be-OI 2025 **Final - SENIOR** Saturday, March 22, 2025	**Fill in this box in CAPITAL LETTERS please** FIRST NAME : . . . . LAST NAME : . . . . SCHOOL :	O **Reserved**

Finals of the Belgian Informatics Olympiad 2025 (time allowed : **2h maximum**)

General information (read carefully before attempting the questions)

1. Verify that you have received the correct set of questions (as mentioned above in the header):
   - for the students in the second year of the Belgian secundary school system or below (US grade 8, UK year 9): category **cadet**.
   - for the students in the third or fourth year of the Belgian secundary school system or equivalent (US grade 9-10, UK year 10-11): category **junior**.
   - for the students in the fifth year of the Belgian secundary school system or equivalent (US grade 11, UK year 12) and above: category **senior**.
2. Write your first name, last name and school **on this page only**.
3. Write **your answers** on the provided answer sheets. Write **clearly and legibly** with a blue or black **pen or bic**.
4. Use a pencil and eraser when working in draft form on the question sheets.
5. You may only have writing materials with you. Calculator, GSM, smartphone … are **forbidden**.
6. You can always request extra scratch paper from the invigilator.
7. When you have finished, **hand in this first page (with your name on it) and the pages with your answers**, you can keep the other pages.
8. All the snippets of code in the exercises are written in **pseudo-code**. On the next pages you will find a description of the pseudo-code that we use.
9. If you have to respond with code, you can do so in **pseudo-code** or in any **current programming language** (such as Java, C, C ++, Pascal, Python, ...). We do not deduct points for syntax errors.

Good Luck !

The Belgian IT Olympiad is possible thanks to the support from our members:

Pseudo-code checklist

Data is stored in variables. We change the value of a variable using $\leftarrow$. In a variable, we can store whole numbers, real numbers, or arrays (see below), as well as Boolean (logical) values: true/correct (**`true`**) or false/wrong (**`false`**). It is possible to perform arithmetic operations on variables. In addition to the four conventional operators ($+$, $-$, $\times$ and $/$), you can also use the operator %. If a and b are whole numbers, then a/b and $a\%b$ denote respectively the quotient and the remainder of the division.
For example, if $a = 14$ and $b = 3$, then $a/b = 4$ and $a\%b = 2$.

Here is a first code example, in which the variable *age* receives 17.

```
$birthYear \leftarrow 2008$
$age \leftarrow 2025 - birthYear$
```

To run code only if a certain condition is true, we use the instruction **`if`** and possibly the instruction **`else`** to execute another code if the condition is false. The next example checks if a person is of legal age and stores the price of their movie ticket in the variable *price*. Look at the comments in the code.

```
if ($age \geq 18$)
{
    $price \leftarrow$ 8 // This is a comment.
}
else
{
    $price \leftarrow$ 6 // cheaper !
}
```

Sometimes when one condition is false, we have to check another. For this we can use **`else if`**, which comes down to executing another **`if`** inside the **`else`** of the first **`if`**. In the following example, there are 3 age categories that correspond to 3 different prices for the movie ticket.

```
if ($age \geq 18$)
{
    $price \leftarrow$ 8 // Price for a person of legal age (adult).
}
else if ($age \geq 6$)
{
    $price \leftarrow$ 6 // Price for children aged 6 or older.
}
else
{
    $price \leftarrow$ 0 // Free for children under 6.
}
```

To handle several elements with a single variable, we use an array. The individual elements of an array are identified by an index (which is written in square brackets after the name of the array). The first element of an array $tab[\,]$ has index 0 and is denoted $tab[0]$. The second element has index 1 and the last has index $n - 1$ if the array contains n elements. For example, if the array $tab[\,]$ contains the 3 numbers 5, 9 and 12 (in this order), then $tab[0]= 5$, $tab[1]= 9$, $tab[2]= 12$. The array is size 3, but the highest index is 2.

To repeat code, for example to browse the elements of an array, we can use a **for** loop. The notation **for** **(**$i \leftarrow a$ **to** b **step** k**)** represents a loop which will be repeated as long as $i \leq b$, in which i begins with the value a and is increased by k at the end of each step. The following example calculates the sum of the elements of the array $tab[\,]$ assuming its size is n. The sum is found in the variable sum at the end of the execution of the algorithm.

```
$sum \leftarrow 0$
for ($i \leftarrow 0$ to $n-1$ step 1)
{
    $sum \leftarrow sum + tab[i]$
}
```

You can also write a loop using the instruction **while**, which repeats code as long as its condition is true. In the next example, we're going to divide a positive integer n by 2, then by 3, then by 4 ... until it is a single digit number (i.e. until $n < 10$).

```
$d \leftarrow 2$
while ($n \geq 10$)
{
    $n \leftarrow n/d$
    $d \leftarrow d+1$
}
```

Often the algorithms will be in a frame and preceded by descriptions. After **Input**, we define each of the arguments (variables) given as input to the algorithm. After **Output**, we define the state of certain variables at the end of the algorithm execution and possibly the returned value. A value can be returned with the instruction **return**. When this instruction is executed, the algorithm stops and the given value is returned.

