be-OI 2023 **Final - JUNIOR** Saturday, March 18, 2023	**Fill in this box in CAPITAL LETTERS please** FIRST NAME : ..... LAST NAME : ..... SCHOOL :	O **Reserved**

Finals of the Belgian Informatics Olympiad 2023 (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 ← 2006
age ← 2023 − 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 ≥ 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.

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.

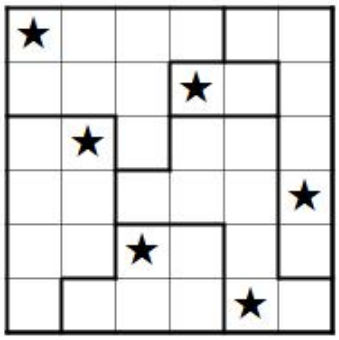
Question 1 – Star Battle

Star Battle is a logic puzzle where one needs to place **n** stars in a grid with **n** rows and **n** columns, divided into **n** zones separated by walls which are represented by thick lines

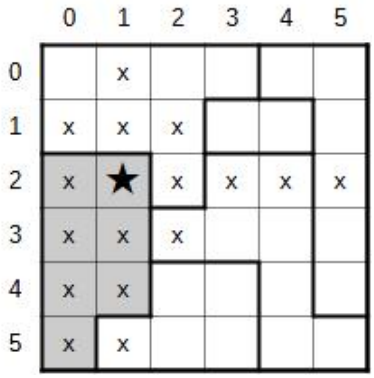
The following rules must be respected.

- **Rule 1:** Each row must contain 1 star.
- **Rule 2:** Each column must contain 1 star.
- **Rule 3:** Each zone must contain 1 star.
- **Rule 4:** The cells that surround a star cannot contain another star.

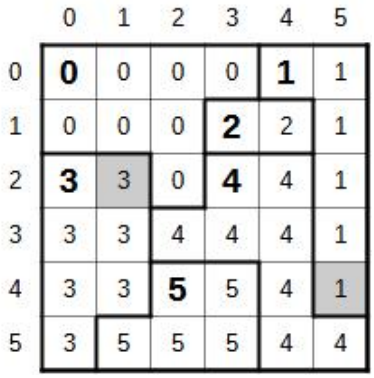
For example: the grid on the right has been filled up according to the four rules.



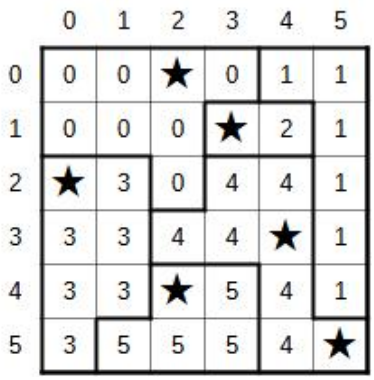
Here is an example with **n=6** where only one star has been placed.
The row numbers (from 0 to 5) are displayed on the left of the grid.
The column numbers (from 0 to 5) are displayed above the grid.
The star has been placed in the cell at the intersection of row 2 and column 1.
The cells where no further star can be placed are marked with 'x':
• the cells in row 2 (rule 1),
• the cells in column 1 (rule 2),
• the (gray) cells in the zone that contains the star (rule 3),
• the 8 cells surrounding the star (rule 4).



Zones are numbered from **0** to **n-1** by moving through the grid according to the English reading direction (left-right), starting from the top left corner.
We write **0** in all the cells of the first zone. We write **1** in all the cells of the next zone that has no numbers yet, and we continue like that until we write **n-1** in the last zone (zone **5** in the example).
The first cells of each zone, according to the reading direction, are displayed in bold.
In the programs, the grid is represented by the array **t[ ][ ]** containing the zone numbers.
• **t[2][1]=3**, because the cell in row 2 and column 1 (in gray) is in zone 3.
• **t[4][5]=1**, because the cell in row 4 and column 5 (in gray) is in zone 1.

