

# Initiation to Algorithmics with Scratch

Benoit Gaudou  
with inputs from Géraldine Abrami

v1, 14th July 2015



# Initiation to Algorithmics with Scratch

The aim of this document is to provide basic knowledge in algorithmics for trainees taking part in agent-based modeling and simulation training sessions. It has been produced for MISS-ABMSS 2015 as a prerequisite distant teaching for participants with no or very little experience in computer coding and algorithmics.

The objective is that the participants who go through this document and do the associated exercises get basic definitions and practice of what is the basis of computer coding, independently of any programming language : what are statements, variables, procedures, functions, loops and how can they be manipulated. They will also get a first glimpse on what is object-oriented computer coding, which we use as a basis for agent-based models programming.

In this document, we will use Scratch<sup>1</sup> to develop algorithms. It has the advantage to allow developers to implement algorithms using graphical programming tools. In each section of the document, basic definitions are given, and then exercises using Scratch are proposed to get a practice of the concepts defined.

Most exercises are easy and allow to understand the concepts. However more challenging exercises have also been inserted. They are noted with stars \*\*. Beginners should do easier exercises in priority and maybe keep the more tricky \*\*exercises for later !

The solutions to the exercises are given at the very end of the document.

[Algorithms: introduction](#)

[Introduction to Scratch](#)

[First algorithm with Scratch](#)

[Sequence of statements](#)

[Variables](#)

[Conditional statements](#)

[Loop statements](#)

[Manipulation of sets \(or list\) of values](#)

[Sub-algorithm: procedure and function](#)

[Toward object/agent-oriented approach](#)

[References](#)

[Solutions](#)

[Sequence of statements](#)

[Variables](#)

[Conditional statements](#)

[Loop statements](#)

[Manipulation of sets of values](#)

[Sub-algorithm: procedure and function](#)

---

<sup>1</sup> <https://scratch.mit.edu/>

## Algorithms: introduction

**Definition** [Cormen et al., 2001]

*Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.*

The various concepts that will be investigated in the sequel are the following ones :

- Sequence of statements
- Introduction to variables
- Conditional statements
- Loop statements
- Manipulation of sets of values
- Procedure and function
- Toward object approach programming

# Introduction to Scratch

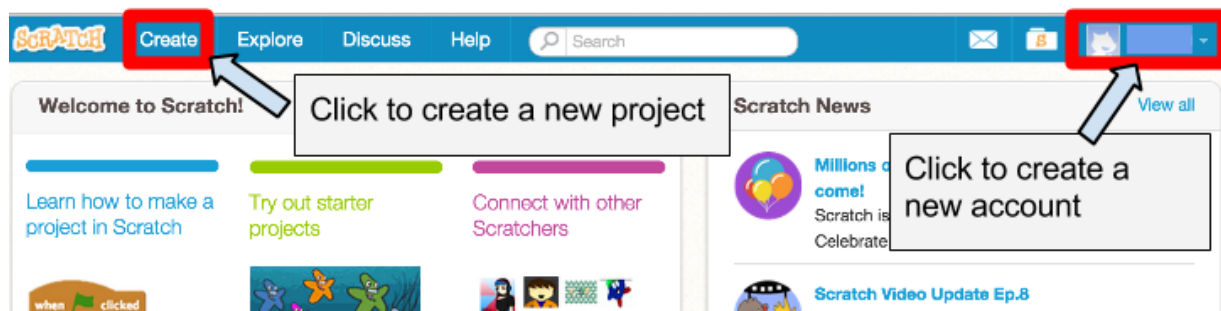
Scratch is a free software and web portal developed by the MIT in order to allow kids to learn how to develop interactive stories and animation in a collaborative manner. In a Scratch project, you can control sprites and make them move, interact... But the language proposed to control these sprites contain all the basics structures necessary in any algorithms. So Scratch can be used to develop any kind of algorithms. In the sequel, for each part there will be exercises aiming at controlling the sprites, but also exercises to design more theoretical classical algorithms.

The tool is available online at the address: <https://scratch.mit.edu/>.

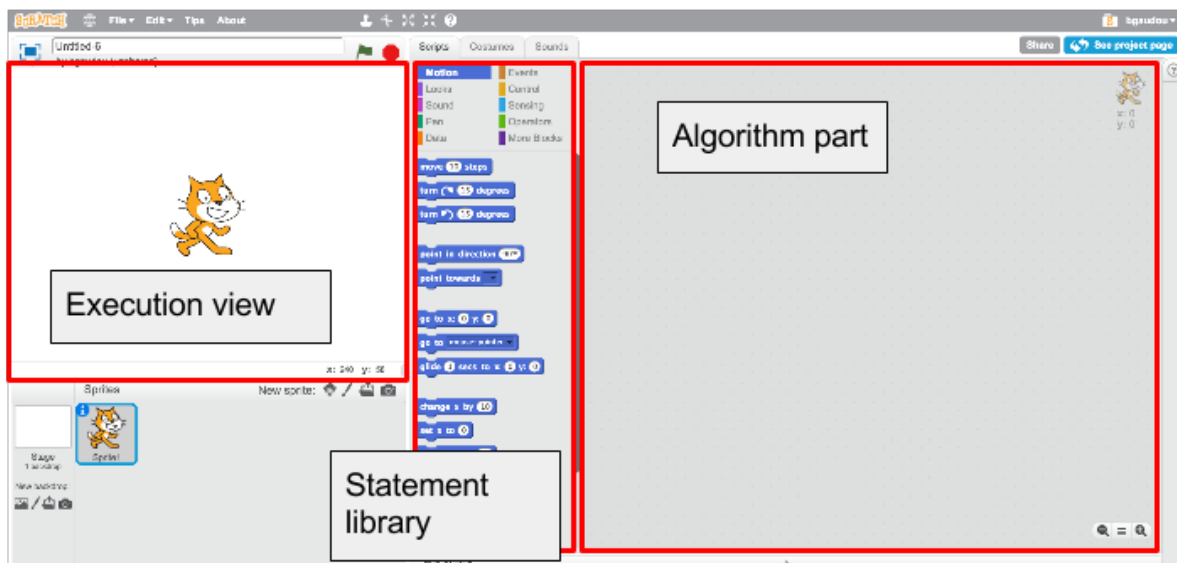
A wiki is also available: [http://wiki.scratch.mit.edu/wiki/Scratch\\_Wiki](http://wiki.scratch.mit.edu/wiki/Scratch_Wiki).

In addition, an offline editor is available: <https://scratch.mit.edu/scratch2download/>.

When you access the website, you can create a free account to save your projects. You can create a new project by clicking on the dedicated menu (Create), even if you have not an account.



After having created a new project, you access to the interface where you can create your algorithms and observe the result of their execution (with the sprite).

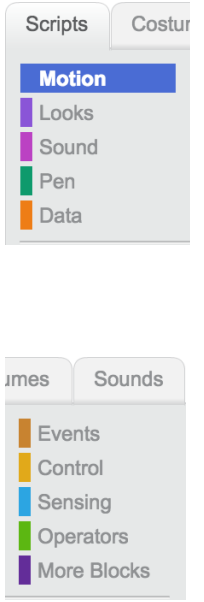


The three main parts of the interface are:

- on the left: the Graphical User Interface showing the execution of the algorithm. In particular, you can observe here the sprites, their moves ...;
- in the center: the library of blocks (statements), that you can use to build the algorithm (in the right part);
- on the right: where the algorithm will be written by putting together blocks.

To build an algorithm, we only need to drag and drop elements from the block library to the algorithm part.

The library contains a huge set of block, ordered by categories. Each category has its own color. For example, all the blocks related to the sprite's motion are blue.

	<ul style="list-style-type: none"><li>● <b>Motion:</b> blocks to move sprites (move, rotate,...) and variables dealing with their position or direction</li><li>● <b>Looks:</b> blocks to modify the way the sprite looks like. It also contains blocks allowing sprites to say anything.</li><li>● <b>Sound:</b> everything related to sound.</li><li>● <b>Pen:</b> blocks allowing the sprite to write on the background.</li><li>● <b>Data:</b> it allows to create new variables and lists. It provides also statements to manage them.</li><li>● <b>Events:</b> blocks to react to events, in particular when the user clicks on the green flag.</li><li>● <b>Control:</b> blocks controlling the execution of the algorithm, e.g. conditionals, loops...</li><li>● <b>Sensing:</b> blocks dealing with interactions with the user, in particular with mouse clicks... It allows also sprites to ask users to give a value.</li><li>● <b>Operators:</b> blocks to do computation (addition, multiplication...), to choose a random number, concatenate two strings, compute conditions...</li><li>● <b>More blocks:</b> this allows to create new blocks, that will be used to defined procedures.</li></ul>
--	---


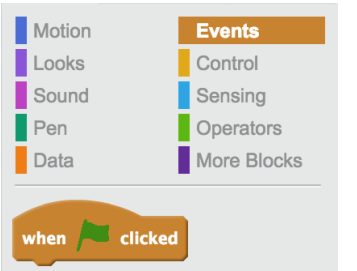


# First algorithm with Scratch

## Aim:

- Run 1 statement when the green flag is clicked.