Here is an example using the calculation of the sum of the elements of an array.

```
Input  : $tab[\,]$, an array of $n$ numbers.
         $n$, the number of elements in the array.
Output : $sum$, the sum of all the numbers in the array.


$sum \leftarrow 0$
for ($i \leftarrow 0$ to $n-1$ step 1)
{
    $sum \leftarrow sum + tab[i]$
}
return $sum$
```

Note: in this last example, the variable i is only used as a counter for the **for** loop. There is therefore no description for it either in **Input** or in **Output**, and its value is not returned.

Question 1 – Alcuin and his boat

The monk Alcuin of York is the happy owner of a mouse, a cat, and a large piece of cheese. He needs to get all of them across the river, but his boat is too small. He can only transport one passenger at a time: either the cat, the mouse, or the cheese. Unfortunately, if the mouse is left alone on the bank with the cheese, it will eat it (if Alcuin is present on the bank, he can watch the mouse). Similarly, the cat cannot be left alone with the mouse, or it will eat it. These two situations are called conflicts. There is no conflict between the cat and the cheese; they can stay together unsupervised.

Alcuin wants to get the cat, the mouse, the cheese, and himself from bank A to bank B of the river. He must make round trips between the two banks, starting on bank A. He explores several scenarios. Here is the first one, where he indicates the direction of each step and what he carries with him in his boat: the cat (🐈‍⬛), the mouse (🐭), the cheese (🧀), or nothing at all (nothing).

Scenario 1

1. bank A → bank B: 🐭
2. bank B → bank A: nothing
3. bank A → bank B: 🧀
4. bank B → bank A: nothing
5. bank A → bank B: 🐈‍⬛

In order to check if his scenario is valid, Alcuin fills in a table to indicate on which bank (A or B) Alcuin, the cat, the mouse, and the cheese are located at each step. The table starts at step ’0’ which represents the initial situation (before the first transport).
The first two lines of the table are already filled in. You must complete it and mark a cross in the last column if there is a conflict at the corresponding step.

Q1(a) /5	**Complete Alcuin’s table for scenario 1.**

	Alcuin	🐈‍⬛	🐭	🧀	conflict?
step 0 :	A	A	A	A	
step 1 :	B	A	B	A	
step 2 :	A	A	B	A	
step 3 :	B	A	B	B	
step 4 :	A	A	B	B	**X**
step 5 :	B	B	B	B	

Alcuin then considers a new scenario. You are asked to complete the table again.

Scenario 2

1. bank A → bank B: 🐭
2. bank B → bank A: nothing
3. bank A → bank B: 🧀
4. bank B → bank A: 🐭
5. bank A → bank B: 🐈
6. bank B → bank A: nothing
7. bank A → bank B: 🐭

Q1(b) /7	**Complete Alcuin's table for scenario 2.**

	Alcuin	🐈	🐭	🧀	conflict?
step 0 :	A	A	A	A	
step 1 :	B	A	B	A	
step 2 :	A	A	B	A	
step 3 :	B	A	B	B	
step 4 :	A	A	A	B	
step 5 :	B	B	A	B	
step 6 :	A	B	A	B	
step 7 :	B	B	B	B	

Alcuin then inherits a larger boat, which allows him to carry two passengers in addition to himself.
We say that he has a size 2 boat (before, he had a size 1 boat).

Q1(c) /3	**Give a scenario in three steps to get the cat, the mouse, and the cheese from bank A to bank B, with a size 2 boat. For each step, you must name 0, 1, or 2 passengers transported by Alcuin.**

1. bank A → bank B : 🐭
2. bank B → bank A : nothing
3. bank A → bank B : 🐈 , 🧀

There are several other solutions.

Now suppose the cat decides to like cheese: there is a **new conflict**, the cat and the cheese can no longer be left together unsupervised.

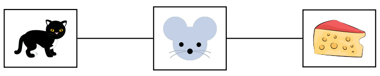
Q1(d) /3	**Give a scenario in 3 steps with a size 2 boat. You must take into account the new conflict between the cat and the cheese. The mouse must cross during the first step.** **For each step, you must name 0, 1, or 2 passengers transported by Alcuin.**

1. bank A → bank B : 🐭, 🐈‍⬛
2. bank B → bank A : 🐈‍⬛
3. bank A → bank B : 🐈‍⬛, 🧀

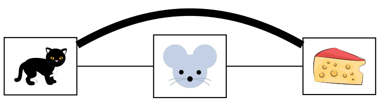
There are several other solutions.