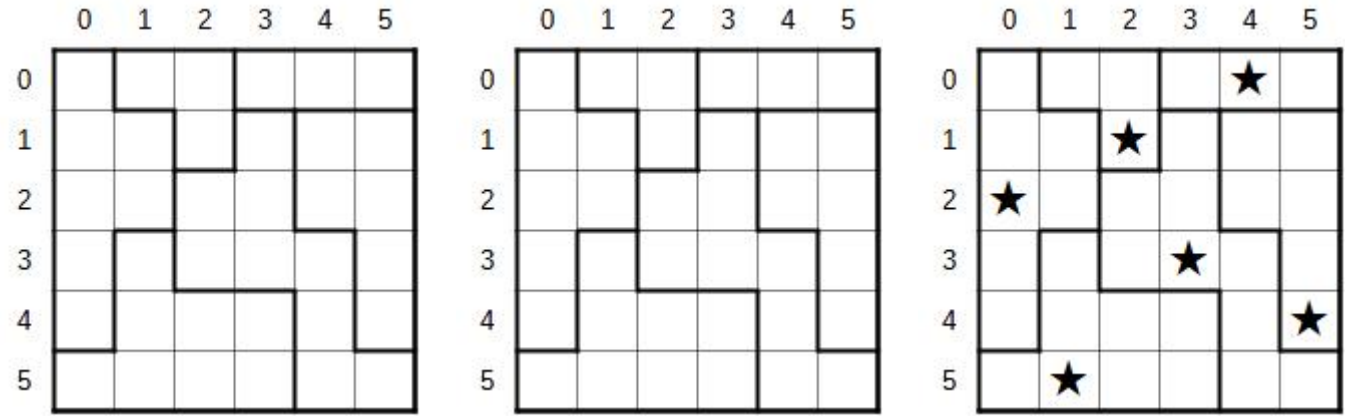


The player can suggest a solution by means of an array **col[ ]**.
For each row number **i**, the player must give the column number **col[i]** where they want a star to be placed.
In the example on the right, **col[ ]=[2,3,0,4,2,5]**
(this is a shorthand for **col[0]=2**, **col[1]=3**, **col[2]=0**, **col[3]=4**, **col[4]=2** and **col[5]=5**).



This suggestion is very bad since only rule 1 is enforced.
• Rule 2 is violated as there are multliple stars in column 2.
• Rule 3 is violated as there are multiple stars in zone 4.
• Rule 4 is violated by the stars in row 0 and row 1.

The questions on this page rely on the **Star Battle** grid which is given three times hereunder.
(The answer is given in the grid on the right.)



Q1(a) /4	**What are the values of the following cells from t[ ][ ] on this grid?** `t[1][5]=..., t[2][3]=..., t[3][0]=..., t[4][3]=...`
Solution : `t[1][5]`=4, `t[2][3]`=3, `t[3][0]`=0, `t[4][3]`=5	

For the next four questions, you need to **tick** the rules which **are respected**, and *avoid ticking* the rules that are *not respected*. You will score points only if you give the right answer for the four rules.

Q1(b) /2	**Tick the rules that are respected if col[ ]=[5, 2, 4, 1, 0, 3].**
	☒ rule 1 ☒ rule 2 ☐ rule 3 ☐ rule 4

Q1(c) /2	**Tick the rules that are respected if col[ ]=[4, 2, 1, 3, 5, 0].**
	☒ rule 1 ☒ rule 2 ☒ rule 3 ☐ rule 4

Q1(d) /2	**Tick the rules that are respected if col[ ]=[4, 2, 5, 3, 0, 2].**
	☒ rule 1 ☐ rule 2 ☒ rule 3 ☒ rule 4

Q1(e) /2	**Tick the rules that are respected if col[ ]=[5, 2, 0, 3, 1, 4].**
	☒ rule 1 ☒ rule 2 ☐ rule 3 ☒ rule 4

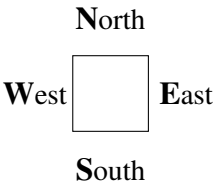
Now, find a solution !

Q1(f) /2	**Place six stars in the grid, while respecting the four rules.**
Solution : The answer is displayed in the grid above right.	

Q1(g) /2	**What is the value of array col[ ] in your solution ?** `col[]=[..., ..., ..., ..., ..., ...]`
Solution : col[ ]=[4, 2, 0, 3, 5, 1]	

You must now complete **Program SB1** below, which is used to display on screen a puzzle grid, like in the first example on the first page of this question.
The program relies on the size **n**, on the array **t[ ][ ]** and on the array **col[ ]**.

The program controls a robot that moves over the screen to draw lines, walls and stars. The program starts with the robot in the correct starting position to draw the cell in the top left corner of the grid (corresponding to textbft[0][0]).
We can use instructions `Go()`, `line()`, `wall()` and `star()` as explained below.
`Go()`, `line()` and `wall()` need a direction W, E, N or S.



- `Go(W,n)`, `Go(E,n)`, `Go(N,n)` and `Go(S,n)` move the robot **n** cells in the given direction.
  Examples: `Go(E,1)` moves the robot one cell to the right, `Go(S,2)` moves it two cells down.
- Instruction `wall()` draws a wall on one of the edges of the cell where the robot is located.
  `wall(W)` draws a vertical wall on the left and `wall(E)` a vertical wall on the right.
  `wall(N)` draws an horizontal wall on the top of the cell and `wall(S)` an horizontal wall at the bottom.
  The grid and all zones must be surrounded by walls.
- `line()` works like `wall()` but draws a fine line instead of a wall.
  There must be either a fine line or a wall on each side of each cell.
- `star()` draws a star in the cell where the robot is located.

