be-OI 2022 **Finale - SENIOR** Saturday 23rd April 2022	**Remplissez ce cadre en MAJUSCULES et LISIBLEMENT, svp** PRÉNOM : .... NOM : .... ÉCOLE :	O **Réservé**

Finals of the Belgian Informatics Olympiad 2022 (time allowed : **2h maximum**)

Solutions

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 **cadets**.
   - 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 (PRÉNOM), first name (NOM) and school (ÉCOLE) **on the first page only**.
3. Write **your answers** on the provided answer sheets, which you will find **at the end of the booklet**.
4. If you have to write outside the answer boxes due to a mistake, then continue writing **on the same sheet of paper**.
5. Write **clearly and legibly** with a blue or black **pen or ballpoint**.
6. You may only have writing materials with you. Calculator, GSM, . . . are **forbidden**.
7. You can always request extra scratch paper from the invigilator.
8. 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.
9. 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.
10. 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:

©2022 Olympiade Belge d'Informatique (beOI) ASBL
This work is made available under the terms of the Creative Commons Attribution 2.0 Belgium License

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 ← 2003
age ← 2020 − 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*. Note the comments in the code.

```
if (age ≥ 18)
{
    price ← 8 // This is a comment.
}
else
{
    price ← 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 ≥ 18)
{
    price ← 8 // Price for a person of legal age (adult).
}
else if (age ≥ 6)
{
    price ← 6 // Price for children aged 6 or older.
}
else
{
    price ← 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.

Solutions

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 ← 0
for (i ← 0 to N − 1 step 1)
{
    sum ← 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 ← 2
while (N ≥ 10)
{
    N ← N/d
    d ← 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.

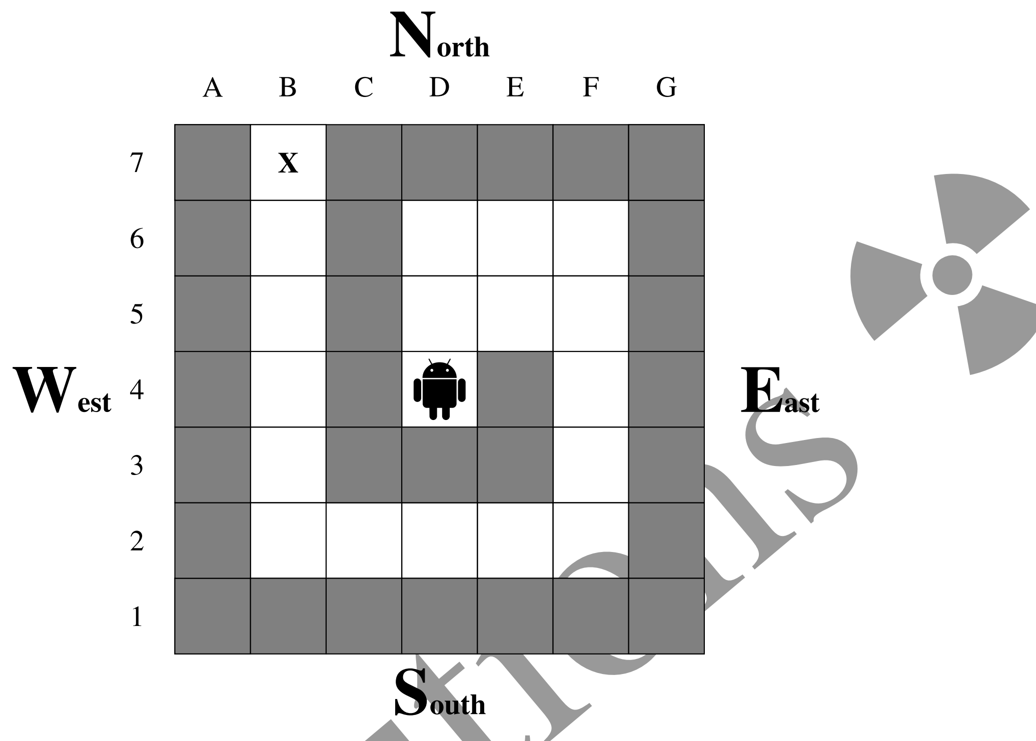
```
Input  : tab, an array of N numbers.
         N, the number of elements in the array.
Output : sum, the sum of all the numbers in the array.


sum ← 0
for (i ← 0 to N − 1 step 1)
{
    sum ← 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.

Solutions

Question 1 – Nono the little robot

Nono, the little robot, is stuck in the middle of a labyrinth.



Nono can move along the white squares but cannot land on grey squares, which represent walls. In the beginning, Nono stands in the middle of the labyrinth on the square with coordinates **D4**. Nono has to reach the exit: the square with coordinates **B7** marked with a **X**.

Nono moves one square at a time with the command `Go()`, which receives one of the following parameters: `N`, `S`, `E` ou `W`. Every parameter corresponds to one of the four cardinal points which are shown on the picture above, with North at the top according to the usual convention.

On the following page, we propose several algorithms to get Nono out of the maze. We want to know if Nono is on the square **X** at the end of the execution of each programs or if he collides with a wall (and, in this case, we ask the coordinates of the wall with which the collision takes place).

