



DEPARTAMENTO DE DESARROLLO PRODUCTIVO Y TECNOLÓGICO
Introducción a Licenciatura en Sistemas 2024
Guía Algoritmos (Ult. Act. Nov/2023)

Fundamentos de algoritmos

¿Qué es un algoritmo?

Un algoritmo es una secuencia ordenada de instrucciones o pasos que representan una forma de resolver un problema. La aplicación (o ejecución) del algoritmo es llevar a cabo cada una de las instrucciones que lo componen de manera secuencial (una tras otra y en el orden en que se indica en el algoritmo). Al finalizar la ejecución, tendremos la solución de dicho problema. Y si bien un algoritmo representa una solución para un problema, no es necesariamente la única. Puede haber más de una solución a un problema, y cada una de ellas estará representada por un algoritmo distinto.

Todo algoritmo debe ser ejecutado por algún agente: una persona, una máquina o cualquier otra cosa capaz de ejecutar instrucciones de manera secuencial. El ejemplo usual es el de una receta de cocina: si nuestro problema es hacer una torta de chocolate, el algoritmo es la receta de torta de chocolate. Aplicamos el algoritmo (seguimos las instrucciones de la receta en orden desde la primera hasta la última) y como resultado, obtenemos nuestra ansiada torta de chocolate.

Estamos omitiendo aquí algo muy importante para poder llevar adelante nuestra receta: los ingredientes necesarios para preparar la torta. Ambos (receta e ingredientes) son igualmente importantes. En los algoritmos que procesan información (los que veremos en este curso), la materia prima con la que opera el algoritmo se denomina datos.

¿Qué es un dato?

Un dato es una entidad que representa un hecho concreto del problema que queremos resolver. Si por ejemplo, nuestro problema a resolver es “hallar la suma de dos números” los datos necesarios para aplicar el algoritmo “sumar dos números” son los números a sumar.

¿Un algoritmo es un programa?

Es una buena pregunta. Si definimos programa como algo que corre en una computadora, no. Una receta de cocina es un algoritmo y sin embargo no es un programa de acuerdo a nuestra definición. Sin embargo, en el presente curso escribiremos nuestros algoritmos como si fueran programas, porque el objetivo del mismo es aprender a desarrollar algoritmos para resolver problemas con computadoras; de modo que nuestros algoritmos en algún momento deberán transformarse en un programa que una computadora ejecute. Y además, si bien hay programas que implementan un algoritmo determinado, en general un programa de cierta envergadura incluye más de un algoritmo para llevar adelante su tarea. Cualquier aplicación que se ejecute en un teléfono inteligente, por ejemplo, emplea una multitud de algoritmos: para conectarse a través de la red, para mostrar datos por pantalla, para reproducir sonidos, para comprimir y encriptar datos, etc.

Pseudocódigo

Vamos a escribir nuestro algoritmo de manera formal. Para ello, emplearemos la sintaxis especificada en el apunte “Sintaxis pseudocódigo”. Se denomina “pseudocódigo” porque es similar al código que se escribe en un lenguaje de programación y que puede ejecutarse en una computadora. El pseudocódigo es una versión en castellano (en los lenguajes de programación se emplea el inglés) y mucho más simplificada que un lenguaje de programación propiamente dicho. El pseudocódigo se utiliza para poder formalizar y transmitir algoritmos a otras personas, sin necesidad de hacerlo en un lenguaje de programación determinado. Es una manera de especificar algoritmos independiente del lenguaje de programación en que vaya a implementarse.

Nos preguntamos entonces: ¿En qué máquina correrá el algoritmo que escribamos en pseudocódigo?. la respuesta es simple: En nuestras cabezas. Para ejecutar un algoritmo en pseudocódigo haremos lo que se denomina “prueba de escritorio”, que es una simulación de la corrida del algoritmo paso a paso, especificando los valores que toman sus variables tras cada uno de ellos.

Comencemos entonces, con algo realmente simple: Un algoritmo para sumar dos números:

```
Algoritmo SumaDosNumeros_V2 //para calcular la suma de los números almacenados
//definimos 3 variables de tipo Entero sumando1, sumando2, total
definir sumando1 como Entero;
definir sumando2 como Entero;
definir total como Entero;

//asignamos el valor de las variables sumando1, sumando2
sumando1 <- 3;
sumando2 <- 4;

//asignamos en la variable total la suma de sumando1 + sumando2
total <- sumando1 + sumando2;

//mostramos un mensaje y el resultado de la suma
escribir "El total de la suma es: ",total;
escribir sumando1,"+",sumando2, "=",total;
FinAlgoritmo
```

Veamos las distintas partes que componen este algoritmo:

A cada uno de los renglones que componen el algoritmo los denominamos “líneas de código” no importa si se trata de pseudocódigo o código en algún lenguaje de programación, siempre hablamos de líneas de código. En cada línea de código hay una sentencia. Cada sentencia tiene un significado y representa una acción determinada a ejecutarse cuando el flujo del algoritmo pase por ella. En el caso de la receta de cocina, “agregar tres cucharadas de azúcar” es una sentencia.

El flujo del algoritmo es la secuencia en la que las instrucciones se ejecutan e indica cuál es la sentencia que se encuentra en ejecución en un momento dado. Por ejemplo, la primera sentencia de nuestro algoritmo:

Algoritmo SumaDosNumeros_V2

nos indica que nos encontramos en el principio del algoritmo cuyo nombre es “SumaDosNumeros_V2”. Todos los algoritmos deben llevar un nombre que los identifique y describa de manera mínima que es lo que hacen. V2 indica que se trata de la segunda versión del mismo. La última sentencia:

FinAlgoritmo

nos indica que hemos llegado al fin de nuestro algoritmo y que ya no deberemos hacer nada. Es decir, esta instrucción indica el punto en el que la ejecución de nuestro algoritmo termina.

Las líneas de código

```
//para calcular la suma de los números almacenados
//definimos 3 variables de tipo Entero sumando1, sumando2, total
//asignamos el valor de las variables sumando1, sumando2
//asignamos en la variable total la suma de sumando1 + sumando2
```

```
//mostramos un mensaje y el resultado de la suma
```

son comentarios. Los comentarios no tienen ninguna acción asociada, es decir, al pasar por dichas líneas no deberemos ejecutar acción alguna. Sólo están para información de la persona que lee el algoritmo.

