Version POK/3

EGT3
ENGINEERING TRIPOS PART IIB

Monday 5 May 2025 14.00 to 15.40

Module 4M26

ALGORITHMS AND DATA STRUCTURES

*Answer not more than **three** questions.*

All questions carry the same number of marks.

*The **approximate** percentage of marks allocated to each part of a question is indicated in the right margin.*

*Write your candidate number **not** your name on the cover sheet.*

STATIONERY REQUIREMENTS
Single-sided script paper

SPECIAL REQUIREMENTS TO BE SUPPLIED FOR THIS EXAM
CUED approved calculator allowed
Engineering Data Book

10 minutes reading time is allowed for this paper at the start of the exam.

You may not start to read the questions printed on the subsequent pages of this question paper until instructed to do so.

You may not remove any stationery from the Examination Room.

1 You are developing the logic of a video game in which the player navigates a cave during a cave-in with the following requirements:

The cave is a `width` × `height` grid of tiles that are either empty or rubble.

- The cave has a start tile and a goal tile.
- The player moves one tile up, down, left, or right per step.
- The player enters on the start tile, and must reach the goal tile.
- The player cannot occupy a rubble tile.
- There are one or more rocks falling from the ceiling.
- Each rock is a square of `size` with a top-left `position` and an `impact` step.
- When a rock impacts, it creates one or more rubble tiles.
- The player sees falling rocks when there are just enough steps to avoid them.

Below is an example cave, four time steps, and then a partial listing for the game. `P` is the player, `S`/`G` are the start and goal tiles, `O` is a falling rock, and `#` is rubble.

```
                                          0      1     2     3
Size: 5x5                                ..G..  ..G.. ..G.. ..G..
Entrance: (2, 4)                         .....  ..... .OO.. .##..
Exit: (2, 0)                             .....  ..... .OP.. .##P.
Rocks: [Rock(x=1, y=1, size=2, impact=3)]  .....  ..P.. ..... .....
                                         ..P..  ..S.. ..S.. ..S..
```

```
class Vec2(NamedTuple("Vec2", [("x", int), ("y", int)])):
    # standard vector operations, including length(), __add__, __sub__, __mul__

class Rock(NamedTuple("Rock", [("pos", Vec2),("size", int),("impact_step", int)])):
    def rubble(self):
      br = self.pos + Vec2(self.size, self.size)
      return set([Vec2(x, y)
        for x in range(self.pos.x, br.x)
        for y in range(self.pos.y, br.y)
      ])

class Cave(NamedTuple("Cave", [("index", int), ("start", Vec2), ("goal", Vec2),
                               ("width", int), ("height", int), ("rocks", List[Rock]),
                               ("obstacles", Set[Vec2]), ("falling", Set[Vec2]),
                               ("step", int)])):
    def at_step(self, step: int) -> Cave:
        # incomplete
    def shortest_path(self, player: Vec2) -> Optional[List[Vec2]]:
        # incomplete
    def game_state(self, player: Vec2) -> GameState:
        # incomplete

def reconstruct_path(came_from, current):
    # ...
```

 (cont.

```
def astar(distance, heuristic, neighbors, is_goal, start):
    # incomplete
```

(a) Write an implementation for the `at_step` method, which returns the state of the game at the specified step. `obstacles` is a set of all tiles where a rock has fallen, and `falling` is a set of tiles that the player can see falling. [30%]

(b) Write the Python implementation for `astar` (assume `reconstruct_path` works as you need it to). For reference, the A* algorithm is as follows: [30%]

function ASTAR(distance, heuristic, neighbours, is_goal, start)
    O ← {start}
    C ← {}
    G ← {start: 0}
    F ← {start: heuristic(s)}
    while O ≠ ∅ **do**
        x ← arg $\min_n$ F[n]
        if is_goal(x) **then**
            return reconstruct_path(x, C)
        O ← O - {x}
        for y ∈ neighbours(x) **do**
            g ← G[x] + distance(x, y)
            if g < G[y] **then**
                C[y] ← x
                G[y] ← g
                F[y] ← g + heuristic(y)
                O ← O ∪ {y}
    return failure

