Problème 1.

Consider a $m \times n$ game board in which each cell has a numeric value, e.g:

	A	B	C	D	E	F
G	1	2	2	3	4	2
H	3	4	4	4	4	1
I	1	4	1	3	1	4
J	2	3	1	4	1	2
K	3	3	2	1	4	2

A player starts the game with a token in the top-left cell (the cell GA in this example) and the player finishes the game by moving to the bottom-right cell (the cell KF in this example). In each round of the game, the player can move in four directions (up, down, left, and right). The distance of each move is determined by the value of the cell. When going over the border of the game board, one ends up on the other side. For example, if the player is in the cell JB, which has value 3, then the player can move 3 steps up (reaching GB), 3 steps right (reaching JE), 3 steps down (reaching HB), and 3 steps left (reaching JE). The score of a player is determined in the total number of rounds the player needs to reach the bottom-right cell.

P1.1

Model the above problem as a graph problem: What are the nodes and edges in your graph, do the edges have weights, and what problem are you trying to answer on your graph?

The game will be modelled as an unweighted directed graph, where the problem is to find the shortest path (minimum number of rounds) to get from top-left cell to bottom-right cell:

- **Nodes**: Each cell in the game board is a node.
- **Edges**: edges are possible moves from one cell to another. The edges will have unweighted, as each moves takes one round regardless of the distance. For example, cell JB will be connected to GB, JE, JE, BH

P1.2

Provide an efficient algorithm that given a $m \times n$ game board, will find an optimal solution if such a solution exists. If the game board has no solution, then the algorithm should report that the game board is invalid. The runtime of the algorithm should be worst-case $\mathcal{O}(mn)$

Implement a breadth-first search to find shortest path:

Algorithm Shortest Path

```
procedure ShortestPath(board, m, n)
    $start \leftarrow (0, 0)$
    $end \leftarrow (m - 1, n - 1)$
    $queue \leftarrow$ empty queue
    $visited \leftarrow$ boolean array of size $m \times n$ initialized to $false$
    $distance \leftarrow$ integer array of size $m \times n$ initialized to $\infty$
    $visited[start] \leftarrow true$
    $distance[start] \leftarrow 0$
    $queue$.enqueue($start$)
    while $queue$ is not empty do
        $(i, j) \leftarrow queue$.dequeue()
        if $(i, j) = end$ then
            return $distance[end]$
        end if
        $value \leftarrow board[i][j]$
        for $(x, y)$ in $\{(i - value, j), (i + value, j), (i, j - value), (i, j + value)\}$ do
            $(x, y) \leftarrow (x \bmod m, y \bmod n)$
            $new \leftarrow (x, y)$
            if $visited[new] = false$ then
                $visited[new] \leftarrow true$
                $distance[new] \leftarrow distance[cell] + 1$
                $queue$.enqueue($new$)
            end if
        end for
    end while
    return $\infty$ //No solution exists
end procedure
```

P1.3

Explain why your algorithm is correct and has a complexity that is worst-case $\mathcal{O}(mn)$

For BFS, for shortest path in unweighted path has runtime complexity of $\mathcal{O}(V + E)$ worst-case. In this setup, $V = mn$ (number of vertices, which is mn), and $E \leq 4mn$ (number of possible moves, up, left, right, down, which would be $\leq 4mn$). Thus, worst-case runtime complexity is $\mathcal{O}(mn)$

Let $T(m,n)$ be maximum number of operations performed by the algorithm. The boundary function can be defined as:

$$T(m,n) \leq c \cdot mn$$

where c is a constant. Invariance requires BFS traversal holds true.

Base case: starts cell is $(0, 0)$, therefore $T(1,1) = 1 \leq c \cdot 1 \cdot 1$

Assume that invariance holds for all cell up to k^{th} cell, that is, for any (i,j) we have $T(i,j) \leq c \cdots ij$

Consider the $(k+1)^{\text{th}}$ cell (i,j) processed. The number of operations is as followed:

- dequeue (i,j): constant c_2
- check if (i,j) is the end cell: constant c_3
- retrieve value of cell: constant c_4
- iterate through the 4 possible moves: constant c_5
- check if neighbor is visited and mark nodes: constant c_6

Total number of operations is $c_2 + c_3 + c_4 + c_5 + c_6 = c_7$

Thus, number of operations performed for the $(k+1)^{\text{th}}$ cell is:
$T(i,j) \leq c \cdot ij + c_7 \leq c \cdot mn + c_7$

$$T(i,j) \leq c \cdot mn + c_7 \leq c \cdot mn + c \leq 2c \cdot mn$$

Therefore, it holds for $(k+1)^{\text{th}}$ cell.