## Exercise 0:




As a first algorithm, we want that, when the user clicks on the green flag, the sprite moves by 10 steps from its actual location.

<p>First, we need a block that is activated when the event “click on the green flag” is triggered. In the library, chose <b>Events</b> and the block . Drag and drop this block in the algorithm part.</p>	
<p>The block describing the sprite’s move can be found in the <b>Motion</b> set of statements.</p> <p>Chose  and drag and drop it into the algorithm part.</p> <p>Drop the second block (move) just below the former one, they will stick together.</p>	

Now the user can click on the green flag and observe the sprite’s move. Try to click again on the green flag to make it move again.



Note: in Scratch, the shape of the values within various blocks has a meaning. In particular:

	● for a number value
	● for a string value
	● for a boolean value (i.e. anything that can be evaluated to true or false). Test operators can be found in the Operators Menu.

## Sequence of statements

As defined in the first section, an *algorithm* is thus a sequence of computational steps that transforms the input into the output.

We can define each of these steps as a statement (e.g. variable declaration or affectation, loop statement, conditional statement...).



To compute the output from the input, several statements are generally needed, that can be executed sequentially. In particular, sequence of statements are needed to do several computations, store intermediate results in variables, repeat several times a same subset of statements with various variable values or execute two or more alternative statements depending of the result of a previous computation or interaction with the user.


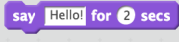
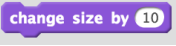
We can distinguish two kinds of statements:

- simple statements: these statements are often an imperative command, basically only 1 line of code or 1 block in Scratch.
- embedding statements: these statements enclose a sequence of statements. It is for example the case for the loops or conditionals. Loops will execute the enclosed sequence of statements several times. Conditional statements will only execute the enclosed sequence of statements if a given condition is met.

### Aim:


- Combine several statements;
- Add an infinite loop;
- Manipulate the various menus.


**Exercise 1:** From the previous algorithm (with the 2 blocks:  and ), add the following statements:

- if on edge, bounce: 
- say Hello! for 2 seconds 
- change the size 



**Hint:** the color of the block is similar to the color in the menu of the library.

**Try it:** try to change the order of the blocks and see what happens

**Exercise 2:** With the previous algorithm, the user has to click several times on the green flag in order that the sprite moves several times. We will had a block to repeat forever the 4 blocks we have already added. Thus add the  block to repeat the four statements.


**Note:** the  block is different from the 4 previous blocks we have used previously as it embeds a set of statements inside itself. It has a beginning and an end; between the



begin and end a set of statements can be enclosed. You can thus drag and drop the  block below the  block and around the 4 statements.

**Try it:** try to put some statements out of the  block and see what happens. Change what is in and out of the block.

**Hint:** in Scratch the manipulation of the statements inserted in an algorithm may seem a little tricky. Note that when you move a statement from a block, it takes with it all the statements stuck below it.

Note also that you can put statements or block of statements that you are not using anywhere in the algorithm window. If they are not stuck under the , they will not be considered in the execution of the code.

In this way you can easily “put on the side” statements or blocks of statements you are not currently using and potentially reuse them later.

## Variables

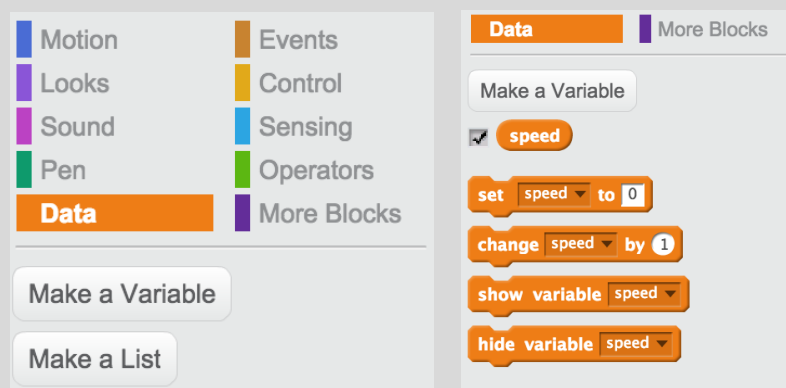
In an algorithm, it is often useful and necessary to store results of intermediate computations and to be able to reuse them in the sequel of the algorithm. It is done using variables. In a variable, we can store a numerical value (integer or float), string value or boolean (only true or false). But a variable can also store a set (a list) of elements.

In any algorithm, variables should first be declared and initialized, i.e. the first statements of an algorithm should be dedicated to name and give an initial value to variables.

It is important to choose carefully the variable name, so that it can be meaningful for the developer himself, but also for any other developer who could read and reuse the algorithm.

In Scratch, sprites have built-in variables such as their location ( **x position** and **y position** ), or the size of their shape ( **size** ). These variables cannot be modified directly by the user (as other variables defined by the user can), but need particular statements to be modified (e.g. **set x to 0** ). Most of the Motion blocks will alter these two variables.

Scratch allows the developer to create his own variables. The menu Data is dedicated to this purpose. To create a new variable, click on “Make a variable” button. Scratch asks the variable name. In the screenshot, we chose “speed”. Scratch also creates a set of new blocks allowing to manage the variable: **set speed to 0** will change the variable value, **change speed by 1** will increment it by a given value.



### Aim:




- Display the position: manipulate operators on built-in string variables (concatenate)

- Increase the sprite speed: manipulate operators on a numerical user defined variable
- Swap 2 variables.

### Exercise 1:

We use the algorithm of Exercise 2 from previous section as startpoint.

Modify it in order to display the coordinates of the sprite:



- Make the sprite say the x position (instead of Hello) with .
- Make the sprite say « My x position is » and the value of the x position variable, using the  block to concatenate strings (it works to concatenate a string and a number too).
- Make the sprite say « My location is : x= x-position and y= y-position » (instead of x-position and y-position, it should display its current x position and y position variable), using several  blocks.

**Hint :** the built-in variables of sprites can be found within the different library menus, after the statements.

### Exercise 2:

Let consider the previous algorithm. The sprite will move forever at the same speed (10). We now aim at increasing the sprite speed after each move.

- Create a new variable (to represent the speed of the sprite).
- Initialize that speed at 0 at the beginning of the algorithm.
- Use the speed variable in the move statement.
- Increase the speed after each move.

**Hints:** The following blocks can be useful: , .

### \*\*Exercise 3: (theoretical exercise)

Create a new algorithm, with 2 variables x and y, initialized to 10 and 50.

Write the algorithm that swap the value of the two variables x and y. At the end, x should be equals to 50 and y to 10.

**Hint:** an additional variable can be useful.

## Conditional statements

The flow of the algorithm should be able to adapt to the various possible variable or inputs values. To this purpose algorithms use conditional statements.

These statements can execute a set of statements if and only if a condition is true or choose between two sets of statements depending on the condition value.

It can take the two following forms:

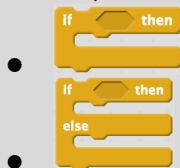
```
IF (condition is true)
THEN
  execute a set of statements
END IF
```

or





```
IF (condition is true)
THEN
  execute set of statements1
ELSE
  execute set of statements2
END IF
```

The condition could be for example equality between variables ( $x=y$ ), the sign of a variable ( $x>0$ ) or any other expression that can be evaluated to either true or false. For example in the algorithm of the Exercise 2 of the previous section, the speed could be increased only if it remains below a maximum value.

In Scratch, these two forms of conditional statements are:



In each of them a set of statements can be embedded after the then and the else.

To write the conditions, Scratch provides following operators:  or . In addition, logical operators allows to combine conditions:  (returns the converse of the condition value, i.e. true if the condition is false and false if the condition is true),  (returns true if and only if both conditions are true, and false otherwise).

For example, the condition  is true if and only if i equals 7 and x lower than y. In other cases, it is false.

**Aim:**

- Add a max speed variable and increase (by 1) the speed if and only if the speed is below the max speed;
- Add a max speed variable and increase (by 1) the speed if the speed is below the max speed, otherwise (if the speed has reached the max speed), set the speed to 0;
- Add a max speed variable and increase (by 1) the speed if the speed is below the max speed. When the speed reaches the max speed, decrease it until reaching 0. When it reaches 0, increase it to Max speed...

**Exercise 1:**

From the algorithm of the Exercise 2 of the previous section:

- add an additional variable to define the maximum speed (initialize it at 50 at the beginning)
- add a conditional in order to increase the speed if and only if the speed is lower than the maximum speed.

**Try it:** notice that the “say” statements takes a long time to execute. Try to remove it from the block that is executed to increase the speed of the program.

**Exercise 2:**

We will modify the previous algorithm in order that:

- if the speed is lower than the maximum speed, increase the speed
- else (if the speed is greater), set the speed to 0.

**\*\*Exercise 3 :**

Modify the previous algorithm in order that:

- the speed is increased from 0 to the maximum speed
- when the speed reaches the maximum speed, it is decreased from maximum speed to 0
- when the speed reaches 0, it is increased from 0 to the maximum speed...

**Hints:** a possible way to do it is to add a new variable used to increment/decrement the speed. Its initial value is 1. It becomes -1 (to decrease speed) when the speed reaches the maximum speed and becomes 1 when the speed is 0.

