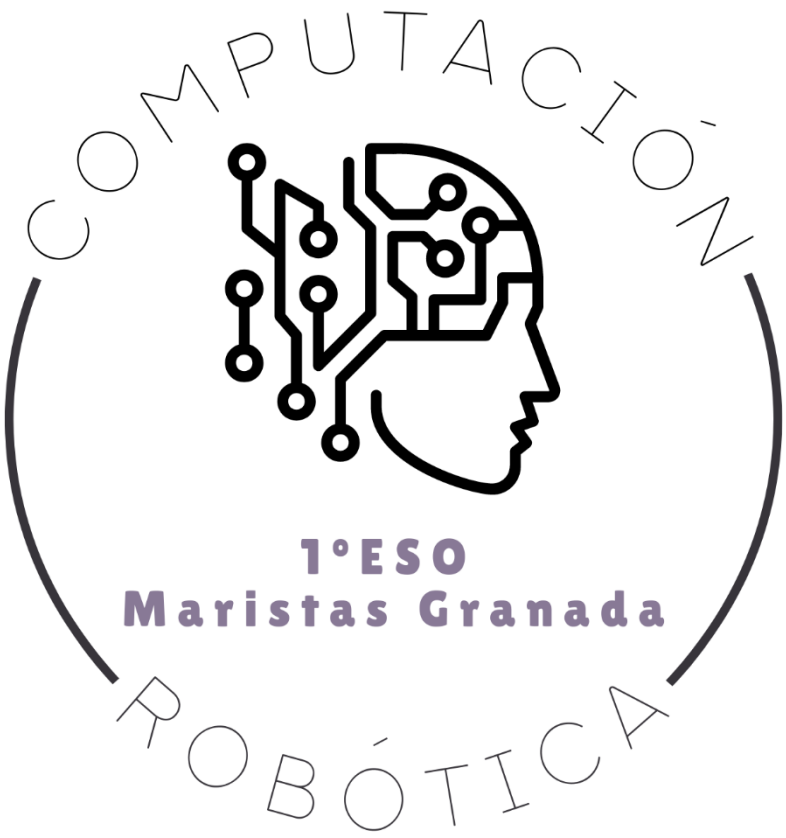
CURSO 2024-2025



DOCUMENTO DE TRABAJO: SESIONES 1-8

COMPUTACIÓN Y ROBÓTICA 1ºESO

COLEGIO MARISTA LA INMACULADA
CALLE SÓCRATES, 8
18002 - GRANADA

Índice

“Knowledge isn't free. You have to pay attention.”

— Richard P. Feynman

Teoría

1. ¿Qué es un algoritmo?

La asignatura de Computación y Robótica nos enseñará a resolver problemas a la manera en que lo resuelven los ordenadores. Trabajaremos **problemas reales** que deben resolverse tras analizar un **conjunto de datos de partida**.

Un ordenador "no comprende" como los humanos. Debemos "explicar" al ordenador el problema de partida dividiéndolo en problemas más pequeños y sencillos. Este método lo llamaremos "**divide y vencerás**". Cada pequeño problema debe resolverse siguiendo un conjunto de pasos ordenados que llamaremos **algoritmo**. Este algoritmo puede ser "entendido" por un ordenador si:

- El orden de ejecución de cada paso está marcado claramente.
- El algoritmo se traduce a un lenguaje técnico reconocible por el ordenador, formando así un código de programación.

¿Qué significa que un ordenador piense como un ser humano? ¿Piensan realmente las inteligencias artificiales (IA) en la actualidad?

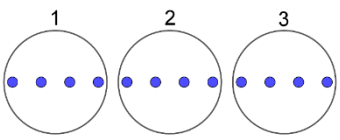
¿Cómo se programa un ordenador para que resuelva un problema?

- Analizamos los datos iniciales.
- Dividimos el problema en pequeñas partes.
- Resolvemos cada parte mediante un conjunto ordenado de pasos llamado algoritmo.
- Escribimos el algoritmo en un lenguaje entendible por el ordenador, creando así un código de programación.

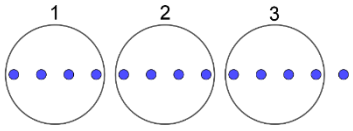
1.1. Ejemplos de algoritmos para saber si un número es divisible entre 4

Dividir es hacer grupos. Si un número "n" es divisible entre 4 significa que podemos encontrar un número exacto de grupos de 4 unidades dentro del número "n", sin que sobre ninguna cantidad. Por ejemplo: El número 12 es divisible entre 4 porque podemos encontrar tres grupos de 4 dentro de 12, y no sobra nada. El número 13 no es divisible entre 4 porque, aunque podemos encontrar tres grupos de 4 dentro de 12, sobra 1 unidad. Y dentro del número 1 no podemos encontrar más grupos de 4.

$\frac{12}{4} = 3$



$\frac{13}{4} = 3 + \frac{1}{4}$



¿Cómo podemos saber si un número grande, como 38.536, es divisible por 4?

Resolvamos esta pregunta aplicando los tres primeros elementos que definen al pensamiento computacional:

- Análisis de datos.
- División del problema en pequeñas partes.
- Creación de un algoritmo de resolución.

Los datos para resolver esta pregunta son: El número de partida (38.536), el divisor (4) y el concepto de divisibilidad entre 4 (dividir el número de partida en un número exacto de grupos de 4 unidades, sin que sobre ninguna cantidad).

Primera propuesta de solución para saber si 38.536 es divisible entre 4: Si sabemos multiplicar (o pensado de otra manera, si un ordenador sabe multiplicar), podemos hacer de manera ordenada la tabla del 4 e ir comprobando si aparece el valor 38.536. Así dividimos el gran problema de partida en miles de pasos intermedios, empezando por la operación matemática 1x4. En cada paso, tras realizar la multiplicación, debemos comprobar si el resultado obtenido coincide con el número que buscamos.

En cada paso aplicamos el siguiente algoritmo:

1. Hacemos la multiplicación.
2. Comparamos el resultado de la multiplicación con el dato 38.356. Pudiendo ocurrir tres cosas:
   2.1. Si el resultado es inferior al dato 38.356, pasamos al siguiente número natural de la tabla del 4. Repetimos el algoritmo desde el inicio.
   2.2. Si el resultado es superior al dato 38.356, el número no es divisible entre 4. Fin del algoritmo.
   2.3. Si el resultado es igual al dato 38.356, significa que el número sí es divisible entre 4. Fin del algoritmo.

A continuación, mostramos la aplicación del algoritmo a unos cuantos pasos del proceso de resolución:

- Paso 1. Una vez 4 es 4. Cantidad inferior a 38.356. Seguimos probando.
- Paso 2. Dos veces 4 es 8. Cantidad inferior a 38.356. Seguimos probando.
- Paso 3. Tres veces 4 es 12. Cantidad inferior a 38.356. Seguimos probando.
- …
- Paso 100. Cien veces 4 es 400. Cantidad inferior a 38.356. Seguimos probando.
- …
- Paso 1.000. Mil veces 4 es 4.000. Cantidad inferior a 38.356. Seguimos probando.
- …
- Paso 5.000. Cinco mil veces 4 es 20.000. Cantidad inferior a 38.356. Seguimos probando.
- …
- **Paso 9.634.** Nueve mil seiscientas treinta y cuatro veces 4 es 38.536. Cantidad idéntica a 38.356. El número es divisible entre 4. **Fin del algoritmo.**

¿Ves este método rápido y cómodo para operar con números muy grandes?

Segunda propuesta de solución para saber si 38.536 es divisible entre 4: Hacer la división con el método que aprendimos en Primaria y comprobar si el resto de la división es igual a 0. Si lo piensas bien, el método de Primaria solo usa multiplicaciones y restas. Por lo tanto, debemos partir de la base de que nuestro ordenador sepa restar además de multiplicar.

3	8.	5	3	6	⌊	4
	2	5				9.634
		1	3			
			1	6		
				0		

El algoritmo de la división opera de izquierda a derecha, hasta llegar a las unidades. Así hacemos grupos de 4 con las decenas de millar, luego con las unidades de millar, luego con las centenas, luego con las decenas y finalmente con las unidades. El algoritmo sería así:

1. Formamos un número de referencia cogiendo la primera cifra del dato 38.536 (en nuestro ejemplo, sería 3).
   1.1. Si es inferior a 4, cogemos también la siguiente cifra (en nuestro ejemplo, tendríamos 38). Con esto garantizamos que aparezcan grupos de 4 unidades.
   1.2. En caso contrario, nos quedamos solo con la primera cifra.
2. Aplicamos la tabla del 4, partiendo de 1x4.
   2.1. Si el resultado es inferior al número de referencia, pasamos al siguiente número natural de la tabla del 4.
   2.2. Si el resultado es igual al número de referencia, significa que el resto de la división vale 0. Escribimos el resto 0 debajo de la última cifra del número de referencia.
   2.3. Si el resultado es superior al número de referencia, significa que nos hemos pasado. Por lo que nos quedamos con el número natural anterior y escribimos el resto debajo de la última cifra del número de referencia.
3. Miramos si quedan más cifras en el dato inicial:
   3.1. Si quedan más cifras en el dato inicial, formamos un número con el resto y con la siguiente cifra del dato inicial. Repetimos el algoritmo desde el apartado 1.1., sabiendo que si tenemos que bajar más de una cifra del número inicial para conseguir una cantidad igual o mayor a 4, deberemos añadir ceros sucesivamente en el número que forma el cociente.
   3.2. Si no quedan más cifras en el dato inicial, fin del algoritmo.
      3.2.1. Si el resto final es 0, el dato inicial será divisible entre 4.
      3.2.2. En caso contrario, no será divisible entre 4.

Así quedarían los pasos de ejecución del algoritmo de la segunda propuesta. Comprobarás que reducimos bastante el número de pasos. Este algoritmo no es aún entendible por un ordenador (no es un código de programación), pero sí es un esquema previo entendible por los humanos, que precede al proceso de escribir el algoritmo en un lenguaje entendible por una máquina.

- Paso 1. Miro la primera cifra del número 38.536. Es el número 3. Como es menor que 4, cogemos las dos primeras cifras: 38.
- Paso 2. Una vez 4 es 4. Inferior a 38. Seguimos probando.
- Paso 3. Dos veces 4 es 8. Inferior a 38. Seguimos probando.
- Paso 4. Tres veces 4 es 12. Inferior a 38. Seguimos probando.
- …
- Paso 10. Nueve veces 4 es 36. Inferior a 38. Seguimos probando.

- Paso 11. Diez veces 4 es 40. Superior a 38. Nos quedamos con el número natural anterior. Nueves veces 4 es 36. Al restar 38 – 36 nos queda resto 2.
- Paso 12. Escribo junto al resto 2 la siguiente cifra, el número 5. Obtenemos 25. Es un número mayor que 4.
- Paso 13. Una vez 4 es 4. Inferior a 25. Seguimos probando.
- Paso 14. Dos veces 4 es 8. Inferior a 25. Seguimos probando.
- Paso 15. Tres veces 4 es 12. Inferior a 25. Seguimos probando.
- …
- Paso 18. Seis veces 4 es 24. Inferior a 25. Seguimos probando.
- Paso 19. Siete veces 4 es 28. Superior a 25. Nos quedamos en el número natural anterior. Seis veces 4 es 24. Al restar 25 – 24 nos queda resto 1.
- Paso 20. Escribo junto al resto 1 la siguiente cifra, el número 3. Obtenemos 13. Es un número mayor que 4.
- Paso 21. Una vez 4 es 4. Inferior a 13. Seguimos probando.
- Paso 22. Dos veces 4 es 8. Inferior a 13. Seguimos probando.
- Paso 23. Tres veces 4 es 12. Inferior a 13. Seguimos probando.
- Paso 24. Cuatro veces 4 es 16. Superior a 13. Nos quedamos en el número natural anterior. Tres veces 4 es 12. Al restar 13 – 12 nos queda resto 1.
- Paso 25. Escribo junto al resto 1 la siguiente cifra, el número 6. Obtenemos 16. Es un número mayor que 4.
- Paso 26. Una vez 4 es 4. Inferior a 16. Seguimos probando.
- Paso 27. Dos veces 4 es 8. Inferior a 16. Seguimos probando.
- Paso 28. Tres veces 4 es 12. Inferior a 16. Seguimos probando.
- Paso 29. Cuatro veces 4 es 16. Coincide con 16. Al restar 16 – 16 el resto vale 0.
- **Paso 30.** El resto vale 0 y compruebo que no quedan más cifras en el número de partida. El número es divisible entre 4. **Fin del algoritmo.**

La primera propuesta daba 9.634 pasos. La segunda propuesta "solo" necesita 30 pasos para resolver el problema. Siempre que redactemos un programa para un ordenador, buscaremos que la máquina pueda leer el programa y ejecutarlo en el menor tiempo posible. Así haremos nuestros programas más **eficientes:** resolver un problema de forma correcta en el menor tiempo posible.

Estamos escribiendo el algoritmo con palabras entendibles por los seres humanos. Esto se conoce como **pseudocódigo: un algoritmo fácilmente entendible por los humanos y que se asemeja a la estructura de un código de programación, como los que usan los ordenadores para funcionar**. Pero un pseudocódigo no es aún un código de programación entendible por un ordenador.

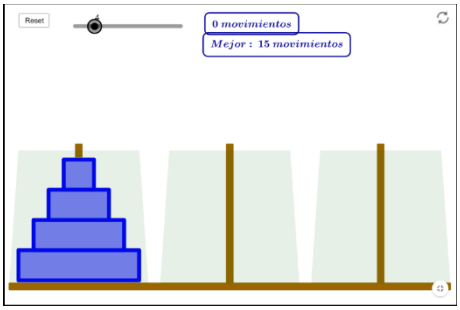
1.2. Encontrar patrones de repetición para acortar los algoritmos

Los ordenadores son muy buenos repitiendo operaciones. Si somos capaces de encontrar patrones de repetición en la solución de un problema, podremos usar las cualidades de un ordenador para resolver el problema más rápido. **La detección de patrones o de simetrías es uno de los objetivos del pensamiento computacional, porque ayuda a dividir el problema inicial en un número cerrado de pequeñas partes que podemos repetir tantas veces como necesitemos**.

Por ejemplo: Todos los números divisibles entre 2 tienen como cifra final un número par (0, 2, 4, 6 u 8). Si nos damos cuenta de este patrón, solo tendremos que mirar la cifra final para resolver el problema de la divisibilidad entre 2. Y ahorraremos mucho tiempo de ejecución en las operaciones de un código de programación que decida si un número es par.

1.3. Ejemplo de patrones de repetición en el juego de las torres de Hanoi

El juego de las torres de Hanoi es otro ejemplo de problema con patrones de repetición. Es un rompecabezas ideado en el s. XIX. Tenemos tres varillas verticales y unos discos de diferente diámetro insertados en la primera de las varillas (extremo izquierdo). Los discos están ordenados, de arriba hacia abajo, de menor a mayor radio.



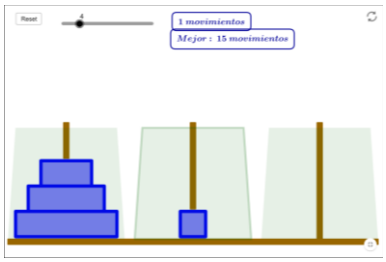
El juego consiste en llevar la formación inicial a la tercera varilla (situada en el extremo derecho), cumpliendo las siguientes normas:

- Solo se puede mover un disco en cada movimiento.
- De cada varilla, en cada movimiento, solo se puede coger el disco que está situado más arriba en la torre.
- Un disco más grande no puede descansar sobre un disco más pequeño.

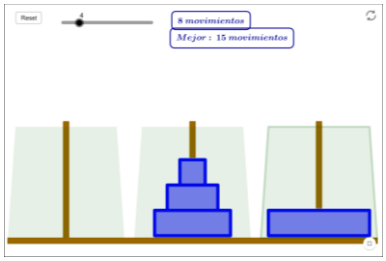
El siguiente enlace de Geogebra te permite jugar de manera online: https://www.geogebra.org/m/NqyWJVra. En la animación de Geogebra puedes cambiar el número de discos iniciales. La animación también te informa del número mínimo de pasos para poder resolver el rompecabezas. Por ejemplo: Para 2 discos se necesitan 3 pasos como mínimo; para 3 discos se necesitan 7 pasos como mínimo. Si consigues superar el juego en el menor número de pasos posibles, la animación escribirá "Perfect!" al finalizar.