Las sentencias:

```
definir sumandol como Entero;  
definir sumando2 como Entero;  
definir total como Entero;
```

nos indican que en el algoritmo emplearemos tres variables del tipo Entero. Una variable es una entidad de un algoritmo que cuyo objeto es almacenar datos para poder referirnos a ellos y manipularlos. En un algoritmo que se ejecuta en una computadora, son espacios reservados en la memoria que, como su nombre indica, pueden cambiar de contenido a lo largo de la ejecución del mismo. En nuestro caso, estamos **declarando** tres variables. Declarar tres variables es un modo de avisar que hará falta espacio para almacenar tres datos: Si ejecutamos nosotros el algoritmo realizando una prueba de escritorio, deberemos reservar un lugar en una hoja de papel para ir anotando los valores que toma cada una de las variables durante la ejecución del algoritmo. Si es una computadora ejecutando un algoritmo escrito en un lenguaje de programación (un programa), reservará espacio en la memoria con dicho propósito.

La declaración consta de tres partes: primero está la palabra clave definir, que nos indica que en esta sentencia se realizará la definición de una variable.

La siguiente parte de la declaración (siempre en la misma línea de código) es el **nombre de la variable**. Las variables deben tener un nombre para poder referirnos a ellas a lo largo del algoritmo. En este caso tenemos la variables sumandol. Los nombres de variables son alfanuméricos y la única restricción es que no deben ser palabras reservadas del lenguaje de nuestro pseudocódigo. Las palabras definidas en el apunte “Sintaxis Pseudocódigo” son reservadas. Esto es para evitar confusiones; la misma restricción suele aplicarse en los lenguajes de programación. Es deseable que los nombres de las variables reflejen aquello que representan. Si bien nuestro algoritmo puede resolverse usando tres variables a, b y c en lugar de sumandol, sumando2 y total, el código queda mucho menos claro y en algoritmos complejos esto puede ser causa de errores. Pero la peor consecuencia es que otra persona que lea nuestro código va a tener que hacer doble trabajo: Comprender el algoritmo y además comprender para qué sirve cada variable que estamos declarando. Si el nombre de la variable refleja su propósito, habremos evitado ese trabajo a otros. Y otros nos lo habrán evitado a nosotros.

La tercera y última parte de la declaración es el tipo de la variable: esto indica de qué tipo es el dato que se almacenará en la misma. En nuestro caso, un número entero. Existen otros tipos de variables, los iremos introduciendo a medida que avancemos en el curso.

Las declaraciones de variables deben ir al principio del algoritmo. Declararemos todas las variables juntas, esto nos permite tener una idea clara de las necesidades de almacenamiento del algoritmo y nos facilitará la definición de la prueba de escritorio, como veremos más adelante.

Las líneas a continuación de la declaración de variables son:

```
sumandol <- 3;  
sumando2 <- 4;
```

Esto se denomina **asignación de variables**. Se llama así porque simplemente le estamos asignando un valor (en este caso el valor 3 a sumandol y el valor 4 a sumando2) a cada variable. Éstos son los **datos** que nuestro algoritmo requiere para funcionar.

La tercera variable, total, se asigna de un modo distinto:

```
total <- sumandol + sumando2;
```

Aquí, antes de asignar a total, hay que calcular el valor necesario. Esto es lo que se hace del lado derecho del signo igual: El **operador +** nos indica que hay que realizar la suma de dos números. Para realizar la suma, lo que hacemos mentalmente es reemplazar cada variable por el valor que tiene asignado en ese momento.

Como el valor de sumando1 es 3 y el de sumando2 es 4, sumamos $3 + 4 = 7$, y ese es el valor que asignaremos a total.

A continuación, la sentencia

```
escribir "El total de la suma es: ",total;
```

nos indica que el algoritmo nos muestra un mensaje y el valor de la variable total de algún modo. Nos mostrará el mensaje "El total de la suma es 7". Cuando realicemos la prueba de escritorio veremos como indicaremos esta **salida** del algoritmo. Por el momento sólo tomemos nota de que escribir es una **función** del lenguaje que recibe **argumentos** y los muestra de algún modo, por pantalla, por impresora, nos los envía por mail, etc. No importa ahora, ni veremos en este curso, de qué manera lo hace.

Finalmente, la sentencia

FinAlgoritmo

nos indica que hemos llegado al fin del algoritmo y que ya no deberemos realizar ninguna acción más.

Prueba de Escritorio

La prueba de escritorio es un método para ejecutar un algoritmo utilizando lápiz y papel. A tal fin, crearemos una tabla como la que sigue:

Paso	sumando1	sumando2	total	Entrada / Salida	Comentarios

Para armar la tabla, definimos una columna "paso" donde indicaremos el paso en que nos encontramos. Los pasos se indican con un número, comenzando por el 1 e incrementándose en 1 cada vez que pasemos por una sentencia. Seguidamente miramos la declaraciones de variables y agregamos una columna por cada variable declarada. Finalmente agregaremos una columna "Entrada/Salida" donde indicaremos las entradas y salidas del algoritmo y una columna "Comentarios" para indicar alguna cosa especial que queramos dejar anotada.

Comenzaremos con sólo dos renglones, agregaremos más a medida que haga falta. Cada renglón representa un paso en el algoritmo. Un paso no es necesariamente una sentencia, cuando veamos estructuras de decisión y de iteración entenderemos por qué. Por lo pronto bastará con saber que cada vez que pasemos de una sentencia a la siguiente deberemos agregar un paso a nuestra tabla.