With his experience, Alcuin decides to leave monastic life and start a business specializing in complicated river crossings. His clients give him *n* passengers (objects or animals) that need to cross the river, along with a diagram indicating those that are in conflict. This diagram is called a 'graph' and is composed of labeled rectangles, called 'nodes', one for each passenger. It also contains links between the nodes, called 'edges' : an edge is drawn between two nodes if the corresponding passengers are in conflict. However, if two passengers can stay together unsupervised, no edge is drawn between the corresponding nodes.

Here is the graph of the initial situation when the cat did not eat cheese. There was a conflict between the mouse and the cat, a conflict between the mouse and the cheese, but no conflict between the cat and the cheese.



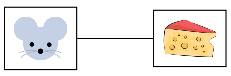
Q1(e) /1	**Modify this graph (by drawing or crossing out edges) to add a conflict between the cat and the cheese.**



Alcuin now wants to know the size of the boat he needs to get a set of passengers across the river given the conflict graph. Naturally, for transporting n passengers, a boat of size n is always sufficient, but Alcuin wants to use the smallest boat possible.

He reasons as follows:
“ *I must divide the passengers into 2 groups. Those in the first group must always be supervised in the boat. The others can stay unsupervised because there is no conflict between them. The passengers to be supervised must be such that if I remove from the graph all the nodes corresponding to them and the edges touching them, there should be no edges left in the graph.*”.

For example, removing the ’cat’ node is not a good solution, because there is still a conflict between the mouse and the cheese.



In other words, we cannot keep the cat alone in the boat and leave the mouse and the cheese on the bank.

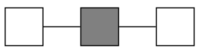
However, at the time when the cat did not like cheese, removing the mouse gives the following graph which contains no conflict:

In other words, we could (when the cat did not like cheese) load the mouse alone in the boat and leave the cat and the cheese on the bank.

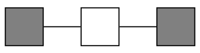
What Alcuin wants to calculate is called a *cover*: **a set of nodes such that, if we remove these nodes and the edges touching them from the graph, no edges remain**.

Here are six graphs, some of whose nodes have been colored gray:

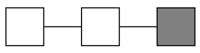
graph 1:



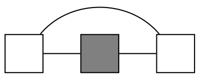
graph 2:



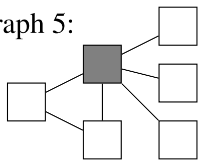
graph 3:



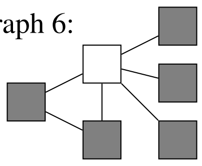
graph 4:



graph 5:

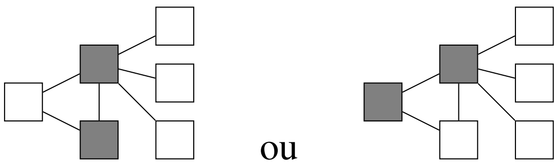


graph 6:



	Yes	No	Indicate whether the gray nodes form a cover in each graph.
Q1(f) /1	☒	☐	The gray nodes in graph 1 form a cover.
Q1(g) /1	☒	☐	The gray nodes in graph 2 form a cover.
Q1(h) /1	☐	☒	The gray nodes in graph 3 form a cover.
Q1(i) /1	☐	☒	The gray nodes in graph 4 form a cover.
Q1(j) /1	☐	☒	The gray nodes in graph 5 form a cover.
Q1(k) /1	☒	☐	The gray nodes in graph 6 form a cover.

Q1(l) /4	**Color the nodes in the graph below. The colored nodes must form a cover containing a minimal number of nodes.**

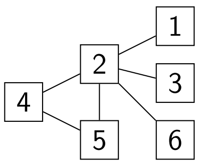


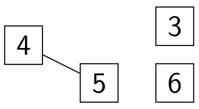
Alcuin wants an algorithm that calculates a cover of a graph. After discussing it at length with the cat, he comes up with the following strategy (which is not necessarily very smart, but it's the cat's idea).
The nodes are numbered from 1 to n and represented by their number in the algorithm.
If there is an edge between nodes numbered i and j, it is denoted (i, j).
The algorithm successively considers all the edges (i, j) of the graph and 'processes' their endpoints, that is to say the nodes numbered i and j.
Processing a node consists of:

1. putting the node in the cover,
2. removing the node and all edges touching it from the graph.

We continue until there are no edges left in the graph.