Además de esta animación online, el profesor repartirá en clase (por grupos) distintos juegos de la torre de Hanoi para practicar manualmente. Comprueba lo siguiente: Si el número de discos iniciales es par (2, 4, 6, 8, etc.) el primer movimiento óptimo es colocar el disco más pequeño en la varilla central. Pero si el número de discos iniciales es impar (3, 5, 7, 9, etc.) el mejor movimiento es colocar el disco más pequeño en la varilla libre del otro extremo (que es la varilla final donde deseamos llevar todos los discos al terminar el juego). Este patrón se puede aplicar, continuamente, conforme vamos jugando.

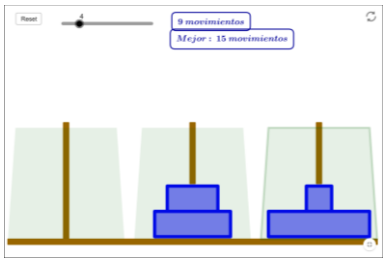
Por ejemplo: Si partimos de 4 discos, el primer movimiento es llevar el disco más pequeño a la varilla central.



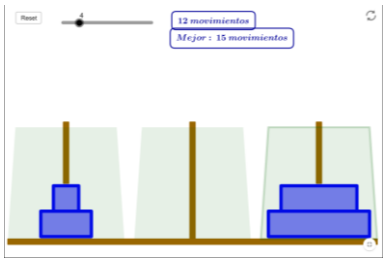
Si realizamos 8 movimientos óptimos, llegaremos a una torre de solo 3 discos.



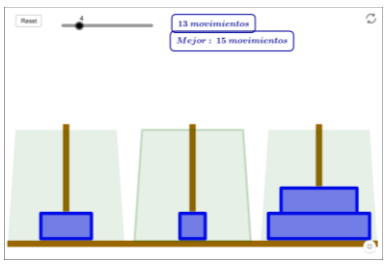
Esta torre de 3 discos está formada por un número impar de discos. Siguiendo el patrón que hemos descrito anteriormente, al tener un número impar de discos, el siguiente movimiento es llevar el disco más pequeño a la varilla final donde deseamos colocar todas las piezas al terminar.



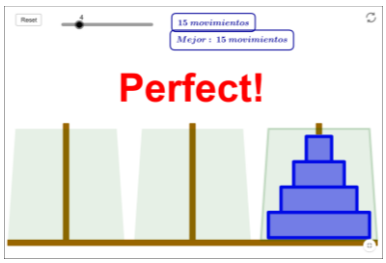
Tras 12 movimientos óptimos, llegaremos a una configuración con una torre de solo 2 discos en la varilla de la izquierda.



Esta torre de 2 discos en la varilla de la izquierda está formada por un número par de piezas. Siguiendo nuestro patrón, el siguiente movimiento es llevar el disco más pequeño a la varilla central.



Los dos movimientos finales son bastante evidentes, para terminar el juego en solo 15 pasos.



1.4. Primer ejemplo de programación por bloques con la web https://studio.code.org

Un **programa** es la traducción de un algoritmo a un lenguaje capaz de ser entendido y procesado por un ordenador. Ahora no vamos a hablar de pseudocódigos entendibles por los humanos, sino de programas entendibles directamente por los ordenadores.

En la actualidad existen miles de lenguajes para programar un ordenador. Como ejemplo inicial, veremos el lenguaje de programación por bloques disponible en la web https://studio.code.org. En primer lugar, visualizaremos el siguiente vídeo:

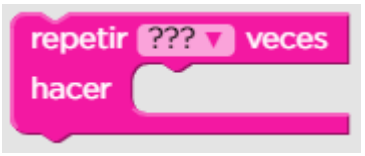
https://www.youtube.com/watch?v=bQilo5ecSX4

En segundo lugar, accederemos a la lección "2. The Maze" compuesta por 20 ejemplos de programación por bloques, de dificultad creciente:

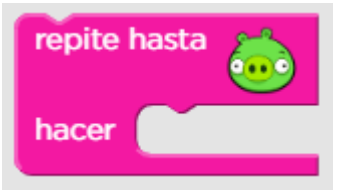
https://studio.code.org/s/20-hour

Para superar los ejemplos, te será útil conocer previamente la funcionalidad de algunos bloques. Podemos desplazar los objetos en pantalla usando los **bloques "avanzar" o "girar"**. Iremos ensamblando unos bloques con otro como piezas de puzzle, y se irán ejecuando de arriba hacia debajo de manera ordenada.

Avanzar: Desplaza el objeto una casilla. **Girar:** Rota el objeto en el sentido que digamos, pero sin desplazarse de la casilla en la que se encuentra.

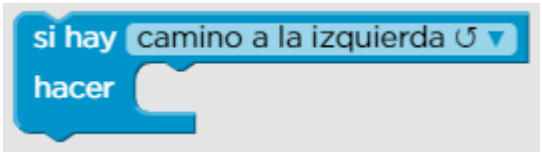


Un **bucle** en programación es una parte del código que permite ahorrar líneas de código, porque repite una y otra vez los mismos comandos para ejecutar una serie de acciones repetitivas. Por ejemplo, el **bloque "repetir"** es un bucle. Todo lo que esté dentro del bloque "repetir" (ver imagen de la derecha) se repetirá tantas veces como indiquemos. Hasta que no se termine de repetir ese número de veces, el ordenador no continuará leyendo el siguiente código que se encuentre por debajo del bloque "repetir".



Otro tipo de bucle es el **bloque "repetir hasta…"**. Este bloque impone una condición: Se repite lo que está dentro del bucle hasta que ocurra una cosa concreta. En la imagen de la derecha, el bloque se ejecutará hasta que se alcance la posición del objeto identificado con un cerdito de color verde.

En ocasiones un programa necesita ejecutar unos comandos si sucede una cosa, y otros comandos si sucede la contraria. Para esto se utilizan los **bloques "Si hay … hacer"** o **"Si hay …. hacer … si no"**. Estos bloques se llaman **condicionales**.



En el bloque "Si hay … hacer" damos una condición y, si se cumple, pasa lo que a continuación este introducido en el bloque.

En el condicional "Si hay … hacer … si no" le damos una condición al programa. Si se cumple la condición, se ejecuta lo indicado en "hacer" (ver imagen de la derecha) . Pero si no se cumple, se ejecuta lo contenido en "si no".

Los lenguajes de programación por bloques son muy populares en la actualidad. Las rutinas que aprendas con uno de ellos te servirán para todos los demás lenguajes por bloques. Nosotros trabajaremos este año, de manera intensiva, con el lenguaje de programación por bloques Scratch y con el lenguaje por bloques para programar la placa micro:bit

2. Diagramas de flujo para representar algoritmos

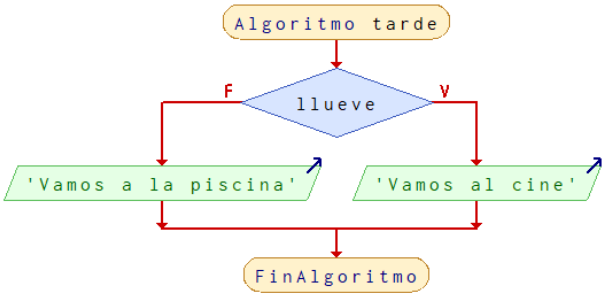
Escribir el pseudocódigo de un algoritmo con frases, tal y como hicimos con el ejemplo de la divisibilidad entre 4, es poco práctico. Hay que escribir mucho texto y no conseguimos una imagen visual y rápida de lo que deseamos que haga el programa. Todo esto se puede resolver con ayuda de unos esquemas llamados **diagramas de flujo**.

Símbolo	Nombre	Función
	Inicio / Final	Representa el inicio y el final de un proceso
	Linea de Flujo	Indica el orden de la ejecución de las operaciones. La flecha indica la siguiente instrucción.
	Entrada / Salida	Representa la lectura de datos en la entrada y la impresión de datos en la salida
	Proceso	Representa cualquier tipo de operación
	Decisión	Nos permite analizar una situación, con base en los valores verdadero y falso

Un diagrama de flujo es un esquema de símbolos y flechas que sintetiza la secuencia de las acciones que debe realizar el programa. Los símbolos que se utilizan vienen representados en la tabla de la derecha.

Un ejemplo: Imagina que organizas una tarde de verano con los amigos. Hay posibilidad de tormenta, y razonáis de la siguiente manera: "Si llueve, vamos al cine. En caso contrario, vamos a la piscina". No haréis las dos opciones. Haréis solo una, según llueva o no llueva.

La lluvia es la condición que determina si vais al cine o si vais a la piscina. El algoritmo de la decisión, escrito con diagramas de flujo, quedaría así:



La expresión "Algoritmo tarde" recoge el nombre del algoritmo. Es el inicio del pseudocódigo, por lo que se usa el símbolo de inicio (óvalo). Asimismo, la expresión "FinAlgoritmo" marca el final del algoritmo y se utiliza también el símbolo del óvalo.

Las flechas marcan el orden de ejecución. La palabra *llueve* aparece dentro de un rombo por ser una decisión. Es un condicional. Si es verdad que llueve seguimos por la flecha indicada con V (Verdadero). En caso contrario, iremos por la flecha señalada con F (falso).

Si la condición es verdadera, diremos (mensaje de salida) que "Vamos al cine". Si la condición es falsa, optaremos por "Vamos a la piscina". Ambas decisiones deben ser comunicadas. Son información de salida (que arroja el programa hacia el exterior). Por lo que optamos por el símbolo de Entrada/Salida del paralelogramo.

2.1. Programa PSeInt para crear diagramas de flujo

Los diagramas de flujo se pueden crear con el programa PSeInt. Permite escribir pseudocódigos de algoritmos con términos en español y tiene un botón para generar automáticamente el diagrama de flujo. Puedes descargar el programa en:

https://pseint.sourceforge.net/?page=descargas.php

Por ejemplo, el pseudocódigo del diagrama de flujo del apartado anterior se escribe así en PSeInt:

```
1  Algoritmo tarde
2      Si llueve Entonces Mostrar "Vamos al cine"
3      SiNo Mostrar "Vamos a la piscina"
4      FinSi
5  FinAlgoritmo
```

El algoritmo tiene un nombre: *tarde*. La expresión **Si** da paso a la condición a evaluar: *llueve*.

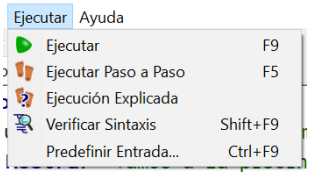
Tras la expresión **Entonces** se muestra en pantalla el mensaje "Vamos al cine" gracias al comando **Mostrar**. Este mensaje aparecerá si la condición *llueve* es verdadera.

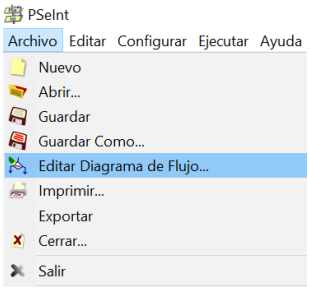
En caso contrario la expresión **SiNo** da paso al mensaje "Vamos a la piscina". Este mensaje se muestra si la condición *llueve* es falsa.

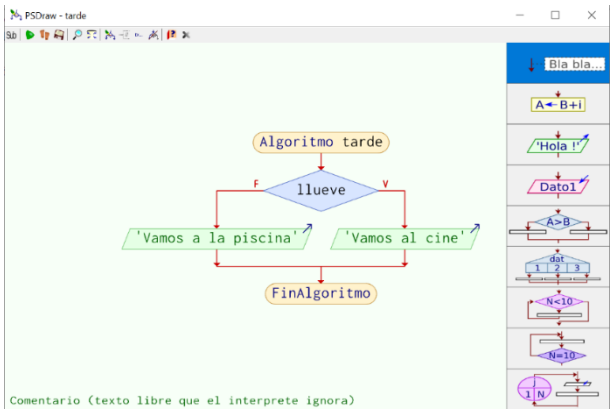
Cerramos el bloque condicional con la expresión **FinSi**. Y cerramos el pseudocódigo con **FinAlgoritmo**.

PSeInt también permite ejecutar el pseudocódigo como un programa. Con el **comando Ejecutar del menú superior** puedes hacer correr el programa y detectar posibles errores en el algoritmo.

En **Archivo / Editar Diagrama de Flujo** se abre la ventana para editar el diagrama asociado al pseudocódigo. A la derecha de esa ventana aparecen los distintos símbolos disponibles para introducir en nuestro diagrama de flujo y hacer al algoritmo cada vez más complejo.







2.2. Algoritmo para sumar dos números introducidos en el ordenador por el usuario

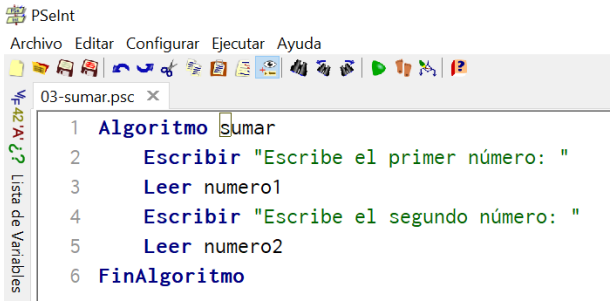
Diseñemos una calculadora rudimentaria donde el usuario pueda introducir dos números que desea sumar. Descompongamos el problema en pequeñas partes, para comprender qué necesita un ordenador para poder sumar dos números:

1. Pedir al usuario que escriba el primer número (mensaje de salida).
2. Guardar en la memoria del ordenador el valor del primer número (entrada de datos en el ordenador).
3. Pedir al usuario que escriba el segundo número (mensaje de salida).
4. Guardar en la memoria del ordenador el valor del segundo número (entrada de datos en el ordenador).
5. Realizar la suma de los dos números almacenados en la memoria del ordenador (proceso del ordenador sobre los datos almacenados).
6. Mostrar el resultado final de la suma (mensaje de salida).

Cuando necesitamos **introducir un dato en un ordenador**, hablamos de **dato de entrada**. En algún momento, el ordenador debe pedir al usuario que introduzca el dato. Por ejemplo, con un mensaje en la pantalla. Esta comunicación máquina-usuario genera una **interfaz**, que es un **programa que facilita la comunicación entre ambas partes**.

Ese dato de entrada debe ser guardado por el ordenador, para luego poder ser utilizado en la operación suma. Guardar un dato significa **almacenar su valor en un lugar de la memoria que sea fácil de recuperar, gracias a un nombre exclusivo**. Es lo que llamaremos **variable**.

En nuestro ejemplo, necesitaremos una variable para guardar el valor del primer número y otra variable para guardar el valor del segundo número. Necesitamos dos variables. En programación es muy recomendable nombrar a las variables con nombres que informen del contenido que almacenan. Fíjate en el siguiente pseudocódigo creado con PSeInt.

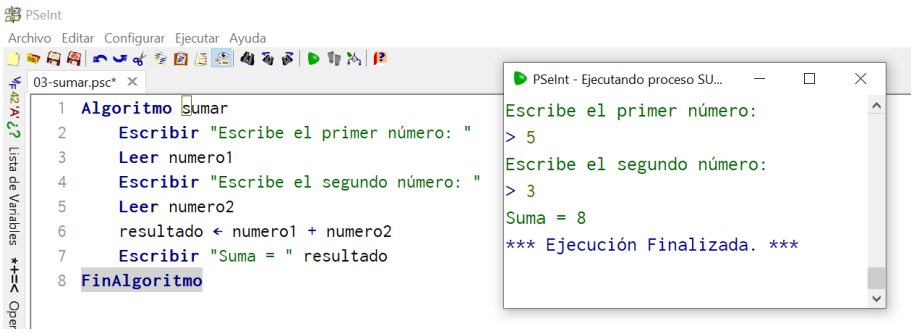


El comando **Leer** permite guardar el dato introducido por el usuario en una posición de la memoria del ordenador, dándole como nombre la expresión *numero1* para el primer número y *numero2* para el segundo número. Es costumbre en programación no usar tildes en los nombres de las variables. Además, el nombre de una variable nunca puede empezar por un número. Una vez que los datos de entrada están almacenados en la memoria, podremos recurrir a ellos fácilmente llamándolos por sus nombres.

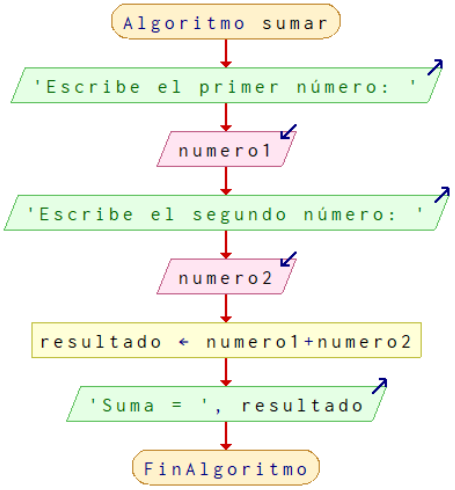
El resultado de la suma lo podemos almacenar en una nueva variable, que utilizaremos para mostrar el valor final en un mensaje de la pantalla. El valor de la suma es un **dato de salida** del programa. Un dato de salida se obtiene **tras realizarse un conjunto de operaciones con los datos de entrada**.