(c) Implement the `shortest_path` method, which returns the shortest path from the player's current position to the goal. [15%]

(d) The game ends when the player is trapped, crushed, or when they reach the goal. Write an implementation for the `game_state` method which returns `GameState.PLAYING`, `GameState.WIN`, or `GameState.LOSE`. Note that a player is trapped if there is no path to the goal at the current step from their location. [5%]

(e) Rewrite `game_state` such that it analyses future rubble positions in order to return `GameState.Lose` when it is no longer possible for the player to reach the goal. [20%]

2 You are writing a rigid physics simulation of bouncing balls in two dimensions. The simulation uses *a priori* collision detection, which means that every collision between the balls and the walls will be calculated ahead of time so that nothing is missed. It has the following properties:

The simulation is constrained by four axis-aligned walls.

- The balls are perfectly elastic and frictionless.
- The balls are masses with a central position, radius, and constant velocity,

The collision time (t) and elastic collision equations for two balls are:

$$\mathbf{x}(t) = \mathbf{p} + \mathbf{v}t \qquad \mathbf{n} = \mathbf{p}_b - \mathbf{p}_a \qquad \mathbf{v}_{ab} = \mathbf{v}_b - \mathbf{v}_a$$

$$\Delta_p = \mathbf{p}_b - \mathbf{p}_a \qquad j = \frac{-2\mathbf{v}_{ab} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}\left(\frac{1}{m_a} + \frac{1}{m_b}\right)}$$

$$\Delta_v = \mathbf{v}_b - \mathbf{v}_a$$

$$r_a^2 + r_b^2 = \Delta_v \cdot \Delta_v t^2 + 2\Delta_p \cdot \Delta_v t + \Delta_p \cdot \Delta_p \qquad \mathbf{v}'_a = \mathbf{v}_a - \frac{j\mathbf{n}}{m_a} \qquad \mathbf{v}'_b = \mathbf{v}_b + \frac{j\mathbf{n}}{m_b}$$

Here is a class listing for the objects in this simulation:

```
class Vec2(NamedTuple("Vec2", [("x", Decimal), ("y", Decimal)])):
    # standard vector maths operations, also dot(v), project(v),
    # normalize(), length(), is_zero()

def elastic_collision(pos0: Vec2, vel0: Vec2, inv_mass0: Decimal,
                      pos1: Vec2, vel1: Vec2, inv_mass1: Decimal) -> Tuple[Vec2, Vec2]:
    # incomplete

class Ball(NamedTuple("Ball", [("pos", Vec2), ("vel", Vec2),
                               ("radius", Decimal), ("inv_mass", Decimal)])):
  def next_collision(self, other: "Ball") -> Decimal:
    # incomplete
  def collide_with(self, other: "Ball") -> Tuple["Ball", "Ball"]:
    # incomplete

class Wall(NamedTuple("Wall", [("horizontal", bool), ("value", Decimal)])):
  def next_collision(self, other: Ball) -> Decimal:
    # incomplete
  def collide_with(self, ball: Ball) -> Ball:
    # incomplete
```

Note the use of the `Decimal` type, which is a more precise representation of floating point numbers than `float`. As such, you do not need to account for errors introduced by floating point arithmetic.

(a) (i) Implement the `elastic_collision` function, which returns the updated velocities of the two rigid bodies provided in order. [15%]

(ii) Implement the `Wall.collide_with` method, which returns a ball with the updated velocity. Note that a wall has infinite mass. [5%]

(iii) Implement the `Ball.collide_with` method, which returns two updated balls for the self and other, respectively. [10%]

(iv) Implement the `Wall.next_collision` method, which returns the duration of time before the next collision. If the ball and wall will never collide, return `INF`. [10%]

(v) Implement the `Ball.next_collision` method, which returns the duration of time before the next collision. If the balls will never collide, return `INF`. [20%]