P1.4

Which of the two graph representation we saw in the course material did you use to store the game board? What would the complexity of your algorithm be if you used the other graph representation?

The graph representation is used as an adjacency list to store the game board.

If the graph representation was an adjacency matrix, the graph as 2D matrix of size $(mn) \times (mn)$, space complexity would increase to $\mathcal{O}((mn)^2)$, time complexity would remain $O(mn + E)$, where $E \leq 4mn$. Given that accessing neighbour cell would be faster since constant-time access of matrix entries. Overall time complexity would still be $\mathcal{O}(mn)$

Problème 2.

Edge-labeled graphs are graphs in which edges have labels that represent the type of relationship that is expressed by that edge. For example, in a social network graph, the edges could be labeled `parentOf`, `friendOf`, and `worksWith`. One way to express graph queries (that express how new information can be derived from edge-labeled graphs) is via a query graph that expresses how new relationships between source node s and target node t can be derived from existing information. The first query relates nodes that represent grandparents and their grandchildren, the second query relates nodes that represent ancestors and their descendants, and the third query relates everyone with a direct family relationship. Let Q be a graph query and G be an edge-labeled graph representing a data set. The graph query evaluation problem is the problem of computing the derived relationship in G expressed by Q. Typically, queries are small and data graphs are enormous. Hence, here we will assume that the size of a query graph is constant.

P2.1

Model the graph query evaluation problem as a graph problem: What are the nodes and edges in your graph, do the edges have weights, and what problem are you trying to answer on your graph?

The graph query evaluation can be modelled as directed graph, where:

- **Nodes**: Each node represents a data elements in the graph
- **Edges**: Each edge represents a relationship between two nodes, for example, 'childOf', 'parentOf'.

The problem is to find all the subgraphs or paths in the graph databases that matches the pattern specified by the query.

For example, consider the following query `grandParentOf(s, t)`, the problems is to find all pairs of node (s, t) in graph such that there exists a path of length 2 from s to t, with both edges labeled "parentOf".

P2.2

Provide an efficient algorithm that, given a graph G, a source node n in graph G, and query Q, will find all nodes m such that the pair (n, m) is in the derived relationship in G expressed by Q. Assuming Q has a constant time, the runtime of your algorithm should be worst-case $\mathcal{O}(|G|)$ in which $|G|$ is the total number of nodes and edges in G.

Algorithm Graph Query Evaluation

```
**Input:** Graph $G$, Source node $n$, Query $Q$
**Output:** All nodes $m$ such that $(n, m)$ is in the derived relationship
$R \leftarrow []$//List to store result nodes
$Visited \leftarrow \{\}$
$Queue \leftarrow$ InitializeQueue()
Enqueue($Queue$, $(n, 0)$)//Enqueue source node with depth 0
**while** Queue is not empty **do**
    $(u, depth) \leftarrow$ Dequeue($Queue$)
    **if** $u \notin Visited$ **then**
        $Visited \leftarrow Visited \cup \{u\}$
        **for all** $v$ such that $Q(u, v)$ is true **do**
            $R$.append($v$)
            **if** Q is transitive **then**
                Enqueue($Queue$, $(v, depth + 1)$)
            **end if**
        **end for**
        **for all** $v$ in Neighbors($G, u$) **do**
            **if** $v \notin Visited$ **then**
                Enqueue($Queue$, $(v, depth + 1)$)
            **end if**
        **end for**
    **end if**
**end while**
**return** $R$
```

P2.3

Explain how you represented your graph G, why your algorithm is correct, and why your algorithm has a complexity that is worst-case $O(|G|)$.

The graph G is represented as an adjacency list, where each node maintains a list of its neighbors and its edge labels. Since we are doing BFS traversal from source node n.

Time complexity for performing a BFS traversal is $O(|G| + |R|)$, where $|G|$ is the number of nodes and edges in the graph, and $|R|$ is the number of nodes in the derived relationship. Since Q has a constant size, number of derived relationship $|R|$ is constant, therefore, overall time complexity is $O(|G|)$.

Correctness of BFS traversal is guaranteed by the invariance of BFS traversal. The algorithm visits all nodes in the graph, and for each node, it visits all its neighbors. The algorithm also checks if the node has been visited before, and if not, it adds the node to the visited set and enqueues its neighbors. Therefore, the algorithm is correct.