Realicemos la prueba con nuestro algoritmo, comenzando desde su definición y siguiendo en orden descendente. La primera sentencia que encontramos es

Algoritmo SumaDosNumeros_V2

Esta sentencia sólo nos indica que estamos frente a un algoritmo. No es necesario indicarla en la prueba de escritorio. Seguimos con la siguiente sentencia:

```
//para calcular la suma de los números almacenados
```

Esto es un comentario, existe sólo para nuestra información y tampoco se indica en la prueba de escritorio. Lo mismo con la línea que sigue a continuación:

```
//definimos 3 variables de tipo Entero sumando1, sumando2, total
```

Pasamos a la siguiente:

```
definir sumando1 como Entero;
```

Esto es una definición de variable. Las definiciones de variables ya las hemos representado en la tabla con las columnas que agregamos al efecto (una por cada variable) y que en este caso, son las columnas sumando1, sumando2 y total. Tampoco es necesario crear un paso para ellas. Ocurre lo mismo con las dos sentencias siguientes:

```
definir sumando2 como Entero;  
definir total como Entero;
```

Saltamos el comentario, ya que solo está a nivel informativo:

```
//asignamos el valor de las variables sumando1, sumando2
```

Pasamos entonces a la sentencia que sigue:

```
sumando1 <- 3;
```

Aquí tenemos, como vimos previamente, una asignación de variable. En este caso sí corresponde agregar un paso e indicar el valor que toma la variable:

Paso	sumando1	sumando2	total	Entrada / Salida	Comentarios
1	3	-	-	-	-

Como podemos ver, sólo especificamos el valor de la variable que se acaba de asignar. De las otras dos variables (sumando2 y total) aún no sabemos nada, porque el flujo de ejecución del algoritmo aún no ha pasado por las sentencias que les asignan un valor. Continuamos con la siguiente sentencia:

```
sumando2 <- 4;
```

Otra asignación de variable. Agregamos un paso más:

Paso	sumando1	sumando2	total	Entrada / Salida	Comentarios
1	3	-	-	-	-
2	3	4	-	-	-

En este paso no sólo especificamos el valor que toma la variable sumando2, sino que repetimos el valor de la variable sumando1 que asignamos en el paso anterior. Entre el paso 1 y el 2 no ocurrió nada que cambie el valor de la variable sumando1, por lo que consignamos otra vez el mismo. Los valores de las variables se mantienen mientras no ocurra otra asignación que cambie su valor.

Continuamos con la siguiente sentencia:

```
total <- sumando1 + sumando2;
```

Es en esta sentencia donde ocurre el verdadero trabajo del algoritmo. Hasta ahora sólo hemos preparado las condiciones para que esto ocurra. Agregamos un paso más:

Paso	sumando1	sumando2	total	Entrada / Salida	Comentarios
1	3	-	-	-	-
2	3	4	-	-	-
3	3	4	7	-	total=sumando1+sumando2

Como hicimos para el paso anterior, consignamos los valores de sumando1 y sumando 2, que no han cambiado, y agregamos el nuevo valor de la variable total. En el comentario sólo indicamos cómo llegamos a ese valor. El comentario no es estrictamente necesario, sólo está para hacernos más fácil el seguimiento de la ejecución del algoritmo.

Podemos ver que en cada fila de la tabla (en cada paso del algoritmo) la prueba de escritorio nos muestra el estado de todas las variables del mismo. Esto nos simplifica el seguimiento, ya que sabemos con exactitud cuál es el valor de cada variable en cualquier paso del algoritmo.

Pasamos a la sentencia siguiente:

```
escribir "El total de la suma es:",total;
```

Esta sentencia, como ya lo indicamos, produce lo que denominamos una **salida** del algoritmo. Una salida es todo lo que el algoritmo nos devuelve, cualquier mensaje para el usuario y la solución al problema que resuelve. En nuestro caso el problema es hallar la suma de dos números, por lo tanto la salida del algoritmo deberá ser el resultado de esa suma. Si nuestro algoritmo se ejecutara en una computadora, recibiríamos una línea de texto por pantalla o en una impresora (en realidad ahora no nos importa mucho) con la leyenda "El total de la suma es 7". Agregamos un paso más a nuestra prueba:

Paso	sumando1	sumando2	total	Entrada / Salida	Comentarios
1	3	-	-	-	-
2	3	4	-	-	-
3	3	4	7	-	total=sumando1+sumando2
4	3	4	7	<El total de la suma es: 7	

Consignamos los mismos valores de las variables, ya que no han cambiado, y agregamos el valor que toma la salida del algoritmo. Cuando consignemos una salida, lo haremos comenzando con el signo <.

Agregamos un paso más para la segunda salida:

Paso	sumando1	sumando2	total	Entrada / Salida	Comentarios
1	3	-	-	-	-
2	3	4	-	-	-
3	3	4	7	-	total=sumando1+sumando2
4	3	4	7	<El total de la suma es: 7	
5	3	4	7	<3+4=7	

La siguiente, y última sentencia es:

```
FinAlgoritmo
```

Que nos indica que hemos llegado al fin del algoritmo y no es necesario agregar un paso para ella. Es en este momento cuando termina la prueba de escritorio.

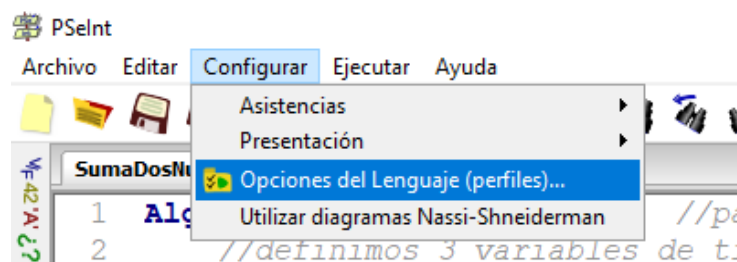
La herramienta PseInt

Además de la prueba de escritorio (en papel), podemos utilizar la herramienta PSeInt para ejecutar nuestro algoritmo. PSeInt es una herramienta para asistir a un estudiante en sus primeros pasos en programación. Mediante un simple e intuitivo pseudolenguaje en español, le permite centrar su atención en los conceptos fundamentales de la algoritmia computacional, minimizando las dificultades propias de un lenguaje y proporcionando un entorno de trabajo con numerosas ayudas y recursos didácticos. Puede descargarse en: <http://pseint.sourceforge.net/>