(b) A skeleton of the `Simulation` class is provided below:

```
Collision = NamedTuple("Collision", [("time", Decimal), ("i", int), ("j", int)])

class Simulation:
  def __init__(self, walls: List[Wall], balls: List[Ball]):
    self.walls = walls
    self.balls = balls
    self.num_balls = len(balls)
    self.num_walls = len(walls)
    self.time = dec(0)
    self.initialise_tocs()

  def initialise_tocs(self):
    # incomplete

  def update_tocs(self, i: int):
    # incomplete

  def next_ball_collision(self) -> Collision:
    # incomplete

  def next_wall_collision(self) -> Collision:
    # incomplete
```

For the `Collision` type, `time` is the absolute time of the next collision, `i` is the index of a ball, and `j` is the index of the ball or wall it collides with, as appropriate. The methods below should be implemented in such a way as to ensure that no collisions are missed while minimising the number of calculations performed.

(i) Implement the `initialise_tocs` method, which sets up any data structures you need. You can also implement any global helper functions you wish to use. [15%]

(ii) Implement the `update_tocs` method, which updates your data structures for the specified ball `i`. [15%]

(iii) Implement the `next_ball_collision` method, which returns the next collision between two balls. [5%]

(iv) Implement the `next_wall_collision` method, which returns the next collision between a ball and a wall. [5%]

3 The pseudocode below shows an algorithm for sorting where A, B and C are arrays. The input is an array $A[1..n]$ with $length[A] = n$. Each element in A is an integer in the range 0 to k. $B[1..n]$ and $C[0..k]$ are arrays. Note that C starts at index zero.

```
SORT(A, B, k)
1    for i ← 0 to k
2        do C[i] ← 0
3    for j ← 1 to length[A]
4        do C[A[j]] ← C[A[j]] + 1
5    for i ← 1 to k
6        do C[i] ← C[i] + C[i − 1]
7    for j ← length[A] downto 1
8        do B[C[A[j]]] ← A[j]
9           C[A[j]] ← C[A[j]] − 1
```

(a) Describe how the algorithm sorts $A = [2, 5, 3, 0, 2, 3, 0, 3]$. [20%]

(b) State the time complexity of the algorithm in Θ-notation. Explain your reasoning. [20%]

(c) Is the sort stable? Explain your reasoning. [20%]

(d) Suppose we change line 7 to read:

for $j \leftarrow 1$ **to** $length[A]$

Does the algorithm still work correctly? Explain your reasoning. [20%]

(e) Describe an algorithm that takes as input n integers in the range 0 to k and outputs how many of the n integers fall within a range $[a, b]$. The algorithm should run in $O(1)$ time, requiring $\Theta(n + k)$ preprocessing time. [20%]

4 Let the array $T[1..n]$ represent text and the array $P[1..m]$ of length $m \leq n$ represent a pattern. Assume the elements of T and P are characters drawn from a finite alphabet Σ. A pattern P exists with a shift s in text T if $0 \leq s \leq n - m$ and $T[s + 1..s + m] = P[1..m]$. A shift s is valid iff P exists in T. The pseudocode below shows an algorithm for this task.

```
PROCESS-PATTERN(T, P)
1    n ← length[T]
2    m ← length[P]
3    for i ← 0 to n − m
4        do if P[1..m] = T[i + 1..i + m]
5            then print i
```

(a) Provide a step-by-step description of the algorithm when $P =$ `aab` and $T =$ `acaabc`. What is the purpose of i? [20%]

(b) State the time complexity of the algorithm in both O and Θ notation. Explain your reasoning by providing an example. [20%]

(c) Explain why the algorithm is not as efficient as it could be by providing an example. [20%]

(d) Suppose all characters in P are different. Describe how to modify the algorithm to run in $O(n)$ time on an n-character T. [20%]

(e) Suppose we allow P to contain a special gap character that can match an arbitrary string of characters, even a string with zero length. The gap character may occur any number of times in P but it may not occur in T. Propose a polynomial-time algorithmic approach for determining whether such a pattern P exists in T and state the time and space complexity. [20%]

END OF PAPER

THIS PAGE IS BLANK