If a test contains several conditions combined with **`or`**, the whole test is **`true`** if one of these conditions is **`true`**.
Moreover, if the first condition of an **`or`** is **`true`**, the second condition is not evaluated (as it is useless).

Q1(h) /10	**Fill in the ____ in program SB1.** **You can score between 0 and 10 points and lose one point per mistake or missing answer.**
Solution : The answers are displayed on a gray background.	

```
Input  : n, t[][], col[]
for (i ← 0 to n-1 step 1) {
   for (j ← 0 to n-1 step 1) {
      if (i=0 or t[i][j] ≠ t[i-1][j]) {wall(N)}
      else { line(N) }

      if (i=n-1) {wall(S)}

      if (j=0 or {t[i][j] ≠ t[i][j-1]) {wall(W)}
      else {line(W)}

      if (j=n-1) {wall(E)}

      if (j=col[i]) {star()}

      Go(E, 1)
   }
   Go(W, n)
   Go(S , 1)
}
```

You must now complete **Program SB2** below which is used to check if a player's suggested solution respects the 4 rules. The program receives size **n**, array **t[ ][ ]**, and the player's suggested solution in array **col[ ]**.

The program relies on an array **bcol[ ]** of **n** Boolean values. Initially, this array is filled up with **`false`** but every time a star is placed in a column, the corresponding cell in the array is changed to **`true`**. In order to test whether a star has already been placed in column 3, for instance, one can then use the test **`if`** `(bcol[3]){...}` **`else`** `{...}` since **bcol[3]** is either **`true`** or **`false`**.

The program also relies on an array **bzone[ ]** of **n** Boolean that has the same role as **bcol[ ]** for zones (instead of columns).

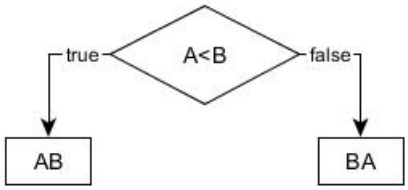
The program starts by checking rule 2. If several stars are placed in the same column, the program stops and returns **`false`**. Next, the program checks rule 3 and stops returning **`false`** if there are several stars in the same zone. When rules 2 and 3 are verified, the program can then check rule 4 and stops, returning **`false`**, if two stars are directly next to each other. The last line gets executed only when the four rules are respected, and returns **`true`**.

Q1(i) /10	**Fill in the ____ in Program SB2.** **You score between 0 and 10 points and lose one point per mistake or missing answer.**
Solution : The answers are displayed on a gray background	

```
Input  : n, t[][], col[]
Output : true or false
for (i ← 0 to n-1) {
   bcol[i] ← false
   bzone[i] ← false
}
//Rule 2 - Regle 2 - Regel 2
for (i ← 0 to n-1 step 1) {
   j ← col[i]
   if (bcol[ j ]) {return false}
   else {bcol[j] ← true}
}
//Rule 3 - Regle 3 - Regel 3
for (i ← 0 to n-1 step 1) {
   z ← t[i][col[i]]
   if (bzone[ z ]) {return false}
   else {bzone[z] ← true}
}
//Rule 4 - Regle 4 - Regel 4
for (i ← 0 to n-2 step 1) {
   if (col[i]=col[i+1]+1 or col[i]=col[i+1]-1) {return false}
}
return true
```

Question 2 – Flowchart

We can represent a test by a diamond shape, in which we write the condition to be checked.
Depending on the outcome of the test, we follow either the **`true`** or the **`false`** branch.
In the example below, `A` and `B` are variables that we want to compare.
If `A<B`, we display "**AB**", otherwise, we display " **BA**".
The display operations are represented by rectangles.