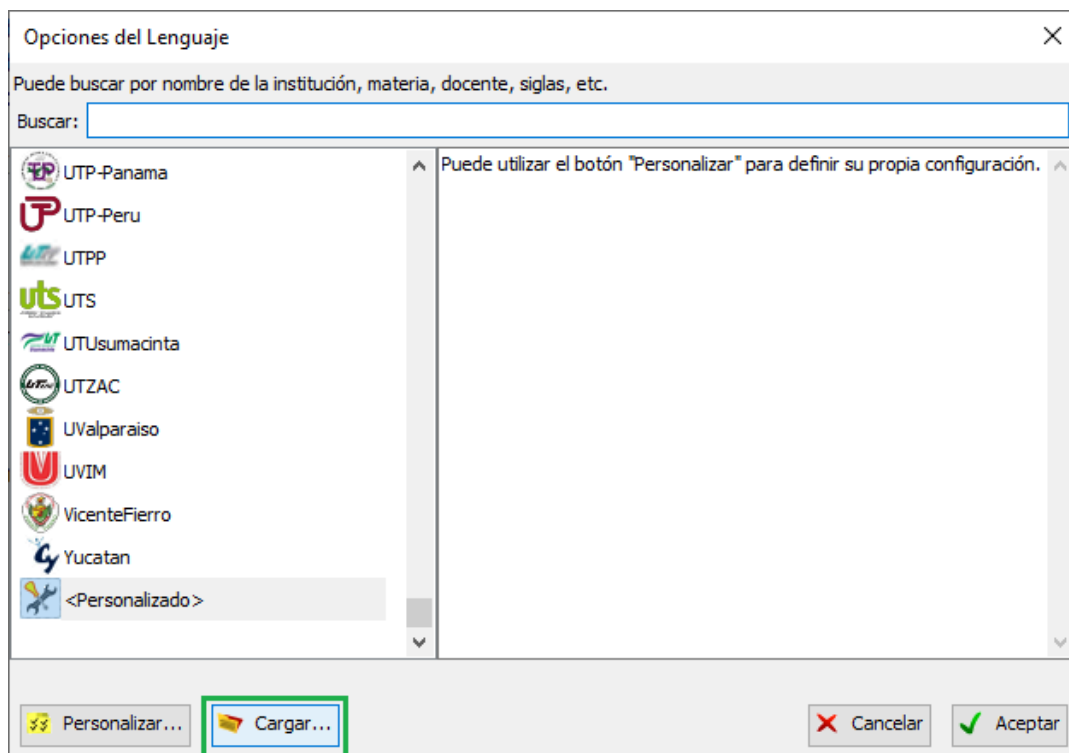
Técnicamente hablando, PseInt es un intérprete de pseudocódigo. Un intérprete (sin profundizar demasiado en el concepto) es un programa que toma un algoritmo escrito en un lenguaje (en este caso, pseudocódigo) y lo interpreta. Interpretar significa recorrer el algoritmo e ir ejecutando las instrucciones que contiene, comenzando por el principio del mismo y paso a paso de manera secuencial hasta el final, del mismo modo en que realizamos la prueba de escritorio.

Teniendo lo anterior en cuenta, y retomando lo que decíamos al principio del apunte (¿Un algoritmo es un programa?), podemos concluir que nuestros algoritmos escritos en pseudocódigo son actualmente programas ejecutables en PseInt.

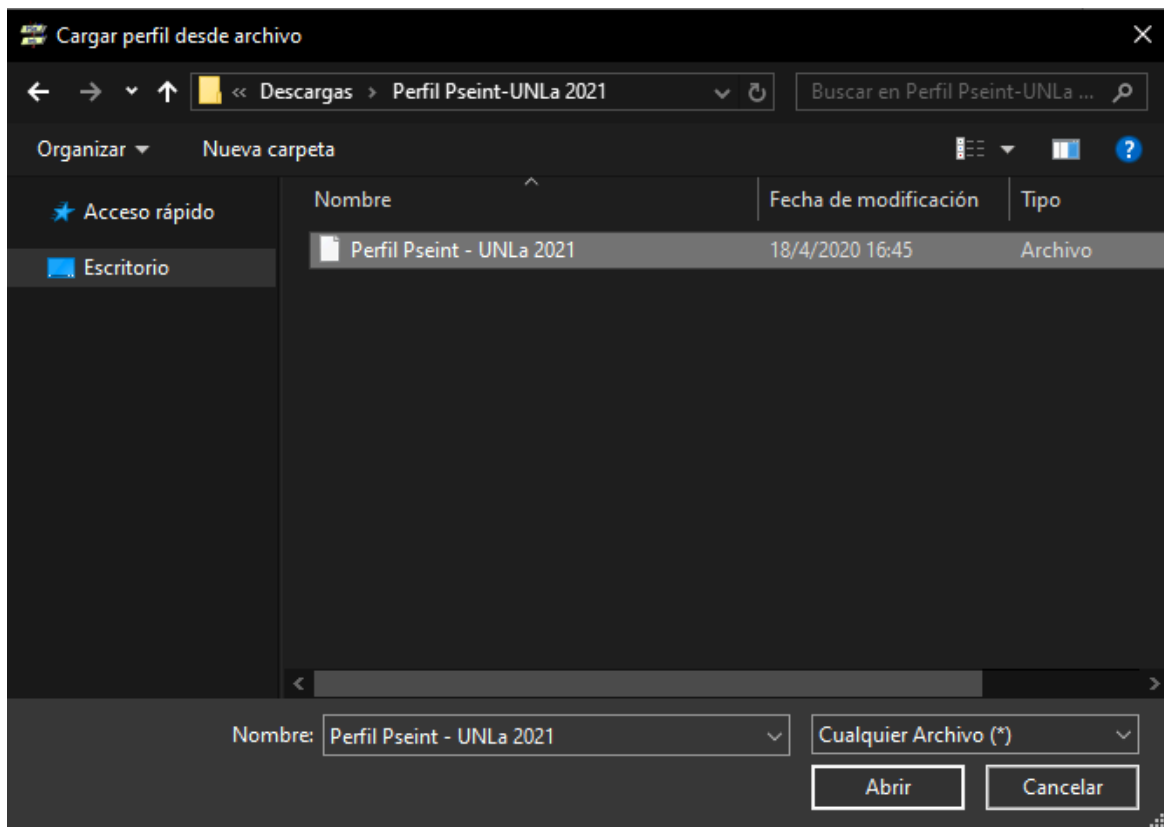
Luego de descargarlo e instalarlo, se debe cargar el perfil "Perfil PseInt Ingreso UNLa", que se encuentra en la misma carpeta donde se encuentra este apunte. Esto se hace desde el menú de PseInt, opción Configurar / Opciones del Lenguaje (perfiles)...":