Some algorithms use arrays. For example `Dir←[N,E,S,W,N]` in algorithm 4 initialises an array with 5 elements, which are `Dir[0]=N`, `Dir[1]=E`, `Dir[2]=S`, `Dir[3]=W` and `Dir[4]=N`.

Solutions

Algorithm 1

```
Go(N)
for (i ← 1 to 2 step 1)
{
  Go(E)
}
for (i ← 1 to 3 step 1)
{
  Go(S)
}
for (i ← 1 to 4 step 1)
{
  Go(W)
}
for (i ← 1 to 5 step 1)
{
  Go(N)
}
```

Algorithm 2

```
Go(N)
i ← 1
while (i≤2)
{
  Go(E)
  i ← i+1
}
while (i≥0)
{
  Go(S)
  i ← i-1
}
while (i≤4)
{
  Go(W)
  i ← i+1
}
i ← 0
while (i<5)
{
  Go(N)
  i ← i+1
}
```

Algorithm 3

```
Go(N)
j ← 2
for (i ← 1 to j step 1)
{
  Go(E)
}
j ← j+1
for (i ← 1 to j step 1)
{
  Go(S)
}
j ← j+1
for (i ← 1 to j step 1)
{
  Go(W)
}
j ← j+1
for (i ← 1 to j step 1)
{
  Go(N)
}
```

Algorithm 4

```
Dir ← [N,E,S,W,N]
Stp ← [1,2,3,4,5]
for (j ← 0 to 4 step 1)
{
  for (i ← 1 to Stp[j] step 1)
  {
    Go(Dir[j])
  }
}
```

Algorithm 5

```
Dir ← [N,E,W,E,S,N,S,W,E,W,N]
Stp ← [1,2,1,1,2,1,2,4,2,1,5]
for (j ← 0 to 10 step 1)
{
  for (i ← 1 to Stp[j] step 1)
  {
    Go(Dir[j])
  }
}
```

Solutions

Q1(a) [5 pts]	**Does algorithm 1 bring Nono onto the square X ? If your answer is « no » and Nono collides with a wall, also provide the coordinates of the first wall that Nono collides with.**
Solution: Yes	

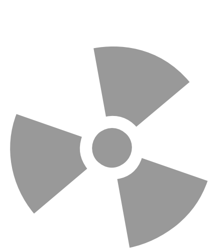
Q1(b) [5 pts]	**Does algorithm 2 bring Nono onto the square X ? If your answer is « no » and Nono collides with a wall, also provide the coordinates of the first wall that Nono collides with.**
Solution: No, F1	

Q1(c) [5 pts]	**Does algorithm 3 bring Nono onto the square X ? If your answer is « no » and Nono collides with a wall, also provide the coordinates of the first wall that Nono collides with.**
Solution: Yes	

Q1(d) [5 pts]	**Does algorithm 4 bring Nono onto the square X ? If your answer is « no » and Nono collides with a wall, also provide the coordinates of the first wall that Nono collides with.**
Solution: Yes	

Q1(e) [5 pts]	**Does algorithm 5 bring Nono onto the square X ? If your answer is « no » and Nono collides with a wall, also provide the coordinates of the first wall that Nono collides with.**
Solution: No, C3	

Solutions



Question 2 – Peak positions

Given an array $a[\,]$ of n numbers, a *peak position of* $a[\,]$ is a position i in that array such that $a[i]$ is at least as large as both the element to its left and the element to its right (if they exist). Note that arrays are 0-indexed: the first element is at position 0 and the last element is at position $n-1$.

For example, if $a = [2, 7, 4, 5]$, then the peak positions of $a[\,]$ are the positions 1 and 3 (corresponding to the values 7 and 5). On the other hand, if $a = [6, 6, 6]$, then all the positions (0, 1 and 2) are peak positions. It is easy to show that any array has at least one peak position.

Q2(a) [2 pts]	**Give *all* the peak positions of array $a = [4, 2, 5, 3]$.**
Solution: 0, 2	

Q2(b) [2 pts]	**Give *all* the peak positions of array $a = [0, 10, 10, 0, 0]$.**
Solution: 1, 2 and 4	

There are quite a few ways to find a peak position, but it is very easy to make mistakes! Your friend Alex sent you the following program listings last night at 3am. He claims that his programs can find a peak position of any array $a[\,]$. You figure that Alex was probably very tired when writing the programs and you decide to check them.
It is assumed that in each program, the arrray of numbers $a[\,]$ has $n \geq 2$ elements.

Program 1:

```
for (i ← 1 to n-2 step 1)
{
    if (a[i] >= i-1 and a[i] >= i+1)
    {
        return i
    }
}
if (a[0] > a[n-1])
{
    return 0
}
else
{
    return n-1
}
```

Q2(c) [1 pt]	**What value does program 1 return if $a = [9, 8, 7, 6, 5]$?**
Solution: 1	

Q2(d) [1 pt]	**What value does program 1 return if $a = [0, 1, 3, 4]$?**
Solution: 2	

Q2(e) [1 pt]	**What value does program 1 return if $a = [0, 0, 0, 0]$?**
Solution: 3	

Solutions

Q2(f) [1 pt]	**What value does program 1 return if $a = [1, 0, 0, 1]$?**
Solution: 3	

Q2(g) [2 pts]	**Does program 1 return a peak position for any array $a[\,]$?**
Solution: No	