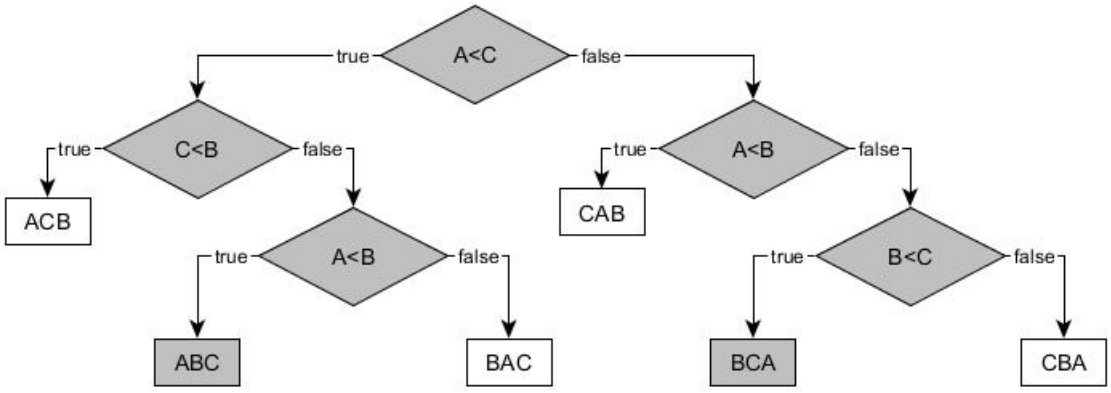


For example, if `A=7` and `B=4`, the above algorithm will follow the **`false`** branch of the test and will display "**BA**".

Let us now consider a similar problem with three variables `A`, `B` and `C` that have **different values**.
You are requested to represent graphically an algorithm that displays the three letters "**A**", "**B**" and "**C**" according to the values of these variables, from the lowest to the largest. For example, if `A=11`, `B=16` and `C=7`, the algorithm must display "**CAB**" since `C` has the smallest value, and `B` has the largest.
To do so, you must write a test in each diamond. You can only write the following tests: `A<B`, `B<A`, `A<C`, `C<A`, `B<C` and `C<B`.

You must also fill in the rectangles with the . . . blank spaces with the three letters in the appropriate order.

Q2(a) /7	**Fill in the . . . below. You will score one point for each correctly filled box.**
Solution : The answers are displayed on a gray background	



Let us assume now that the variables can have **equal values**.

Q2(b) /2	**What is the message that is displayed when the three variables are equal?**
Solution : CBA	

Q2(c) /2	**Which message is never displayed whenever two variables are equal?**
Solution : ACB	

Question 3 – Intersecting lists

We want to find all the common elements in two lists of n elements $L1[\,]$ and $L2[\,]$.
We know that the elements of $L1[\,]$ are all different, as are the elements of $L2[\,]$.
We have to create the list $Li[\,]$ containing all elements that belong to both $L1[\,]$ and $L2[\,]$.
For example, if $n = 8$, $L1[\,] = [1, 3, 6, 10, 15, 21, 28, 36]$ and $L2[\,] = [3, 7, 15, 19, 23, 29, 36, 41]$ then $Li[\,] = [3, 15, 36]$.
As usual, list elements are numbered from 0.

First idea: program P1.
This program relies on two nested **`for`** loops.
The first loop selects a new element $x1$ from $L1[\,]$ during each iteration.
The second loop, which is nested in the first one, compares successively $x1$ to all the elements of $L2[\,]$.
When an equality is found, $x1$ is added to $Li[\,]$.
P1 relies on `Li[]←[]` to create an empty list $Li[\,]$.
It further relies on method `append()` to **append an element at the end of a list**.
For example, if $Li[\,]$ is empty, and if we call `Li[].append(3)` then $Li[\,] = [3]$.
Next, if we execute `Li[].append(15)` followed by `Li[].append(36)`, then $Li[\,] = [3, 15, 36]$.

Note that, in **P1**,lines (1) and (2) are highlighted with a gray rectangle.

```
Input : L1[], L2[], n      Output : Li[]
Li[] ← []
for i1 ← 0 to n-1 step 1 {
    x1 ← L1[i1]        // (1)
    for i2 ← 0 to n-1 step 1 {
        if (L2[i2]=x1)     // (2)
            {Li[].append(x1)}
    }
}
```

Q3(a) /1	**If $n = 100$, how many times will line (1) be executed?**
Solution : 100	

Q3(b) /1	**If $n = 100$, how many times will line (2) be executed?**
Solution : 10000	

Q3(c) /1	**In general, for a given n, how many times will line (1) be executed?**
Solution : n	

Q3(d) /1	**In general, for a given n, how many times will line (2) be executed?**
Solution : n^2	

In the next questions, we test **P1** on lists of different lengths.
We will assume that the total execution time depends only on the number of times line **ligne (2)** is executed (this is clearly an approximation, but it is sufficiently precise for our purpose).

Q3(e) /1	**If the execution time is 3 seconds for a given n, what is the execution time (in *seconds*) to run the program on lists that are 5 times longer?**
Solution : $3 \cdot 5 \cdot 5 = 75$ seconds	

Q3(f) /1	**If the execution time is** 6 **seconds for a given** n**, what is the execution time (in *minutes*) to run the program on lists that are** 10 **times longer?**
Solution : $6 \cdot 10 \cdot 10 = 600$ seconds = 10 minutes	

Q3(g) /1	**If the execution time is** 4 **seconds for a given** n**, what is the execution time (in *hours*) to run the program on lists that are** 60 **times longer?**
Solution : $4 \cdot 60 \cdot 60$ seconds = $4 \cdot 60$ minutes = 4 hours	

Second idea: program P2.
P1 will work on all lists, whether they are sorted or not, but it is relatively slow.
P2 is faster, but will only work when the input lists are sorted in increasing (ascending) order.

