Paradigmas de Programación
Práctica 8

1. Defina en OCaml una función ***queens : int -> (int * int) list list***, de modo que *queens n* sea, para cada *n*, la lista de todas las soluciones al problema de las reinas de dimensión *n*. Así, por ejemplo, podríamos tener

```
# queens;;
- : int -> (int * int) list list = <fun>
# queens 0;;
- : (int * int) list list = [[]]
# queens 1;;
- : (int * int) list list = [[(1, 1)]]
# queens 2;;
- : (int * int) list list = []
# queens 4;;
- : (int * int) list list =
[[(1, 3); (2, 1); (3, 4); (4, 2)]; [(1, 2); (2, 4); (3, 1); (4, 3)]]
```

Cada solución se da como una lista de las coordenadas donde deben situarse las reinas (las filas y columnas deben numerarse de *1* a *n*). Nótese que el orden de la lista de soluciones no es relevante. De la misma forma, tampoco es relevante el orden en que aparecen las casillas dentro de cada solución. Es decir, la función *queens* sería igualmente válida si tuviésemos, por ejemplo

```
# queens 4;;
- : (int * int) list list =
[[(4, 3); (3, 1); (2, 4); (1, 2)]; [(4, 2); (3, 4); (2, 1); (1, 3)]]
```

Defina una función ***is_queens_sol: int -> (int * int) list -> bool*** tal que *is_queens_sol n sol* indique si la lista *sol* es una solución válida para el problema de las *n* reinas. En la implementación de esta función **no debe usarse la función *queens***, pues se pretende utilizarla para comprobar su corrección. Así, por ejemplo, debería cumplirse lo que puede verse en el siguiente ejemplo

```
# is_queens_sol 2 [(1, 1); (2, 2)];;
- : bool = false
# is_queens_sol 2 [(1, 1); (2, 3)];;
- : bool = false
# let check n = List.for_all (is_queens_sol n) (queens n);;
val check : int -> bool = <fun>
# List.for_all check (List.init 13 abs);;
- : bool = true
```

Puede comprobar también que el número de soluciones que da la función queens para cada *n*, se corresponde con el número de soluciones indicado en wikipedia.

```
# List.map (fun i -> i, List.length (queens i)) (List.init 13 abs);;
- : (int * int) list =
[(0, 1); (1, 1); (2, 0); (3, 0); (4, 2); (5, 10); (6, 4); (7, 40); (8, 92);
 (9, 352); (10, 724); (11, 2680); (12, 14200)]
```

Estas definiciones deben escribirse en un archivo "queens.ml", que debe compilar sin errores con el comando

```
ocamlc -c queens.mli queens.ml
```

2. (Ejercicio opcional) ***"Caballeros informáticos y universos 4D"***
Inspirados por *"El Problema del Caballo"* o *"Knight's Tour"* queremos definir una función

tour: int -> int -> (int * int) list -> int * int -> int * int -> (int * int) list

tal que ***tour m n obstaculos ini fin*** sea un recorrido realizado "a salto de caballo" en un tablero de *m* filas y *n* columnas, desde la casilla *ini* a la casilla *fin,* evitando pasar por las casillas de la lista *obstaculos*.
Aunque en el ejercicio anterior numeramos filas y columnas de *1* a *n*, en este caso, siguiendo la costumbre informática (asumida también por los caballeros), empezaremos a numerar en el *0*, de modo que las filas estarán numeradas de *0* a *(m-1)* (de arriba a abajo) y las columnas de *0* a *(n-1)* (de izquierda a derecha). La casilla *ini* debe ser la primera de la lista resultado y *fin* la última, no se debe pasar dos veces por la misma casilla (no puede haber valores repetidos en la lista) y no podemos salirnos del tablero. Si no existiese un recorrido tal, la función debería activar la excepción *Not_found.*

```
# tour 2 5 [] (0,0) (1,2);;
- : (int * int) list = [(0, 0); (1, 2)]

# tour 2 5 [] (0,0) (1,3);;
Exception: Not_found.

# tour 5 6 [] (1,2) (1,1);;
- : (int * int) list =
[(1, 2); (2, 4); (3, 2); (4, 4); (2, 5); (3, 3); (4, 1); (2, 2); (3, 4); (4, 2);
(3, 0); (1, 1)]

# tour 5 6 [(3,2)] (1,2) (1,1);;
- : (int * int) list =
[(1, 2); (2, 4); (4, 5); (3, 3); (4, 1); (2, 2); (3, 4); (4, 2); (3, 0); (1, 1)]

# tour 5 6 [(3,2); (4,5)] (1,2) (1,1);;
- : (int * int) list =
[(1, 2); (2, 4); (0, 5); (1, 3); (2, 5); (3, 3); (4, 1); (2, 2); (3, 4);(4, 2);
(3, 0); (1, 1)]
```

Tenga en cuenta que sus soluciones no tienen por qué coincidir con las mostradas.

Si ha conseguido implementar la función tour, intente definir una función

min_tour : int -> int -> (int * int) list -> int * int -> int * int -> (int * int) list

de modo que ***min_tour m n obstaculos ini fin*** devuelva un camino como los que debe devolver la función *tour,* pero con la característica adicional de ser minimal; es decir que no haya otro camino más corto en esas condiciones.

```
# min_tour 5 6 [] (1,2) (1,1);;
- : (int * int) list = [(1, 2); (3, 1); (2, 3); (1, 1)]

# min_tour 5 6 [(3,1)] (1,2) (1,1);;
- : (int * int) list = [(1, 2); (0, 4); (2, 3); (1, 1)]

# min_tour 5 6 [(3,1); (2,3)] (1,2) (1,1);;
- : (int * int) list = [(1, 2); (2, 0); (3, 2); (1, 1)]

# min_tour 5 6 [(3,1); (2,3); (3,2); (2, 4)] (1,2) (1,1);;
- : (int * int) list = [(1, 2); (3, 3); (4, 1); (2, 2); (3, 0); (1, 1)]

# min_tour 5 6 [] (0,1) (4,4);;
- : (int * int) list = [(0, 1); (2, 0); (3, 2); (4, 4)]
```

En un universo 4D han conseguido curvar los tableros y pegar sus bordes de forma que la última fila vuelve a quedar encima de la primera, y la última columna vuelve a quedar a la izquierda de la primera. De este modo, desde cualquier casilla, cada uno de los ocho saltos de caballo inicialmente posibles, vuelve a caer siempre dentro del tablero.

Defina una función

min_tour4D : int -> int -> (int * int) list -> int * int -> int * int -> (int * int) list

que calcule caminos minimales en el universo 4D. Tenga en cuenta que un camino minimal en el universo 4D puede ser más corto que un camino minimal en nuestro universo.

```
# min_tour4D 5 6 [] (0,1) (4,4);;
- : (int * int) list = [(0, 1); (4, 4)]

# min_tour4D 5 6 [(3,1); (2,3); (3,2); (2, 4)] (1,2) (1,1);;
- : (int * int) list = [(1, 2); (4, 0); (3, 3); (1, 1)]
```

Las tres funciones pedidas en este ejercicio deben implementarse en un archivo “knight.ml” que compile sin errores con el comando

```
ocamlc -c knight.mli knight.ml
```

Si no consigue implementar alguna de las tres funciones puede definirla de modo que devuelva siempre la lista vacía (para que el fichero compile correctamente).