Program 2:

```
i ← 0
for (j ← 1 to n-1 step 1)
{
    if (a[j] > a[i])
    {
        i ← j
    }
}
return i
```

Q2(h) [1 pt]	**What value does program 2 return if $a = [4, 3, 2, 1]$?**
Solution: 0	

Q2(i) [1 pt]	**What value does program 2 return if $a = [0, 1, 0, 5, 0, 5, 0]$?**
Solution: 3	

Q2(j) [2 pts]	**Does program 2 return a peak position for any array $a[\,]$?**
Solution: Yes	

Program 3:

```
if (a[0] > a[n-1])
{
    return 0
}
else
{
    return n-1
}
```

Q2(k) [1 pt]	**What value does program 3 return if $a = [1, 2, 3]$?**
Solution: 2	

Q2(l) [1 pt]	**What value does program 3 return if $a = [2, 2, 1]$?**
Solution: 0	

Solutions

Q2(m) [2 pts]	**Does program 3 return a peak position for any array *a*[ ]?**
Solution: No	

Solutions

Program 4:

```
for (i ← 0 to n-2 step 1)
{
    if (a[i] >= a[i+1])
    {
        return i
    }
}
return n-1
```

Q2(n) [1 pt]	What value does program 4 return if $a = [1, 2, 3, 2, 1]$?
Solution: 2	

Q2(o) [1 pt]	What value does program 4 return if $a = [0, 1, 0, 5, 0]$?
Solution: 1	

Q2(p) [1 pt]	What value does program 4 return if $a = [2, 2, 1, 0, 5]$?
Solution: 0	

Q2(q) [2 pts]	Does program 4 return a peak position for any array $a[\,]$?
Solution: Yes	

Program 5:

```
i = n / 2
while (i > 0 and a[i-1] > a[i])
{
    i ← i-1
}
while (i < n-1 and a[i+1] > a[i])
{
    i ← i+1
}
return i
```

Q2(r) [1 pt]	What value does program 5 return if $a = [1, 2, 3, 4, 5]$?
Solution: 4	

Q2(s) [1 pt]	What value does program 5 return if $a = [5, 4, 3, 2, 1]$?
Solution: 0	

Q2(t) [1 pt]	What value does program 5 return if $a = [4, 3, 2, 2, 6, 5]$?
Solution: 4	

Solutions

Q2(u) [1 pt]	**What value does program 5 return if $a = [0, 4, 2, 5, 3]$?**
Solution: 1	

Q2(v) [2 pts]	**Does program 5 return a peak position for any array $a[\,]$?**
Solution: Yes	

Solutions

The next day, Barbara sent you another program for finding a peak position. It looks intriguing, but you're confident that Barbara wrote it correctly (not least because, unlike Alex, she didn't send it to you in the middle of the night). However, never one to miss an opportunity to challenge her friends' deduction skills, Barbara left some blanks to fill in.

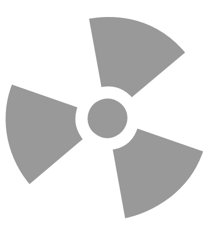
```
l = 0
r = ...           // (e1)
while (...)       // (e2)
{
    mid ← (l + r) / 2
    if (...)      // (e3)
    {
        l ← mid + 1
    }
    else
    {
        r ← mid
    }
}
return l
```

Q2(w) [2 pts]	What is expression (e1) in Barbara's program?
Solution: n-1	

Q2(x) [2 pts]	What is expression (e2) in Barbara's program?
Solution: l<r (other solution l≠r)	

Q2(y) [2 pts]	What is expression (e3) in Barbara's program?
Solution: a[mid] <= a[mid+1] or a[mid] < a[mid+1]	

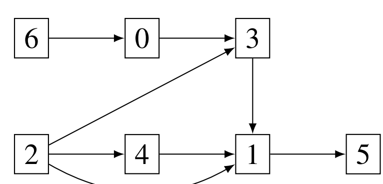
Solutions

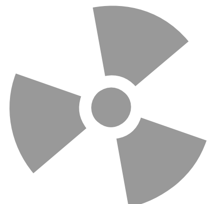


Question 3 – The computer factory

The company be-OI father and son manufactures computer parts. A computer component is a complex object and requires, to be manufactured, other components, which are themselves assembled from other components, *etc*. The company has all the machines to manufacture everything, but cannot make them all work at the same time. So it has to find *the right order to make the parts, assuming that each machine will only be operated once*.

Each machine has a number from 0 to $n-1$ and the company has represented the dependencies between the machines using a diagram as shown in the following example.





On this diagram, each machine is represented by a rectangle with the machine number. An arrow going from machine i to machine j means that machine i must run before machine j, because the parts produced by i are necessary for those produced by j (we don't care about the quantities here: it is enough for i to run only once to allow j to run). For example, it is mandatory to run machine 2 and machine 0 before machine 3. Any operating order of the machines where 3 is run before 2 or before 0 is therefore impossible. In order for the diagram to make sense, we assume that it never contains a *cycle*, i.e. that it is never possible to start from a machine and follow a series of arrows to return to the same machine.

As you can see, some machines can be started up immediately.