Here is the listing of **P2** without explanations.
Note that line (3) is highlighted with a number in a gray rectangle.

```
Input : L1[], L2[], n       Output : Li[]
Li[] ← []
i1 ← 0
i2 ← 0
while (i1 < n and i2 < n) {
    x1 ← L1[i1]        // (3)
    x2 ← L2[i2]
    if (x1 < x2) {i1 ← i1 + 1}
    else {
        if (x1 = x2) {Li[].append(x1)}
        i2 ← i2 + 1
    }
}
```

Q3(h) /1	**If** $n = 10$**, what is the MINimum number of times that line (3) will be executed?**
Solution : 10	

Q3(i) /1	**If** $n = 10$**, what is the MAXimum number of times that line (3) will be executed?**
Solution : 19	

Q3(j) /1	**If** $n = 10$ **and** $L1[\,] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$**, give a list** $L2[\,]$ **of different integer** > 0 **numbers such that line (3) gets executed a MINimal number of times.**
Solution : Any list such that the first element is larger than 19, for example: `L2[] = [20,21,22,23,24,25,26,27,28,29]`	

Q3(k) /1	**If** $n = 10$ **et** $L1[\,] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$**, give a list** $L2[\,]$ **of different integer** > 0 **numbers such that line (3) gets executed a MAXimal number of times.**
Solution : There are many solutions: `L2[]=[2,4,6,8,10,12,14,16,18,n]` with $n \geq 20$ or `L2[]=L1[]=[1,3,5,7,9,11,13,15,17,19]`	

Q3(l) /1	**For a given** n**, what is the MINimum number of times that line (3) will be executed?**
Solution : n	

Q3(m) /1	**For a given n, what is the MAXimum number of times that line (3) will be executed?**
Solution : $2n - 1$	

In the next questions, we test **P2** on lists of different lengths.
We will assume that the total execution time depends only on the number of times line **ligne (3)** is executed (this is clearly an approximation, but it is sufficiently precise for our purpose).

Q3(n) /1	**If the MAXimum execution time is 3 seconds for a given n, what is the MAXimum execution time (in *seconds*) for lists that are 5 times longer?**
Solution : $3 \cdot 5 = 15$ seconds	

Q3(o) /1	**If the MAXimum execution time is 6 seconds for a given n, what is the MAXimum execution time (in *minutes*) for lists that are 10 times longer?**
Solution : $6 \cdot 10 = 60$ seconds = 1 minute	

Q3(p) /1	**f the MAXimum execution time is 4 seconds for a given n, what is the MAXimum execution time (in *minutes*) for lists that are 60 times longer?**
Solution : $4 \cdot 60$ seconds = 4 minutes	

Question 4 – Blocks

$L[\,]$ is a list with n elements $L[0], \ldots, L[n-1]$.
The **length** of a list is the number of elements it contains.
A **block** from $L[\,]$ is a sub-list which is as long as possible and contains consecutive elements of $L[\,]$ which are all equal.
For example: the blocks of $L[\,] = [4, 2, 2, 2, 1, 1, 2]$ are $[4]$, $[2, 2, 2]$, $[1, 1]$ and $[2]$, with lengths 1, 3, 2 et 1 respectively.

Given $L[\,]$, its **associated** list is the list of the lengths of the blocks from $L[\,]$.
Example (continued): The list associated to $[4, 2, 2, 2, 1, 1, 2]$ is $[1, 3, 2, 1]$.

Q4(a) /1	**What is the associated list of** $[1, 2, 3, 4, 5]$**?**
Solution : $[1, 1, 1, 1, 1]$	

Q4(b) /1	**What is the associated list of** $[3, 3, 3, 3, 3]$**?**
Solution : $[5]$	

Q4(c) /2	**What is the length of** $L[\,]$ **if its associated list is** $[3, 2, 2, 3]$ **?**
Solution : 10	

Q4(d) /2	**Which list is equal to its own associated list ?**
Solution : $[1]$	

In all the following questions, we consider lists $L[\,]$ **containing only the numbers** 1 **and** 2**, and starting with** 1.
For example, $L[\,] = [1, 2, 2, 2, 1, 2]$ or $L[\,] = [1, 1, 1, 1]$.
$A[\,]$ will be the list associated to $L[\,]$.
For the two examples above, we have $A[\,] = [1, 3, 1, 1]$ and $A[\,] = [4]$, respectively.

