Crib for 4M26 Algorithms and Data Structures 2024/2025

Version: POK/3

Question 1

a)

Correctness: obstacles, logic for showing falling, filtered set for falling.
Style: use of comprehensions.

```
def at_step(self, step):
  obstacles = set().union(*[r.rubble() for r in self.rocks
                            if r.impact_step <= step])

  def is_falling(r: Rock) -> bool:
    diff = r.impact_step - step
    return diff > 0 and diff <= r.size // 2 + r.size % 2

  falling = set().union(*[r.rubble() for r in self.rocks
                            if is_falling(r)])
  return self._replace(obstacles=obstacles, falling=falling, step=step)
```

[30%]

b)

Correctness: use of arguments, correctly setting $G[]$ and $F[]$, updating `came_from`
Style: using a minheap for the frontier

```
def astar(distance, heuristic, neighbors, is_goal, start):
  from heapq import heappop, heappush
  frontier = []
  heappush(frontier, (0, 0, start))
  C = {}
  G = {}
  C[start] = None
  G[start] = 0

  while frontier:
    _, _, x = heappop(frontier)

    if is_goal(x):
      return reconstruct_path(x, C)

    for y in neighbors(x):
      g = G[x] + distance(x, y)
      if g < G.get(y, float("inf")):
        C[y] = x
        G[y] = g
        h = heuristic(y)
        priority = g + h
        heappush(frontier, (priority, h, y))

  return None
```

[30%]

c)

Correctness: using A* to find the path (i.e. providing valid implementations of all the astar arguments)
Style: using the builtin vector functions

```
def shortest_path(self, player: Vec2) -> Optional[List[Vec2]]:
  def distance(a: Vec2, b: Vec2) -> int:
    return (a-b).length()

  def heuristic(p: Vec2) -> int:
    return distance(p, self.goal)

  def neighbors(p: Vec2) -> List[Vec2]:
    dirs = [Vec2(1, 0), Vec2(-1, 0), Vec2(0, 1), Vec2(0, -1)]

    def is_valid(p: Vec2) -> bool:
      return self.is_inside(p) and p not in self.obstacles

    return [p + d for d in dirs if is_valid(p + d)]

  def is_goal(p: Vec2) -> bool:
    return p == self.goal

  return astar(distance, heuristic, neighbors, is_goal, player)
```

[15%]

d)

Correctness: checking if player crushed, checking if player has one, distance and heuristic functions, neighbors function, is_goal function, using A* to find path. Checking if path points are inside the cave.
Style: using the builtin vector functions

```
def game_state(self, player: Vec2) -> GameState:
  if player in self.obstacles:
    return GameState.LOSE

  if player == self.goal:
    return GameState.WIN

  path = self.shortest_path(player)
  return GameState.PLAYING if path else GameState.LOSE
```

[5%]

e)

Correctness: Custom state with player and step, distance and heuristics, using the shortest_path method to determine valid neighbors, is_goal function, using A* to find path.
Style: using NamedTuple for State

```
def game_state(self, player: Vec2) -> GameState:
  if player in self.obstacles:
    return GameState.LOSE

  if player == self.goal:
    return GameState.WIN
```

```
  State = NamedTuple("State", [("player", Vec2), ("step", int)])

  def distance(a: State, b: State) -> int:
    return abs(b.step - a.step)

  def heuristic(s: State) -> int:
    return (s.player - self.goal).length()

  def neighbors(s: State) -> List[State]:
    dirs = [Vec2(1, 0), Vec2(-1, 0), Vec2(0, 1), Vec2(0, -1)]
    game = self.at_step(s.step + 1)

    def is_valid(p: Vec2) -> bool:
      if not self.is_inside(p) or p in game.obstacles:
        return False

      path = game.shortest_path(s.player)
      if path is None:
        return False

      return True

    ps = [s.player + d for d in dirs]

    return [
      State(p, s.step + 1)
      for p in ps
        if is_valid(p)
      ]

  def is_goal(s: State) -> bool:
    return s.player == self.goal

  path = astar(distance, heuristic, neighbors, is_goal, State(player, self
      .step))

  return GameState.WIN if path else GameState.LOSE
```

[20%]

Question 2

a)

i)