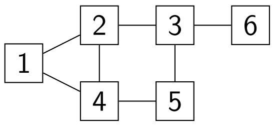
Alcuin observes that there is no direction to the edges, for example, the edge $(3, 1)$ is the same as $(1, 3)$.
He can therefore simply examine the edges (i, j) where $i < j$ (to avoid processing the same edge twice).
Alcuin decides to process the edges in the following order (if they exist):
first the edges $(1, 2)$, $(1, 3)$,... $(1, n)$; then the edges $(2, 3)$, $(2, 4)$,... $(2, n)$; then the edges $(3, 4)$, $(3, 5)$,... $(3, n)$; and so on until the last edge $(n - 1, n)$.

Q1(m) /3	**Draw the graph obtained at the beginning of the algorithm after considering only the first edge** $(1, 2)$ **and thus after processing nodes** 1 **and** 2 **of the graph below.** 

Solution: 

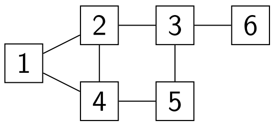
Q1(n) /4	**What is the cover computed after the complete execution of the algorithm on the graph from the previous question? You must give the list of nodes in the computed cover.**

Solution : $\{1, 2, 4, 5\}$

Q1(o) /4	**What is the cover computed by the algorithm on the graph below? You must give the list of nodes in the computed cover.** 

Solution : $\{1, 2, 3, 5\}$

To implement the algorithm, Alcuin uses an array `G` of size $n \times n$ to store the edges.
Since the edges have no direction, he only considers the edges $(\mathtt{i}, \mathtt{j})$ with $\mathtt{i} < \mathtt{j}$.
The array contains **`true`** in the cell `G[i][j]` (with $\mathtt{i} < \mathtt{j}$) if and only if there is an edge $(\mathtt{i}, \mathtt{j})$ in the graph.
The other cells of the array contain **`false`**.

Q1(p) /4	**Give the array `G` that corresponds to the graph below, considering that the cell `G[i][j]` is located at row `i`, column `j`. Write a T in the cells that contain `true`.** 

	1	2	3	4	5	6
1		T		T		
2			T	T		
3					T	T
4					T	
5						
6						

Alcuin wrote his algorithm on a piece of parchment but unfortunately the mouse nibbled on some parts of it. Alcuin needs your help to find the missing parts.

The algorithm starts with two functions:

- `InitCountEdges` counts the number of edges in the graph and stores this number in the variable `countEdges`,
- `ProcessNode(i)` processes the node `i` (as described above).

In addition to the array `G[][]`, a Boolean array `InCover[]` of size `n` is used, with all elements initialized to **`false`**. The algorithm indicates that node `i` is part of the cover by changing the value of `InCover[i]` to **`true`**.

Q1(q) /4	Complete the ▨ in the function **`InitCountEdges`** (answers as short as possible).

```
function InitCountEdges()
{
  CountEdges = 0
  for (i ← 1 to n-1 step 1)
  {
    for (j ← i+1 to n step 1)
    {
      if (G[i][j]) CountEdges ← CountEdges + 1
    }
  }
}
```

Q1(r) /6	Complete the ▨ in the function **`ProcessNode`** (answers as short as possible).

```
function ProcessNode(i)
{
  InCover[i] ← true
  for (k ← 1 to i-1 step 1)
  {
    if (G[k][i])
    {
      G[k][i] ← false
      CountEdges = CountEdges - 1
    }
  }
  for (k ← i+1 to n step 1)
  {
    if (G[i][k])
    {
      G[i][k] ← false
      CountEdges ← CountEdges - 1
    }
  }
}
```

The algorithm is completed by the function `Cover`.
This function receives an array `G[][]` of size `n` × `n` representing the existing edges between the *n* nodes.
The array `InCover[]` (of size `n`) is initialized to **`false`** as explained above before executing the function `Cover`.
The computed cover is composed of all nodes `i` such that `InCover[i]` is **`true`** after the execution of the function `Cover`.

Q1(s) /6	**Complete the ▨ in the function `Cover` (answers as short as possible).**

```
function Cover()
{
  InitCountEdges()
  i ← 1
  j ← 2
  while (CountEdges != 0)
  {
    if (G[i][j])
    {
      ProcessNode(i)
      ProcessNode(j)
    }
    j ← j+1
    if (j = n+1){
      i ← i+1

      j ← i+1
    }
  }
}
```

Question 2 – Naval combat

We want to program the naval combat game that is played on a 10×10 grid where boats are placed.

- There are 10 rows numbered from 1 to 10 from top to bottom.
- There are 10 columns numbered from 1 to 10 from left to right.
- A horizontal boat occupies consecutive squares on the same row.
- A vertical boat occupies consecutive squares on the same column.
- We use `(r,c)` to denote the coordinates of the square at the intersection of row `r` and column `c`.

Example:
– 3 boats **A**, **B** and **C** are drawn in gray on the image.
– **A** is horizontal, **B** is vertical, **C** is both horizontal and vertical.
– **B** occupies the squares with coordinates `(5,7)`, `(6,7)` and `(7,7)`.