Q4(e) /2	**What is** $A[\,]$ **if** $L[\,]$ **and** $A[\,]$ **have length** 6 **?**
Solution : $A[\,] = [1, 1, 1, 1, 1, 1]$	

Q4(f) /2	**What is** $L[\,]$ **if** $L[\,]$ **and** $A[\,]$ **have length** 6 **?**
Solution : $L[\,] = [1, 2, 1, 2, 1, 2]$	

You need now to complete **Program B1** below which takes as input a list $L[\,]$ and its length n (where $n > 0$) and which produces as output the list $A[\,]$ associated to $L[\,]$.

The program relies on `A[]←[]` to create a an empty list $A[\,]$.
The program also relies on the method `append()` which appends an element to the end of a list.
For example, if $A[\,]$ is empty and we call `A[].append(3)` then $A[\,] = [3]$.
If we further execute `A[].append(1)`, then $A[\,] = [3, 1]$.

Q4(g) /6	**Fill in the ____ in program B1.** **You will score between 0 and 6 points, and you will lose 1 point per mistake or missing answer.**
Solution : Solutions are displayed on a gray background.	

```
Input  : n, L[]
Output : A[]
A[] ← []
value ← L[0]
length ← 0
for (i ← 0 to n-1 step 1)
{
    if (L[i] = value)
    {
        length ← length + 1
    }
    else
    {
        A[].append(length)
        value ← L[i]
        length ← 1
    }
}
A[].append(length)
return A[]
```

We can consider infinite lists as well.
For example, the infinite list $[1, 2, 1, 2, 1, 2, 1, 2, \dots]$ has $[1, 1, 1, 1, 1, 1, 1, 1, \dots]$ as associated list, which is infinite as well.

There exists a unique "magic" infinite list with the following three properties:

- It starts with 1.
- It contains 1's and 2's only.
- It is equal to its associated list.

We call the magic list $M[\,]$.

Q4(h) /4	**What are the 6 first elements of the magic list $M[\,]$?**
Solution : $[1, 2, 2, 1, 1, 2]$	

You now need to complete **Program B2** below, which generates the first elements of the magic list.
The input of the algorithm is an integer number n, which is the number of elements that the algorithm must compute. The output is an array $M[\,]$ of length n that contains the n first elements of the magic list.

The program relies on `M[]←[1]` to create a list $M[\,]$ that contains a unique element $M[0]$ equal to 1. On top of the `append()` method that we have introduced already, this program relies on the method `pop()` that **deletes the last element of a list**.
For example, if $L[\,] = [1, 2, 2, 4]$ and we execute `L[].pop()` then $L[\,] = [1, 2, 2]$.
We must sometimes call this method at the end of the program because it can generate, during the **`while`** loop, one element more than the requested n elements.

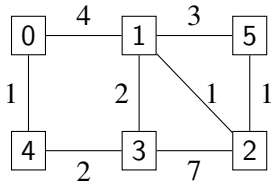
Q4(i) /8	**Fill in the ____ in Program B2.** **You will score between 0 and 6 points, and you will lose 1 point per mistake or missing answer.**
Solution : The answers are displayed below, on a gray background	

```
Input  : n
Output : M[]

M[] ← [1]
iL ← 1
iA ← 1
value ← 2

while (iL < n)
{
    M[].append(value)

    if (M[iA] == 2) { M.append(value) }

    iL ← iL + M[iA]
    iA ← iA + 1
    value ← 3 - value
}
if (iL = n+1) {M.pop()}
return M[]
```

Question 5 – The journey of the Smurfs

The Smurfs love to travel, everybody knows that! They regularly embark on a trip, lead by Papa Smurf. But these travels are very long (« How much farther, Papa Smurf ? ») and Papa Smurf is starting to feel his age now that he has turned 542...

Computer Smurf decides to help him. He asks Geographer Smurf for assistance to draw a map of all the paths that are known in the Magic Forest, together with the times that are needed to go from one point to another. The map is given below.



On this map, the different places are represented by squares.
0 is the village, which is the starting point of all journeys; 1 is the big oak; 2 is Gargamel's hovel; and so forth.
The lines between the places represent the paths.
The numbers on the paths show the time needed to go from one place to the other.
For example, one needs 4 hours to travel from 0 to 1.

But Geographer Smurf's map displays *direct* paths only.
Computer Smurf wants to compute the fastest *itinerary* to reach any place.
An *itinerary* is a sequence of places that one can visit by following directly connected paths, starting from the village.
For example 0, 4, 3, 2 is an itinerary, but 0, 3, 2 is not because there is no direct path between 0 and 3.
The *duration* of an itinerary is the sum of the durations of all the paths in the itinerary.
For example, the duration of 0, 4, 3, 1 is $1 + 2 + 7 = 10$ hours.