Correctness: elastic collision formula implemented correctly
Style: using vector maths

```
def elastic_collision(pos0, vel0, inv_mass0,
                      pos1, vel1, inv_mass1):
  n = pos1 - pos0
  v_ab = vel1 - vel0
  j = -2 * v_ab.dot(n) / (n.dot(n) * (inv_mass0 + inv_mass1))
  impulse = n * j
  return vel0 - inv_mass0 * impulse, vel1 + inv_mass1 * impulse
```

[15%]

ii)

Correctness: checking if horizontal, calculating the wall values properly, returning the new ball

```
def collide_with(self, ball: Ball) -> Ball:
  if self.horizontal:
    v0, _ = elastic_collision(ball.pos, ball.vel, ball.inv_mass,
                              Vec2(ball.pos.x, self.value), Vec2(0, 0),
                                  0)
  else:
    v0, _ = elastic_collision(ball.pos, ball.vel, ball.inv_mass,
                              Vec2(self.value, ball.pos.y), Vec2(0, 0),
                                  0)

  return ball._replace(vel=v0)
```

Alternate solution that does not use the `elastic_collision` method:

```
def collide_with(self, ball: Ball) -> Ball:
  if self.horizontal:
    v0 = ball.vel._replace(y=-ball.vel.y)
  else:
    v0 = ball.vel._replace(x=-ball.vel.x)

  return ball._replace(vel=v0)
```

[5%]

iii)

Correctness: correctly passing things to elastic_collision, returning the new balls

```
def collide_with(self, other):
  v0, v1 = elastic_collision(self.pos, self.vel, self.inv_mass,
                             other.pos, other.vel, other.inv_mass)
  return self._replace(vel=v0), other._replace(vel=v1)
```

[10%]

iv)

Correctness: checking if horizontal, checking if greater than or less than, calculating the gap and velocity, returning the time to collision, returning INF if the velocity is negative

```
def next_collision(self, ball):
  if self.horizontal:
    pos = ball.pos.y
    vel = ball.vel.y
  else:
    pos = ball.pos.x
    vel = ball.vel.x

  if self.value < pos:
    gap = pos - ball.radius - self.value
    vel = -vel
  else:
    gap = self.value - pos - ball.radius

  if vel <= 0:
    return INF

  t = gap / vel
```

```
  return t
```

[10%]

v)

Correctness: calculating dpos and dvel correctly, calculating a, b, c and using the quadractic formula to find t, returning the time to collision, returning INF if the balls will not collide
Style: using the determinant to early exit, using the second formulation of the quadratic formula for numerical stability

```
def next_collision(self, other) -> float:
  dpos = other.pos - self.pos
  dvel = other.vel - self.vel

  b = 2 * dpos.dot(dvel)

  if b >= 0:
    return INF

  a = dvel.dot(dvel)
  c = dpos.dot(dpos) - (self.radius + other.radius) ** 2
  discriminant = b * b - 4 * a * c
  if discriminant <= 0:
    return INF

  # math.sqrt() is acceptable here
  t1 = (2 * c) / (-b + (discriminant.sqrt()))

  return t1
```

[20%]

b)

Correctness: calculating and storing the ball-ball collisions, calculating and storing the wall-ball collisions, calculating the absolute time of collision, returning the minimum TOI properly
Style: only storing the collision time between a pair of balls once, only updating the collisions for i, O(1) lookups (e.g. using a dict or a set)
There are several possible valid solutions, but this is one typical example:

i)

```
def min_toi(tois: Mapping[Tuple[int, int], d.Decimal]) -> Tuple[int, int]:
  def toi(x):
    return x[1]

  item = min(tois.items(), key=toi)
  return Collision(toi(item), *item[0])

def initialise_tocs(self):
  self.ball_tocs = {}
  self.wall_tocs = {}

  for i in range(self.num_balls):
    b = self.balls[i]
    for j in range(i+1, self.num_balls):
```

```
      self.ball_tocs[(i, j)] = b.next_collision(self.balls[j])

    for j in range(self.num_walls):
      self.wall_tocs[(i, j)] = self.walls[j].next_collision(b)

  self.next_wall_collision_ = min_toi(self.wall_tocs)
  self.next_ball_collision_ = min_toi(self.ball_tocs)
```