Los lenguajes de programación ofrecen funciones predefinidas que facilitan la tarea de programar. Por ejemplo, al usar "+" los lenguajes de programación entienden que queremos sumar dos números. Y la flecha hacia la izquierda "←" indica que el valor de la suma de ambos números se almacena en una variable llamada *resultado*.

En la siguiente imagen de PSeInt aparece el algoritmo completo del programa *sumar*, con la pantalla de ejecución del programa mostrando el 5 y el 3 como datos de entrada a modo de ejemplo. La pantalla de la derecha es un interfaz máquina-usuario.



El anterior pseudocódigo genera el diagrama de flujo en PSeInt que muestra la siguiente imagen. Las flechas que apuntan hacia fuera en los símbolos de los paralelogramos indican que es una información de salida, hacia el usuario (por ejemplo, el mensaje de texto para solicitar que escriba el primer número). Las flechas que apuntan hacia dentro en los paralelogramos indican que es una información de entrada, hacia la computadora (por ejemplo, leer el valor introducido por el usuario y almacenarlo en la variable *numero1*). Fíjate que el proceso de sumar ambos números y almacenarlo en la variable *resultado* se representa con el símbolo del rectángulo (símbolo de proceso).

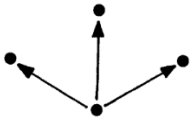


2.3. ¿Existen juegos donde es imposible ganar a un ordenador bien programado?

El siguiente juego aparece descrito en la página 12 del recurso web de *Shell Centre for Mathematical Education Publications*:

https://www.mathshell.com/publications/tss/ppn/ppn_teacher.pdf

Es un juego de estrategia para dos jugadores, donde cada movimiento es simbolizado por una flecha. Un jugador comienza en el punto de inicio de la imagen de la derecha y debe desplazarse, con una flecha, a un punto adyacente que se encuentre a una altura superior. La siguiente imagen muestra las tres opciones iniciales de desplazamiento que tiene el jugador inicial.

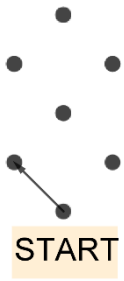
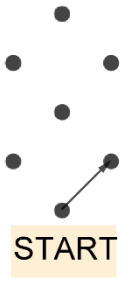
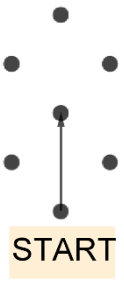


Tras el movimiento inicial, es el turno del segundo jugador, que debe seguir las mismas reglas de movimiento descritas anteriormente. Cada jugador dibuja flechas en turnos alternos, ganando el jugador que llegue con su flecha al punto final.

¿Posee este juego una estrategia ganadora? La expresión "estrategia ganadora" significa que si un jugador aplica esa estrategia, ganará seguro. Independientemente de los movimientos que realice el rival. Es decir, si encontramos una estrategia ganadora podríamos programar un ordenador al que sería imposible vencer.

Una partida de ejemplo: En la imagen inferior aparecen las flechas de una partida. El jugador 1 está simbolizado por las flechas de trazo continuo. El jugador 2 está simbolizado por las flechas de trazo discontinuo. Gana el jugador 2, por ser quien llega al punto final del juego.

Para descubrir la estrategia ganadora, vamos a estudiar la simetría del juego. Para ello, descomponemos el juego en partes más sencillas de analizar. En vez de jugar con 7 niveles de altura de puntos vamos a jugar con solo 5 niveles. En este caso, hay dos movimientos iniciales del jugador 1 que le dan la victoria segura si aplica la estrategia ganadora.

5 niveles de altura de puntos		
Jugador 1 gana seguro si sigue aplicando la estrategia ganadora	**Jugador 1 gana seguro si sigue aplicando la estrategia ganadora**	**Jugador 1 pierde la oportunidad de ganar seguro**
		

Las dos primeras imágenes de la tabla superior llevan al jugador 1 a una victoria segura. Haga lo que haga el jugador 2, el jugador 1 siempre puede terminar en el punto final.

La tercera imagen, sin embargo, lleva al jugador 1 a la derrota, porque en el siguiente turno el jugador 2 puede llegar directamente al punto final.

Es decir, para 5 niveles de altura de puntos, hemos descompuesto las posibilidad del jugador 1 en tres opciones distintas y hemos detectado cuáles de ellas llevan a una victoria segura.

¿Ocurre lo mismo para 7 niveles de puntos? El análisis de esta situación se plantea en los retos a resolver de la Sesión 2. Recomendación: descompón el movimiento inicial del jugador 1 en sus tres únicas opciones posibles, y analiza para cada caso si existe o no estrategia ganadora.

3. Tipos de lenguajes de programación

Una clasificación sencilla de los lenguajes de programación es la siguiente:

- **Lenguaje máquina.** Es el lenguaje que entienden directamente los ordenadores. Usa el alfabeto binario (0 y 1) para indicar la ausencia o la presencia de carga eléctrica en cada uno de los millones y millones de microcircuitos eléctricos que forman una computadora.
- **Lenguajes de alto nivel.** Diseñados para que los humanos escriban instrucciones de programación lo más parecidas al lenguaje humano. Esto provoca que se necesite menos tiempo para aprender a programar, en comparación con el lenguaje máquina. Estos lenguajes siempre necesitan de un compilador, que "traducen" el lenguaje de alto nivel a lenguaje máquina.
- **Lenguajes visuales.** Especialmente usados en entornos educativos para aprender a programar, utilizando formas y colores en vez de texto para facilitar el montaje correcto de los programas. Scratch y el lenguaje de bloques de micro:bit son dos ejemplos.

3.1. El transistor para almacenar información en los ordenadores

Los ordenadores cuentan con millones y millones de pequeñísimos circuitos electrónicos llamados **transistores**. Esos transistores pueden almacenar energía eléctrica. Cada transistor puede estar en dos estados de funcionamiento: almacenando energía o no almacenando energía. Si almacena energía, diremos que el transistor está en 1. Si no almacena energía, diremos que el transistor está en 0.

Por lo tanto, un transistor puede encontrarse en dos estados: 0 o 1. Esto significa que un ordenador que estuviese formado por un único transistor podría almacenar únicamente dos valores de información. Esto se llama **bit**: La menor cantidad de información posible que puede almacenar un ordenador.

La agrupación de ocho transistores forman ocho bits de información. Esto se conoce como **Byte**. Este término seguro que te suena al leer las prestaciones de un ordenador o de un teléfono móvil, donde la memoria o la capacidad de almacenamiento vienen indicados como número de Bytes:

$$\mathbf{8}\ \boldsymbol{bits} = \mathbf{1}\ \boldsymbol{Byte}$$

Según el número de bits que utilicemos para almacenar números en un ordenador, tendremos la siguiente secuencia:

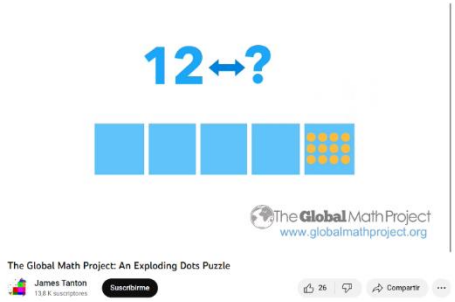
- 1 bit almacena dos números: 0 y 1.
- 2 bits almacenan 4 números: 0, 1, 2 y 3.
- 3 bits almacenan 8 números: 0, 1, 2, 3, 4, 5, 6 y 7.
- …
- n bits almacenan 2^n números: 0, 1, 2, 3, 4, … , $2^n - 1$.

¿Puede ocurrir que haya un número muy grande que no se pueda almacenar en un ordenador? ¿Alguna vez te ha pasado eso operando con una calculadora?

La información se guarda en el interior del ordenador como una combinación de diferentes 0 y 1 de los transistores. Esta combinación se llama **código binario**. Un número es un tipo de dato, al igual que lo es una letra de un texto o un píxel de una imagen. En los próximos apartados vamos a estudiar como almacenar números en código binario, usando solo las cifras 0 y 1. Fíjate que nuestro **código decimal** emplea más cifras (0, 1, 2, 3, 4, 5, 6, 7, 8 y 9) para formar cualquier número.

3.2. ¿Cómo codificar un número decimal en binario?

Mira el siguiente vídeo del canal YouTube de James Tanton y explica con tus palabras qué ocurre:

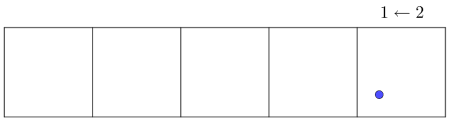


https://www.youtube.com/watch?v=q-ucOwlZRqQ

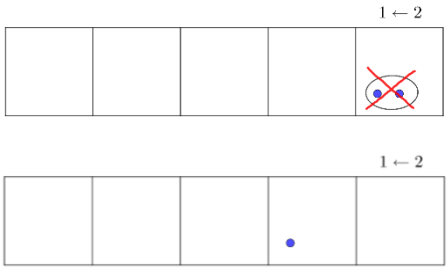
Cada punto representa una unidad de cantidad. Acumulamos puntos en la casilla de la derecha y aplicamos la siguiente regla: **Cuando tengamos dos puntos en una misma casilla, explotan y desaparecen, y son sustituidos por un punto en la casilla situada justo a la izquierda**. Esta regla crea la **máquina dos-uno**, que se representa al contrario de cómo se lee:

$$Máquina\ 1 \leftarrow 2$$

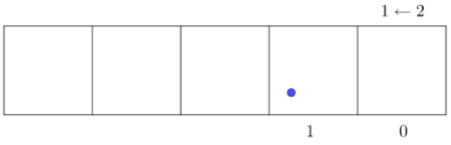
El nombre de la máquina lo situamos encima de la casilla de la derecha. Si colocamos un único punto, la máquina no hace gran cosa:



Si colocamos dos puntos en la casilla de la derecha, explotan, desaparecen, y dan paso a un único punto justo en la casilla situada a la izquierda:



Es decir, nuestra máquina expresa la cantidad inicial 2 de una forma muy especial: un punto en la penúltima casilla y ninguno en la última casilla. Vamos a completar la última imagen situando debajo de las dos últimas casillas un 1 o un 0 según contengan o no contengan puntos.



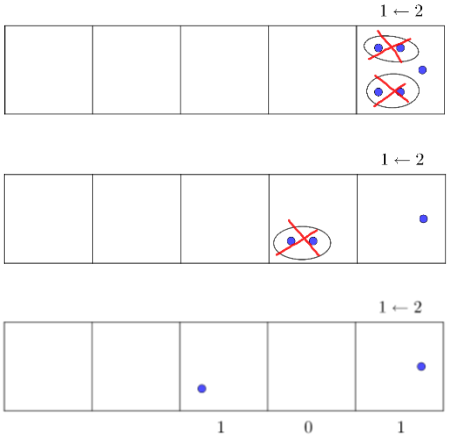
¿Para qué nos sirve situar esos números debajo de las casillas? Para representar la cantidad 2 de otra manera. ¿De qué manera? **De la misma forma en que lo codifican las máquinas: en sistema binario**. El sistema de numeración que estamos acostumbrados a utilizar emplea cifras del 0 a 9. Como son diez cifras distintas, lo llamamos sistema decimal. Los ordenadores, por el funcionamiento de los transistores que comentamos anteriormente, solo pueden almacenar 0 o 1 en su unidad básica de memoria. Como son únicamente dos cifras, lo llamamos sistema binario.

La máquina $1 \leftarrow 2$ es un ejemplo práctico de cómo convertir un número decimal en número binario. Si escribimos:

$$(2)_{10} = (10)_2$$

Quiere decir que el número 2 en decimal se expresa con el código 10 en binario. El número representado como subíndice, fuera del paréntesis, indica el sistema en el que estamos trabajando (10 significa sistema decimal, 2 significa sistema binario). **¡Ojo al leer el código $(10)_2$ en binario! No se lee "diez". Se lee "uno-cero". Leemos cada cifra de manera independiente.**

¿Qué ocurre en nuestra máquina si situamos 5 puntos en la casilla de la derecha?



Es decir, la cantidad 5 en decimal puede escribirse en nuestra máquina con el código 101.

$$(5)_{10} = (101)_2$$

Por lo tanto, un ordenador necesita al menos 3 bits para poder codificar el número 5.

3.3. Otro ejemplo de codificación: base hexadecimal

¿Sólo podemos escribir números con base decimal o con base binaria? No. Podemos usar la base que queramos. Por ejemplo, la base hexadecimal emplea dieciséis símbolos distintos: las cifras del 0 al 9 y las letras de la A hasta la F. Fíjate en la siguiente tabla.

Decimal (cifras de 0 a 9)	Hexadecimal (cifras de 0 a 9 y letras de A a F)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

La base hexadecimal es muy utilizada en informática. Como ejemplo, una secuencia de la película "Marte". El actor principal (Matt Damon) queda atrapado en Marte por accidente. Para comunicarse con la Tierra emplea una antigua cámara que gira sobre su eje. Cada giro de la cámara da lugar a un símbolo del código hexadecimal, con el que recibe (con paciencia) mensajes desde la Tierra.

En clase vamos a visionar la secuencia de la película donde aparece el código hexadecimal (desde minuto 44:30 hasta minuto 52:50; película disponible en Disney+).

3.4. Pseudocódigo y diagrama de flujo para pasar de binario a decimal

Imagina que tienes un número codificado en binario, con 4 bits de profundidad. Y necesitas pasar el número a formato decimal, para lo cual diseñas un programa de ordenador. El pseudocódigo del algoritmo podría ser el siguiente:

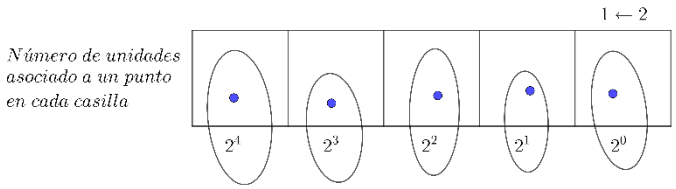
1. Solicitar al usuario que escriba el primer dígito del número binario (comenzando por la izquierda).
2. Almacenar el primer dígito en una variable.
3. Solicitar al usuario que escriba el segundo dígito del número binario (comenzando por la izquierda).
4. Almacenar el segundo dígito en una variable.
5. Solicitar al usuario que escriba el tercer dígito del número binario (comenzando por la izquierda).
6. Almacenar el tercer dígito en una variable.
7. Solicitar al usuario que escriba el cuarto dígito del número binario (comenzando por la izquierda).
8. Almacenar el cuarto dígito en una variable.
9. **Operar con las variables almacenadas para pasar el número a base decimal. ¿Cómo conseguirlo?**
10. Mostrar el resultado en pantalla.

Para ejecutar el paso 9 del pseudocódigo necesitamos una fórmula matemática que nos permita pasar de binario a decimal. ¿Cómo razonar esa fórmula? Tanto el sistema binario como el sistema decimal son sistemas posicionales. Pensemos en base decimal: No es lo mismo tener "2" en la posición de las unidades, que tener "2" en las decenas, o tener "2" en las centenas.

- El símbolo 2 significa dos unidades.
- El símbolo 22 significa veintidós unidades: $20 + 2 \rightarrow 2 \times 10 + 2 \times 1 \rightarrow 2 \times 10^1 + 2 \times 10^0$
- El símbolo 222 significa doscientas veintidós unidades: $200 + 20 + 2 \rightarrow 2 \times 100 + 2 \times 10 + 2 \times 1 \rightarrow 2 \times 10^2 + 2 \times 10^1 + 2 \times 10^0$

Fíjate que el número de las centenas se multiplica por 10^2. El número de las decenas se multiplica por 10^1. Y el número de las unidades se multiplica por 10^0. Si hubiésemos tenido unidades de millar, multiplicaríamos por 10^3. Y así sucesivamente.