	1	2	3	4	5	6	7	8	9	10
1										
2			A	A	A	A				
3										
4										
5							B			
6							B			
7							B			
8				C						
9										
10										

We are hesitating between 2 systems to represent the boats in our program.
Each system uses 4 attributes to indicate the position of a boat on the grid.

Attributes in the **Orientation** system.
- The row `r` of the upper left square of the boat.
- The column `c` of the upper left square of the boat.
- The length `L` of the boat.
- The direction `hor` of the boat:
  `true` if it is horizontal, **`false`** if it is vertical.

Orientation attributes of the boats on the image.

Boat	`r`	`c`	`L`	`hor`
A	2	3	4	**`true`**
B	5	7	3	**`false`**
C	8	4	1	**`true`** or **`false`** at choice

Attributes in the **Rectangle** system.
- The row `r1` of the upper left square of the boat.
- The column `c1` of the upper left square of the boat.
- The row `r2` of the lower right square of the boat.
- The column `c2` of the lower right square of the boat.

Rectangle attributes of the boats on the image.

Boat	`r1`	`c1`	`r2`	`c2`
A	2	3	2	6
B	5	7	7	7
C	8	4	8	4

We will write some functions in both systems to decide which one to choose for our program.

Conversions between the 2 systems

Let's first practice switching from one system to the other.

Orientation ⟶ Rectangle
A boat is described by its 4 attributes `(r,c,L,hor)` in the **Orientation** system.
Give its attributes `(r1,c1,r2,c2)` in the **Rectangle** system.

Q2(a) /1	**`(4, 5, 2, True)`** ⟶ **`( 4 , 5 , 4 , 6 )`**
Q2(b) /1	**`(3, 6, 3, False)`** ⟶ **`( 3 , 6 , 5 , 6 )`**
Q2(c) /1	**`(7, 5, 1, True)`** ⟶ **`( 7 , 5 , 7 , 5 )`**
Q2(d) /1	**`(1, 9, 5, False)`** ⟶ **`( 1 , 9 , 5 , 9)`**

Rectangle ⟶ Orientation
A boat is described by its 4 attributes `(r1,c1,r2,c2)` in the **Rectangle** system.
Give its attributes `(r,c,L,hor)` in the **Orientation** system.

Q2(e) /1	`(2, 7, 5, 7)` ⟶ `( 2 , 7 , 4 , False )`

Q2(f) /1	`(4, 3, 4, 3)` ⟶ `( 4 , 3 , 1 , True )`

Q2(g) /1	`(8, 3, 8, 6)` ⟶ `( 8 , 3 , 4 , True )`

Q2(h) /1	`(1, 4, 1, 10)` ⟶ `( 1 , 4 , 7 , True )`

Let's create functions to automate these conversions.

Function O2R
The `O2R` function converts attributes from the **Orientation** system to equivalent attributes in the **Rectangle** system.
For example, using boat **A** from the image, `O2R(2,3,4,true)` returns the result `(2,3,2,6)`.

Q2(i) /5	**Complete the ▭ in the `O2R` function.**

```
function O2R(r,c,L,hor)
{
  if (hor)
    { return ( r , c , r , c+L-1 ) }
  else
    { return ( r , c , r+L-1 , c ) }
}
```

Function R2O
The `R2O` function converts attributes from the **Rectangle** system to equivalent attributes in the **Orientation** system.
Use a logical expression (see next page) to calculate the value of `hor`.
For example, using boat **A** from the image, `R2O(2,3,2,6)` returns the result `(2,3,4,true)`.

Q2(j) /5	**Complete the ▭ in the `R2O` function.**

```
function R2O(r1,c1,r2,c2)
{
  L ← (r2-r1) + (c2-c1) + 1

  hor ← (r1==r2)

  return (r1, c1, L, hor)
}
```

Validity tests

The program must check that the attributes describe a boat that can be placed on the 10×10 grid.
It does this by evaluating logical expressions, whose result is either **`true`** or **`false`**.
This type of expression uses the comparison operators (==, !=, >, >=, <, <=) and the logical operators (**`and`**, **`or`**, **`not`**) as shown in the examples below.

Logical expression	Value
`c==4`	**`true`** if `c` is equal to 4, **`false`** otherwise.
`r!=5`	**`true`** if `r` is different from 5.
`r1>3`	**`true`** if `r1` is greater than 3.
`c2>=c1`	**`true`** if `c2` is greater than or equal to `c1`.
`r+L<9`	**`true`** if `r+L` is less than 9.
`c2<=c1+2`	**`true`** if `c2` is less than or equal to `c1+2`.
`(c<2) and (r>3)`	**`true`** if both inequalities are true.
`(c1==3) or (r1==2)`	**`true`** if at least one of the 2 equalities is true.
`not(hor)`	**`true`** if `hor` is equal to **`false`** (and **`false`** if `hor` is equal to **`true`**).