## Loop statements

In algorithms, it is often useful to repeat several times the same statements. At each new call of the set of statements, only the value of a variable can be modified. It avoids write several times the same lines in the algorithm.

It becomes also necessary to use loop statements when the number of repetitions is not known when the algorithm is written. The number of repetition can be an input of the user, as in an algorithm computing the sum or average of the N first integer numbers. Or it can depend on the state of a variable, for example when an algorithm should compute the sum of the elements of set or a list.

Finally, in several cases, the number of repetition is not known even at the beginning of the loop. For example, an algorithm can ask the user to type a number greater than 10 and the algorithm repeats this request until the user has typed a correct number.

Loop statements can take following forms:

**LOOP** (N times)  
    set of statements  
**END LOOP**

or

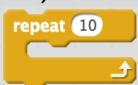
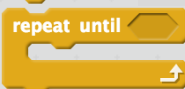
**LOOP UNTIL** (condition is true)  
    set of statements  
**END LOOP UNTIL**

or

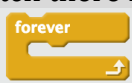
**LOOP WHILE** (condition is true)  
    set of statements  
**END LOOP WHILE**

The first form is used when the number of repetitions is known when the loop begins. In contrary, LOOP UNTIL and LOOP WHILE have an unknown number of repetitions. LOOP UNTIL is repeated until the condition becomes true whereas the LOOP WHILE is repeated while the condition is true.

In Scratch, the two kinds of loop statements are:

-  A yellow Scratch 'repeat' block with the number '10' in a white circle, a notch on the left, and a small arrow on the right.
-  A yellow Scratch 'repeat until' block with a notch on the left, a small arrow on the right, and a small arrow on the right side of the block.

In Scratch there is also a specific block for forever repeat of the embedded statements:

-  A yellow Scratch 'forever' block with a notch on the left, a small arrow on the right, and a small arrow on the right side of the block.

**Aim:**

- Repeat N times;
- Repeat until loop with 1 condition;
- Equivalence between repeat N times and repeat until;
- Repeat until loop with several conditions.

**Exercise 1:**

Write an algorithm that repeats 10 times the following statements:

- the sprite moves why 10 steps
- waits 1 second.

**Hint:** wait is in the Control menu

**Exercise 2:**

Add to the previous algorithm the fact that the sprite says “Step 1” after its first move, “Step 2” after the second one and so on...

**Hint:** an additional variable can be created in order to store the number of repetitions.

**Exercise 3:**

Write an algorithm that moves the sprite until its x position is greater or equals to 100.

**Exercise 4:**

Write an algorithm that computes and displays the sum of the 12 first integers with 2 alternative algorithms, one with a “repeat N times” loop and one with a “repeat until ” loop.

**Exercise 5:**




Write an algorithm to play the following game with the sprite.

The program chooses a random number between 1 and 20. Then it asks the user to choose a number and waits for an answer. It displays (says):

- “Too low! Try again”: if the user has proposed a number lower than the chosen number
- “Too high! Try again”: if the user has proposed a number greater than the chosen number
- “You win!”: if the user has proposed the chosen number.

The program should continue while the user does not find the right number.

Hints: the following blocks can be useful:

-  : to get a random value between 1 and 20. The value can thus be stored in a variable to be kept all along the algorithm.
-  shows a message and waits for the user to type a value. The user’s answer is stored in the  variable.

## Manipulation of sets (or list) of values

Up to now, variables we have created can only store one single value.

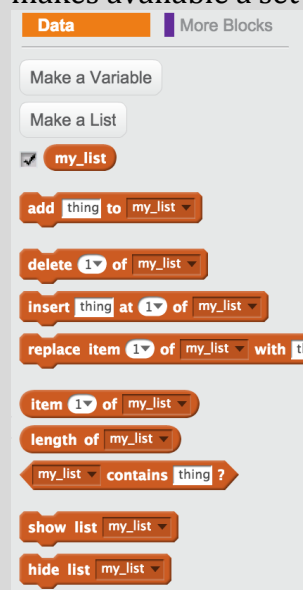
It can be useful to manipulate a set of values. The simplest examples in mathematics are vectors or matrices.

For example, if we want to create an application helping a teacher to manage marks of his students, it could be useful to manage together all the marks of a student to be able, for example, to compute his average mark. In addition, the number of marks can be different for two distinct students, as one can have missed some exams. A list has the advantage that the number of elements it contains is not fixed a priori.

In general, a list keeps the order in which elements have been inserted. In a list, elements can be added (at the end) or inserted at a given location (defined by an integer index value). They can also be removed or replaced.

Finally it is possible to get the *i*th element of the list (without removing it) or to get the length of the list (number of elements).

In Scratch, in the Data menu, we can also make a list (and give it a chosen name). This makes available a set of statements to manage lists.



### Aim:

- Store data in a list;
- Manipulate a list;
  - Get the max element;
  - Sort it.

### Exercise 1:



We consider the algorithm of the Exercice 3 of the section Conditional statements as startpoint (the sprite increases and decreases its speed from 0 to maximum speed).

We aim at storing all the x positions where the sprite has reached the maximum speed:

- Create a new list variable
- At initialization, empty the list (i.e. delete all the element of the list variable)
- Add to the list the x location each time the sprite reaches the maximum speed.

### **\*\*Exercice 2:**

Modify the previous algorithm in order that the sprite always says the maximum x of the list (i.e. the maximum of all the x at which it has reached the maximum speed)

**Hint:** a new variable could be useful. This variable can be computed each time the maximum speed is reached and an element added to the list.

### **\*\*Exercice 3:**

Modify the algorithm of the Exercice 1, in order that the list is always sorted. This means that every time a value is added, it should not be added at the end, but inserted at a location which keeps the list sorted.

### **Exercice 4:** (Memory game).

The sprite will ask you to memorize a list of numbers that increases every step.

The algorithm will randomly choose a number between 1 and 100 and store it in a list of numbers. It will say the new number to the user and ask him to memorize it. It then asks the user to type one by one each number of the list. When it has asked all the number of the list, it will choose a new random number, and so on...

The game ends as soon as the user type a wrong answer.

Example of game flow:

Sprite says: "The new number is: 35". (wait for 2 seconds)

Sprite says: "What is the number 1 ?"

User types: "35"

Sprite says: "The new number is: 47". (wait for 2 seconds)

Sprite says: "What is the number 1 ?"

User types: "35"

Sprite says: "What is the number 2 ?"

User types: "47"

Sprite says: "The new number is: 2". (wait for 2 seconds)

Sprite says: "What is the number 1 ?"

User types: "35"

Sprite says: "What is the number 2 ?"

User types: "47"

Sprite says: "What is the number 3 ?"

User types: "3"

Sprite says: "You lose. Game Over"

## Sub-algorithm: procedure and function

When algorithms become long in terms of number of lines and statements, it becomes useful in order to keep the algorithm clear and readable, to split a single algorithm in several parts (that can be procedures or functions).

Procedures or functions can also be useful when a same sequence of statements is used at several places in the algorithm. Then it becomes useful to factorize this part in a procedure or function.

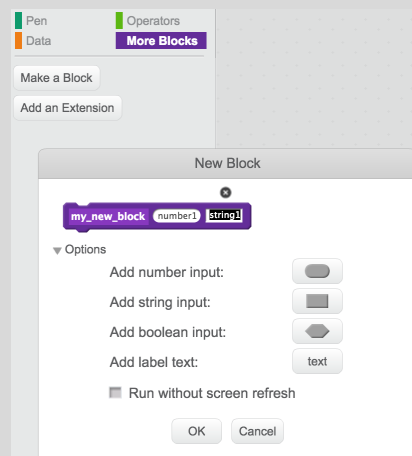
Procedures or functions can be defined as sub-algorithms which take some input and use them in computation to produce a result.

A function is a sub-algorithm which takes some inputs and returns a result. The inputs are not modified.

In contrarily, a procedure is a sub-algorithm which does not return any result but modify the value of inputs. A procedure is used when a sub-algorithm needs to produce several outputs.

Scratch allows the user to define new blocks to define sub-algorithms. It is limited on this point because a new block can neither return any value, nor modify its input parameters. It will be used to modify directly the variables of the sprite.

The More Blocks menu allows the user to define a new block, to modify its name and to add inputs (from various types).



The creation of a new block creates a new beginning of algorithm and a new block allowing to call this sub-algorithm anywhere in other algorithms. In the example below, the my\_new\_block is a new sub-algorithm, in which number1 and string1 are the two input parameters. number1 and string1 becomes also variables which can (and should) be used in the sub-algorithm only.



**Aim:**

- Define input / output of a procedure;
- Write a procedure;
- Call a procedure;
- Idea on functions.

**Exercise 1:**




The aim of this algorithm is to listen mouse clicks on the screen, to get the coordinates of the mouse, and to locate the sprite on the mouse location, its direction being in direction 90.

To this purpose, first define a new block MoveSprite, with 2 number inputs (new\_x and new\_y). It will locate the sprite on this new coordinates and make it point in direction 90. The two following blocks can be useful:  and .