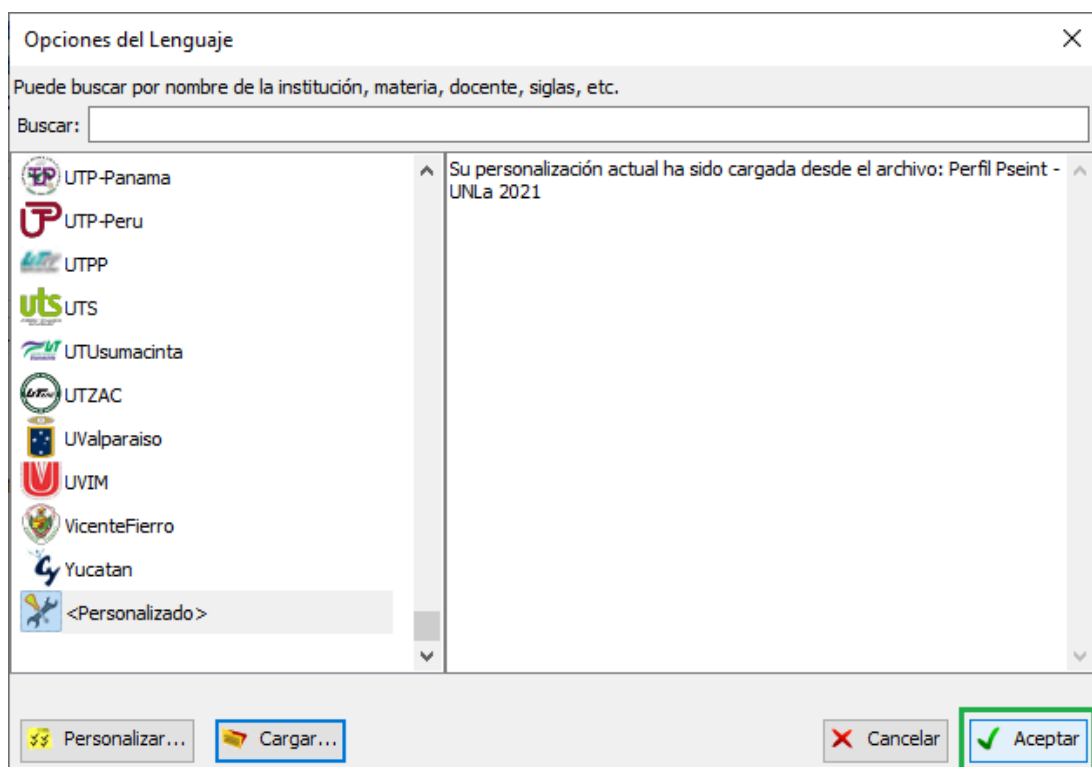
y en la ventana que se abre elegir el botón cargar :



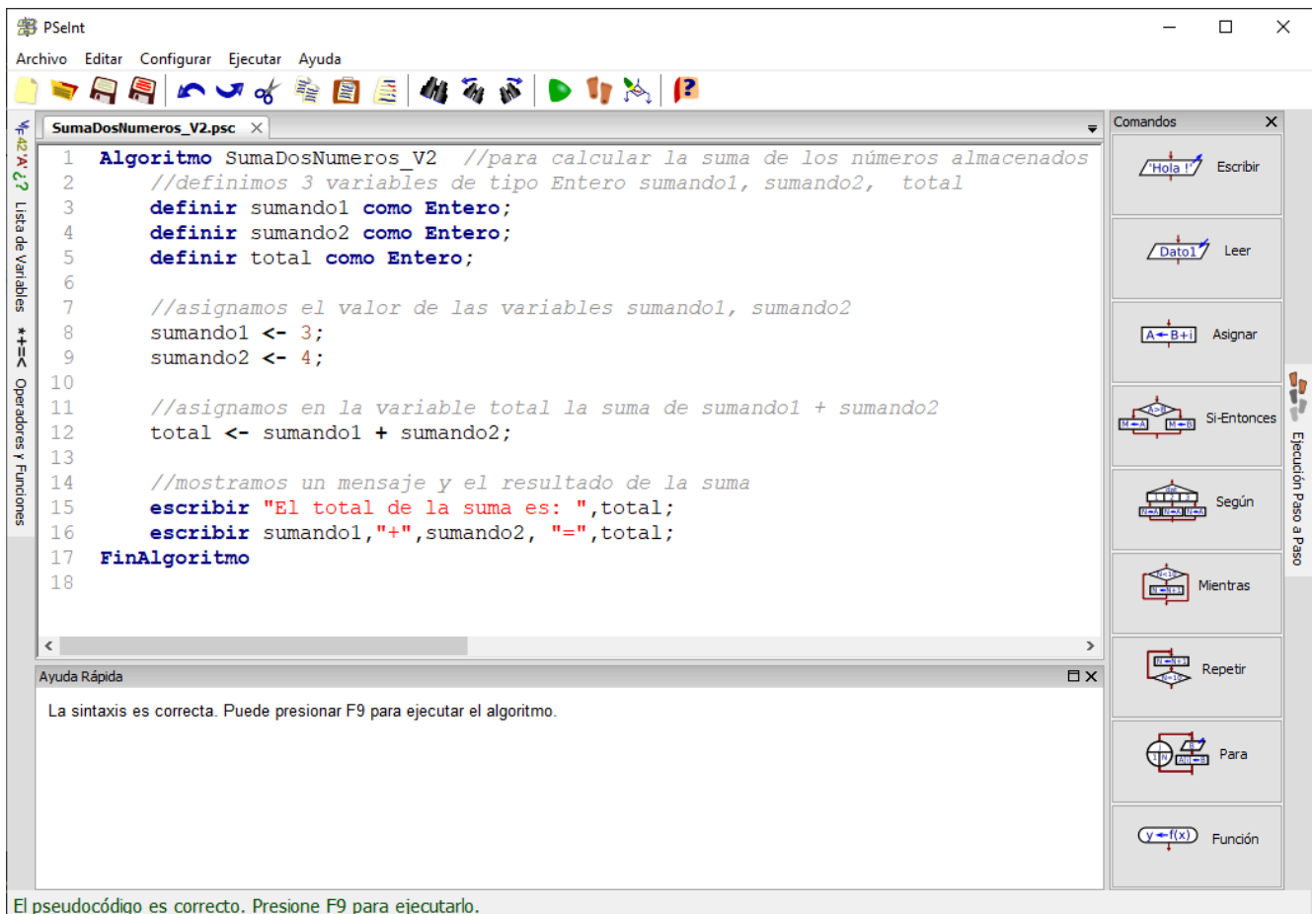
Se debe seleccionar el perfil de la UNLa:




Y luego se debe hacer clic en Aceptar:



En la ventana del editor, podemos tipear nuestro algoritmo SumaDosNumeros_V2:



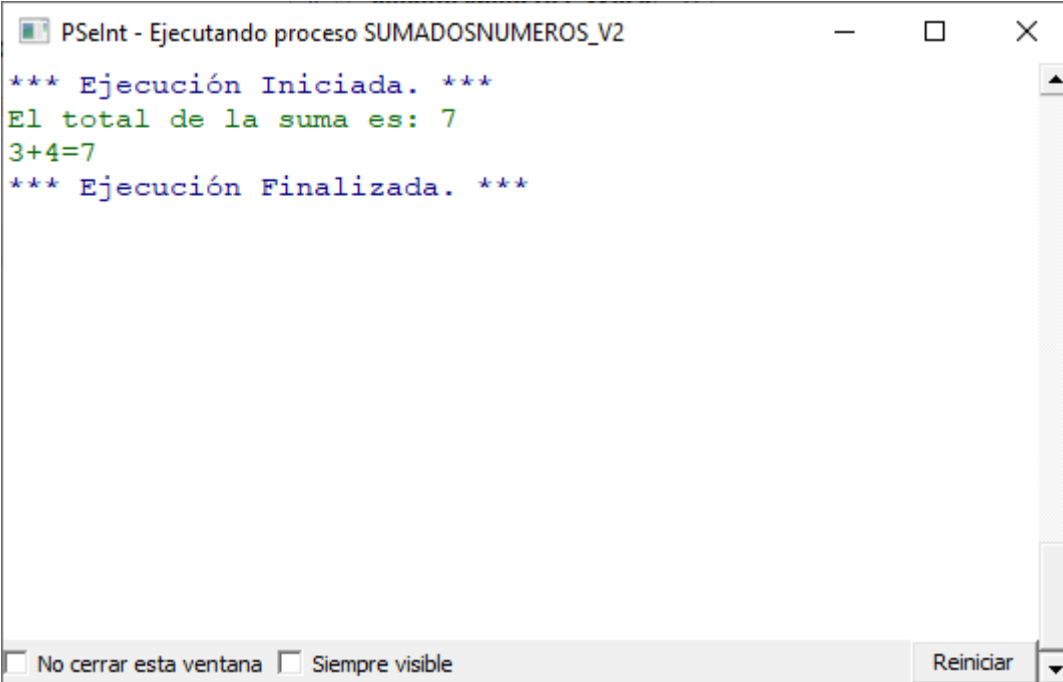
Y presionamos F9 para ejecutarlo. Alternativamente, podemos dar click al botón “Ejecutar”  de la barra de tareas.

El editor de Pselnt irá subrayando en rojo las sentencias que tengan errores sintácticos al mismo tiempo que las vayamos ingresando. Además, antes de ejecutar nuestro algoritmo hará una revisión del mismo y nos dará un reporte de los errores de sintaxis que pueda contener. Si el algoritmo contiene algún error, no se ejecutará. De ser así, deberemos corregir los errores indicados y reintentar la ejecución del mismo. Se recomienda la lectura del archivo de ayuda de Pselnt para familiarizarse con su funcionamiento.

Si bien Pselnt captura los errores de sintaxis y de tipos que existan en nuestro código, estos errores son sólo una parte de los errores posibles del algoritmo, y son los que impiden su ejecución simplemente porque Pselnt “no entiende” lo que escribimos. Un intérprete de un lenguaje como es Pselnt, es muy estricto en cuanto a la forma en que se escriben las sentencias del pseudocódigo. No es lo mismo escribir “definir sumando1 como Entero” que “definir como Entero sumando1”. Si bien nosotros entendemos claramente que ambas oraciones significan lo mismo, Pselnt nos indicará un error en el segundo caso. Es fundamental tener presente esto todo el tiempo, para evitar errores que de otro modo nos parecerán incomprensibles.

El otro tipo de errores posible, y que **Pselnt no detecta**, son los errores lógicos. Éstos errores no impiden que el algoritmo se ejecute, pero sí que llegue a un resultado correcto. Los errores lógicos sólo pueden detectarse leyendo atentamente el código y buscándolos, ayudados por medio de la “ejecución paso a paso” del menú ejecutar o “Ejecución explicada” del mismo menú. Si no disponemos de Pselnt, la prueba de escritorio sirve para el mismo fin.