Function Rok

In the **Rectangle** system, the squares `(r1,c1)` and `(r2,c2)` must be in the same row or column, all coordinates must be between 1 and 10, and the first square must be above or to the left of the second square.
Complete the `Rok` function that returns **`true`** if the attributes `(r1,c1,r2,c2)` meet all of these conditions.
Since the expression is very long, it is evaluated in several steps using the logical variable `ok`.

Q2(k) /5	Complete the ▨ in the **`Rok`** function.

```
function Rok(r1, c1, r2, c2)
{
  ok ← (r1==r2) or (c1==c2)
  ok ← ok and (1<=r1 and r1<=r2 and r2<=10)
  ok ← ok and (1<=c1 and c1<=c2 and c2<=10)
  return ok
}
```

Function Ook

In the **Orientation** system, the length must be greater than or equal to 1.
Complete the `Ook` function that returns **`true`** if the boat `(r,c,L,hor)` can be placed on the 10×10 grid.

Q2(l) /5	Complete the ▨ in the **`Ook`** function.

```
function Ook(r,c,L,hor)
{
  ok ← (1<=L and 1<=r and 1<=c)
  if (hor)
    { return (ok and r<=10 and c+L-1<=10) }
  else
    { return (ok and r+L-1<=10 and c<=10) }
}
```

Hit boat

The program must check if a boat is hit when we shoot at a square whose coordinates are given.
This is the case if the boat occupies the targeted square.

In the following questions, try to find the shortest and simplest answers possible using a minimum of comparisons and logical operators.

Function hitR (Rectangle system)

The `hitR(rr,cc, r1,c1,r2,c2)` function returns **`true`** if the boat with attributes `(r1,c1,r2,c2)` occupies the square with coordinates `(rr,cc)` and returns **`false`** if it does not.

With the situation described in the image:

- `hitR(2,5, 2,3,2,6)` returns **`true`** because boat **A** occupies the square with coordinates `(2,5)`.
- `hitR(7,6, 5,7,7,7)` returns **`false`** because boat **B** does not occupy the square with coordinates `(7,6)`.

Q2(m) /6	Complete the ▨ in the **`hitR`** function (answers as short as possible).

```
function hitR(rr,cc, r1,c1,r2,c2)
{
  return (r1<=rr) and (rr<=r2) and (c1<=cc) and (cc<=c2)
}
```

Function hitO (Orientation system)

The `hitO(rr,cc, r,c,L,hor)` function returns **`true`** if the boat with attributes `(r,c,L,hor)` occupies the square with coordinates `(rr,cc)` and returns **`false`** if it does not.

With the situation described in the image:

- `hitO(2,5, 2,3,4,`**`true`**`)` returns **`true`** because boat **A** occupies the square with coordinates `(2,5)`.
- `hitO(7,6, 5,7,3,`**`false`**`)` returns **`false`** because boat **B** does not occupy the square with coordinates `(7,6)`.

Q2(n) /6	Complete the ▨ in the **`hitO`** function (answers as short as possible).

```
function hitO(rr,cc, r,c,L,hor)
{
  if (hor)
    { return (rr==r) and (c<=cc) and (cc<c+L) }
  else
    { return (cc==c) and (r<=rr) and (rr<r+L) }
}
```

Collision

Two boats cannot occupy the same square on the grid, otherwise there is a collision.

Function collisionR (Rectangle system)

The `collisionR` function returns **`true`** if two boats are in collision and **`false`** if they are not.
The attributes of the boats are `(r1A,c1A,r2A,c2A)` and `(r1B,c1B,r2B,c2B)`.
Use logical expressions that are as short as possible. Nonetheless it's very long so the expression is evaluated in several steps using the logical variables `rbool` and `cbool`.

Q2(o) /7	Complete the ▭ in the **`collisionR`** function (answers as short as possible).

```
function collisionR(r1A,c1A,r2A,c2A, r1B,c1B,r2B,c2B)
{
    rbool ← (r1A<=r1B and r1B<=r2A) or (r1B<=r1A and r1A<=r2B)

    cbool ← (c1A<=c1B and c1B<=c2A) or (c1B<=c1A and c1A<=c2B)
    return rbool and cbool
}
```

Function collisionO (Orientation system)

The `collisionO` function returns **`true`** if two boats are in collision and **`false`** if they are not.
The attributes of the boats are `(rA,cA,LA,horA)` and `(rB,cB,LB,horB)`.
To avoid even longer logical expressions, we use the `hitO` function defined above.