We will now write the main part of the algorithm. Details of the algorithm:

- Wait until the user has clicked
- move the sprite at the location of the mouse (calling the new block MoveSprite)
- Then repeat forever:
  - move the sprite by 10 steps and if on edge, bounce
  - if the user has clicked somewhere, move the sprite at this location.

**Hints:**

- to create inputs for a block, click the “options” menu when you create it or, after it has already been created, right-click on it, select “edit” and click the “options” menu
- when defining a block in the algorithm window, its inputs can be taken from the definition of the block and slided as variables within the statements
- The block  if the user has clicked.
- If the user has clicked, the two variables  and  contains the coordinates of the click.

**\*\*Exercise 2: (Compute factorial)**

The aim of this algorithm is to compute and store in a list the factorial of the integers from 1 to 10 (or any N...). Note that the factorial of a positive integer number is defined by:  $n! = 1*2*3*...*(n-1)*n$ . For example:  $4! = 1*2*3*4 = 24$ .

- First create a variable fact and a list variable.
- Create a new block which computes the factorial of a number given in input. The result is stored in the fact variable.
- Create the main algorithm which computes, for each i from 1 to 10, the value of i! and add it in the list. The list contains thus at each index i the value of i!.

## Toward object/agent-oriented approach

We have now a global overview of the main principles and structures of algorithms.

In agent-based (and object-oriented) algorithms, all these principles are used. But in addition, the algorithm is designed by considering entities composing the phenomenon to model.

Each type of agent (or object) has its own variables describing its state and its own algorithms (functions and procedures) describing its behaviors. For instance all “car” objects or agents can be described with variables such as “color” or “brand”, and have behaviors such as “turn\_wheel” or “change\_break”.

All agents or objects of a same type will have the same variables and same algorithms describing their behaviors, but they will not act identically, as their state variables will be distinct for each agent or object. For example, 2 cars may not have the same brand, and depending on the brand, the “change\_break” behavior may act differently.

An idea of objects / agents can be found in Scratch with the sprites : you can define different sprites and give them different variables and algorithm, so they will act differently when you press the green flag. However it is not possible to define types of sprites as it is possible to do when using object-oriented programming.

The essence of object-oriented programming and agent-based models is to define agents/objects states and behaviors and then manage how they can interact with each others. This can be done by defining in agents / objects behaviors how they react when they perceive something from another agent / objects, and / or define behaviors which send messages to others.

Another level of control is generally done through a “scheduler” which controls centrally when an agent / object should trigger a behavior.

In Scratch, sprites can perceive distance to other sprites and interact by broadcasting messages and reacting to messages they receive. But these messages cannot be directed to one particular sprite and there is no other way for sprites to interact. Also there is no sense of a global “scheduler”, all sprites behaviors being triggered by events.

All these concepts of agents / objects and scheduler will be part of the MISS-ABMSS training, so we have not included exercises about it in this document. However if you are already curious, check for instance this scratch project where you can see how 2 sprites can be defined and interact with each other

<https://scratch.mit.edu/projects/12367177/>

### **Aim :**

- create different objects (sprites) with different behaviors and different variables
- understand the difference between local variables and global variables

- manage interactions between objects (sprites) through message sending and global variables writing

### Exercise 1

- we will use the exercise 5 from the “loops” section. The objective is to have sprites playing the guessing game together : one “game master” sprite who is having the others guessing and two “player” sprites who will have 2 different strategies to guess
- create 1 sprite “game master” and 1 sprite “player”
- create the “game master” behavior by modifying the “loops” section exercise 5 within the “game master” algorithm window
  - make sure chosen\_number is a local variable for “game master”. This means only “game master” sprite can access (read and write) this variable : chosen\_number is unknown to all other sprites than “game master”
  - replace interactions with user by interactions with “player” sprite :
    - replace ask statement by broadcast and wait statement. It means that “game master” will send a message to the “player” sprite and wait for the “player” sprite to run its reaction to the message to continue
    - add broadcast and wait statements to say statements (so that is still possible to follow what happens - messages do not appear in the execution window)
  - create a global variable named “answer”. This means this variable can be accessed (read and write) by all sprites : it will be written by “player” and read by “game master”. Replace the answer sensing (blue) by the answer global variable (orange)
- create the “player” behavior within the “player” algorithm window
  - player will need 2 local variables “lower” and “higher” so that it can adjust its guess. These variables should be initialised to respectively 0 and 20 at the beginning of the game
  - player should react to the message sent by game master :
    - when receiving “choose a number”, it should choose a number between lower and higher and write it in the “answer” variable
    - when receiving “too low” or “too high”, it should adjust its “lower” or “higher” variable
    - when receiving “you win”, it should be happy!

### Hints

- sprites can be added and modified in the sprites window in the lower left part of the screen: click on “new sprite” to create a new sprite and right-click on an existing sprite and then “info” to modify its name
- each sprite has its own algorithms : click on a sprite in the sprites window to get its algorithm window. This is equivalent to objects / agents having their own behaviors
- in the execution view, local variables appear with the name of their sprites, global variables appear with nothing
- messages statements are in the events section

**Try it :** create a second player, maybe with a different strategies and check who wins between the two players! As Scratch do not allow directed message, you will have to manage differentiation between player 1 and 2. Maybe the easiest way would be to have

2 global variables, answer1 for player 1 and answer2 for player 2, as well as distinct messages "1 - too low" / "2 - too low", etc.. This is a big limit of Scratch compared to proper object oriented programming where interactions between different objects can be managed much more transparently.

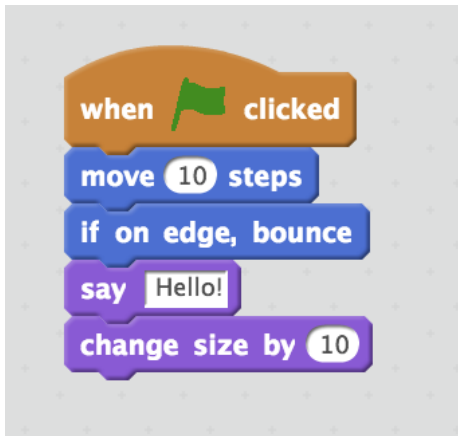
## References

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. Introduction to Algorithms (2nd ed.). McGraw-Hill Higher Education.

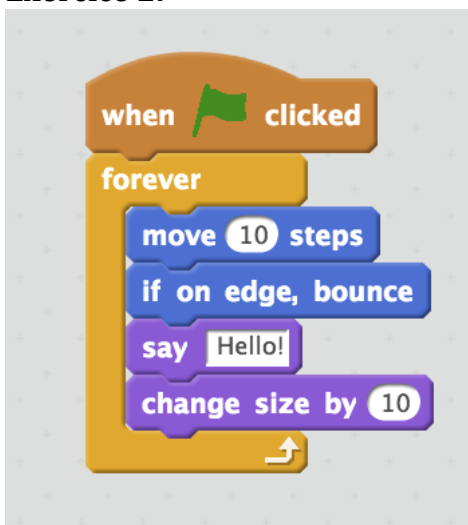
## Solutions

### Sequence of statements

#### Exercise 1:



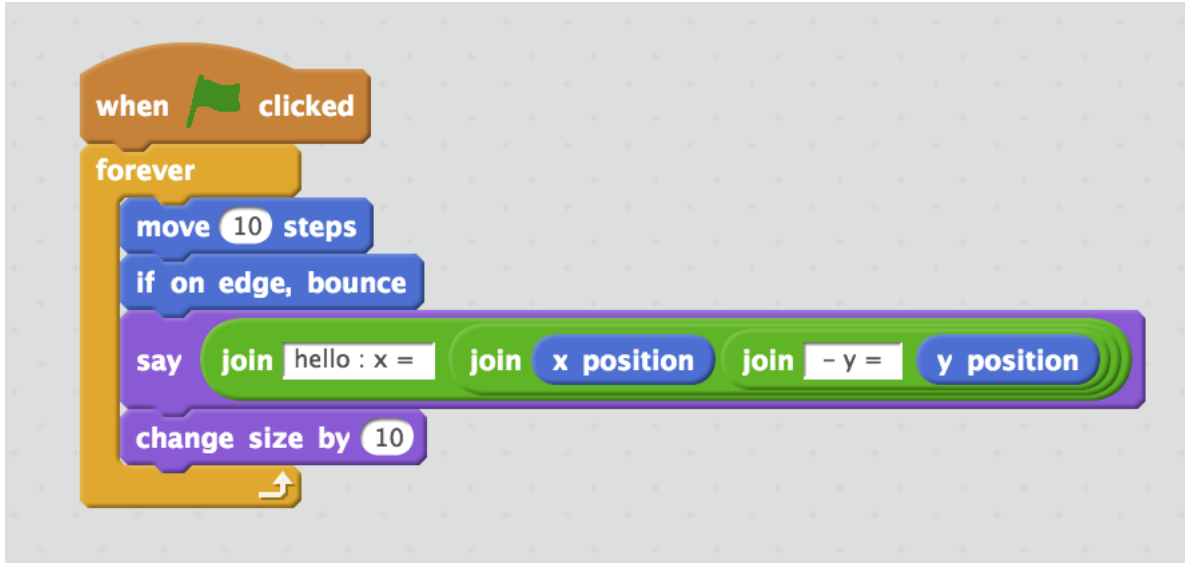
#### Exercise 2:



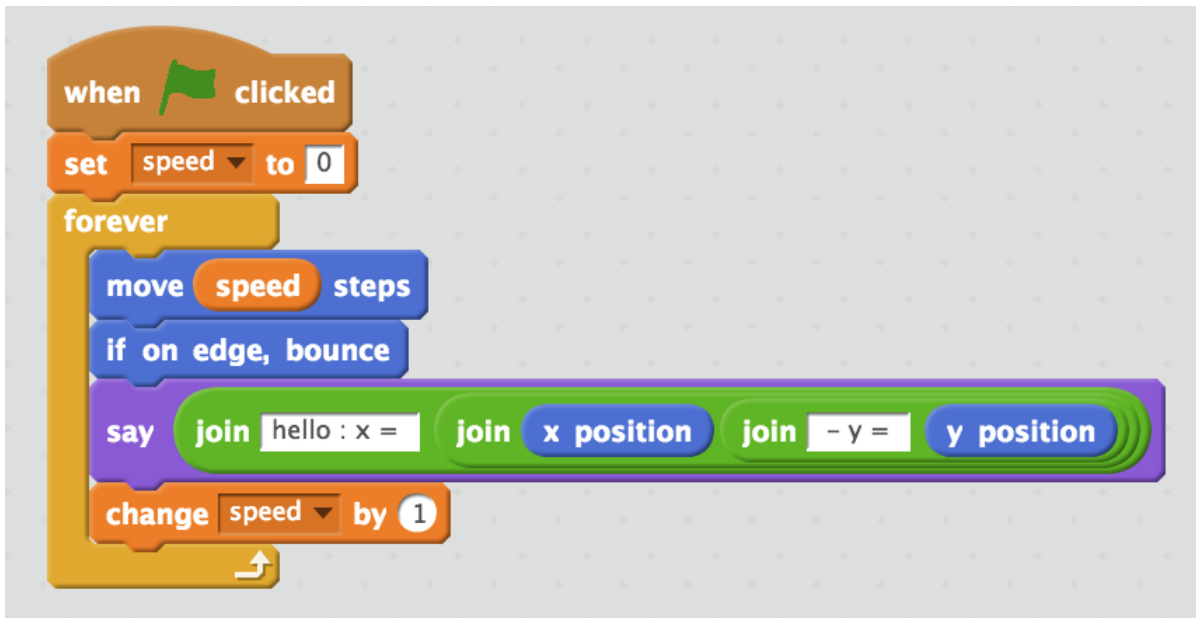
### Variables



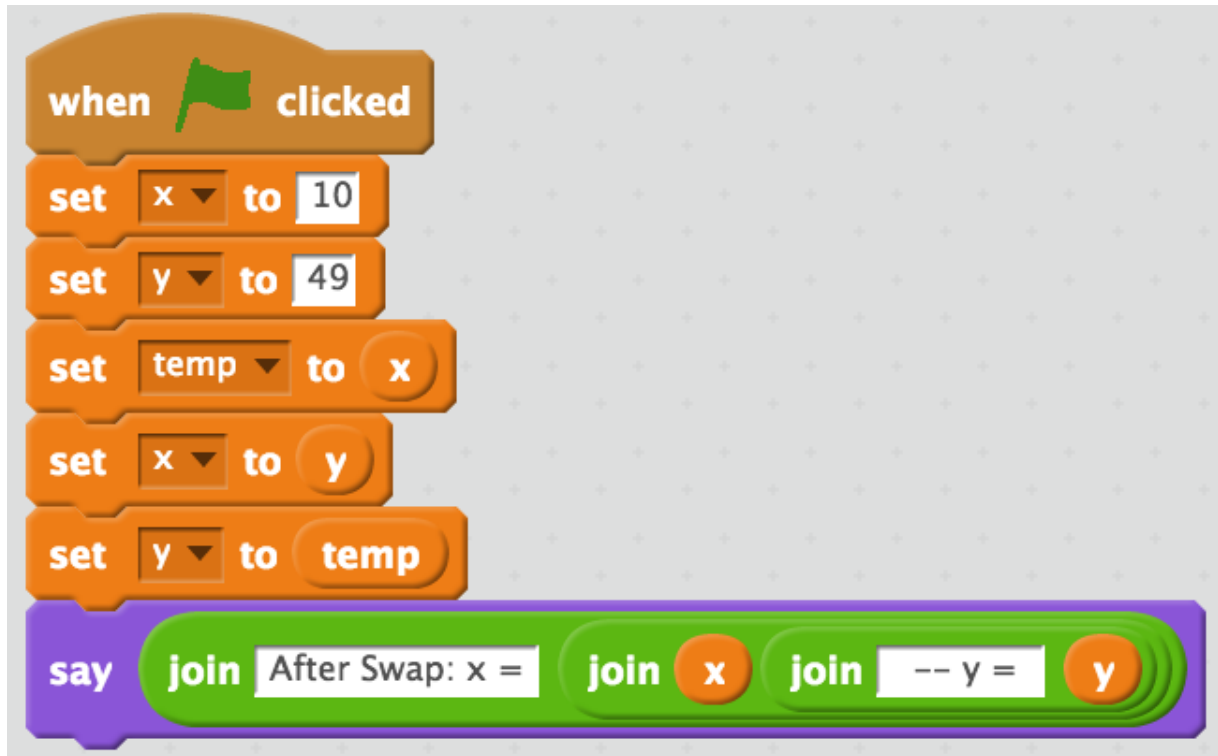
### Exercise 1:



### Exercise 2:

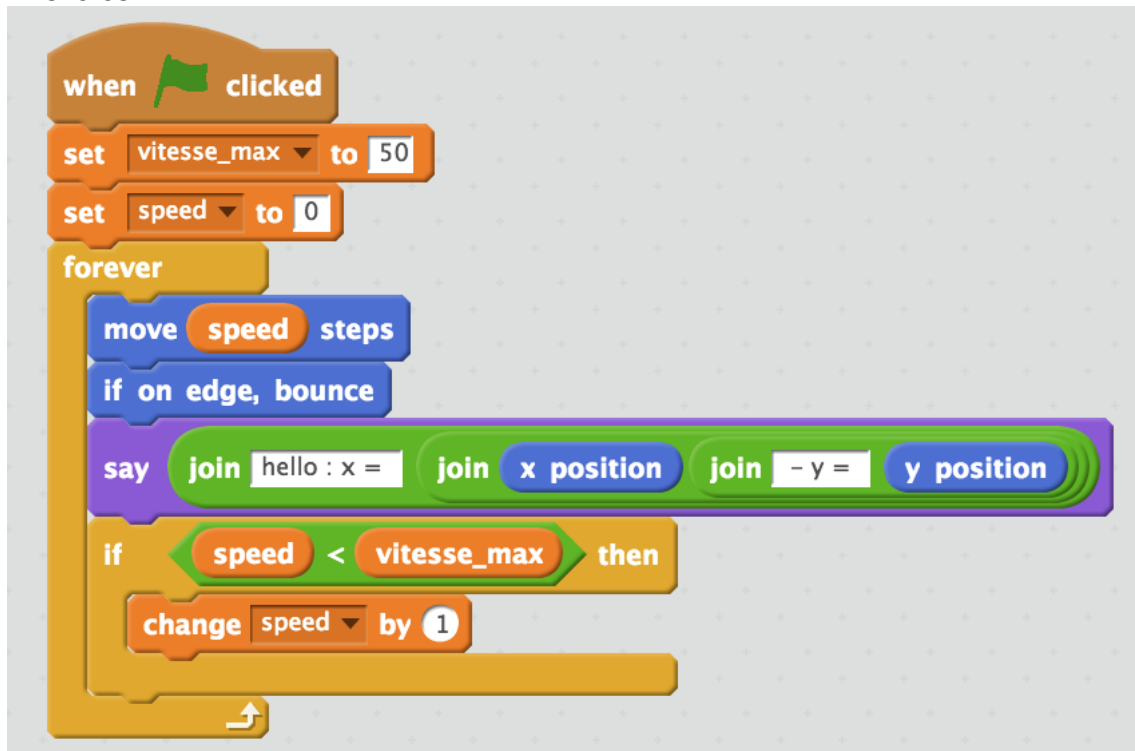


### Exercise 3:



### Conditional statements

#### Exercise 1:

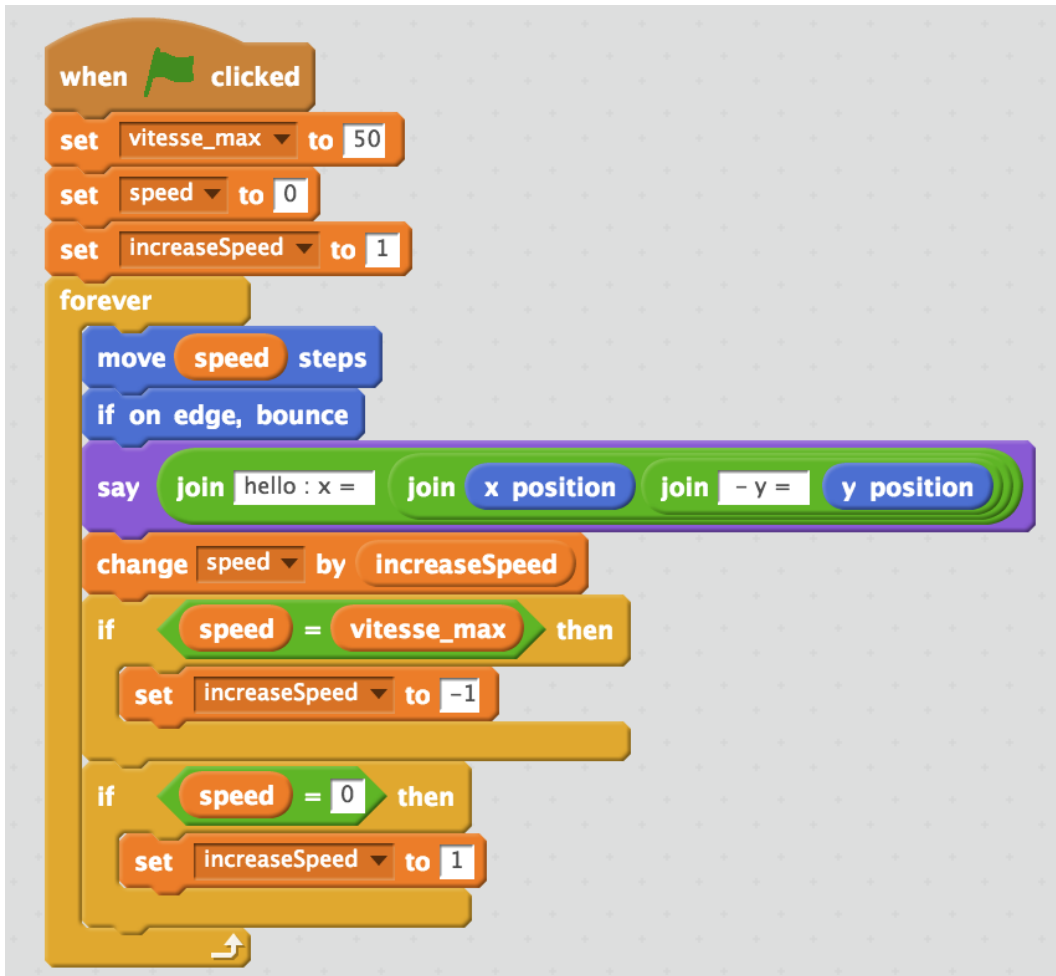


Exercise 2:

```
when green flag clicked
  set vitesse_max to 50
  set speed to 0
  forever loop
    move speed steps
    if on edge, bounce
    say join hello : x = join x position join - y = y position
    if speed < vitesse_max then
      change speed by 1
    else
      set speed to 0
```

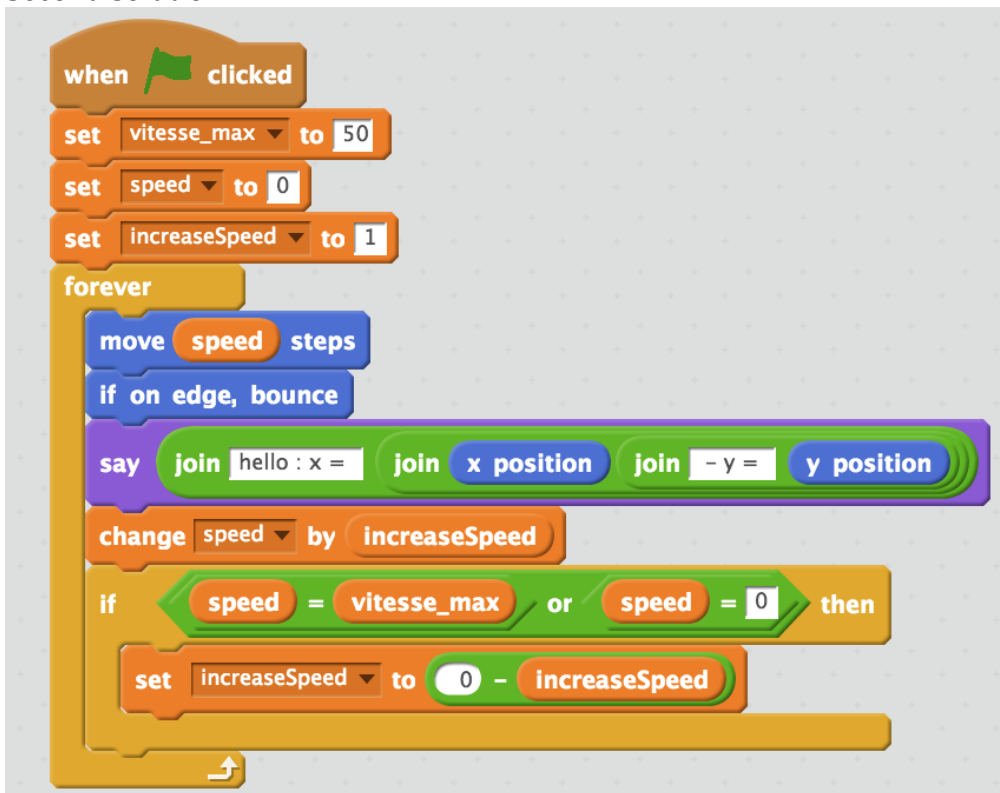
The image shows a Scratch script for a bouncing ball simulation. It starts with a 'when green flag clicked' event block. This is followed by two 'set' blocks: 'set vitesse\_max to 50' and 'set speed to 0'. A 'forever' loop contains the following blocks: 'move speed steps', 'if on edge, bounce', a 'say' block with three 'join' blocks ('hello : x =', 'x position', '- y =', 'y position'), an 'if' block with the condition 'speed < vitesse\_max'. The 'if' block has two branches: a 'then' branch with 'change speed by 1' and an 'else' branch with 'set speed to 0'. The loop ends with an upward arrow.

### Exercise 3: First solution:



```
when clicked
  set vitesse_max to 50
  set speed to 0
  set increaseSpeed to 1
  forever
    move speed steps
    if on edge, bounce
    say join hello: x = join x position join -y = y position
    change speed by increaseSpeed
    if speed = vitesse_max then
      set increaseSpeed to -1
    if speed = 0 then
      set increaseSpeed to 1
```

### Second solution:



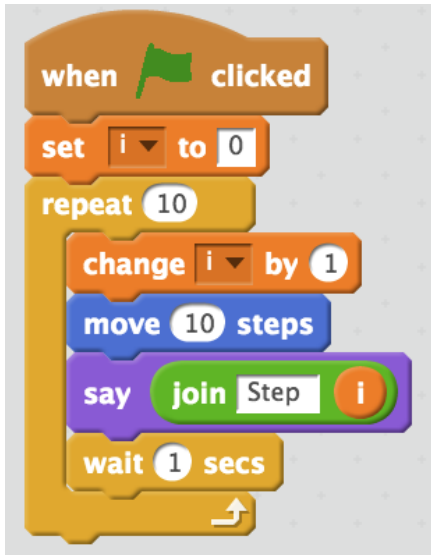
```
when clicked
  set vitesse_max to 50
  set speed to 0
  set increaseSpeed to 1
  forever
    move speed steps
    if on edge, bounce
    say join hello: x = join x position join -y = y position
    change speed by increaseSpeed
    if speed = vitesse_max or speed = 0 then
      set increaseSpeed to 0 - increaseSpeed
```

## Loop statements

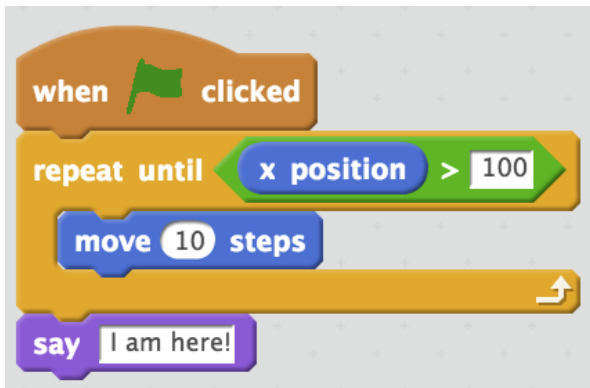
### Exercise 1:



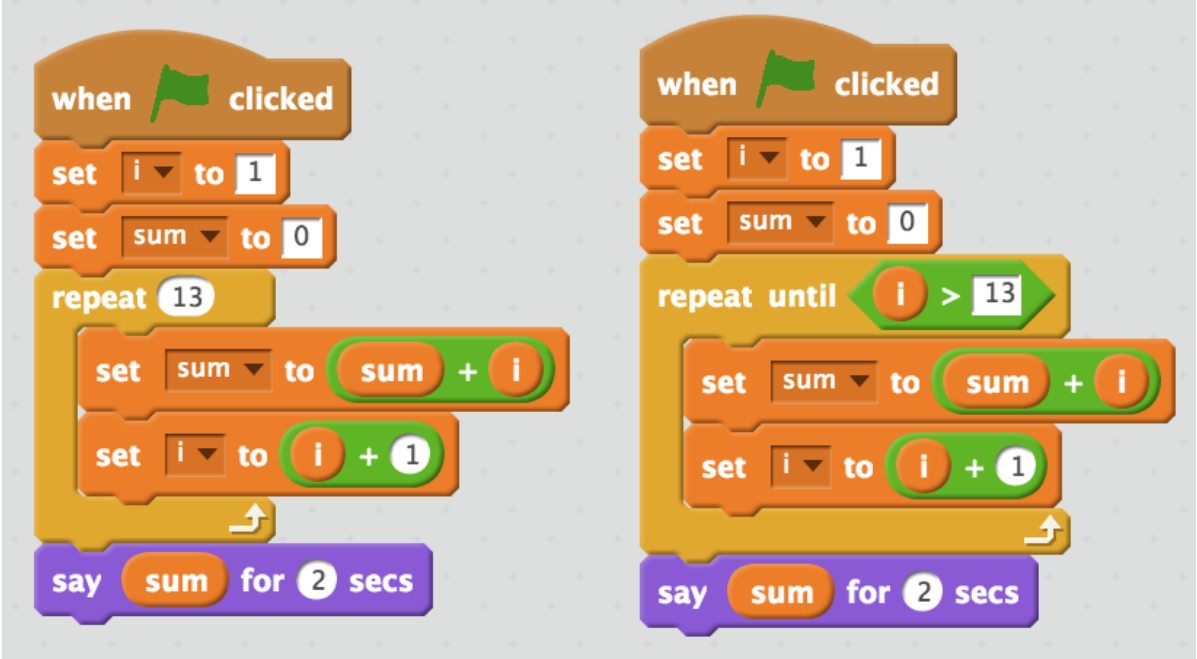
### Exercise 2:



### Exercise 3:



Exercise 4:



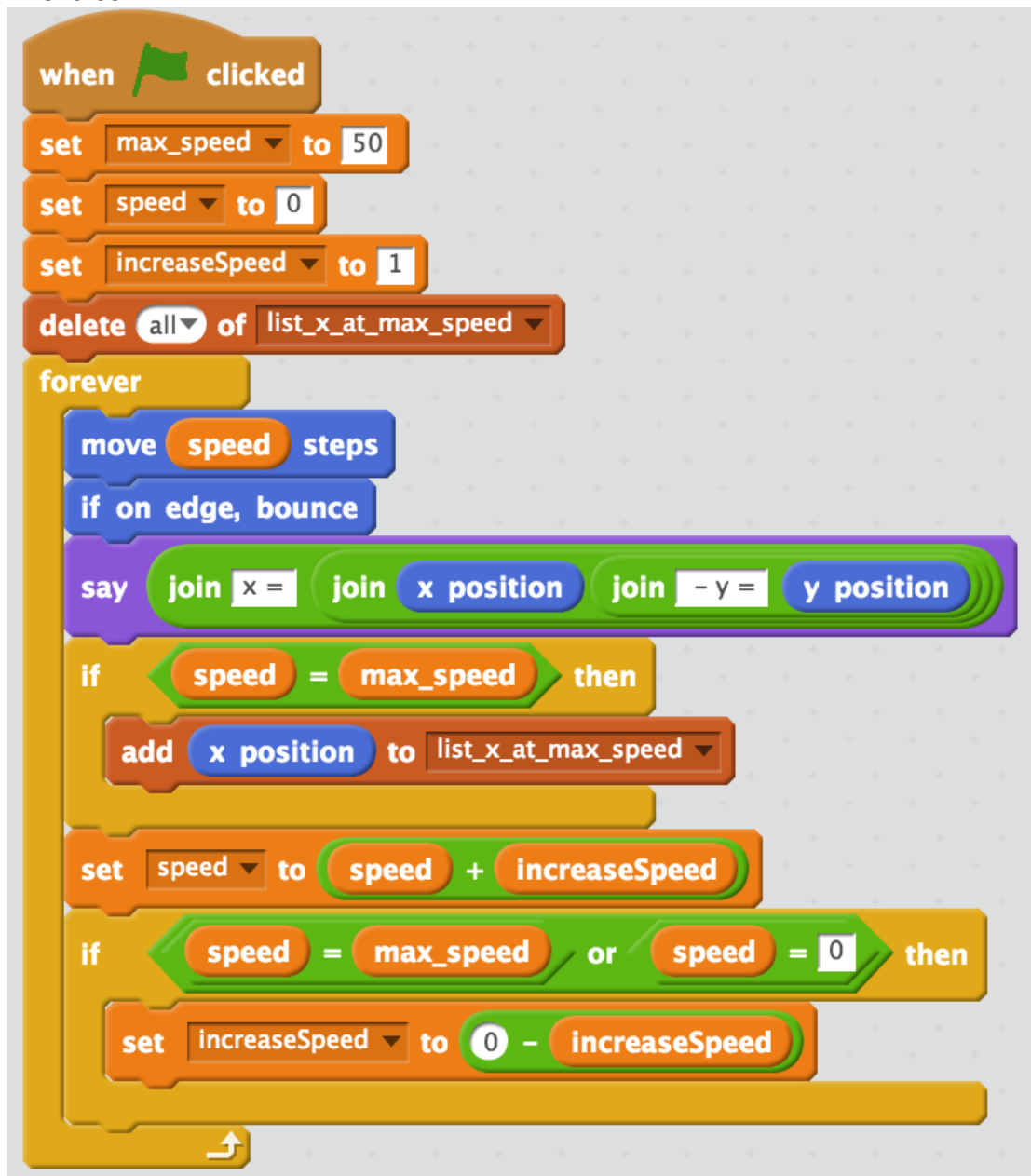
Exercise 5:

```
when clicked
  set chosenNumber to pick random 1 to 20
  repeat until answer = chosenNumber
    ask Chose a number and wait
    if answer < chosenNumber then
      say Too low ! Try again for 2 secs
    else
      if answer > chosenNumber then
        say Too high ! Try again for 2 secs
      else
        say Youv win for 2 secs
```

The image shows a Scratch script for a number-guessing game. It starts with a 'when clicked' event block. The first block is 'set chosenNumber to pick random 1 to 20'. This is followed by a 'repeat until' loop with the condition 'answer = chosenNumber'. Inside the loop, there is an 'ask' block with the text 'Chose a number and wait'. Below the 'ask' block is an 'if' block with the condition 'answer < chosenNumber'. If true, it says 'Too low ! Try again for 2 secs'. If false, it goes to an 'else' block. Inside the 'else' block, there is another 'if' block with the condition 'answer > chosenNumber'. If true, it says 'Too high ! Try again for 2 secs'. If false, it says 'Youv win for 2 secs'. The script ends with a return arrow block.

Manipulation of sets of values

### Exercise 1:

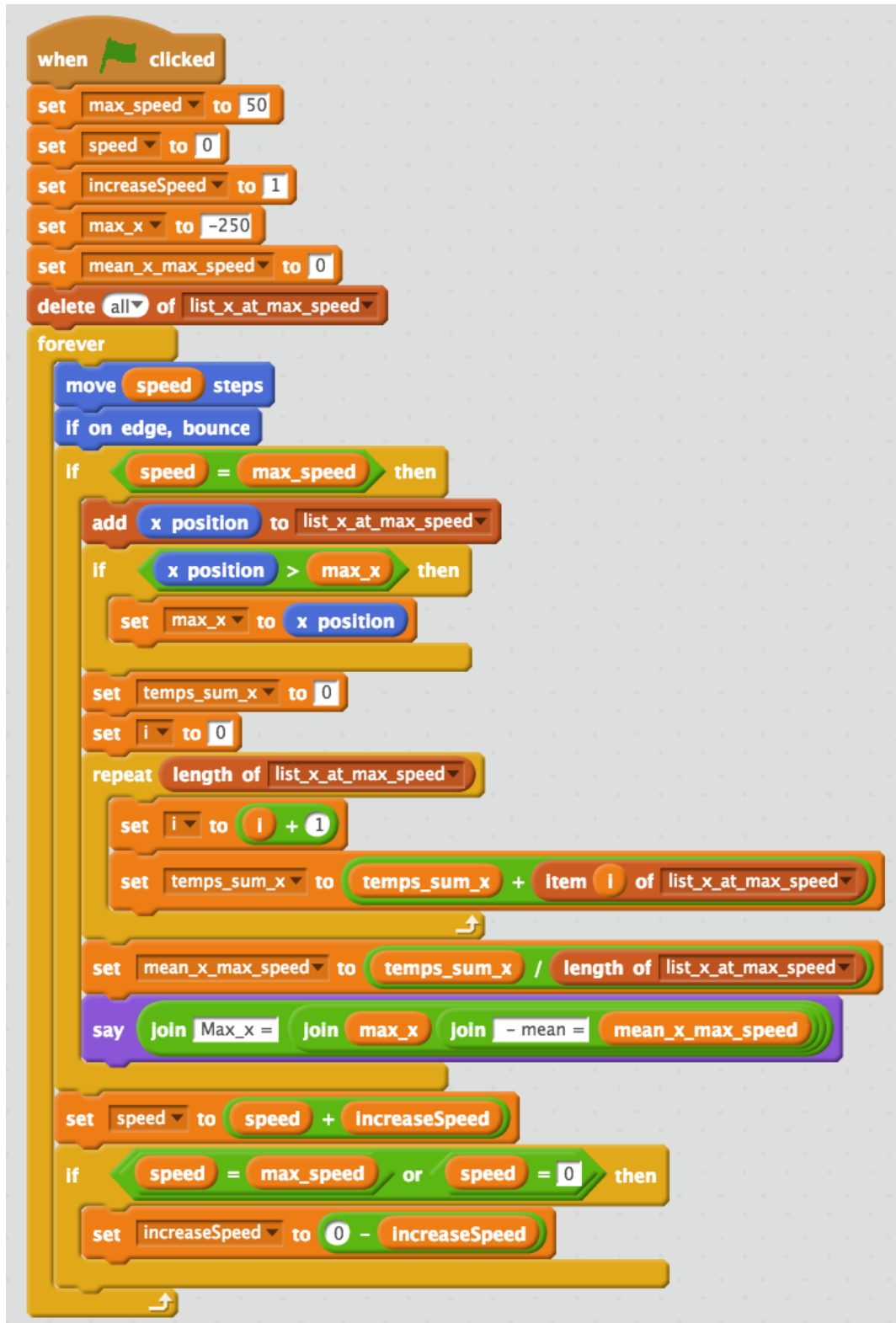


```
when green flag clicked
  set max_speed to 50
  set speed to 0
  set increaseSpeed to 1
  delete all of list_x_at_max_speed
  forever loop
    move speed steps
    if on edge, bounce
    say join x = join x position join -y = y position
    if speed = max_speed then
      add x position to list_x_at_max_speed
    set speed to speed + increaseSpeed
    if speed = max_speed or speed = 0 then
      set increaseSpeed to 0 - increaseSpeed
```