Q5(a) /1	**What is the duration of** 0, 1, 3, 2, 5 **?**
Solution : 14	

Q5(b) /1	**Can we travel from** 0 **to** 5 **in exactly 7 hours?** **If your answer is 'yes', give the itinerary; otherwise answer 'no'.**
Solution : 0, 1, 5	

Q5(c) /1	**Can we travel from** 0 **to** 5 **in less than 5 hours?** **If your answer is 'yes', give the itinerary; otherwise answer 'no'.**
Solution : non	

Q5(d) /1	**What is the fastest itinerary to go from** 0 **to** 5 **?**
Solution : 0, 1, 2, 5	

Computer Smurf then proceeds to explain the algorithm (given below) that he came up with to compute the fastest itineraries.

My algorithm relies on an array `D[]`, indexed by the place numbers. At the end of the algorithm, for each place `x`, `D[x]` will be the duration of the fastest itinerary from the village to `x`.
Initially, I let `D[x]`$=+\infty$ for all places `x` thanks to a **`for`** loop.
Straight away after that, I let `D[0]`$=0$, since the village is always the starting point of all itineraries.

> The $+\infty$ symbol means 'infinite value'.
> It is some sort of mathematical equivalent to 'It's still very far, my little Smurfs!'
> It is a special value that is larger than any number.
> Moreover, adding an integer value to $+\infty$ yields $+\infty$.
> For example, $+\infty + 5 = +\infty$.

During its execution, my algorithm identifies two types of places.

- The places for which the fastest itinerary (from the village) is already know.
  For these places, the value `D[x]` will not change anymore.
  We will call these places the *known places*.
- The places for which the fastest itinerary (from the village) is not known yet.
  For those places, `D[x]` can still change during the rest of the execution.
  We will call these places the *unknown places*.

At the beginning, the algorithm assumes that all places are *unknown*.
At the end of the execution, when all the fastest itineraries have been computed, all places will be considered as *known*.
In practice, the algorithm maintains an array of Boolean values `K[]` such that `K[x]` is **`true`** for the *known* places and **`false`** for the *unknown* places.

When we are certain that `D[x]` will not change anymore, it means that `x` becomes *known*, and this information is stored by setting `K[x]` to **`true`**.

The most important operations of the algorithm take place in the **`while`** loop. This is what gets executed at each iteration of the loop:

- Among all *unknown* places, pick the place `m` such taht `D[m]` is minimum.
- Set `m` as a *known* place.
- Consider all the *successors* of `m` to check whether we can find an itinerary that is faster than the one we already know. The successors of `m` are all the places that can be reached from `m` by following a direct path.
  For example, the successors of 0 are 1 and 4.
  I then check what is the duration `D[s]` of the fastest known itinerary to each successor `s` and I try to improve it with an itinerary going through `m`.

Finally, I have to explain how I represent the map with the paths.
I rely on an array `G[][]` such that `G[i][j]` is the duration of the direct path between place `i` and place `j`, if this path exists.
If there is no direct path between place `i` and place `j`, I let `G[i][j]`=$+\infty$. For example, with the map of Geographer Smurf, we have `G[0][1]=4` and `G[0][3]`=$+\infty$.

The algorithm relies on function `minFalse(D[],K[])` that returns the number (between 0 and $n-1$) of the *unknown* place with the smallest value `D[]`.
If there is a tie between several *unknown* places, the function returns the one with the smallest number.
If all the places are known, the function returns -1.
Example: if the unknown places are 2, 4 and 5, and if `D[]=[0,7,9,15,17,9]`, then `minFalse(D[],K[])` will look for the smallest value among `D[2]=9`, `D[4]=17` and `D[5]=9`. The result will be 9.
Since there is draw between 2 and 5, `minFalse(D[],K[])` returns the smallest number, which is 2.

I hope you have smurfed—I mean, 'understood'—this whole explanation!

Here is Computer Smurf's algorithm:
Input : n, the number of places; G[][], the map.
Output : D[], such that D[m] is the duration of the fastest itinerary from 0 to m.

```
for (i ← 0 to n-1 step 1) {
    D[i] ← +∞
    K[i] ← false
}
D[0] ← 0
m ← minFalse(D[],K[])

while(m ≠ -1) {
   K[m] ← true
   for(s ← 0 to n-1 step 1) {
       if ((not K[s]) and (G[m][s] ≠ +∞)) {
           if (D[m] + G[m][s] < D[s]) {
               D[s] ← D[m] + G[m][s]
           }
       }
   }
   m ← minFalse(D[],K[])
}
return D[]
```

Here are the first few steps of execution of the algorithm, using the map of Geographer Smurf.
The arrays represent D[] and K[]. As an abbreviation, we represent **true** by T and **false** by F.