Q2(p) /7	Complete the ▭ in the **`collisionO`** function using the **`hitO`** function.

```
function collisionO(rA,cA,LA,horA, rB,cB,LB,horB)
{
  if (horA==horB)
    { return hitO(rB,cB, rA,cA,LA,horA)) or hitO(rA,cA, rB,cB,LB,horB) }
  else
  {
    if (horA)
      { return hitO(rA,cB, rA,cA,LA,horA) and hitO(rA,cB, rB,cB,LB,horB) }
    else
      { return hitO(rB,cA, rA,cA,LA,horA) and hitO(rB,cA, rB,cB,LB,horB) }
  }
}
```

Risk of collision

Boats must not be too close to each other.
More precisely, two squares occupied by two different boats must not have a side or corner in common.
For example, all the boats in the image below are too close to another boat.
A and **B** touch, **C** and **D** touch, **E** and **F** touch by a corner.

	1	2	3	4	5	6	7	8	9	10
1								C		
2		A	A	A	A			C		
3			B					C		
4			B					C	D	
5									D	
6										
7										
8		E	E	E	E					
9						F	F	F		
10										

Function riskR (Rectangle system)

The `riskR` function returns **`true`** if two boats are too close and **`false`** if they are not.
The attributes of the boats are `(r1A,c1A,r2A,c2A)` and `(r1B,c1B,r2B,c2B)`.
The simplest approach is to use the `collisionR` function defined above.

Q2(q) /6	**Complete the ▭ in the `riskR` function using the `collisionR` function.**

```
function riskR(r1A,c1A,r2A,c2A,r1B,c1B,r2B,c2B)
{
  return collisionR( r1A,c1A,r2A,c2A, r1B-1,c1B-1,r2B+1,c2B+1 )
}
```

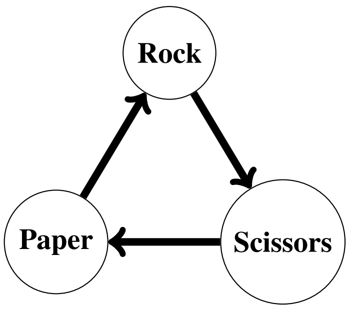
Function riskO (Orientation system)

This function is harder to write.
We give up and decide to use the `Rectangle` system in our program.

Question 3 – Rock-Scissors-Paper

In a famous game, the 2 players simultaneously mime a weapon with their hand: **Rock** or **Scissors** or **Paper**.

The result of the battle is determined by the circular graph opposite:
Rock beats **Scissors** which beats **Paper** which beats **Rock**.
There is a tie if the players choose the same weapon.



A toy manufacturer wants to market small robots capable of playing **Rock-Scissors-Paper** with each other.

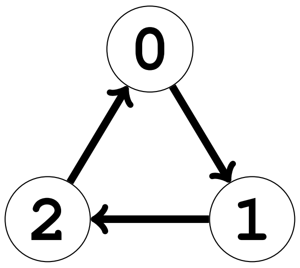
Two robots placed near each other communicate wirelessly. A player presses a button on their robot to propose a match, the other player presses a button on their robot to accept the match.
During a match, the robots play 10 games of **Rock-Scissors-Paper**, each victory earns 1 point for the winning robot.
The games and scores can be followed on the robots' screens.
The one who wins the most games wins the match.
There can be a tie if both robots win the same number of games.

The robots do not play randomly: each follows a **strategy** determined by a program.
The manufacturer wants to offer a large number of robots of different colors, shapes, and especially different strategies.

Codings

In the programs, **Rock** is coded by `0`, **Scissors** by `1` and **Paper** by `2`.
So `0` beats `1` which beats `2` which beats `0` and the game graph becomes the one opposite.

In the following, we will say that a robot plays the move `0`, `1` or `2`.



Strategies

The strategy of a robot is determined by 2 elements.

- The variable `init` which contains the move to play in the first game.
- The table `strat`, with 3 rows and 3 columns, used to determine the moves of the other games.
  If the robot played `r` and its opponent played `s`, then it will play `strat[r][s]` in the next game.

An example of a `strat` table is given on the right.
Its row and column numbers start at `0` and are noted in gray.

	0	1	2
0	1	2	0
1	2	1	0
2	2	2	1

With this table and depending on the moves of the previous game, the robot must play:
- `strat[1][2]=0` if the robot played `1` and its opponent `2`.
- `strat[2][0]=2` if the robot played `2` and its opponent `0`.
- `1` if there was a tie because `strat[0][0]=strat[1][1]=strat[2][2]=1`.

Encoding strategies

In the following questions, we do not care about the `init` variable.
A strategy is described by a text explaining what the robot should play based on what it and its opponent played in the previous game.
You must encode the `strat` table that corresponds to the strategy described by the text.
You can use the grids on the next page as a draft.
Don't forget to **copy your solutions onto the answer sheets**.