Si aplicamos este razonamiento a sistema binario, la base a utilizar será 2. Y el número de puntos situados en cada una de las casillas de una $Máquina\ 1 \leftarrow 2$ se multiplicará respectivamente por 2^0, 2^1, 2^2, 2^3, etc. Fíjate en la siguiente imagen:



Es decir, el número binario $(11111)_2$ lo podemos escribir descompuesto en potencias de base 2.

$$(11111)_2 = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

¿Para qué sirve esta forma de descomponer el número en binario? Para obtener la cantidad en base decimal:

$$(11111)_2 = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 8 + 4 + 2 + 1 = 31$$

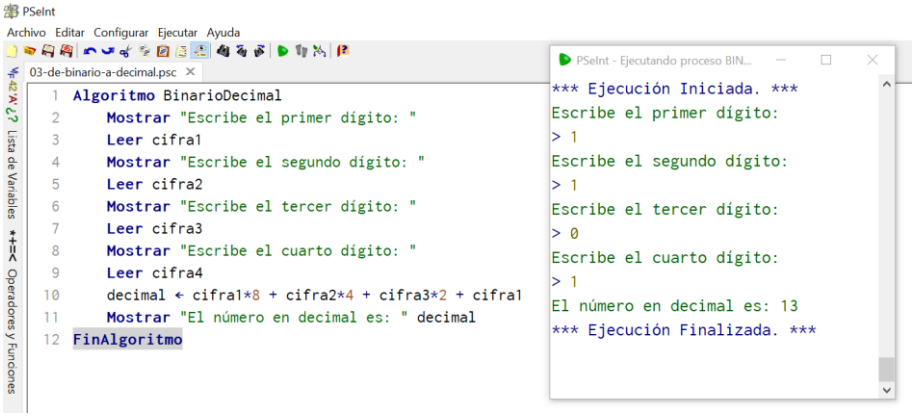
¡Hemos demostrado una fórmula matemática para pasar de binario a decimal!

Funciona con cualquier número binario, por muy largo que sea. Fíjate en el siguiente número de 8 bits de tamaño:

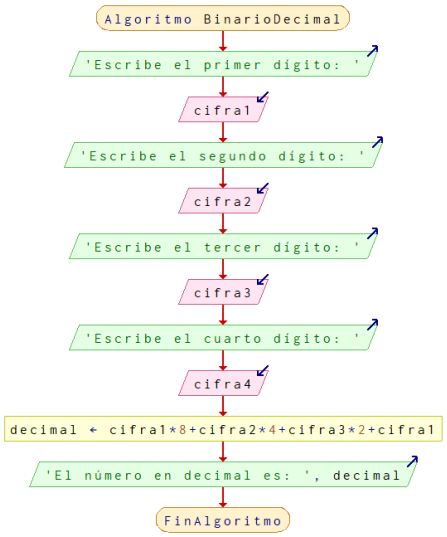
$$(10101010)_2 = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$(10101010)_2 = 128 + 0 + 32 + 0 + 8 + 0 + 2 + 0 = 170$$

En el pseudocódigo con el que hemos empezado este apartado, la fórmula a utilizar será multiplicar cada una de las variables por las potencias 2^3, 2^2, 2^1 y 2^0 respectivamente. O lo que es lo mismo, multiplicar por 8, por 4, por 2 y por 1. La siguiente imagen muestra el algoritmo en PSeInt, aplicado al número binario $(1101)_2$ que se traduce en el número 13 decimal.



Este algoritmo, en PSeInt, genera automáticamente el siguiente diagrama de flujo:



4. Procesador y microcontroladora de un ordenador. Hardware y software

El cerebro de un ordenador es un microchip llamado **procesador**, que contiene un programa que está siempre funcionando y que controla completamente al ordenador. El procesador es el encargado de leer el código de programación creado por los seres humanos.

Este cerebro está situado dentro de una **microcontroladora** (controladora pequeña). La microcontroladora permite conectar el procesador con los **dispositivos de entrada** y con los **dispositivos de salida**. Ejemplos de dispositivos de entrada son el teclado o el micrófono. Ejemplos de dispositivos de salida son la pantalla o los altavoces.

Si la microcontroladora está insertada en un objeto móvil, que actúa en función de las decisiones del procesador, tendremos un robot. El cerebro del robot puede gestionar la información que recibe de los sensores de entrada del robot (sensor de luz, medidor de distancias por ultrasonido, etc.) y poner en funcionamiento los actuadores de salida (motores de movimiento, luces LED, zumbadores, etc.). Todas las acciones que puede realizar un robot están contenidas en el programa instalado en el procesador. Este curso trabajaremos con la **microcontroladora micro:bit**, cuyo procesador nos permitirá controlar al **robot maqueen**.

Todo lo que se pueda tocar con las manos en un ordenador o robot se denomina **hardware**. Todo aquello que no podamos tocar (el código de los programas o los datos almacenados en la memoria) se denomina **software**.

4.1. Leyes de la robótica

Isaac Asimov (1920-1992), escritor y científico nacido en Rusia y nacionalizado estadounidense, es una figura imprescindible de la literatura de ciencia-ficción del siglo XX. Obras como "Yo Robot" o "Fundación e Imperio" son libros de referencia a nivel mundial. En sus libros imagina una sociedad donde los robots toman decisiones propias e interaccionan con los seres humanos. La palabra robot viene del checo "robota", que significa "realizar tareas forzadas".

Hoy en día podemos definir **robot como una máquina automática programable que realiza tareas repetitivas**. En esta definición aparecen palabras que debemos comprender bien:

- "Máquina" es todo aquello que realiza un trabajo y es controlado por un ser humano (por ejemplo, una máquina de coser o una máquina para cortar el césped del jardín).
- "Automática" es una máquina que realiza un trabajo repetitivo sin necesidad de supervisión humana (por ejemplo, un ascensor que continuamente está subiendo y bajando, o una lavadora).
- "Programable" implica que la máquina puede tomar decisiones sobre el trabajo a realizar gracias al código de programación y a los sensores de entrada (por ejemplo, un coche que viaja sin conductor, o un robot que limpia el suelo de la casa).

Según la complejidad de las decisiones que toma el robot y de su capacidad de aprender con la práctica, podremos hablar de Inteligencia Artificial.

Asimov establece tres leyes básicas que todo robot debe cumplir, por el bien de la sociedad y de las personas que la forman:

1. **Un robot no hará daño a un ser humano o, por falta de acción, no permitirá que un ser humano sufra daño.**
2. **Un robot debe cumplir las órdenes dadas por los seres humanos, siempre que no entren en conflicto con la primera ley.**
3. **Un robot debe proteger su propia existencia, siempre que no entre en conflicto con la primera y segunda ley.**

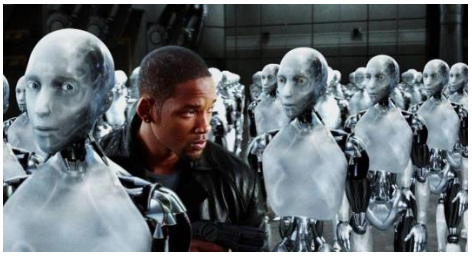
Estas leyes aparecen en los libros de ciencia-ficción. No se utilizan en la programación actual de robots. Pero la fama de las tres leyes de Asimov es tan grande, que hoy en día están presentes en el debate sobre cómo han de funcionar los robots del futuro (que se parecerán a los que Asimov había imaginado en sus libros).

Según Asimov, estas tres leyes buscan evitar una rebelión de los robots. Si un robot intentase saltarse alguna de estas leyes, su código de programación debería provocar la "desactivación" del robot. Para superar situaciones en que el cumplimiento de las tres leyes entrasen en conflicto entre sí, al final de su vida Asimov establece una ley cero (previa a las otras tres):

0. **Un robot no hará daño a la Humanidad o, por falta de acción, no permitirá que la Humanidad sufra daño.**

¿Funcionarán en la práctica las leyes de la robótica de Asimov? Difícil saberlo, porque las técnicas de programación en informática y de creación de robots actuales (y en los próximos años) son mucho más complejas de las que Asimov conocía en su momento.

¿Podrán las Inteligencias Artificiales implantadas en ordenadores y robots aprender hasta tal punto en convertirse más inteligente que los seres humanos, y tener capacidad para alterar sus propios códigos de programación?



Veamos un trozo de la película "Yo Robot" (disponible en Disney+; desde el minuto 28:00 hasta el minuto 31:40). En la conversación entre el actor humano y el robot, aparecen varias aplicaciones de estas leyes de la robótica, que debatiremos en clase. Piensa que, en un futuro que todos veremos, la influencia de los robots en la vida diaria de las personas será un tema de debate público.

4.2. Programación por bloques de la microcontroladora micro:bit

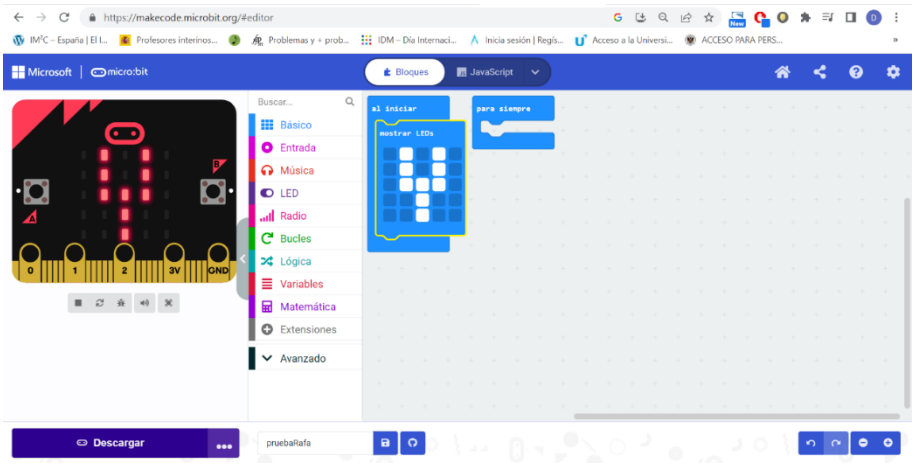
La web oficial de micro:bit es https://microbit.org. Se accede a la zona de programación online en https://makecode.microbit.org. Es recomendable que te registres en la zona de makecode con tu email de @maristasmediterranea.com. Como es una cuenta vinculada a Microsoft, puedes elegir loguearte en makecode con la opción "Continue with Microsoft". Así tendrás siempre disponible todos los programas que vayas creando, independientemente del ordenador que utilices.

La placa microcontroladora micro:bit se conecta al ordenador mediante un clave USB (ver imagen de la derecha). La placa se verá en nuestro ordenador con la apariencia de un disco externo. El cable USB alimenta eléctricamente a la placa y permite cargar el programa con el código.

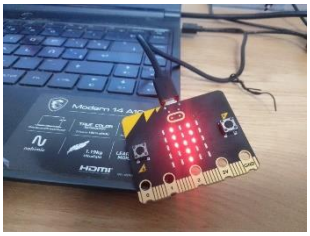
La imagen inferior muestra el ejemplo básico de encender algunos sensores LEDs gracias al bloque "al iniciar" y al bloque "mostrar LEDs". Podemos seleccionar con el ratón los LEDs que se iluminarán.

Pulsando en "Descargar", el programa se descarga al ordenador. Y arrastrando el archivo de extensión .hex sobre el directorio de nuestro ordenador donde aparece MICROBIT, el programa se ejecuta de inmediato. Con la última versión de Chrome y de Explorer, se puede elegir que al descargar el programa se ejecute directamente sin necesidad de tener que arrastrar el archivo sobre el directorio de micro:bit. Para que funcione esta ejecución directa, debemos emparejar la placa con el ordenador (el botón simbolizado por tres puntitos ··· junto a "Descargar" permite conectar la placa para emparejarla).

El interfaz online de MakeCode posee un simulador, que permite ver cómo se ejecuta el programa en la placa sin necesidad de tener físicamente la placa conectada al ordenador. Esto es muy útil para practicar en casa, ya que la placa micro:bit solo la tendrás en clase (salvo que quieras comprarte una, lo cual no es mala idea si te gusta la programación y la robótica; su precio ronda los 20 €).



Bajo el simulador online aparece un botón de "play/stop" para iniciar/detener la simulación en MakeCode. Cada código que creemos se guarda como un nuevo proyecto, que podemos modificar en posteriores accesos siempre y cuando nos hayamos registrados previamente.



La imagen de la derecha muestra la placa ejecutando el programa de encender los LEDs seleccionados.

La web oficial www.microbit.org posee infinidad de recursos, ilustrados con vídeos y ejemplos de código, para aprender poco a poco. Muchos de los vídeos se explican con un inglés muy sencillo, con subtítulos en YouTube. Por lo que es una buena actividad para practicar inglés y robótica a la vez.

Un corazón dibujado en el display de LEDs es otra sencilla actividad:

https://microbit.org/projects/make-it-code-it/heart

El display de la actividad anterior muestra un corazón fijo. Nuevamente utilizamos el bloque "al iniciar". Fíjate que el bloque "mostrar icono" engancha, como un puzle, con el bloque "al iniciar". Esta es una de las grandes ventajas educativas de la programación por bloques: la forma de cada bloque da pistas de cómo deben conectarse entre sí.

Vamos a dar vida al corazón, simulando latidos. La idea básica es alternar, en el tiempo, un corazón grande con un corazón pequeño. Y crear un bucle que repita de manera indefinida este proceso. Todo viene explicado en:

https://microbit.org/projects/make-it-code-it/beating-heart/

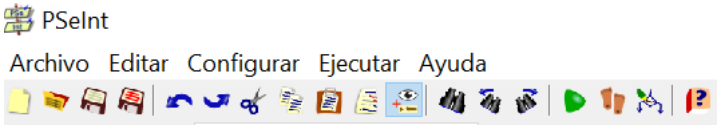
Como deseamos que el latido se mantenga siempre, usaremos el bloque "para siempre" (bucle infinito) para implementar el código (ver imagen de la derecha).

Introducimos un bloque "mostrar icono" con el corazón grande dibujado, seguido de un bloque "pausa" con un tiempo de espera de 200 ms (milisegundos). Acto seguido añadimos otro bloque "mostrar icono" con el corazón pequeño, y otro bloque de "pausa" con 200 ms de espera. Así tendremos un bucle infinito.

4.3. Depurar errores en nuestros códigos de programación. Escribir comentarios en el código

Conforme nuestros programas aumentan en complejidad, es inevitable que los seres humanos cometamos errores a la hora de escribir el código de programación. Da igual que sea un código con texto o con bloques; da igual que seamos programadores expertos o noveles. Los errores a la hora de programar siempre aparecen, y debemos educar un espíritu paciente y ordenado para detectarlos.

Nunca diremos la siguiente frase: "He escrito mi código bien y el programa no funciona". Eso no es posible. El ordenador nunca se equivoca. Siempre ejecuta lo que nosotros le digamos que haga. Si el programa no hace lo que nosotros queremos que haga, es sencillamente porque nos hemos equivocado al crear el código. No desesperes si tu algoritmo no funciona a la primera.



La imagen superior muestra el menú superior y los botones de acceso rápido de PSeInt. Son muy útiles para analizar, paso a paso, un programa que está dando fallos. El botón simbolizado con un triángulo verde ejecuta al programa desde el principio hasta el fin. El botón simbolizado por dos pies permite avanzar y detener el programa poco a poco. El botón representado por un esquema muestra el diagrama de flujo del pseudocódigo. Y el botón que contiene un libro con un signo de interrogación da acceso a la ayuda, con decenas de ejemplos sencillos con los que practicar.

Una buena táctica para minimizar errores, y para recordar de un día para otro qué hace exactamente cada parte de un código, es introducir comentarios. Un comentario es un texto escrito por el programador que el ordenador ni lee ni ejecuta. Solo queda visible a los ojos del programador, como una ayuda para comprender el código. En PSeInt (y en muchos lenguajes de programación de alto nivel) se utilizan las dos barras invertidas // para escribir el texto del comentario. Fíjate en el siguiente algoritmo con comentarios.

```
1  Algoritmo menu
2      Escribir 'Escribe un primer número A: '
3      Leer A //Esta variable almacena el valor del primer número
4      Escribir 'Escribe un segundo número B: '
5      Leer B //La variable B contiene el valor del segundo número
```

Visualmente, el comentario suele aparecer como un texto en gris y en cursiva. A diferencia del código de programación, que suele aparecer en colores.

4.4. Bucle de control "mientras" y operador "modulo" aplicado a un programa con PSeInt

Demos un paso hacia delante en la complejidad de nuestros programas. Y añadamos una nueva estructura de control: el **bucle "mientras"**. En programación es muy frecuente tener que repetir una misma parte del código varias veces. Los bucles de control facilitan esta tarea.

Imagina que deseamos obtener los divisores primos de un número. Tenemos que ir probando por 2, luego por 3, etc. Sin saber exactamente cuántas veces tendremos que probar con cada uno de los números primos. Por ejemplo, el número 8 contiene tres veces al divisor 2. Mientras que el número 10 solo contiene una vez al divisor 2. ¿Cómo podemos adaptar nuestro programa a las posibles repeticiones que necesitemos?

Con ayuda de un bucle de control repetitivo. En PSeInt podemos usar el bucle "mientras" (en inglés, while). Si se cumple una condición, el código del bucle se repite. En caso contrario, el programa sale del bucle. La sintaxis del bucle "mientras" en PSeInt es la siguiente:

Mientras <condición> Hacer

<instrucciones>

FinMientras

El siguiente algoritmo muestra, con ayuda del bucle "mientras", todos los divisores primos (del 2 al 7, para no alargar demasiado el código) de un número introducido por el usuario a través del teclado. La imagen inferior también muestra un ejemplo de aplicación con el número 84.