Q3(a) [1 pt]	According to the diagram above, what are all the machines can be started immediately?
Solution: 2 and 6	

Q3(b) [1 pt]	According to the diagram above, what are all the machines that can be started if only machine 2 has been run (remember: each machine is only run once) ?
Solution: 4 and 6	

Which of the following orders of operating the machines are possible within the constraints given in the above diagram ?

	Possible	Impossible	Order of operating the machines
Q3(c) [1 pt]	☐	☒	0,1,2,3,4,5,6
Q3(d) [1 pt]	☒	☐	6,0,2,3,4,1,5
Q3(e) [1 pt]	☐	☒	6,5,4,3,2,1,0
Q3(f) [1 pt]	☒	☐	2,4,6,0,3,1,5
Q3(g) [1 pt]	☐	☒	2,4,1,6,0,3,5

Clearly, there is a very specific condition that the machines must meet (in a diagram like the one above) so that we can make them work immediately.

Solutions

Q3(h) [2 pts]		**Among the propositions below, which specifically characterizes the machines that can be operated immediately?**
	☐	Those machines that have an even number of incoming arrows.
	☐	Those machines that have at most 2 incoming arrows.
	☒	Those machines that have no incoming arrows.
	☐	Those machines that have at least one incoming arrow.
	☐	Those machines that have no incoming arrows and an even (machine) number.

We are now looking for an algorithm to find an order to operate the machines, which is compatible with the diagram. The diagram is given as a matrix of Booleans $M[n][n]$, where $M[i][j]$ is equal to **`true`** if and only if there is an arrow going from machine i to machine j.

Here is the matrix for the example diagram.
Boxes with a value of **`true`** have been highlighted to improve readability.

	0	1	2	3	4	5	6
0	false	false	false	**true**	false	false	false
1	false	false	false	false	false	**true**	false
2	false	**true**	false	**true**	**true**	false	false
3	false	**true**	false	false	false	false	false
4	false	**true**	false	false	false	false	false
5	false	false	false	false	false	false	false
6	**true**	false	false	false	false	false	false

M[ ][ ]

$M[2][4]$ is surrounded by a solid line, while $M[4][2]$ is surrounded by a dotted line.
$M[2][4]$ = **`true`** because there is an arrow going from machine 2 to machine 4.
$M[4][2]$ = **`false`** because there is no arrow going from machine 4 to machine 2.

The proposed algorithm is based on a data structure called *queue*. It is a queue, with a beginning and an end. We add the elements at the end of the queue and we process the elements starting with the one at the beginning of the queue. We have the functions: `Empty()`, which empties the queue (it no longer contains any elements); `Insert(v)`, which adds the element v to the **end** of the queue; `Remove()`, which returns the element of the **start** of the queue and removes it from the queue; and finally `IsEmpty()` which returns **`true`** if the queue is empty and **`false`** if it contains at least one element.

The algorithm is the following

1. We start by inserting into the queue all the machines that we can immediately start running.
2. Then we repeat the same actions until the queue is empty:
   (a) take a machine at the beginning of the queue and run the machine;
   (b) if operating this machine makes it possible for other machines to start running, add these new machines to the queue.

Remarks.
A boolean value **bool** can be used directly in an instruction **if** (**bool**){...}.
The **not** operator acts as follows on Boolean values: **not true** is equal to **false** and **not false** is equal to **true**.

Here is an implementation of the algorithm where some parts of the code are missing:

```
Input : A matrix M[n][n] of size n × n, which represents the dependencies between the
    n machines.
Output : Runs the machines in an order compatible with M.

Empty()
D ← [0,...,0] // (an array of length n, initialized with all elements equal to 0)

for (j ← 0 to n−1 step 1)
{
  for (i ← 0 to n−1 step 1)
  {
    if(M[i][j])
    {
      D[...] ← ... // (A) and (B)
    }
  }
  if (...) // (C)
  {
    Insert(j)
  }
}

while (not IsEmpty())
{
  m ← Remove()
  Operating the machine m
  for (i ← 0 to n−1 step 1)
  {
    if (M[m][i])
    {
      D[i] ← ... // (D)
      if (...) // (E)
      {
        Insert(i)
      }
    }
  }
}
```

Q3(i) [2 pts]	Which expression **(A)** is missing in the algorithm ?
	Solution: j

Q3(j) [2 pts]	Which expression **(B)** is missing in the algorithm ?
	Solution: $D[j]+1$

Solutions

Q3(k) [2 pts]	Which condition **(C)** is missing in the algorithm ?
Solution: $D[j] = 0$	

Q3(l) [2 pts]	Which expression **(D)** is missing in the algorithm ?
Solution: $D[i] - 1$	

Q3(m) [2 pts]	Which condition **(E)** is missing in the algorithm ?
Solution: $D[i] = 0$	