[15%]

ii)

```
def update_tocs(self, i: int):
  b = self.balls[i]
  for j in range(0, i):
    self.ball_tocs[(j, i)] = self.time + b.next_collision(self.balls[j])

  for j in range(i+1, self.num_balls):
    self.ball_tocs[(i, j)] = self.time + b.next_collision(self.balls[j])

  for j in range(self.num_walls):
    self.wall_tocs[(i, j)] = self.time + self.walls[j].next_collision(b)

  self.next_wall_collision_ = min_toi(self.wall_tocs)
  self.next_ball_collision_ = min_toi(self.ball_tocs)
```

[15%]

iii)

```
def next_ball_collision(self) -> Collision:
  return self.next_ball_collision_
```

[5%]

iv)

```
def next_wall_collision(self) -> Collision:
  return self.next_wall_collision_
```

[5%]

Question 3

(a)

The initial loop performs initialisation by setting all elements in the working array C to zero. The second loop checks if the value of any input element in A is i. If it is, $C[i]$ is incremented. This results in $C[i]$ containing the number of input elements in A equal to i for each $i = 0..k$. In the third loop the algorithm calculates for each $i = 0..k$ how many input elements are less than or equal to i. In the fourth loop the algorithm places each input element $A[j]$ into its sorted position in the output array B. Since all n elements may not be distinct, the algorithm decrements $C[Aj]]$ each time a value of $A[j]$ is placed into the output array B. Decrementing like this ensures the next hypothetical input element with a value equal to $A[j]$ to take the position immediately before $A[j]$ in the output array B. Thus B is the output array and C is a working array. The final output is $B = [0, 0, 2, 2, 3, 3, 3, 5]$. [20%]

(b)

There are four loops. Loop 1 requires $\Theta(k)$ time, loop 2 requires $\Theta(n)$ time, loop 3 requires $\Theta(k)$ time and loop 4 requires $\Theta(n)$ time. Hence, the overall time complexity is $\Theta(k+n)$ (which in practice, if $k = O(n)$, will yield $\Theta(n)$). [20%]

(c)

The sort is stable. We can see this by examining lines 7–8 where we create the output array B. Assume we have some element k in positions i and j with $i < j$. Since $j > i$, line 8 will examine $A[j]$ before $A[i]$. It then correctly places $A[j]$ in the output array B by placing it in position $B[C[k]]$. Since position $C[k]$ is then decremented on line 9 and never incremented again, when the loop examines $A[i]$ the new position must be lower than the original position $C[k]$. [20%]

(d)

The algorithm still works correctly since the order in which elements are taken from the working array C and put into the output array B does not affect the placement of elements with the same key. [20%]

(e)

Perform preprocessing as indicated in lines 1–6 in the algorithm given in the question. This requires $\Theta(n+k)$ preprocessing time. Then compute:

$$n = \begin{cases} C[b] & \text{if } a = 0 \\ C[b] - C[a-1] & \text{otherwise} \end{cases}$$

This requires $O(1)$ time.

[20%]

Question 4

(a)

The algorithm slides the pattern across the text, outputting i for each shift i where the pattern exists in the text. Hence the purpose of i is to track the valid shifts of P in T. The loop starting on line 3 considers all shifts. The test on line 4 checks if the shift is valid. Line 5 outputs each valid shift. P will not match for $i = 0$ or $i = 1$. $i = 2$ will yield a match and hence $i = 2$ is valid shift. Finally, $i = 3$ is not a valid shift. [20%]

(b)

The time complexity of the algorithm is $O((n-m+1)m)$. This upper bound is also the tight bound. For example, set T to n successive `x` characters and P to m successive `x` characters. There are $n-m+1$ possible valid shifts and line 4 must execute m times to validate each shift. Hence, worst-case running time is $\Theta((n-m+1)m)$. [20%]

(c)

The algorithm is not as efficient as it could be since it has no memory of any prior solution for a previously valid shift. For example, if $P =$ `aaab` and a shift at 0 is valid then the shifts at 1, 2 and 3 must be invalid since the fourth character in P is `b`. [20%]

(d)

Suppose $T[i] \neq P[j]$. Then for some k in the interval $[1, j)$, $T[i-k] = P[j-k] \neq P[0], [i-k, i)$ are all invalid shifts which do not require checking. Hence, in the next iteration we can compare $T[i]$ with $P[0]$. [20%]

(e)

This reduces to a dynamic programming problem with time and space complexity $O(nm)$. [20%]