03-primos.psc

```
1  Algoritmo primos
2      Mostrar "Introduce el número a descomponer en divisores primos: "
3      Leer descomponer
4      Mientras (descomponer Mod 2)==0 Hacer
5          Mostrar "El número es divisible por 2"
6          descomponer ← descomponer/2
7      FinMientras
8      Mientras (descomponer Mod 3)==0 Hacer
9          Mostrar "El número es divisible por 3"
10         descomponer ← descomponer/3
11     FinMientras
12     Mientras (descomponer Mod 5)==0 Hacer
13         Mostrar "El número es divisible por 5"
14         descomponer ← descomponer/5
15     FinMientras
16     Mientras (descomponer Mod 7)==0 Hacer
17         Mostrar "El número es divisible por 7"
18         descomponer ← descomponer/7
19     FinMientras
20 FinAlgoritmo
```

```
PSeInt - Ejecutando proceso PR...
*** Ejecución Iniciada. ***
Introduce el número a descomponer
en divisores primos:
> 84
El número es divisible por 2
El número es divisible por 2
El número es divisible por 3
El número es divisible por 7
*** Ejecución Finalizada. ***
```

En el código anterior, además del bucle "mientras", debemos explicar un operador matemático muy útil a la hora de programar: el **operador "modulo"** (recuerda que es costumbre no usar tildes en los términos de programación). Ese operador devuelve el resto entero de la división, es decir, el resto sin sacar decimales. Por ejemplo:

- 7 Mod 2 devuelve el número 1 (porque al dividir 7 entre 2, queda de resto 1).
- 9 Mod 3 devuelve el número 0 (porque al dividir 9 entre 3, queda de resto 0).

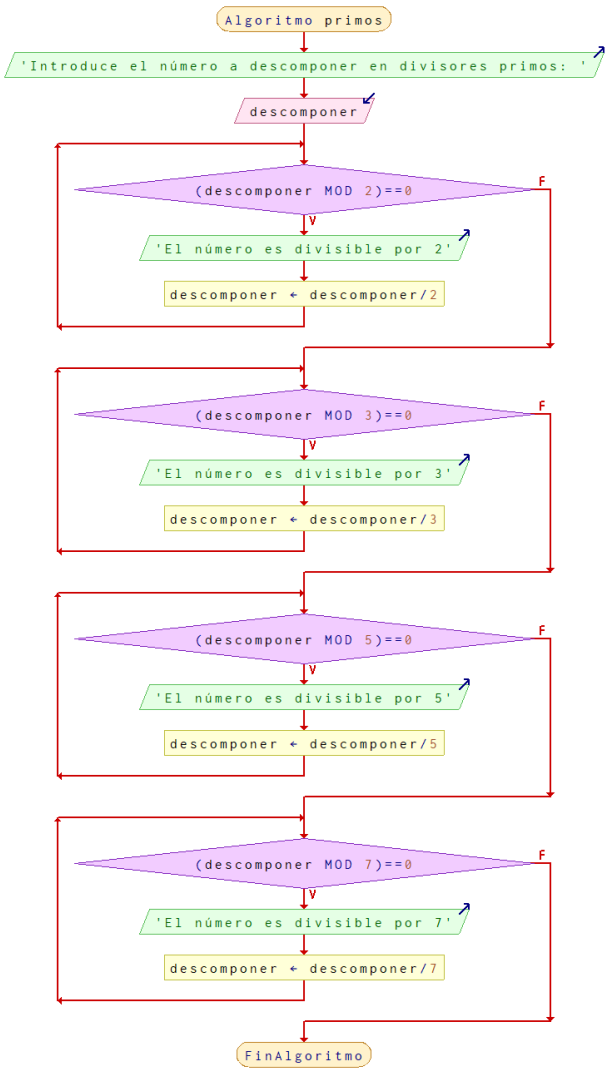
Usamos este operador "modulo" para decidir si se cumple o no la condición de cada bucle "mientras". Por ejemplo:

- La condición (descomponer Mod 2) == 0 es cierta solo si el número almacenado en la variable *descomponer* genera un resto 0 al dividir entre 2.
- La condición (descomponer Mod 3) == 0 es cierta solo si el número almacenado en la variable *descomponer* genera un resto 0 al dividir entre 3.
- La condición (descomponer Mod 5) == 0 es cierta solo si el número almacenado en la variable *descomponer* genera un resto 0 al dividir entre 5.
- La condición (descomponer Mod 7) == 0 es cierta solo si el número almacenado en la variable *descomponer* genera un resto 0 al dividir entre 7.

Fíjate que **para comparar** el resultado del operador "módulo" con el número 0 **se usa el doble signo igual ==** en vez de un solo signo igual. Esto es muy típico en los lenguajes de programación de alto nivel.

El gráfico del bucle "mientras" en un diagrama de flujo es un rombo, al igual que los condicionales, porque ambos casos son estructuras de control. Con la diferencia de que si la condición es verdadera (V) la flecha de flujo vuelve otra vez al inicio del bucle "mientras" para comprobar nuevamente la condición.

La imagen inferior muestra el diagrama de flujo del programa. Cada vez que es Verdadera la condición de un bucle "mientras", el valor del número almacenado en la variable *descomponer* debe ser modificado por el resultado de realizar la división exacta. Por eso aparece, tomando como ejemplo el primer bucle, el gráfico del rectángulo del proceso $descomponer \leftarrow descomponer / 2$. **La barra invertida "/" es el operador de división** al que estamos acostumbrados, que devuelve el resultado de la división (es decir, devuelve el cociente de la división).

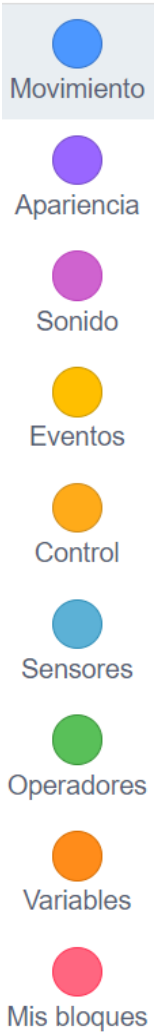


5. Elementos principales en la programación por bloques aplicado a Scratch

Las características principales de los lenguajes de programación las vamos a estudiar a partir del caso concreto de Scratch. La web oficial de este entorno de programación online es https://scratch.mit.edu. Y pulsando en la opción "Crear" llegamos al editor.

Scratch es un entorno de programación por bloques desarrollado por el MIT (Instituto Tecnológico de Massachusetts). Sirve para crear juegos online y animaciones. Si te registras (con tu cuenta de correo electrónico) tendrás siempre disponibles tus creaciones, independientemente del ordenador que utilices. Como siempre, nunca compartas con nadie tus contraseñas de internet y acostúmbrate a utilizar contraseñas distintas para las páginas web y aplicaciones que utilices.

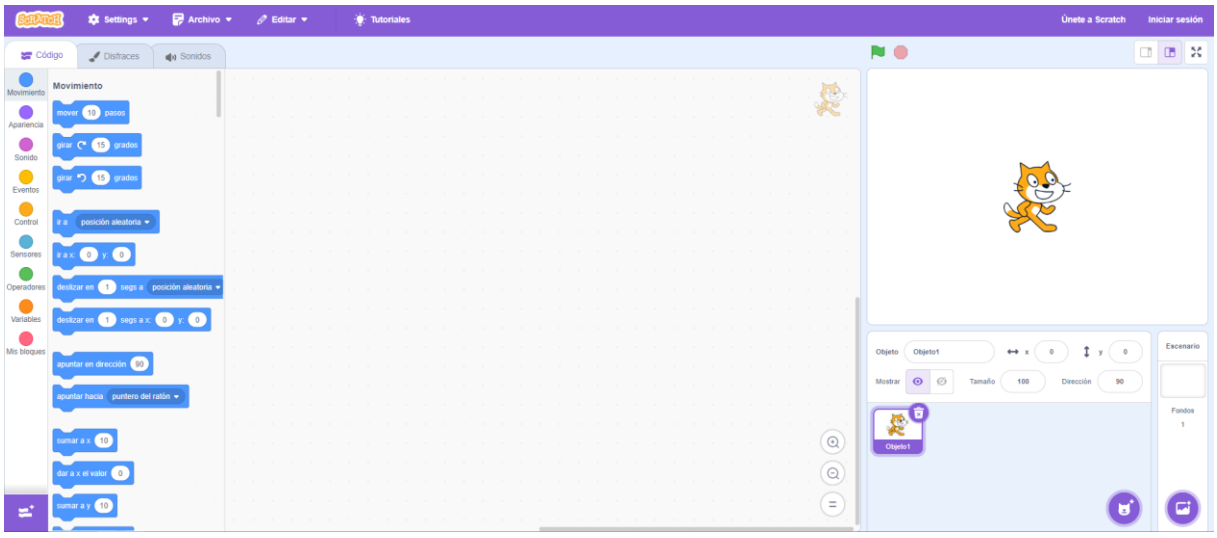
Cualquier elemento que aparezca en el **escenario** del programa creado con Scratch se llama **objeto**. Y estos objetos se pueden programar mediante bloques. La imagen de la derecha muestra, con círculos de colores, las categorías que clasifican todos los bloques disponibles en Scratch:



- "Movimiento" engloba los bloques que permiten desplazar los objetos por el escenario: mover un número de pasos, girar una cantidad de grados, ir a una posición (x, y) del plano que contiene al escenario, sumar una cantidad fija a una de las dos componentes, etc.
- "Apariencia" permite crear diálogos de texto asociados a los objetos, cambiar su apariencia (disfraz), modificar su tamaño, mostrar u ocultar un objeto, etc.
- "Sonido" aglutina los bloques relacionados con los archivos de audio vinculados a cada objeto, su volumen, su inicio y su fin.
- "Eventos" son acciones que se usan para comenzar un programa o desencadenar una acción (por ejemplo, hacer clic en un objeto o presionar una tecla concreta).
- "Control" contiene las estructuras de bucles y condicionales.
- "Sensores" hace sensible a los objetos cuando se tocan entre sí, cuando tocan un color determinado, cuando quedan a la espera de una respuesta ante una pregunta que debe responder el usuario, o a la posición del ratón en el escenario del programa.
- "Operadores" incluye operaciones matemáticas, comparación entre números (menor, mayor, igual) y las opciones lógicas y, o, negación. Además contiene bloques para trabajar con las distintas letras de una palabra o cadena de texto.
- "Variables" permite crear las variables del programa: espacios de memoria del ordenador donde podemos almacenar información, modificarla y que podemos llamarlas cómodamente por un nombre único. Si las variables están disponibles para todos los objetos, se denominan variables globales. Y si solo afectan a un objeto determinado, diremos que son variables locales.
- "Mis bloques" ofrece al usuario la posibilidad de crear bloques propios, distintos a los ya definidos por defectos en Scratch. Es una opción especialmente útil cuando nuestros programas crecen en complejidad y no encontramos ninguna secuencia de bloques que ejecute las acciones que deseamos.

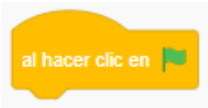
Cualquier entorno de programación por bloques (como el que utilizamos en makecode de micro:bit en la sesión anterior) sigue una estructura similar a las categorías que acabamos de describir.

La imagen inferior muestra una visión global de las distintas secciones del entorno de programación online de Scratch. A la izquierda apreciarás los iconos circulares de las categorías de los bloques de programación. En la parte central aparece un gran espacio en blanco donde poder enlazar los bloques entre sí para generar la lógica del código de programación. A la derecha encontramos otro espacio en blanco más pequeño, que muestra el escenario del programa y los diferentes objetos que hemos creado. La parte superior está reservada para el menú (elegir idioma del entorno, cargar archivo externo, acceder a tutoriales, etc.). Y justo en la esquina superior derecha podemos iniciar sesión, con nuestro usuario y contraseña de Scratch.



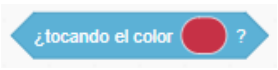
5.1. Algunos bloques de Scratch especialmente importantes.

Destacamos, por su importancia ahora que estamos aprendiendo a programar, los siguientes bloques:



Evento “al hacer clic en bandera verde”. Usaremos este bloque para que, al pulsar sobre él, se inicie la ejecución de nuestro programa. Encima del escenario del programa encontramos el icono de la bandera verde, que es el que lanza la secuencia de todos los bloques situados por debajo de este Evento. El botón octogonal rojo de Stop detiene la ejecución del programa.

El bloque “al presionar tecla” también pertenece a la categoría de Evento. Cuando pulsamos la tecla del teclado indicada en este bloque, se lanza toda la secuencia de los bloques situados por debajo de este Evento. El desplegable situado dentro del bloque permite elegir el tipo de tecla que deseamos controlar.

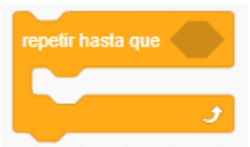


Situado dentro de Sensores, este bloque es un condición que devuelve SÍ cuando el objeto toca alguna parte (del escenario u otro objeto) del color que hayamos prefijado. Su forma hexagonal nos da pistas de que podremos colocarlo, por ejemplo, dentro de un bloque de control.

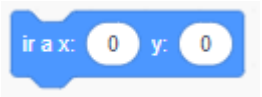
El bloque condicional se encuentra en la categoría Control. Acepta una condición de forma hexagonal. Si esta condición se cumple (si es Verdadera) se ejecutan los bloques situados a continuación de “entonces”. Fíjate que este bloque solo da paso a su interior si se cumple la condición; a diferencia del siguiente bloque que vamos a estudiar, que distingue entre lo que se ejecuta si se cumple la condición y lo que se ejecuta si no se cumple la condición.

Este bloque “si …. entonces … si no” también acepta una condición hexagonal. Si la condición se cumple (es Verdadera) se ejecuta todo lo que aparece a continuación de “entonces”. Si la condición no se cumple (es Falsa) se ejecuta lo que sigue a “si no”.

Este bloque es especialmente útil para controlar el flujo del programa cuando llegamos a un punto donde solo hay dos opciones posibles, opuestas entre sí. O bien ocurre una cosa o bien ocurre su contrario. No pueden ocurrir ambas cosas a la vez. Conforme avance el curso, comprobarás rápidamente que nuestros códigos de programación se inundarán de estos condicionales.



En la categoría Control también encontramos el bloque “repetir hasta que”. Este bloque es muy parecido al bucle “mientras” que utilizamos en PSeInt en sesiones pasadas para el programa de descomposición en números primos. En PSeInt el bucle “mientras” se ejecutaba siempre que la condición fuese Verdadera. Ahora, en este bloque de Scratch, el bucle se repite hasta que la condición sea Verdadera. Si te das cuenta, hay un sutil cambio en la interpretación de la condición.



El escenario de Scratch sitúa la posición (0, 0) en su centro. La primera componente es la componente horizontal (x). Mientras que la segunda componente es la componente vertical (y). Ambas componentes pueden tomar valores positivos (hacia la derecha y hacia arriba respecto del centro) o valores negativos (hacia la izquierda y hacia abajo respecto del centro).

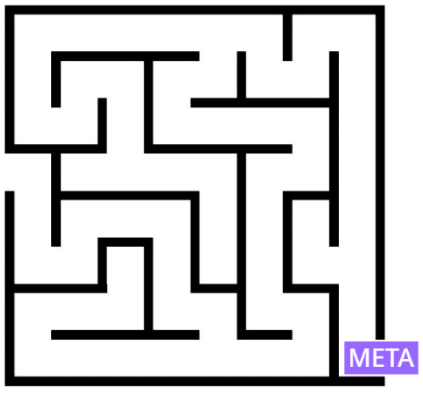
5.2. Juego del laberinto con Scratch: condicional “si … entonces” y sensor de color

Ya podemos crear nuestro primer programa de cierta complejidad con Scratch: mover un objeto circular dentro de un laberinto, con ayuda de las flechas del teclado, sin que toque las paredes. Todo viene explicado, paso a paso, en el siguiente vídeo tutorial: https://www.youtube.com/watch?v=9mApLZFft0U


Scratch Cap 1 - Juego del laberinto en Scratch - Conociendo el interfaz

La imagen del laberinto te la hará llegar el profesor por Teams. Se inserta como escenario nuevo en Scratch con la opción “Carga un fondo”. Verás que el tamaño del objeto se ajusta perfectamente a las dimensiones del escenario.