Finally, we would like to build the queue and its four functions used in the algorithm using an array $f[n]$ of size n and two indices s and e. The elements that are in the queue will be all the elements with index $s, s+1, \ldots, e-1$. The element with index s is the one at the beginning and the one with index $e-1$ is at the end. For example, if $s = 3$ and $e = 6$, the beginning of the queue will be $f[3]$, then the queue will contain the elements $f[4]$ and $f[5]$. If one of the indices reaches the last cell $n-1$ of the array, we can still extend the queue by processing the array in a circular way, which is the same as setting the index back to 0. For example, if $n = 7$, $s = 5$ and $e = 2$, the queue will contain the elements $f[5], f[6], f[0], f[1]$, in this order, with $f[5]$ at the beginning. You are asked to complete the code below to obtain this implementation of the queue. Hint: use the mathematical function *modulo* (represented by the operator %): $a\%b$ is the remainder of the division of a by b.

```
Empty()
{
  s ← 0
  e ← 0
}

Insert(v)
{
  f[e] ← v
  e ← ... // (F)
}

IsEmpty()
{
  return (...) // (G)
}

Remove()
{
  v ← f[s]
  s ← ... // (H)
  return v
}
```

Q3(n) [2 pts]	Which expression **(F)** is missing in the algorithm ?
Solution: $(e+1)\%n$	

Solutions

Q3(o) [2 pts]	**Which condition `(G)` is missing in the algorithm ?**
Solution: $e = s$	

Q3(p) [2 pts]	**Which expression `(H)` is missing in the algorithm ?**
Solution: $(s + 1)\%n$	

Solutions

Question 4 – LIDAR

LIDAR is a technology that uses light or lasers to determine distances or heights of objects. You are part of a team that wants to map out how high exactly different parts of the Great Wall of China are. While flying high above the Wall, you suddenly notice that your LIDAR equipment has a problem: instead of measuring the height of a single segment, it measures and sums together the heights of several consecutive segments. You only had permission to fly over the Wall today, and there is no time to go back to reconfigure the LIDAR. As you are the algorithmic expert of the team, your colleagues look to you to recover the actual heights of the segments from the data gathered in the scan.

For simplicity, we represent the Wall as a list $W[\,]$ of n positive or null integers ($W[i] \geq 0$).
Each $W[i]$ is the height of a segment of the Wall.
The LIDAR will take q scans, each will be the sum of consecutives elements of $W[\,]$.

Example: take for instance a wall with $n = 4$ segments: $W = [4, 3, 5, 8]$.
As usual the index start at 0, so here $W[0] = 4$, $W[1] = 3$, $W[2] = 5$ and $W[3] = 8$.
The results of 2 scans by the LIDAR could be (pay attention to the notation $W[i \ldots j] = W[i] + \ldots + W[j]$):
$W[1 \ldots 3] = W[1] + W[2] + W[3] = 3 + 5 + 8 = 16$ and $W[0 \ldots 1] = W[0] + W[1] = 4 + 3 = 7$.

For the following four questions, consider a wall $W[\,]$ with 4 segments.
The LIDAR has measured $W[0 \ldots 2] = 31$, $W[0 \ldots 3] = 42$ and $W[2 \ldots 3] = 17$.

Q4(a) [1 pt]	How high is $W[3]$?
Solution: 11, as $W[3] = W[0 \ldots 3] - W[0 \ldots 2] = 42 - 31 = 11$	

Q4(b) [1 pt]	How high is $W[2]$?
Solution: 6, as $W[2] = W[0 \ldots 2] + W[2 \ldots 3] - W[0 \ldots 3] = 31 + 17 - 42 = 6$	

Q4(c) [1 pt]	What is the value of $W[0 \ldots 1]$?
Solution: 25, as $W[0 \ldots 1] = W[0 \ldots 3] - W[2 \ldots 3] = 42 - 17 = 25$	

Q4(d) [1 pt]	How many different walls are compatible with the LIDAR measurements ?
Solution: 26, since only $W[0]$ and $W[1]$ are unknown but they must sum to 25 (don't forget that a height can be equal to 0).	

Before solving the real problem, we first try to simulate the measurements effectively. This will allow us to generate test cases to check whether we have done things correctly.

Let's first implement a simple, but slow algorithm. The inputs are the wall $W[\,]$ and its length n, the number of scans q, and the left and right endpoints for the scans in the tables $left[\,]$ and $right[\,]$. The output is an array $Lidar[\,]$ whose elements are the sums $W[left[i] \ldots right[i]]$ measured by the scans.

```
Input  : n, W[], q, left[], right[]
Output : Lidar[]

Lidar[] is initialized to q zeroes.

for (i ← 0 to ... step 1) // (i)
{
    for (j ← left[i] to ... step 1) // (ii)
    {
```

Solutions

```
        Lidar[i] ← Lidar[i] + ... //(iii)
    }
}
```

There are a few blanks in this implementation, can you fill them in?

Q4(e) [1 pt]	What is expression (i) in the above algorithm ?
Solution: $q-1$	

Q4(f) [1 pt]	What is expression (ii) in the above algorithm ?
Solution: $right[i]$	

Q4(g) [1 pt]	What is expression (iii) in the above algorithm ?
Solution: $W[j]$	

Here is a faster algorithm, with only a single pass over both the values of $W[\,]$ and of $left[\,]$ and $right[\,]$. The idea is to preprocess a prefix table $P[\,]$ such that $P[0]=0$ and $P[i+1]=W[0\dots i]$ and then use it to calculate $Lidar[\,]$. The inputs and output are the same as in the first algorithm.

```
Input  : n, W[], q, left[], right[]
Output : Lidar[]

P[] ← [0,0,---,0] //n+1 "0".

for (i ← 0 to ... step 1) // (i)
{
    P[i + 1] ← ...   // (ii)
}

for (i ← 0 to ... step 1) // (iii)
{
    Lidar[i] ← ... // (iv)
}
```

You just have to fill in a few blanks.