Si todo anduvo bien, al presionar f9 deberíamos ver algo similar a la imagen siguiente:



```
PSeInt - Ejecutando proceso SUMADOSNUMEROS_V2

*** Ejecución Iniciada. ***
El total de la suma es: 7
3+4=7
*** Ejecución Finalizada. ***

☐ No cerrar esta ventana ☐ Siempre visible Reiniciar
```

Mejorando nuestro algoritmo

Nuestro primer algoritmo nos ha servido para iniciarnos en la escritura de los mismos y para realizar nuestra primera prueba de escritorio, pero en realidad sólo nos sirve para sumar los números 3 y 4. Para que puedan sumarse otros números habría que cambiar las líneas

```
sumando1 <- 3;
sumando2 <- 4;
```

por otras que asignen otros valores, por ejemplo:

```
sumando1 <- 10;
sumando2 <- 2;
```

Pero esto causaría que tengamos un algoritmo distinto para cada par de números que queramos sumar. Para poder tener un sólo algoritmo que pueda sumar cualquier par de números, tendríamos que encontrar alguna instrucción que nos permita obtener ese número de la persona que ejecuta el algoritmo. Si nos fijamos en el apunte “Sintaxis Pseudocódigo” veremos que en la sección “entrada y salida” hay una instrucción llamada leer:

```
Leer a; // solicita un valor por teclado
```

Leer es una instrucción de entrada: cada vez que nos la encontremos al realizar la prueba de escritorio, deberemos elegir una valor a asignar a la variable que recibe como parámetro, en el caso del ejemplo, a. La instrucción Leer es la contraparte de **Escribir**: así como Escribir nos muestra los resultados del algoritmo, Leer nos permite introducir datos para que los procese.

Vamos a reescribir el algoritmo empleando la instrucción leer:

```

Algoritmo SumaDosNumeros_V2
    //Calcular la suma de los números ingresados por el usuario
    definir sumando1 como Entero;
    definir sumando2 como Entero;
    definir total como Entero;

    Escribir "Ingrese sumando 1:";
    Leer sumando1;

    Escribir "Ingrese sumando 2:";
    Leer sumando2;

    //asignamos en la variable total la suma de sumando1 + sumando2
    total <- sumando1 + sumando2;

    //mostramos un mensaje y el resultado de la suma
    escribir "El total de la suma es: ",total;
    escribir sumando1,"+",sumando2, "=",total;
FinAlgoritmo

```

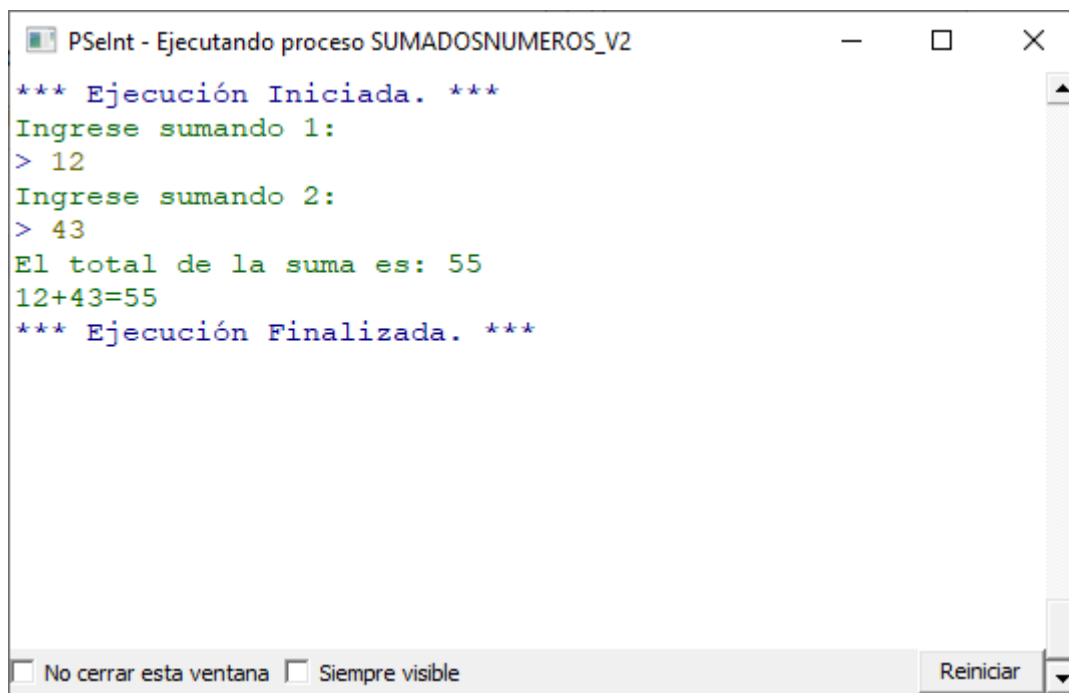
El único cambio que hemos realizado es que las asignaciones de sumando1 y sumando2, en lugar de ser directas como en la primera versión, se realizan por medio de la instrucción leer. Empleamos una instrucción por cada variable a leer, con una sentencia escribir antes en la que indicamos qué dato se debe ingresar. En un algoritmo que se ejecuta en una computadora, cada vez que llegamos a una sentencia leer el programa nos mostraría el mensaje de la misma y quedaría esperando a que nosotros introduzcamos el dato pedido por teclado.

Aplicando las mismas reglas que para el caso anterior, la prueba de escritorio queda como sigue:

Paso	sumando1	sumando2	total	Entrada / Salida	Comentarios
1	-	-	-	<Ingrese Sumando 1:	< indica salida
2	12	-	-	>12	> indica entrada
3	12	-	-	<Ingrese Sumando 2:	
4	12	43	-	>43	
5	12	43	55		total=sumando1+sumando2
6	12	43	55	<El total de la suma es: 55	
7	12	43	55	<12+43=55	

En los pasos 2 y 4 hemos pasado por las sentencias que contienen las instrucciones leer. Los ingresos del usuario se consignan en la columna Entrada / Salida y debe llevar el signo > adelante. **< indica salida y > indica entrada**. Además, como la instrucción leer además asigna el valor ingresado a la variable, indicamos el valor ingresado en la columna de las variables correspondientes, en este caso sumando1 y sumando2. En el paso 2 se ingresa y asigna sumando1 y en el paso 4, sumando2.

Ejecutando nuestro algoritmo con PseInt, deberíamos obtener un resultado similar al de la figura siguiente:



```
*** Ejecución Iniciada. ***
Ingrese sumando 1:
> 12
Ingrese sumando 2:
> 43
El total de la suma es: 55
12+43=55
*** Ejecución Finalizada. ***
```

Buenas prácticas de programación

Es importante que la escritura de un algoritmo sea clara y concisa. Si nos preocupamos por nuestro algoritmo exhiba estas dos características, resultará más sencillo de seguir (más sencilla la prueba de escritorio) y de comprender (entenderemos los que hace con mayor facilidad).