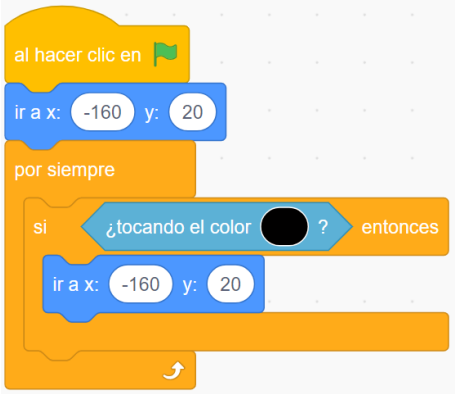
Usaremos el objeto “Ball” que viene cargado por defecto en las opciones de Scratch, y que representa una pelota. Lo encontrarás en “Elige un objeto”. El tamaño de “Ball” es demasiado grande para los pasillos del laberinto, por lo que escalamos el objeto al tamaño 40.



Las flechas derecha, izquierda, arriba y abajo del teclado controlan el movimiento del objeto. Aquí tienes el código de programación por bloques para guiar el movimiento:

El bloque “sumar a x” desplaza horizontalmente el objeto el número de unidades que le indiquemos, respecto de la posición que se encuentre. Puedes escribir un número (positivo o negativo) para fijar el espacio de desplazamiento con cada pulsación de una flecha. El bloque “sumar a y” hace lo mismo, pero verticalmente.

Cuando pulsemos bandera verde, el objeto debe colocarse en la entrada del laberinto. Además, durante todo el funcionamiento del programa, el código debe controlar que el objeto no toque las líneas negras del borde del laberinto. Esto lo conseguimos con el bloque “para siempre”, con el bloque condicional “si … entonces” y con el bloque sensor “¿tocando el color …?”. La siguiente imagen muestra esta parte del código.



Si el objeto toca una línea negra, se reinicia a su posición de partida. Los valores “x: -160”, “y: 20” en el bloque “ir a x: … y: …” son los que sitúan, para el ejemplo del vídeo tutorial, el objeto en la entrada del laberinto. Tú puedes utilizar los valores numéricos que más te convengan para tu juego. ¡Y ya tendríamos terminado nuestro primer videojuego en Scratch!

Si te sobra tiempo de trabajo en clase, puedes animarte a ampliarlo añadiendo algún texto o imagen si el objeto llega a la meta del laberinto, o algún sonido si el objeto cocha con las paredes del laberinto.

6. Los datos generan información: entrada y salida de datos

Ya hemos estudiado que la cantidad más pequeña de información que puede almacenar un ordenador se llama bit. Ahora vamos a emplear nuevamente la palabra “información” con otro significado: como aquello que transmite a una persona un mensaje completo con sentido.

La expresión “Tengo 46 años” es información, porque utiliza datos (números y letras) para informar de la edad de una persona. Si entendemos de esta manera el concepto de información, diremos que **un dato es la unidad mínima de información. Es decir, la información se compone de datos**. En la frase “Tengo 46 años” cada letra y cada cifra es un dato. Por sí solo, un dato no nos dice nada. Pero muchos datos juntos sí nos pueden informar, por ejemplo, de la edad de una persona.

Gracias a los datos tomamos decisiones o hacemos cálculos. No solo en computación y robótica, sino en cualquier faceta de nuestra vida: para organizar unas vacaciones, para comprar el regalo a un compañero que nos invita a una fiesta, etc. Siempre trabajamos con datos para generar información y así para orientar nuestra forma de actuar.

Los datos pueden ser:

- Estructurados: Son los que pueden ser almacenados en tablas y poseen un formato bien definido (fechas de nacimiento, números de DNI, etc.).
- No estructurados: Son los datos recogidos en su forma original y no están recogidos en una tabla: el texto de una carta, la información de un archivo de audio o la imagen captada por una fotografía.

Piensa en una fotografía digital. Un dato es la imagen visual que recibimos. Pero hay más datos: la fecha de realización, el lugar, el zoom aplicado en la lente, el tipo de dispositivo (móvil, cámara compacta, etc.), la luz y el color de cada uno de los píxeles digitales que forman la imagen, etc. Otro ejemplo: El tiempo que una persona dedica a consultar su teléfono móvil: ¿Qué aplicaciones utiliza? ¿Qué mensajes de texto envía? ¿Su ubicación por GPS?, etc.

El volumen de datos digitales que cada persona puede generar en un día es tan grande que, unido a los datos del resto de millones de personas de un país o del planeta, da lugar a una rama de la informática muy de actualidad: El estudio de los **Big data** (datos masivos). Las personas (y los programas) dedicados al Big data recopilan la información de pantallas, cámaras, micrófonos, etc. Los analizan para buscar patrones de comportamiento y repeticiones. Y generan informes de salida con los que tomar conclusiones. Por ejemplo: tecleo en el navegador la dirección de páginas web relacionadas con el baloncesto; comparto por redes sociales imágenes relacionadas con el baloncesto; visualizo vídeos sobre partidos de baloncesto; etc. Este comportamiento, mantenido en el tiempo y analizado por el Big data, puede dar lugar a que las recomendaciones de vídeos, cuando use YouTube, estén relacionados con el baloncesto.

Más aún. El comportamiento de muchos pacientes que sufren la misma enfermedad y sus rutinas con la medicación, generan información importantísima que, bien analizada, puede llevar a tratamientos más eficaces. Los datos de los sensores de temperatura, lluvia y viento almacenados durante décadas son la base de las previsiones meteorológicas cada vez más acertadas. Y los datos de las cámaras de tráfico y de los GPS de los móviles de los conductores ayudan a predecir posibles atascos en la carretera.

6.1. Operar con datos numéricos: sumar en binario

Al igual que escribimos un diario de las experiencias del verano, o tachamos de la lista de la compra los productos que vamos metiendo en el carrito del supermercado, un ordenador también realiza acciones con la información almacenada. Un ejemplo muy intuitivo es realizar operaciones matemáticas. Al igual que nosotros sumamos o restamos datos cuando lo necesitamos, un ordenador también puede operar matemáticamente con la información de tipo numérica.

Un ordenador distingue los datos numéricos según sean de tipo entero (naturales con signo + o con signo -) o real (con decimales). E incluso clasifica según el número de bits que, como máximo, se dedica a cada número (el número de bits determina el número más grande y más pequeño que puede almacenar el ordenador en sus operaciones).

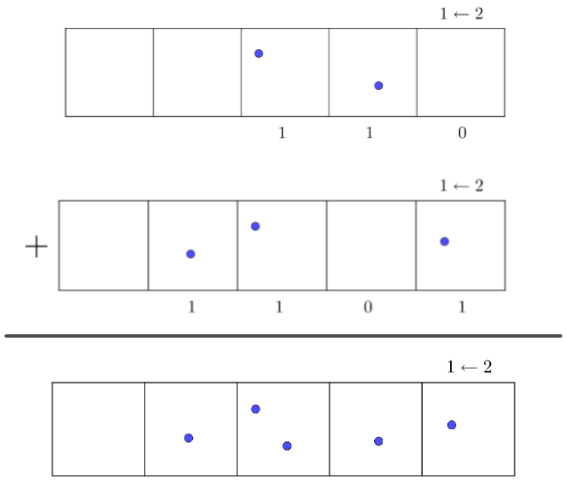
TIPOS DE DATOS | Introducción a los ALGORITMOS y la PROGRAMACIÓN | #2

TodoCode 175 K... Unirme Suscribirme 4,4 K Compartir

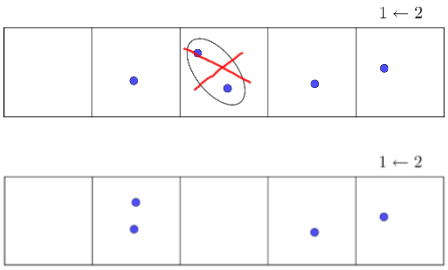
Las computadoras también distinguen los datos formados por letras (**tipo carácter**) y los datos que solo admiten dos opciones lógicas: Verdadero o Falso (**tipo lógico**). El siguiente vídeo te explica un resumen de los tipos de datos numéricos:

https://www.youtube.com/watch?v=_INtSsEcnwc (ver hasta minuto 3:50)

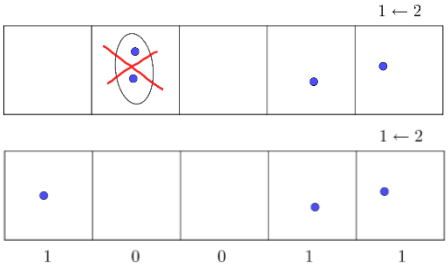
A modo de introducción al procesado de datos por parte del ordenador, vamos a estudiar cómo sumar números en binario. Usando nuestras *Máquinas* $1 \leftarrow 2$ es muy sencillo. Agrupamos una máquina encima de la otra, cada una con su correspondiente número en binario, y sumamos los puntos almacenados en las casillas de las mismas posiciones. En el siguiente ejemplo vamos a sumar los números binarios $(110)_2$ y $(1101)_2$.



En la casilla que aparecen dos puntos, se produce una explosión. Y surge un punto nuevo en la casilla situada justamente a la izquierda.



De nuevo aparecen dos puntos en una misma casilla. ¡Volvemos a explotarlos!



¿Cuál es el resultado de la suma? Un número en binario de código $(10011)_2$.

Si pensamos en decimal, con la descomposición en potencias de base 2 que estudiamos en clases anteriores, podemos comprobar que hemos sumado en formato decimal 6 y 13 para obtener 19 como resultado final.

$$(110)_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6$$

$$(1101)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$$

$$(10011)_2 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 19$$

6.2. ¿Qué información contiene el píxel de una imagen?

Una imagen se divide en pequeños cuadraditos, llamados píxeles. **Cada píxel almacena una información de color e intensidad lumínica**. Y esta información, en el fondo, es una escala de números naturales. Por lo tanto, la información de cada píxel se puede almacenar en código binario o, por ejemplo, en base hexadecimal (que ya estudiamos en sesiones anteriores).

Un ejemplo sencillo para codificar colores es dedicar una cifra hexadecimal a la cantidad de rojo del color (R), otra cifra hexadecimal para la cantidad de verde (G) y otra cifra hexadecimal para la cantidad de azul (B). Generándose así el código de color RGB. Por ejemplo:

$000: negro\ puro\ más\ oscuro$

$FFF: blanco\ puro\ más\ brillante$

$F00: rojo\ puro\ más\ brillante$

$0F0: verde\ puro\ más\ brillante$

$00F: azul\ puro\ más\ brillante$

¿Cuántos colores distintos podemos almacenar con este código RGB? $16 \times 16 \times 16 = 4.096$ colores distintos.

Si dedicamos dos cifras decimales para el rojo, dos cifras hexadecimales para el verde y otras dos cifras hexadecimales para el azul, el número de combinaciones sube bastante: $16 \times 16 \times 16 \times 16 \times 16 \times 16 = 16.777.216$ colores diferentes. Esta cantidad surge de mezclar 256 rojos distintos con 256 verdes distintos y con 256 azules distintos.

En el siguiente enlace puedes practicar con este tipo de codificación hexadecimal, al igual que lo hacen los programas de retoque fotográfico más usuales: La cantidad de color rojo oscila de 0 (negro) a 255 (rojo más brillante), la cantidad de azula oscila de 0 (negro) a 255 (verde más brillante) y la cantidad de verde oscila de 0 (negro) a 255 (azul más brillante).

https://www.geogebra.org/m/wchd89kc#material/qkcmausf

6.3. Bucle de control "segun" en PSeInt

El bucle de control "segun" (en inglés, which; recuerda que no escribimos tildes en los comandos de programación) permite ejecutar varias instrucciones, en función del valor de una variable. En PSeInt esta variable debe ser de tipo numérico. La sintaxis de "según" es:

```
Segun <variable> Hacer
      <número1>: <instrucciones>
      <número2>,<número3>: <instrucciones>
      <...>
      De Otro Modo: <instrucciones>
FinSegun
```

La instrucción final "De Otro Modo" indica qué instrucciones se ejecutarán si el valor de la variable de control no coincide con ninguno de los valores indicados en el condicional.

En la siguiente imagen tienes un programa que pide al usuario dos números (variables *A* y *B*) y que seleccione la operación que desea realizar con esos números:

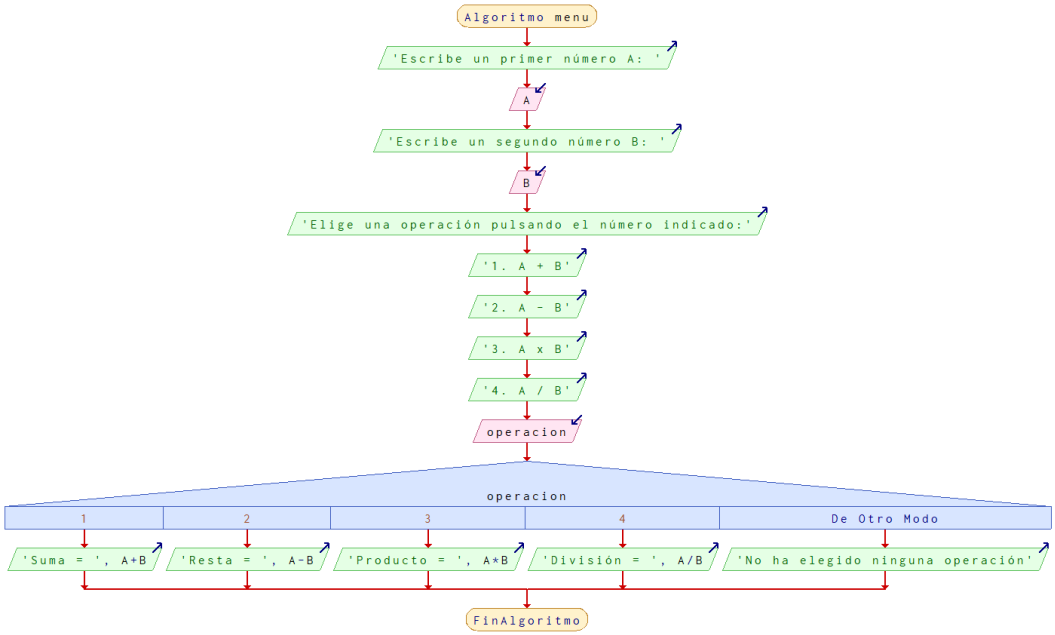
- Si el usuario pulsa 1, se realiza la suma.
- Si pulsa 2, se realiza la resta.
- Si pulsa 3, el producto.
- Si pulsa 4, la división.

En la imagen también verás, en el lateral derecho, el resultado de una ejecución del programa, donde el usuario introduce los números 19 y 4 y elige multiplicarlos para obtener el producto 76.

```
Algoritmo menu
    Mostrar "Escribe un primer número A: "
    Leer A
    Mostrar "Escribe un segundo número B: "
    Leer B
    Mostrar "Elige una operación pulsando el número indicado:"
    Mostrar "1. A + B"
    Mostrar "2. A - B"
    Mostrar "3. A x B"
    Mostrar "4. A / B"
    Leer operacion
    Segun operacion Hacer
        1: Mostrar "Suma = " A+B
        2: Mostrar "Resta = " A-B
        3: Mostrar "Producto = " A*B
        4: Mostrar "División = " A/B
        De Otro Modo: Mostrar "No ha elegido ninguna operación"
    FinSegun
FinAlgoritmo
```

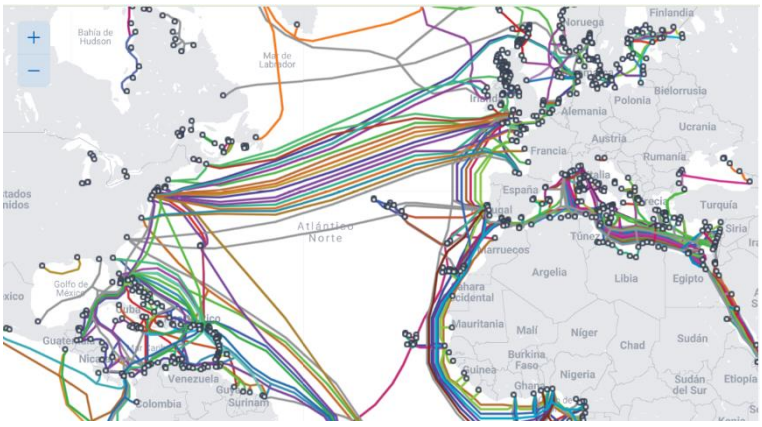
```
PSeInt - Ejecutando proceso MENU
*** Ejecución Iniciada. ***
Escribe un primer número A:
> 19
Escribe un segundo número B:
> 4
Elige una operación pulsando el número indicado:
1. A + B
2. A - B
3. A x B
4. A / B
> 3
Producto = 76
*** Ejecución Finalizada. ***
```

El diagrama de flujo de este programa muestra visualmente qué instrucción se realiza en función del valor tecleado por el usuario y almacenado en la variable *operacion* (recuerda que no escribimos tildes en los nombres de las variables). Todas las opciones posibles del condicional "segun" están a la misma altura en el diagrama de flujo para indicar que no hay ninguna operación que se ejecute antes que otras. Todas las instrucciones están al mismo nivel de prioridad. Es el valor de la variable *operacion* la que determina qué instrucción ejecutar.