Q4(h) [1 pt]	What is expression (i) in the above algorithm ?
Solution: $n-1$	

Q4(i) [2 pts]	What is expression (ii) in the above algorithm ?
Solution: $P[i]+W[i]$	

Q4(j) [1 pt]	What is expression (iii) in the above algorithm ?
Solution: $q-1$	

Solutions

Q4(k) [3 pts]	What is expression (iv) in the above algorithm ?
Solution: $P[right[i]+1]-P[left[i]]$	

If we could calculate the $P[\,]$ table in the previous algorithm from the scans, we could easily deduce all the $W[i]$ values. Let's try to do it effectively. The inputs are the number of segments n in the wall, the number of scans q, the endpoints of the scans in the tables $left[\,]$ and $right[\,]$, and the values of the scans in $Lidar[i] = W[left[i] \dots right[i]]$. **Important note: for this algorithm, we assume that the scans are all consistent (that is, there are no contradictions), and that the scans uniquely determine $P[\,]$.**

The $K[\,][\,]$ array will be used to contain all measurements we know for each endpoints of each scan. As an example, if a scan is $W[3\dots7] = 61$, then we know that $P[8] = W[0\dots7] = W[0\dots2] + W[3\dots7] = P[3] + 61$ and also that $P[3] = P[8] - 61$. To take that into account, we register the constraint $(8, 61)$ in the list $K[3]$ and the constraint $(3, -61)$ in the list $K[8]$. Pay close attention to the indices !

Note that $K[\,][\,]$ is an array of lists and that the elements in each list $K[i][\,]$ are couples of numbers $(indice, value)$.
The line **foreach** `((index, scan)in K[current])` in the code will start a loop which will be executed once for each element in the list $K[current][\,]$. The values of `index` and `scan` can be used into that loop.

The `append` x `to` y operation in the code adds x as a new element to the right of the list y. For example, if $x = 4$ and $y = [6, 2]$, afterwards y will be $[6, 2, 4]$.
The inverse operation is `remove last element` : it transforms $y = [6, 2, 4]$ into $[6, 2]$ and assigns 4 to the specified variable.

```
Input  : n, q, left[], right[], Lidar[]
Output : P[]

Initialize K[][] to n+1 empty arrays.

for (i ← 0 to q - 1 step 1)
{
    append (..., Lidar[i]) to K[left[i]]     // (i)
    append (...) to K[...] // (ii), (iii)
}

Initialize processed[] to n values false.
P[0] ← 0
processed[0] ← true
Initialize todo[] to [0]

while (todo[] is not empty)
{
    current ← remove last element of todo[]
    foreach ((index, scan) in K[current][])
    {
        P[index] ← ... // (iv)
        if (not processed[index])
        {
            processed[index] ← true
            append index to todo
        }
    }
}
```

Solutions

Q4(l) [1 pt]	**What is expression (i) in the above algorithm ?**
Solution: $right[i] + 1$	

Q4(m) [3 pts]	**What is expression (ii) in the above algorithm ?**
Solution: $left[i], -Lidar[i]$	

Q4(n) [1 pt]	**What is expression (iii) in the above algorithm ?**
Solution: $right[i] + 1$	

Q4(o) [5 pts]	**What is expression (iv) in the above algorithm ?**
Solution: $P[current] + scan$	

Solutions

The following scans of **a wall with 9 segments** are consistent and uniquely determine $P[\,]$ (you can use the previous algorithm) and $W[\,]$.

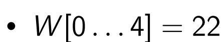
- $W[0\ldots0] = 4$
- $W[0\ldots3] = 18$
- $W[0\ldots5] = 20$
- $W[1\ldots4] = 14$
- $W[2\ldots8] = 23$
- $W[3\ldots4] = 1$
- $W[4\ldots7] = 15$
- $W[5\ldots7] = 15$
- $W[5\ldots8] = 18$
- $W[7\ldots8] = 7$
- $W[8\ldots8] = 3$

Q4(p) [4 pts]	What are the heights of the 9 segments in the wall $W[\,]$?
Solution: $[4, 9, 4, 1, 0, 2, 9, 4, 3]$	

Solutions

The following scans of **a wall with 9 segments** are consistent but don't uniquely determine $W[\,]$.

- $W[0\ldots3] = 22$
- $W[0\ldots4] = 22$
- $W[1\ldots1] = 2$
- $W[1\ldots4] = 16$
- $W[1\ldots5] = 24$
- $W[2\ldots4] = 14$
- $W[3\ldots6] = 19$
- $W[3\ldots8] = 30$
- $W[7\ldots8] = 11$



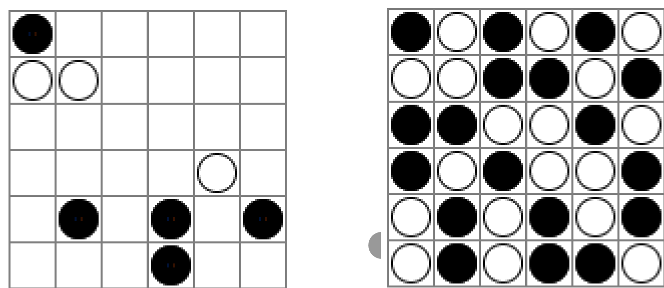
Q4(q) [4 pts]	How many walls are consistent with the given scans ?
Solution: 144, any wall $[6, 2, 14-x, x, 0, 8, 11-x, y, 11-y]$ where x and y can each take any value from 0 to 11 is possible.	

