, 0), "B": (0, 40), "C": (30, 0), "D": (30, 40), "E": (0, 70), "F": (30,
70)}
graph = {
    "A": {"B": 80, "C": 30},
    "B": {"A": 80, "D": 30, "E": 30},
    "C": {"A": 30, "D": 40},
    "D": {"B": 30, "C": 40, "F": 30},
    "E": {"B": 30, "F": 30},
    "F": {"D": 30, "E": 30},
}

def dijkstra(graph, start, goal):
    pass
```

The hint students can ask for: Keep a distance for every node, starting at infinity except the start, and a record of which node each best distance came from. Repeatedly take the unvisited node with the smallest distance and try to improve its neighbours. Rebuild the path by following the records back from the goal. Each leg's move is the difference between two nodes' coordinates.

A solution

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

coords = {"A": (0, 0), "B": (0, 40), "C": (30, 0), "D": (30, 40), "E": (0, 70), "F": (30,
70)}
graph = {
    "A": {"B": 80, "C": 30},
    "B": {"A": 80, "D": 30, "E": 30},
    "C": {"A": 30, "D": 40},
    "D": {"B": 30, "C": 40, "F": 30},
    "E": {"B": 30, "F": 30},
    "F": {"D": 30, "E": 30},
}

def dijkstra(graph, start, goal):
    dist = {node: float("inf") for node in graph}
    previous = {}
    dist[start] = 0
```

```
    unvisited = set(graph)
    while unvisited:
        current = min(unvisited, key=lambda n: dist[n])
        unvisited.remove(current)
        if current == goal:
            break
        for neighbour, weight in graph[current].items():
            if dist[current] + weight < dist[neighbour]:
                dist[neighbour] = dist[current] + weight
                previous[neighbour] = current
    path = [goal]
    while path[-1] != start:
        path.append(previous[path[-1]])
    path.reverse()
    return path, dist[goal]

path, cost = dijkstra(graph, "A", "F")
print("path:", " ".join(path))
print("cost:", cost)
for here, there in zip(path, path[1:]):
    dx = coords[there][0] - coords[here][0]
    dy = coords[there][1] - coords[here][1]
    if dx > 0:
        right(50, distance=dx)
    elif dx < 0:
        left(50, distance=-dx)
    if dy > 0:
        forward(50, distance=dy)
    elif dy < 0:
        backward(50, distance=-dy)
led("green")
```

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