7. Transporte y almacenaje de datos

La transmisión de datos entre equipos informáticos se realiza principalmente por cable, con redes de fibra óptica terrestre y submarina (la imagen de la derecha, tomada de la web Submarine Cable Map, muestra la conexión submarina por el Atlántico Norte). También existe la transmisión inalámbrica, a través de antenas y satélites.



Hay datos que se almacenan en nuestros ordenadores y teléfonos personales: un archivo de texto que estoy redactando, una fotografía tomada con la cámara del móvil, etc. Y otros datos quedan almacenados en centros de datos llamados servidores, que son ordenadores de gran potencia, que almacenan y gestionan millones y millones de datos, y que nos permite acceder a la información desde diferentes dispositivos (es lo que se conoce como **almacenaje en la nube**).

El tamaño de los datos nos da una idea del volumen de información que pueden llegar a albergar. Es común utilizar la siguiente terminología:

- 1 Byte (B)
- 1 KiloByte (KB) = 2^{10} B = 1.024 B
- 1 MegaByte (MB) = 1.000 KB = 2^{20} B = 1.048.576 B
- 1 GigaByte (GB) = 1.000 MB = 2^{30} B = 1.073.741.824 B
- 1 TeraByte (TB) = 1.000 GB = 2^{40} B = 1.099.511.627.776 B

7.1. Seguridad en la transmisión de la información

Transmitir datos con seguridad es fundamental: enviar correos electrónicos, recibir mensajes de WhatsApp, realizar un pago por BIZUM o hacer la declaración de la renta de manera online son ejemplos de cómo la transmisión de datos afecta a aspectos muy sensibles de nuestra vida.

La transmisión de datos no segura puede generar una gran cantidad de riesgos: robo de identidad, filtración de datos y pérdidas financieras. Los ciberdelincuentes pueden interceptar la transmisión de datos no segura y acceder a información confidencial.

El cifrado es una de las formas más efectivas de proteger la transmisión de datos. El cifrado implica convertir texto sin formato en texto cifrado, que es ilegible sin la clave de descifrado. Si los ciberdelincuentes interceptan los datos, no podrán leerlos sin la clave de descifrado. El cifrado garantiza que los datos permanezcan seguros mientras están en tránsito de un equipo a otro.

Los certificados SSL/TLS proporcionan cifrado y una conexión segura entre el dispositivo del usuario y el servidor. Garantizan que el usuario se comunica con el servidor deseado y no con uno falso. El siguiente enlace explica de forma sencilla las aplicaciones de los certificados:

https://www.youtube.com/watch?v=tHhFQaurGAg

Terminamos nombrando las redes privadas virtuales (VPN). Garantizan que los datos estén cifrados y también proporcionan anonimato, lo que permite que las actividades en línea del usuario no puedan rastrearse hasta ellas.

7.2. Extensión de los archivos informáticos

Un código de programación se guarda en un archivo informático. Todo archivo ocupa un tamaño al ser almacenado en un ordenador. Cuanto menos texto (o bloques) contenga un código de programación, menos tamaño ocupará el archivo informático que lo contiene.

Por ejemplo, en Scratch podemos descargar un archivo con el código de programación con extensión .sb3. En makecode de micro:bit la extensión de sus archivos es .hex. Cuanto más bloques empleamos, mayor tamaño tendrá el archivo. La extensión identifica el tipo de archivo y el programa o aplicación que puede abrirlo para ser editado.

El siguiente listado muestra las extensiones de archivos más usadas actualmente:

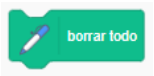
- Documentos de texto: .docx, .pdf, .txt
- Imágenes: .jpg, .png, .gif
- Audios: .mp3, .wav, .flac
- Videos: .mp4, .avi, .mkv
- Ejecutables: .exe, .dmg, .deb
- Comprimidos: .zip, .rar, .7z
- Archivos web: .html, .css, .js

7.3. Acortar el código de programación con el bucle “repetir”

Una de las funciones de los bucles es acortar código de programación: Hacer que el programa funcione correctamente con la menor cantidad de líneas de código (o bloques de programación). Veamos un ejemplo con Scratch, dibujando polígonos regulares.

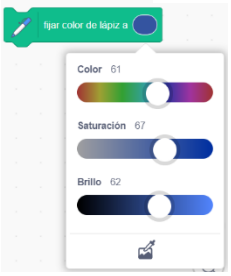
Para dibujar en Scratch, como si de un lápiz se tratara, tenemos que añadir un nuevo módulo a nuestra biblioteca de tipos de bloques. Para ello pinchamos en el botón que ves en la imagen de la derecha. Y en el desplegable de opciones seleccionaremos la opción “Lápiz”.

Esta acción añadirá un nuevo tipo en la biblioteca de bloques, con opciones para poder dibujar. Destacamos las más útiles:



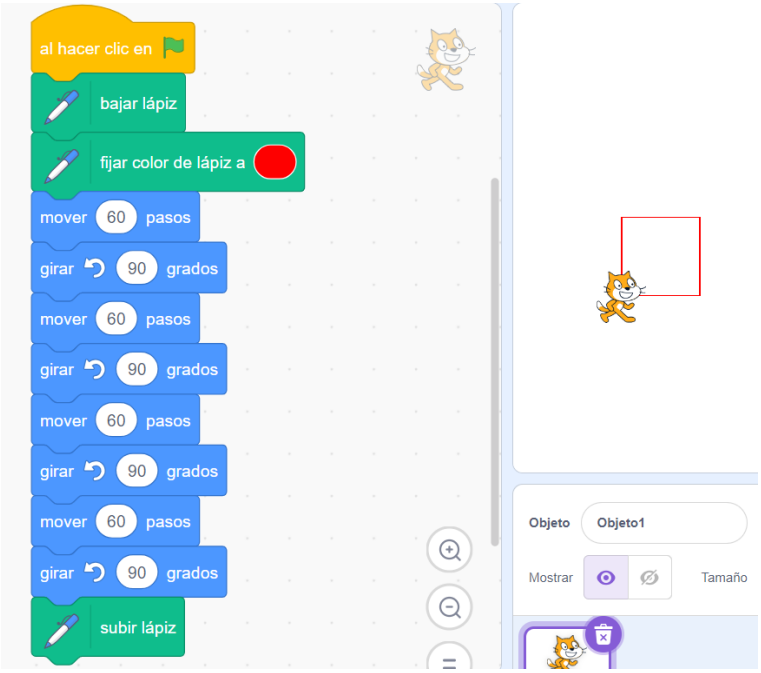
El bloque “borrar todo” elimina todas las líneas y trazos que hayamos dibujado sobre el escenario. Si deseamos dibujar, debemos en primer lugar elegir la opción “bajar lápiz”.

Los bloques “bajar lápiz” y “subir lápiz” funcionan tal y como indican sus nombre. En la vida real, si no bajo el lápiz sobre el papel no podré escribir. De la misma forma, si no bajamos el lápiz sobre el escenario no podremos dibujar en Scratch. Cada vez que deseemos pintar, debemos asegurarnos de que previamente hemos bajado el lápiz. ¡Ojo! Si bajamos el lápiz, cualquier movimiento del objeto dejará su rastro dibujado en el escenario, hasta que utilicemos “subir lápiz”.



El bloque “fijar color de lápiz a …” permite seleccionar el color del trazo, además de la saturación (cantidad de color) y del brillo (cantidad de luz). Con el cuentagotas podemos seleccionar el color exacto de cualquier elemento del escenario.

Con estos bloques ya podemos dibujar en Scratch. Vamos a dibujar un cuadrado (polígono regular de cuatro lados). Si bajamos el lápiz y desplazamos el objeto por el escenario, su movimiento dejará un rastro del color que hayamos seleccionado. El pseudocódigo de nuestro programa, y su codificación con bloques en Scratch, podemos verlo en la siguiente imagen:



1. Bajo el lápiz.
2. Elijo un color.
3. Avanzo el objeto (por ejemplo, 60 pasos).
4. Giro a la izquierda 90 grados.
5. Avanzo el objeto 60 pasos.
6. Giro a la izquierda 90 grados.
7. Avanzo el objeto 60 pasos.
8. Giro a la izquierda 90 grados.
9. Avanzo el objeto 60 pasos.
10. Giro a la izquierda 90 grados (para dejar al objeto con la misma orientación de partida).
11. Subo lápiz (por si deseo mover al objeto sin que dibuje su rastro sobre el escenario).

¿Imaginas hacer este programa para un polígono regular de 8 lados? ¿Y de 20 lados? ¿Y de 100 lados? No es nada práctico tener que añadir continuamente la secuencia "mover 60 pasos" y "girar a la izquierda 90 grados". El bucle "repetir" viene en nuestra ayuda.

Podemos meter la secuencia "mover 60 pasos" y "girar a la izquierda 90 grados" dentro de un bloque "repetir", indicando el número 4 como el número de repeticiones. Quedaría como la imagen de la derecha.

Este código sigue siendo mejorable. Con ayuda de una variable podemos hacer el código más flexible, para que nos permita dibujar cualquier polígono regular.

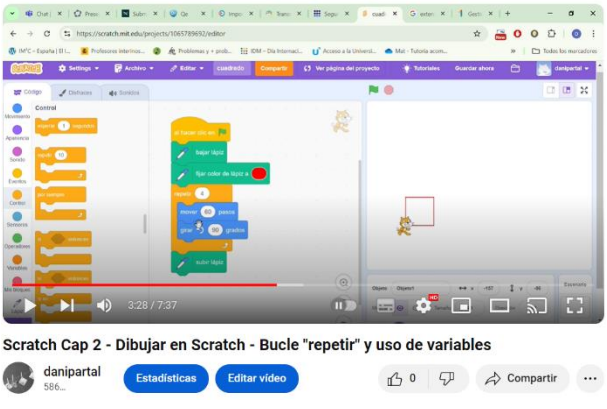
¿Cómo hacerlo? Creamos una nueva variable (categoría "Variables"), que podemos llamar "numeroLados". Recuerda que ya utilizamos variables en el programa PSeInt. Scratch permite indicar si la variable puede ser usada por cualquier objeto (variable global) o solo por un único objeto (variable local). Mientras no digamos lo contrario, las variables que vamos a crear este año en 1ºESO serán siempre globales.

Damos a "numeroLados" el valor inicial 4. Y dentro del bloque "repetir" introducimos el nombre de la variable, en vez del número 4. De esta forma, cambiando el valor inicial de la variables estaremos cambiando también el número de repeticiones del bucle. El avance de 60 pasos podemos dejarlo tal cual. Pero ¿qué ángulo giramos?

En un cuadrado, el giro es de 90 grados. En un triángulo equilátero, el ángulo de giro sería 120 grados. Y en un hexágono regular, 60 grados. ¿Cómo adaptar nuestro programa para que, con ayuda de la variable, sepamos qué ángulo girar? Aplicando la fórmula matemática $360/numeroLados$, porque una vuelta de circunferencia son 360 grados. Dentro de la categoría "Operadores" encontrarás los bloques relacionados con operaciones matemáticas.

¡Y ya lo tendríamos! La imagen de la derecha muestra el código de programación por bloques. Y el enlace de abajo da acceso al video tutorial sobre el programa, explicado paso a paso:

https://youtu.be/7cmrxhDd1nQ



8. Seguridad: creación de números aleatorios

Los números aleatorios son esenciales en los juegos de azar. Un dado, una ruleta o un bingo son ejemplos de juegos donde continuamente se crean números aleatorios. Si deseamos diseñar un dado, una ruleta o un bingo para un programa informático, tendremos que aprender cómo generar un número aleatorio en un algoritmo de programación. Además, los números aleatorios son muy utilizados en la seguridad informática.

Los ordenadores están diseñados para ejecutar instrucciones ordenadas muy bien determinadas. ¿Cómo es posible generar números aleatorios con una computadora programada de manera determinista?

Podemos hablar de dos tipos de números aleatorios creados por los ordenadores:

- **Números pseudoaletorios**. Se generan con un algoritmo que toma un número de partida (semilla) y un conjunto de ecuaciones matemáticas. Estos números son "aparentemente" aleatorios ya que siguen un patrón determinista (la semilla y las ecuaciones matemáticas) pero es imposible predecir, a priori, qué valor aleatorio se va a generar si no se conoce la semilla ni las ecuaciones originales.
- **Números puramente aleatorios**. Utilizan una base física totalmente impredecible para generar los números aleatorios. Por ejemplo: El tiempo de descarga de un condensador, el nivel de ruido recogido por un micrófono en un espacio de tiempo, el tiempo de oscilación de un muelle, la presión al pulsar las teclas del ordenador o la velocidad de movimiento del ratón.

Los números aleatorios son importantes en seguridad informática porque son **impredecibles**. Esto dificulta, por ejemplo, que un ciberdelincuente pueda saber, con antelación, el comportamiento concreto que va a tener un código de programación. Cuando un ordenador genera un número aleatorio, ofrece un conjunto de bits (ceros y unos) totalmente impredecibles, con los cuales se generan nuevos procesos: se crean cálculos matemáticos, se eligen caminos dentro de un algoritmo, se ejecutan determinados condicionales o se generan claves para codificar información.

Cuando decimos que los bits generados son impredecibles significa dos cosas:

- Aunque conozcamos el valor concreto de un conjunto de bits, es imposible saber seguro cuál será el valor de los siguientes bits (impredecible hacia delante).
- Aunque conozcamos el valor concreto de un conjunto de bits, es imposible saber seguro cuáles fueron los valores de los bits que les precedieron (impredecible hacia detrás).

Cuando afirmamos **"imposible saber"** significa que un ordenador tardaría años y años en analizar la información para poder decidir, con total seguridad, el valor del resto de bits aleatorios. ¡Ojo! Con los ordenadores cuánticos (que ya existen en la actualidad) y la evolución de la Inteligencia Artificial (que está en pleno desarrollo) estos parámetros de seguridad se van a ver comprometidos a corto y medio plazo.

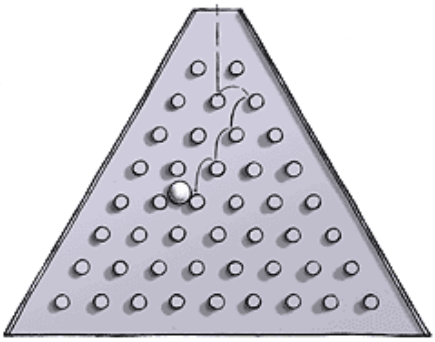
Todos los lenguajes de programación poseen alguna instrucción para generar números aleatorios (random, en inglés). Scratch y makecode de micro:bit lo tienen. Y existen empresas especializadas que venden programas específicos para generar números aleatorios "de alta calidad" que puedan utilizarse en el cifrado de información de seguridad crítica: funcionamiento de centrales nucleares, bases de datos del ministerio de defensa, control de transacciones económicas de un banco, etc. ¡E incluso se utiliza el periodo de desintegración de sustancias radiactivas para la generación de números aleatorios con ordenadores!

8.1. ¿Dos números aleatorios siempre tienen la misma probabilidad de ser elegidos? La tabla de Galton

Lanzar un dado (no trucado) es un experimento aleatorio. No podemos predecir el resultado que va a salir. Si lanzamos un dado, la probabilidad de obtener cualquier número del 1 al 6 es la misma. Hay seis casos posibles, por lo que la probabilidad de cada caso es igual a 1/6 (cada número es un caso favorable dividido entre el número de casos totales). Este razonamiento se conoce como regla de Laplace (científico francés del periodo 1749-1827).

¿Y si lanzamos dos dados y sumamos sus puntuaciones? ¿La probabilidad de obtener cualquier número entre el 2 y el 12 es la misma? Las respuestas a estas preguntas sobre los dos dados formarán una de las actividades de la sesión que debes realizar en tu cuaderno.

En un experimento aleatorio no podemos predecir el resultado que vamos a obtener. Pero esto **no significa que todos los posibles resultados aleatorios tengan la misma probabilidad de ser elegidos**. Veamos esto con el clásico experimento de la tabla de Galton (científico inglés que vivió de 1822 a 1911).