Question 5 – Binairo

Binairo alias **Takuzu** is a logic game with simple rules but complex solutions.
The game is played on a square grid of even size ($2n$ rows of $2n$ cells).
At the start, some cells contain a piece, black or white. All other cells are empty.
The goal is to put a piece in each cell according to the following rules.

- **Rule 1:** Each row or column must contain n white pieces and n black pieces.
- **Rule 2:** In each row or column, you can have two consecutive pieces of the same color, but not three.
- **Rule 3:** Each row and each column is unique (we won't need this rule in what comes next).

Here is an example of a 6x6 binairo grid (left) and the unique solution (right).



Below are 2 examples of rows that do not respect the rules.

○○●○●○○● Rule 1 not respected (there are more white pieces than black ones).

○○●●●○○● Rule 2 not respected (there are more than 2 consecutive black pieces).

Complete the following rows **while respecting the first 2 rules**.
If necessary, answer here in draft form before **copying to the answer sheet**.

Q5(a) [1 pt]	**Complete the row** ●● _ _
	Solution: ●●○○

Q5(b) [1 pt]	**Complete the row** ○ _ _ ●● _ _ ○
	Solution: ○●○●●○●○

Q5(c) [1 pt]	**Complete the row** ● _ ● _ _ _ _ _ ●●
	Solution: ●○●○○●○○●●

In how many ways can the following rows be completed **while respecting the first 2 rules** ?

Q5(d) [1 pt]	**Number of ways to complete** ○● _ _
	Solution: 2

Solutions

Question		Grid
Q5(e) [1 pt]	**Number of ways to complete**	○ ○ ☐ ☐
Solution: 1		

Question		Grid
Q5(f) [1 pt]	**Number of ways to complete**	☐ ○ ☐ ☐ ● ☐
Solution: 3		

Question		Grid
Q5(g) [1 pt]	**Number of ways to complete**	○ ☐ ☐ ● ○ ☐ ☐ ●
Solution: 4		

Question		Grid
Q5(h) [1 pt]	**Number of ways to complete**	○ ○ ● ○ ☐ ☐ ☐ ☐
Solution: 2		

Question		Grid
Q5(i) [1 pt]	**Number of ways to complete**	○ ☐ ○ ☐ ☐ ☐ ☐ ☐ ○ ○
Solution: 1		

Question		Grid
Q5(j) [1 pt]	**Number of ways to complete**	○ ☐ ☐ ○ ☐ ☐ ○ ☐ ☐ ○
Solution: 6		

Question		Grid
Q5(k) [1 pt]	**Number of ways to complete**	● ☐ ○ ○ ☐ ● ☐ ○ ☐ ●
Solution: 0		

Question		Grid
Q5(l) [1 pt]	**Number of ways to complete**	☐ ☐ ☐ ☐
Solution: 6		

Question		Grid
Q5(m) [1 pt]	**Number of ways to complete**	☐ ☐ ☐ ☐ ☐ ☐
Solution: 14		

Question		Grid
Q5(n) [1 pt]	**Number of ways to complete**	○ ○ ● ☐ ☐ ☐ ☐ ☐ ☐ ● ○ ○
Solution: 1		

Question		Grid
Q5(o) [1 pt]	**Number of ways to complete**	☐ ○ ● ☐ ☐ ○ ☐ ☐ ● ☐ ☐ ○
Solution: 5		

Solutions

The **program 1** checks if a row of black and white pieces respects the first 2 rules. Some parts of the code are missing, they must be completed.

The row to check is represented by an array of integers $binairo[\,]$. White pieces are represented by 1 and black pieces by 2. A number n is also given: the array $binairo[\,]$ contains $2n$ elements (numbered from 0 to $2n-1$) and according to **Rule 1**, there must be the same number n of white and black pieces. The program must return **true** if the first two rules are respected and **false** otherwise.

Examples.

- For ●○○●○●○●●○ $n = 5$, $binairo = [2, 1, 1, 2, 1, 2, 1, 2, 2, 1]$, the program returns **true**.
- For ○●●●○○ $n = 3$, $binairo = [1, 2, 2, 2, 1, 1]$, the program returns **false**.
- For ●○●●○●○● $n = 4$, $binairo = [2, 1, 2, 2, 1, 2, 1, 2]$, the program returns **false**.

The program traverses $binairo[\,]$ and uses the variable $t1$ to count the **total** number of white pieces and the variable $c1$ to count the number of **consecutive** white pieces among the last ones scanned. The variables $t2$ and $c2$ do the same for the black pieces.