Q3(a) /4	**Fill in the `strat` table for the strategy described by this sentence.** **"Play the opponent's previous move."**

	0	1	2
0	0	1	2
1	0	1	2
2	0	1	2

Q3(b) /4	**Fill in the `strat` table for the strategy described by this sentence.** **"Play the move that wins against the opponent's previous move."**

	0	1	2
0	2	0	1
1	2	0	1
2	2	0	1

Q3(c) /4	**Fill in the `strat` table for the strategy described by this sentence.** **"If there was a tie, play the robot's previous move; otherwise play the previous winner's move."**

	0	1	2
0	0	0	2
1	0	1	1
2	2	1	2

Q3(d) /4	**Fill in the `strat` table for the strategy described by this sentence.** **"If there was a tie, play the move that wins against the previous move; otherwise replay the robot's previous move."**

	0	1	2
0	2	0	0
1	1	0	1
2	2	2	1

In the following, we use the notations described below.

- `r`: the move played by the robot in the previous game.
- `s`: the move played by the opponent in the previous game.
- `w`: the move played by the winner of the previous game (or by both players in case of a tie).
- `x`: the move played by the loser of the previous game (or by both players in case of a tie).
- `W(c)`: the move that wins against the move `c` (for example `W(0)=2` because `2` beats `0`).
- `X(c)`: the move that loses against the move `c` (for example `X(0)=1` because `0` beats `1`).

Q3(e) /4	Fill in the `strat` table for the strategy described by this sentence. “If there was a tie, play `X(r)`; otherwise play the move that was not played.”

	0	1	2
0	1	2	1
1	2	2	0
2	1	0	0

Q3(f) /4	Fill in the `strat` table for the strategy described by this sentence. “If `s=0`, play `r`; if `s=1` and there was no tie, play `w`; if `s=2` and there was no tie, play `x`; in other cases, play `W(r)`.”

	0	1	2
0	0	0	0
1	1	0	2
2	2	1	1

Beating a strategy

In the following questions, a strategy A is fully given using `initA` and the `stratA` table.
Find a strategy B that wins with a score of 10 to 0 in a match against strategy A.
You must give the **first move `initB`** and **a minimum number of values in the `stratB` table**.
Leave empty the cells of `stratB` that will never be used by your strategy in a match against strategy A.
The row and column numbers are no longer written; add them if you feel the need.

Q3(g) /4	**Give a strategy B that wins 10 to 0 against strategy A.** **Only specify the necessary values in `stratB`.**

`initA=0, stratA=`

2	0	0
1	0	1
2	2	1

$\longrightarrow$ `initB=` 2 `, stratB=`

2		

Q3(h) /4	**Give a strategy B that wins 10 to 0 against strategy A.** **Only specify the necessary values in `stratB`.**

`initA=1, stratA=`

2	0	1
0	1	2
1	2	0

$\longrightarrow$ `initB=` 0 `, stratB=`

	2	
0		

Q3(i) /4	**Give a strategy B that wins 10 to 0 against strategy A.** **Only specify the necessary values in `stratB`.**

`initA=2, stratA=`

2	0	1
2	0	1
2	0	1

$\longrightarrow$ `initB=` 1 `, stratB=`

	1	
		2
0		

Q3(j) /4	**Give a strategy B that wins 10 to 0 against strategy A.** **Only specify the necessary values in `stratB`.**

`initA=1, stratA=`

1	1	2
0	1	0
0	2	2

$\longrightarrow$ `initB=` 0 `, stratB=`

	2	
		1
1		

Don't forget to **copy your solutions onto the answer sheets**.

The Tenzero robot

Professor Zarbi has modified a toy robot to create the Tenzero robot that wins all its matches 10 to 0.
His goal is to show the manufacturer that toy robots are not secure enough.

At the beginning of the match, Tenzero exploits a flaw in the wireless communication system to retrieve the strategy of its opponent. Before the first game starts, it adapts its strategy to ensure a 10 to 0 victory.

Tenzero uses the `Tenzero(initA, stratA)` program which receives the strategy A of the other robot and modifies certain values `(initTZ, stratTZ)` of its own strategy.
After these modifications, the match begins and Tenzero wins 10 to 0.
Can you complete the listing of the program used by Tenzero?

The program uses the `W(c)` function which receives a move `c` and returns the move that wins against `c`.
Remember that `0` beats `1` which beats `2` which beats `0`.

Q3(k) /10	Complete the ▭ in the **`Tenzero`** program.

```
Tenzero(initA, stratA)
{
  initTZ ← W(initA)

  stratTZ[0][1] ← W(stratA[1][0])

  stratTZ[1][2] ← W(stratA[2][1])

  stratTZ[2][0] ← W(stratA[0][2])
}
```

Don't forget to **copy your solutions onto the answer sheets**.