El profesor te mostrará un tablero con el experimento de Galton, parecido a la imagen de la derecha. Se colocan chinchetas separadas la misma distancia unas de otras y se dejan caer canicas. Cuando una canica impacta sobre una chincheta, asumimos que tiene una probabilidad del 50% de ir hacia la izquierda y una probabilidad del 50% de ir hacia la derecha. Al final de la tabla se recogen las canicas que van cayendo. ¿Podemos saber, a priori, en que apertura del final caerá la canica? ¿Todas las aperturas tienen la misma probabilidad de recoger canicas? ¿Cómo quedaría la distribución de canicas si repetimos el experimento con un número inmensamente grande de canicas?

8.2. Comando azar(x) en PSeInt para generar números aleatorios: juego de adivinar número

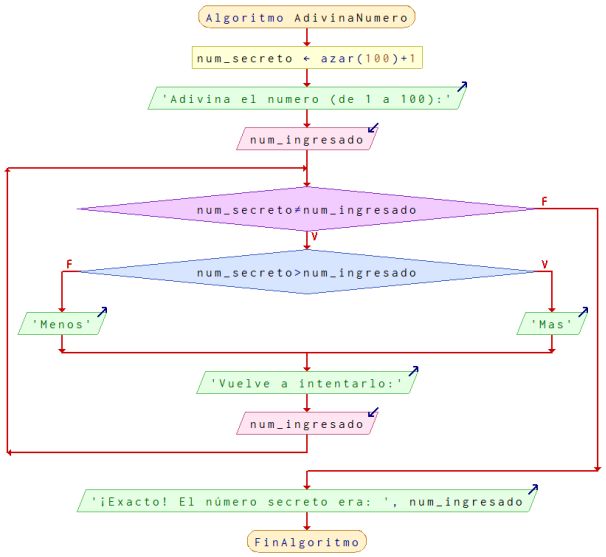
El programa PSeInt dispone de un comando para generar números aleatorios. Si escribes "azar(x)" se genera un número entero entre 0 y x-1. Es decir, si escribimos "azar(100)" tendremos un número aleatorio entero entre 0 y 99.

Imagina que programamos un juego en el que el ordenador "piensa" un número aleatorio secreto entre 0 y 100 y el usuario debe adivinarlo. Cada vez que el usuario propone un número, el ordenador informa si el número secreto es mayor o menor que el número propuesto. Y el programa ofrece la oportunidad de seguir intentándolo, hasta que se adivine.

¿Qué necesitaremos para escribir el pseudocódigo en PSeInt?

1. La función azar(x).
2. Una variable donde almacenar el número aleatorio secreto.
3. Interactuar con el usuario para solicitar que escriba su propuesta de número, que será almacenado en una segunda variable.
4. Un bloque de control "mientras" que se ejecuta siempre que no se acierte el número secreto.
5. Un condicional "si … sino …" para saber si el número secreto es mayor o menor que el número propuesto.
6. Informar al usuario si ha acertado el número secreto.

El diagrama de flujo de este programa podría quedar como el de la imagen de la derecha. Expliquemos paso a paso el diagrama:



- Como "azar(100)" genera un entero entre 0 y 99, escribimos "azar(100) + 1" para obtener un entero entre 1 y 100. El resultado de esta operación lo almacenamos en una variable que vamos a llamar *num_secreto*.
- Solicitamos (proceso de salida) que el usuario escriba un número. Y almacenamos ese número (proceso de entrada) en una nueva variable llamada *num_ingresado*.
- Si *num_secreto* coincide con num_ingresado, el juego termina e informamos al usuario que ha acertado. Si ambas variables no coinciden, usamos un condicional para determinar si *num_secreto* es mayor o menor que *num_ingresado*.
- Fíjate que el primer diagrama de decisión es un bloque "mientras" que se ejecuta si *num_secreto* no coincide con *num_ingresado*. Si es Verdadero (V) que no coinciden, da paso al siguiente diagrama de decisión que es el condicional "si … sino …". Pero si es Falso (F) que no coinciden, significa que ambas variables son iguales y hemos terminado el juego. ¡Ojo con las dobles negaciones! Negar algo falso da como resultado que sea verdadero.

El algoritmo en PSeInt podría escribirse como muestra la siguiente imagen (en gris, tras los símbolos //, aparecen los comentarios aclaratorios):

```
1  // Juego que pide al usuario que adivine un numero entre 1 y 100
2  Algoritmo AdivinaNumero
3      num_secreto ← azar(100)+1
4      // Genera un número al azar entre 1 y 100
5      Escribir 'Adivina el numero (de 1 a 100):'
6      Leer num_ingresado
7      // Almacena el valor del número escrito por el usuario
8      Mientras num_secreto≠num_ingresado Hacer
9          // Si no acierta, se informa si el número secreto es mayor o menor
10         Si num_secreto>num_ingresado Entonces
11             Escribir 'Mas'
12         SiNo
13             Escribir 'Menos'
14         FinSi
15         Escribir 'Vuelve a intentarlo:'
16         Leer num_ingresado
17         // Se almacena el nuevo valor tecleado por el usuario en el nuevo intento
18     FinMientras
19     Escribir '¡Exacto! El número secreto era: ', num_ingresado
20     // Cuando acierta el número, se muestra mensaje de conformación en pantalla
21 FinAlgoritmo
```

Retos para resolver

Siempre, siempre, siempre, siempre debes anotar en tu cuaderno las explicaciones de clase y los esquemas y ejemplos que el profesor escriba o proyecte en la pizarra. Si no tienes el cuaderno limpio, completo y ordenado con las explicaciones de clase, los siguientes retos no serán calificados.

Todos los retos deben estar resueltos en el cuaderno, salvo los archivos informáticos con código de programación que el profesor te indique que el enseñes en clase o que le envíes por chat privado de Teams, para que pueda comprobar el correcto funcionamiento de los programas.

Retos de la Sesión 1

1.1. Deseamos saber si el número 5.146 es divisible entre 6. Diseña dos algoritmos que resuelvan este problema:

- El primer algoritmo solo puede utilizar la operación de multiplicación y la comparación entre números (mayor, menor o igual).
- El segundo algoritmo puede utilizar la operación de multiplicación, la resta y la comparación entre números (mayor, menor o igual).

Indica claramente en tu cuaderno el número de pasos que se dan en cada algoritmo hasta resolver el problema. Puedes tomar como referencia el apartado 1.1., donde hemos realizado una actividad idéntica, pero dividiendo un número entre 4.

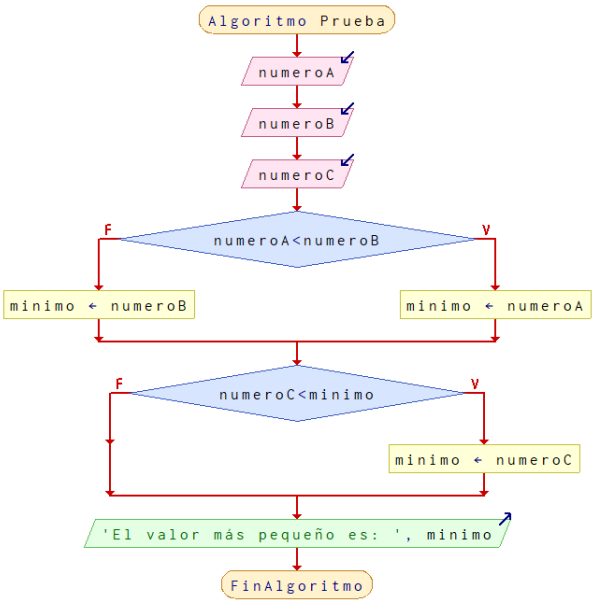
1.2. Copia y completa la siguiente tabla en tu cuaderno, jugando con las torres de Hanoi. ¿Sabiendo el número de discos de partida eres capaz de obtener una fórmula para calcular el número de pasos mínimos con los que resolver el juego?

Torres de Hanoi	
Número de discos de partida	**Número de pasos mínimos para resolver el juego**
2	3
3	7
4	15
5	
6	
7	
8	
9	
10	
n (cualquier número natural)	

1.3. Enseña al profesor, en tiempo de clase, tu pantalla de ordenador con los 20 retos superados del enlace https://studio.code.org/s/20-hour. El profesor debe comprobar que, efectivamente, los 20 retos están en color verde como señal de haber sido superados. Si reinicias el navegador, y pierdes el trabajo realizado, deberás empezar nuevamente desde el primer reto.

Retos de la Sesión 2

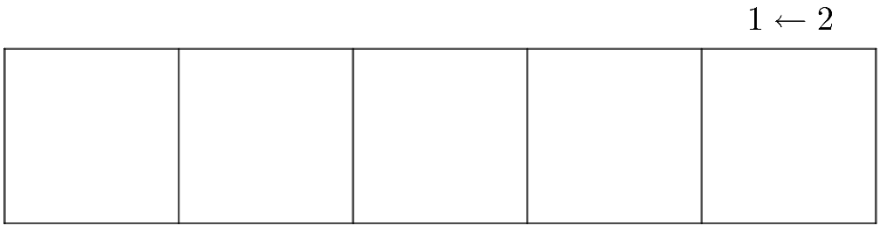
2.1. Mira el siguiente diagrama de flujo. Cópialo en el cuaderno. Explica paso a paso, de forma ordenada, qué hace cada símbolo en el programa



2.2. Analiza el juego con 7 niveles de puntos y determina si existe estrategia ganadora para el jugador 1. Haz los dibujos que necesites en tu cuaderno. Se limpio y ordenado. Y redacta tu razonamiento. Recomendación: descompón el movimiento inicial del jugador 1 en sus tres únicas opciones posibles, y analiza para cada caso si existe o no estrategia ganadora. Ayúdate de lo explicado en la Sesión 2 sobre este juego.

Retos de la Sesión 3

3.1. Toma como ejemplo lo que hemos estudiado sobre la máquina $1 \leftarrow 2$. Utiliza la máquina para representar el número decimal 29 en formato binario. Haz en tu cuaderno todos los dibujos paso a paso, colocando en primer lugar todos los puntos en la casilla de la derecha y señalando poco a poco todas las explosiones de manera ordenada. El profesor te entregará fotocopias de las casillas que necesites para que puedas pegarlas en tu cuaderno y hacer todas las explosiones. Se ordenado y limpio. Utiliza colores para distinguir los puntos de las explosiones.



3.2. No solo podemos codificar números en binario. También podemos codificar texto. Nuestro alfabeto (desde la letra “a” hasta la letra “z”, incluyendo la letra “ñ”) cuenta con 27 caracteres. Vamos a codificar cada letra a binario con ayuda de 5 bits, siguiendo el siguiente orden:

- a: $(00000)_2$
- b: $(00001)_2$
- c: $(00010)_2$
- d: $(00011)_2$
- e: $(00100)_2$
- …

Completa en tu cuaderno toda la tabla de codificación, desde la “a” hasta la “z”.

3.3. Reproduce en PSeInt el algoritmo que hemos explicado en la sesión para convertir un número binario de 4 bits en un número decimal. Debes enseñar el programa funcionando correctamente al profesor, en tiempo de clase. Además, debes copiar en tu cuaderno el diagrama de flujo que ilustra al programa y que tienes como imagen al final de la sesión.

Retos de la Sesión 4

4.1. Crea en makecode el programa de latidos del corazón que hemos explicado en la sesión. Debes enseñar el programa al profesor, en tiempo de clase, funcionando en el simulador y en la placa micro:bit . Además, debes copiar en tu cuaderno el bloque “para siempre” que hace funcionar el programa, con el resto de bloques que contiene en su interior.

4.2. Crea en PSeInt el programa de descomposición en número primos que hemos trabajado en la sesión. Debes enseñar el programa funcionando correctamente al profesor, en tiempo de clase. Además, debes copiar en tu cuaderno el diagrama de flujo que ilustra al programa y que tienes como imagen al final de la sesión. Y estar preparado para explicar al profesor, si te pregunta, los gráficos del diagrama.

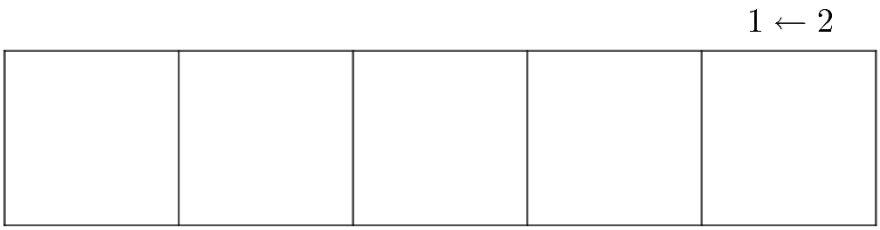
Retos de la Sesión 5

5.1. Copia en tu cuaderno el dibujo y la explicación de los bloques especialmente resaltados en el apartado 5.1 de la sesión 5. Cuida la presentación y la estética de tus dibujos, por favor.

5.2. Enseña al profesor, en tiempo de clase, el juego del laberinto funcionando correctamente en Scratch. Además de las explicaciones de clase, aquí tienes un vídeo tutorial con la creación del código de programación, paso a paso: https://www.youtube.com/watch?v=9mApLZFft0U

Retos de la Sesión 6

6.1. Realiza en binario la suma $(1001)_2 + (1010)_2$. Expresa el resultado final en base decimal, indicando claramente las operaciones de descomposición en base 2. Haz en tu cuaderno todos los dibujos paso a paso. Coloca una máquina encima de la otra para representar por separado la suma inicial de ambos números. Señala poco a poco todas las explosiones de manera ordenada. El profesor te entregará fotocopias de las casillas que necesites para que puedas pegarlas en tu cuaderno y hacer todas las explosiones. Se ordenado y limpio. Utiliza colores para distinguir los puntos de las explosiones.



6.2. Imagina que tienes fotografía de tamaño 1024x720 píxeles. La información de cada píxel se almacena con dos cifras hexadecimales. ¿Qué tamaño en Bytes ocupa la fotografía almacenada en el ordenador? Recuerda que 1 Byte son 8 bits.

6.3. Crea en PSeInt el programa de selección para sumar, restar, multiplicar o dividir dos números que hemos explicado en la sesión. Debes enseñar el programa funcionando correctamente al profesor, en tiempo de clase. Y estar preparado para explicar al profesor, si te pregunta, los gráficos del diagrama.

Retos de la Sesión 7

7.1. Indica diez ejemplos de la vida cotidiana donde el uso de certificado SSL/TSL sea de vital importancia para garantizar la seguridad de nuestra navegación por internet.

7.2. Enseña al profesor, en tiempo de clase, el dibujo de polígonos regulares en Scratch, utilizando el bucle repetir y variables. Recuerda que tienes acceso al video tutorial sobre el programa: https://www.youtube.com/watch?v=7cmrxhDd1nQ

Retos de la Sesión 8

8.1. Explica cinco fenómenos que puedan usarse para generar números aleatorios. Deben cumplir ser impredecibles y cambiar rápidamente de valor. Por ejemplo: Cuando el usuario de una página web pulsa la tecla “Enter” el ordenador toma como valor el número de milisegundos que han pasado desde el 1 de enero de 1970. No sabemos cuando el usuario pulsará “Enter” (impredecible) y el valor de los milisegundo cambia continuamente conforme pasa el tiempo.

8.2. Cada alumno de la clase escribe en un papel un número natural entre el 1 y el 10. No lo comparte con nadie. La intuición nos dice que si cada alumno elige un número al azar, el número de alumnos que escogería el 1 sería igual al número de alumnos que elegirían el 2, e igual al número de alumnos que optarían por el 3, … , e igual al número de alumnos que elegirían el 10. ¿Ocurre esto, realmente, cuando comprobamos en clase los resultados que cada compañero ha escrito? Haz una tabla de valores indicando, por filas, cada número del 1 al 10 y el número de veces que ha sido escogido ¿Encuentras alguna explicación a lo que ha ocurrido? ¿Si repetimos por segunda vez el experimento aleatorio obtendríamos los mismos resultados? Escribe de manera razonada y redactando adecuadamente tu opinión.

8.3. Al lanzar dos dados y sumar sus puntuaciones, obtenemos cualquier número entre 2 y 12. Escribe todas las combinaciones posibles (por ejemplo: 1+1=2, 1+2=3, 1+3=4, … , 2+1=3, 2+2=4, 2+3=5, … , 6+1=7, 6+2=8, …). El orden es importante: el primer resultado representa el primer dado y el segundo resultado el segundo dado. Haz una tabla indicando, por filas, la suma de ambas puntuaciones y el número de veces que aparece esa suma. Calcula la probabilidad de obtener cada valor de la suma, usando la regla de Laplace, tal y como hemos explicado en clase (casos favorables dividido entre casos totales).

8.4. Escribe en PSeInt el programa para adivinar el número aleatorio generado por el ordenador, que se explica en la sesión. Debes enseñar el programa funcionando correctamente al profesor, en tiempo de clase. Y estar preparado para explicar al profesor, si te pregunta, los gráficos del diagrama.