Program 1: The solutions follow after the . . ..

```
Input  : n, binairo[]
Output : true or false
t1 ← 0; t2 ← 0; c1 ← 0; c2 ← 0
for (i ← 0 to 2*n-1 step 1){
    if (binairo[i] = 1) {
        t1 ← ...t1+1
        if (t1 > ...n) {return false}
        c1 ← ...c1+1
        if (c1 = ...3) {return false}
        c2 ← ...0
    }
    else {
        t2 ← ...t2+1
        if (t2 > ...n) {return false}
        c2 ← ...c2+1
        if (c2 = ...3) {return false}
        c1 ← ...0
    }
}
return ...true
```

Q5(p) [6 pts]	**Complete the . . . in program 1.** Score between 0 and 6. You lose 1 point for each mistake.
Solution: The solutions follow after the . . .	

Solutions

Program 2 is an improvement on Program 1. It allows to check if a row, *which may still have some empty cells*, respects the first 2 rules. Attention: the program does not try to check if it is possible to fill the row while respecting the rules, it only checks if the first 2 rules are respected *by the pieces that are already in the row*.

The array $binairo[\,]$ can contain 0 which represent empty cells (remember: 1 and 2 represent pieces). The variables $t1$ and $t2$ are replaced by an array $t[\,]$ with 3 elements. $t[0]$ is unused, $t[1]$ counts the total number of white pieces, $t[2]$ counts the total number of black pieces. The variables $c1$ and $c2$ are replaced by an array $c[\,]$ with 3 elements. $c[0]$ is unused, $c[1]$ counts the number of consecutive white pieces, $c[2]$ counts the number of consecutive black pieces.

Examples.

- For | ● | ○ | | | ○ | | | | | ○ | $n = 5$, $binairo = [2, 1, 0, 0, 1, 0, 0, 0, 0, 1]$ and the program returns **true**.
- For | | ● | ● | ● | | ○ | $n = 3$, $binairo = [0, 2, 2, 2, 0, 1]$, the program returns **false** because there are more than two consecutive black pieces in a row.
- For | ● | | ● | ● | | ● | | ● | $n = 4$, $binairo = [2, 0, 2, 2, 0, 2, 0, 2]$, the program returns **false** because there are too many black pieces.

Program 2: The solutions follow after the . . .

```
Input  : n, binairo[]
Output : true or false
t ← [0,0,0]
c ← [0,0,0]
for (i ← 0 to 2*n-1 step 1){
    j = binairo[i]
    if (...j ≠ 0) {
        t[j] ← ...t[j]+1
        if (...t[j] > n) {return false}
        c[j] ← ...c[j]+1
        if (...c[j] = 3) {return false}
        c[...3-j] ← ...0
    }
    else {
      c[1] ← 0
      c[2] ← 0
    }
}
return ...true
```

Solutions

Q5(q) [6 pts]	**Complete the . . . in program 2.** **Score between 0 and 6. You lose 1 point for each mistake.**
Solution: The solutions follow after the . . .	

Program 3 counts the number of ways to fill a row with n white pieces and n black pieces (respecting the first 2 rules, we won't remind it anymore).

Let us illustrate the idea of the program in the case where $n = 2$.
Notation: we write # in front of a line to indicate the number of ways to complete this line. We have successively :

□□□□ = # ○□□□ + # ●□□□
= (# ○○□□ + # ○●□□) + (# ●○□□ + # ●●□□)
= (# ○○●□ + (# ○●○□ + # ○●●□)) + ((# ●○○□ + # ●○●□) + # ●●○□)
= (# ○○●● + (# ○●○● + # ○●●○)) + ((# ●○○● + # ●○●○) + # ●●○○)
$= \Big(1 + \big(1 + 1\big)\Big) + \Big(\big(1 + 1\big) + 1\Big) = 6$

Program 3 uses only one input variable: the number n.
We first define a function $TotalBinairo(i, t1, c1, t2, c2)$ that calculates the number of ways to complete a row, knowing that:

- the cells with a number lower than i are already filled,
- there are already $t1$ white pieces and $t2$ black pieces,
- the last placed cells are $c1$ consecutive white pieces (and in this case $c2 = 0$) or $c2$ consecutive black pieces (and in this case $c1 = 0$).

To perform this calculation, the function first checks if $i = 2n$, which means that the row is filled. In this case, the value 1 is returned to count this way of filling the row. Otherwise, the function checks if a white piece can be placed in cell i. If this is the case, it calls itself with adapted values of the parameters and receives the number of ways to complete the cells that remain after this new white piece. Then the function does the same by trying to place a black piece in cell i. If this is not possible (no white or black piece can be added to cell i), the function must return 0.

The last line of the program calls the function to calculate the number of ways to complete an empty row.

Programme 3: The solutions follow after the . . .

```
TotalBinairo( i, t1, c1, t2, c2){
    if (i = 2*n){return 1}
    else {
        TotBin ← 0
        if (t1<n and c1<2){
            TotBin ← TotBin + TotalBinairo( ...i+1, ...t1+1, ...c1+1, ...t2, ...0)
        }
        if (...t2<n and c2<2){
            TotBin ← TotBin + TotalBinairo( ...i+1, ...t1, ...0, ...t2+1, ...c2+1)
        }
        return ...TotBin
    }
}
return TotalBinairo(0, 0, 0, 0, 0)
```

Q5(r) [6 pts]	**Complete the . . . in program 3.** **Score between 0 and 6. You lose 1 point for each mistake.**
Solution: The solutions follow after the . . .	

Solutions