The image shows a Scratch script for a ball simulation. It starts with a 'when green flag clicked' event block. This is followed by four 'set' blocks: 'set max\_speed to 50', 'set speed to 0', and 'set increaseSpeed to 1'. The next block is 'delete all of list\_x\_at\_max\_speed'. A 'forever' loop contains several blocks: 'move speed steps', 'if on edge, bounce', a 'say' block with a join of 'x =', 'x position', '-y =', and 'y position', an 'if speed = max\_speed then' block with 'add x position to list\_x\_at\_max\_speed', a 'set speed to speed + increaseSpeed' block, and an 'if speed = max\_speed or speed = 0 then' block with 'set increaseSpeed to 0 - increaseSpeed'.



## Exercise 2:



```
when green flag clicked
  set max_speed to 50
  set speed to 0
  set increaseSpeed to 1
  set max_x to -250
  set mean_x_max_speed to 0
  delete all of list_x_at_max_speed
  forever loop
    move speed steps
    if on edge, bounce
    if speed = max_speed then
      add x position to list_x_at_max_speed
      if x position > max_x then
        set max_x to x position
      set temps_sum_x to 0
      set i to 0
      repeat length of list_x_at_max_speed
        set i to i + 1
        set temps_sum_x to temps_sum_x + item i of list_x_at_max_speed
      set mean_x_max_speed to temps_sum_x / length of list_x_at_max_speed
      say join Max_x = join max_x join - mean = mean_x_max_speed
    set speed to speed + increaseSpeed
    if speed = max_speed or speed = 0 then
      set increaseSpeed to 0 - increaseSpeed
```

The image shows a Scratch script for a physics simulation. It starts with a 'when green flag clicked' event. The script sets several variables: 'max\_speed' to 50, 'speed' to 0, 'increaseSpeed' to 1, 'max\_x' to -250, and 'mean\_x\_max\_speed' to 0. It then deletes all items from a list named 'list\_x\_at\_max\_speed'. A 'forever' loop follows, containing several steps: moving the object by 'speed' steps, bouncing off edges, and checking if the current speed equals the maximum speed. If so, it adds the current x position to the list, updates 'max\_x' if the current position is further right, and calculates the mean of the list. It then increases the speed by 'increaseSpeed' and checks if the speed has reached the maximum or become zero, in which case it reverses the direction of 'increaseSpeed'.

### Exercise 3:

```
when green flag clicked
  set max_speed to 50
  set speed to 0
  set increaseSpeed to 1
  delete all of list_x_at_max_speed

  forever
    move speed steps
    if on edge, bounce
    if speed = max_speed then
      set i to 1
      set found_location to false
      repeat until (i > length of list_x_at_max_speed or found_location = true)
        if item i of list_x_at_max_speed > x position then
          set found_location to true
          set i to i - 1
        set i to i + 1
      insert x position at i - 0 of list_x_at_max_speed
    set speed to speed + increaseSpeed
    if speed = max_speed or speed = 0 then
      set increaseSpeed to 0 - increaseSpeed
```

#### Exercise 4:

```
when green flag clicked
  delete all of Numbers_to_memorize
  set user_error to false
  repeat until user_error = true
    set new_number to pick random 1 to 100
    add new_number to Numbers_to_memorize
    say join The New number is new_number for 2 secs
    set i to 0
    repeat until i = length of Numbers_to_memorize or user_error = true
      set i to i + 1
      ask join What is the number join i ? and wait
      if not item i of Numbers_to_memorize = answer then
        set user_error to true
        say You lose! for 2 secs
  stop all
```

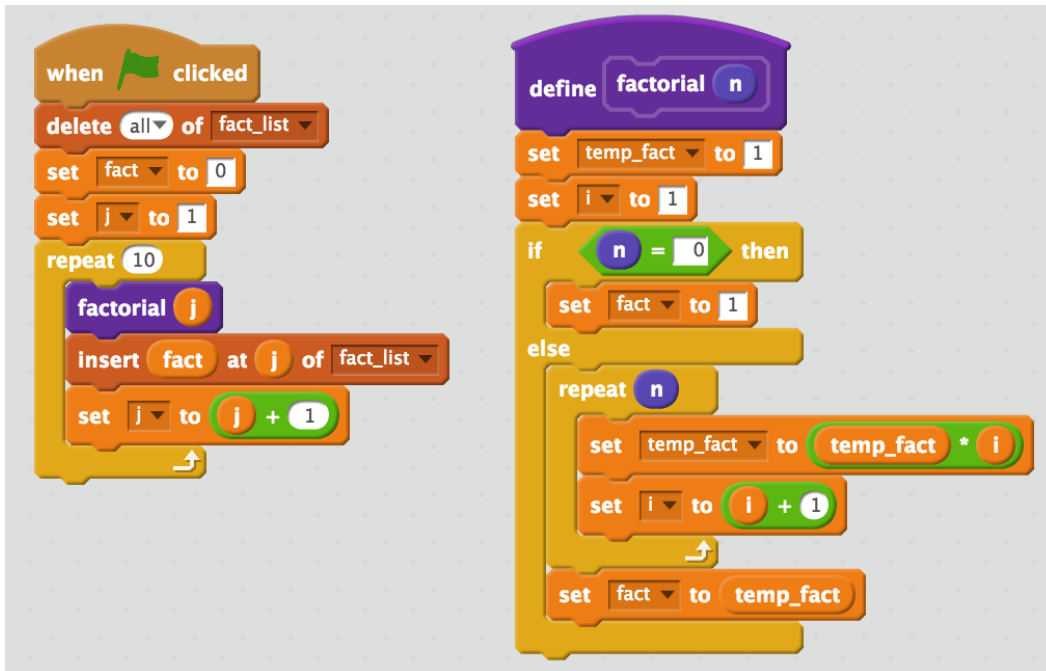
#### Sub-algorithm: procedure and function

#### Exercise 1:

```
when green flag clicked
  wait until mouse down?
  MoveSprite mouse x mouse y
  forever
    move 10 steps
    if on edge, bounce
    if mouse down? then
      MoveSprite mouse x mouse y

define MoveSprite new_x new_y
  go to x: new_x y: new_y
  point in direction 90
```

## Exercise 2:



Towards object /agent-oriented approach

## Exercise 1:

Sprites X: 31 y: -108

New sprite:

game mas...

player

x: -147  
y: -15

```

when clicked
  set chosen_number to pick random 1 to 20
  repeat until answer = chosen_number
    broadcast choose a number and wait
    if answer < chosen_number then
      say too low.. try again! for 2 secs
      broadcast too low.. try again! and wait
    else
      if answer > chosen_number then
        say too high try again! for 2 secs
        broadcast too high.. try again! and wait
      else
        say you win! for 2 secs
        broadcast you win! and wait
  
```

```

when clicked
  set higher to 20
  set lower to 0

when I receive choose a number
  set answer to pick random lower to higher
  say answer for 2 secs

when I receive too high.. try again!
  set higher to answer - 1

when I receive too low.. try again!
  set lower to answer + 1

when I receive you win!
  point in direction 180
  wait 1 secs
  point in direction 90
  say Youpi!
  
```

x: 145  
y: 112