- Initialisation:

	0	1	2	3	4	5
D:	0	+∞	+∞	+∞	+∞	+∞

	0	1	2	3	4	5
K:	F	F	F	F	F	F

- During the first iteration of the loop, the algorithm selects m= 0 and inspects its successors 1 and 4.
  We have G[0][1]= 4 and G[0][4]= 1. Since D[0]+G[0][1] $= 0 + 4 = 4 < +\infty$, we update D[1] with this value.
  Similarly, D[4] will be updated and we obtain:

	0	1	2	3	4	5
D:	0	4	+∞	+∞	1	+∞

	0	1	2	3	4	5
K:	T	F	F	F	F	F

What are the next steps of the algorithm?

Q5(e) /4	**Give the arrays D[] and K[] at the end of the second iteration of the while loop.**
Solution : D[] = [0,4,+∞,3,1,+∞] K[] = [T,F,F,F,T,F]	

Q5(f) /4	**Give the arrays D[] and K[] at the end of the third iteration of the while loop.**
Solution : D[] = [0,4,10,3,1,+∞] K[] = [T,F,F,T,T,F]	

Q5(g) /4	**Give the arrays D[] and K[] at the end of the fourth iteration of the while loop.**
Solution : D[] = [0,4,5,3,1,7] K[] = [T,T,F,T,T,F]	

Now, Computer Smurf is reflecting on the efficiency of his algorithm to compute the fastest itineraries, and on the answers it could return...

Q5(h) /2	**With the map from Geographer Smurf, how many iteration of the `while` loop will be executed?**
Solution : 6	

Q5(i) /2	**If the map has n places, and the largest duration of a direct path is M, how many iteration of the `while` loop will be executed?**
Solution : n	

Next, Computer Smurf shows his algorithm to Papa Smurf.
He answers: 'This is really good, my little Smurf, but your algorithm is of little use to me if it computes only the *durations* of the best itineraries! I want to know the actual itineraries as well!'

Computer Smurf goes back to work and figures out that he needs to compute, for each place `x`, the place `y` that precedes directly `x` in the fastest itinerary to reach `x`. He calls `y` the *predecessor* of `x`.
For example, the fastest itinerary to reach 3 is 0, 4, 3.
Hence, we must remember that the predecessor of 3 is 4, and that the predecessor of 4 is 0.
Computer Smurf enhances his algorithm with an array `P[]` which is meant to remember, for each location `x`, its predecessor `P[x]`.
Initially, `P[x]` must be equal to a special value indicating that nothing has been computed yet.
Computer Smurf chooses as special value: -1.
Next, he needs to modify the **`while`** loop to be able to update `P[x]`.
Based on this information, can you help him to fill in the two missing lines in his algorithm?

Q5(j) /4	**Fill in the _____ in Computer Smurf's algorithm to compute `P[]`.**
Solution : The answers are displayed on a gray backgruond.	

```
for (i ← 0 to n-1 step 1) {
    D[i] ← +∞
    K[i] ← false
    P[i] ← -1
}
D[0] ← 0
m ← minFalse(D[],K[])
while(m ≠ -1) {
   K[m] ← true
   for(s ← 0 to n-1 step 1) {
       if ((not K[s]) and (G[m][s] ≠ +∞)) {
           if (D[m] + G[m][s] < D[s]) {
               D[s] ← D[m] + G[m][s]
               P[s] ← m
           }
       }
   }
   m ← minFalse(D[],K[])
}
```

Q5(k) /2	**What is the content of array `P[]` at the end of the execution of the algorithm, when executed on the map from Geographer Smurf?**
Solution : `P[] = [-1, 0, 1, 4, 0, 2]`	

Unfortunately, Gargamel, the archenemy of the Smurfs, has installed Smurf traps on some paths.
The Smurfs cannot use those paths anymore, and Geographer Smurf has therefore decided to delete them from the map.
Hence, some place can now be totally unreachable from the village!
Computer Smurf now wonders how his algorithm will behave.
Brainy Smurf, who always believes he understands everything that has to do with computing, comes up with a series of statements.
Which ones are true?

	True	False	Brainy Smurf's propositions
Q5(l) /1	☐	☒	There is no map for which the algorithm returns an array `P[]` containing a value $+\infty$.
Q5(m) /1	☒	☐	On certain maps, the algorithm can return an array `D[]` containing one or several values $+\infty$.
Q5(n) /1	☒	☐	The algorithm returns an array `D[]` where `D[x]`$=+\infty$ for all the places `x` such that there is no itinerary to reach `x` from the village (place 0).