La claridad en un algoritmo se expresa, en general, a través de:

- a) El orden en que realizamos la escritura de las sentencias que lo componen. Si nos fijamos en los ejemplos anteriores, veremos que siguen un patrón semejante:
 - i) Inicio del algoritmo, en el caso de pseudocódigo señalado con la palabra "Algoritmo" y el nombre del mismo.
 - ii) Un comentario explicando qué es lo que hace el algoritmo
 - iii) Declaración de variables (definir...).
 - iv) Asignación y/o lectura inicial (inicialización) de variables (sumando1<-4 ó leer sumando1)
 - v) Las instrucciones del algoritmo propiamente dicho (procesamiento)
 - vi) Salida de los resultados del algoritmo.
 - vii) Fin del algoritmo, señalado con la palabra FinAlgoritmo en el caso de pseudocódigo
- b) Nombres adecuados para el algoritmo y las variables declaradas. Los nombres deben reflejar el cometido de la variable o algoritmo; los nombres que no tengan significado dentro del ámbito del algoritmo nos obligan a un esfuerzo mayor para comprenderlo. Si en lugar de sumando1, sumando2 y total hubiéramos denominado a dichas variables como a, b y c (nombres permitidos en el pseudocódigo), al encontrarnos las variables en el código nos veríamos obligados a recordar cuál era el propósito de la variable para entender el algoritmo. Con tres variables esto no parece un problema, pero con algunas más comienza a complicarse. Es más fácil equivocarse e introducir errores, y luego es más trabajoso encontrarlos.
- c) Cuando resulte necesario, comentarios para aclarar las partes más difíciles de comprender del código. Sólo los comentarios estrictamente necesarios, una gran profusión de los mismos también oscurece el código.

Respetar el orden indicado en el punto a) nos reporta la ventaja de que al encontrarse cada parte del algoritmo en lugares predecibles, es más sencillo encontrar las sentencias que lo componen. Si necesitamos verificar el

valor inicial de una variable, nos referiremos a la sección de inicialización. Si declaráramos o inicializáramos las variables en cualquier parte del código (por ejemplo, inmediatamente antes de utilizarla), esto nos obligaría a recorrerlo desde el principio hasta encontrar la que buscamos. Esto hace que las pruebas de escritorio y la búsqueda de errores resulten más trabajosas.

Nota importante: No alcanza con que el algoritmo “funcione bien”. El algoritmo debe funcionar bien, por supuesto, pero además debe estar escrito de manera ordenada, clara, concisa y de acuerdo a las reglas del arte y las buenas prácticas. Recordemos que nos estamos preparando para una carrera profesional en la que es importante el trabajo en equipo, y adherir a las reglas establecidas de trabajo es una muestra de respeto hacia aquellos que trabajan con nosotros y una demostración de calidad profesional.

Escenarios de ejecución de un algoritmo o programa

La primera consecuencia de escribir un algoritmo al que se le puedan ingresar distintos conjuntos de datos para su ejecución es que existe una gran cantidad de casos para probar el algoritmo. Al algoritmo que acabamos de escribir podemos ejecutarlos con el par de valores 12 y 43 para sumando1 y sumando2 como lo acabamos de hacer, pero perfectamente podríamos haber elegido 3 y 7, 48 y 100, 1000 y 4, etc, etc. A cada uno de los conjuntos de datos elegidos para ejecutar un algoritmo lo denominaremos **escenario**. Así cuando en los problemas de práctica digamos “Realizar la prueba de escritorio (o ejecutar el algoritmo, o probar el algoritmo) para el **escenario** sumando1=4 y sumando2=23” nos estaremos refiriendo al conjunto de datos de entrada que recibirá el algoritmo.

Práctica

1. Diseñar un algoritmo que utilice dos variables una **a** y **b** calcule la suma, resta, multiplicación y división, en la sentencias de salida que tengan este formato ejemplo: 3+4=7. Realizar la prueba de escritorio para a=27 y b=-5.
2. Diseñar un algoritmo que dado el radio de un círculo calcule el área y la longitud de circunferencia. Realizar la prueba de escritorio para r=3.

HELP: $\text{área de un círculo} = \pi \cdot r^2$ $\text{longitud de circunferencia} = 2 \cdot \pi \cdot r$

<pre> Algoritmo AlgValorDePi Definir valorPi Como Real; valorPi <- pi; Escribir valorPi; FinAlgoritmo </pre>	<pre> *** Ejecución Iniciada. *** 3.1415926536 *** Ejecución Finalizada. *** </pre>
---	---

3. Diseñar un algoritmo que dados los catetos de un triángulo rectángulo, calcule la hipotenusa, el área y el perímetro. Realizar la prueba de escritorio para catetoMenor = 3 y catetoMayor=5.
4. Diseñar un algoritmo que lea dos variables lógicas **p** y **q**. Calcule en tres variables **NO p**, **NO q**, **p Y q** y **p O q** y por último muestre los resultados. Realizar la prueba de escritorio para los Escenario 1: p=V, q=V; Escenario 2: p=V, q=F; Escenario 3: p=F, q=V; Escenario 4: p=F, q=F;

HELP:

<pre> Algoritmo NO_p //Definición de variables Definir p como Logico; Definir noP como Logico; //Entrada Escribir "p?"; Leer p; noP <- NO p; Escribir "p =", p; Escribir "NO P =", noP; FinAlgoritmo </pre>	<pre> *** Ejecución Iniciada. *** p? > VERDADERO p =VERDADERO NO P =FALSO *** Ejecución Finalizada. *** </pre>	<pre> *** Ejecución Iniciada. *** p? > FALSO p =FALSO NO P =VERDADERO *** Ejecución Finalizada. *** </pre>
---	---	---

La tablas de verdad

Negación: es el contrario	Conjunción: es V solo si ambas son V	Disyunción: es F solo si ambas son F																																				
<table><tr><td>p</td><td>NO p</td></tr><tr><td>V</td><td>F</td></tr><tr><td>F</td><td>V</td></tr></table>	p	NO p	V	F	F	V	<table><tr><td>p</td><td>q</td><td>p Y q</td></tr><tr><td>V</td><td>V</td><td>V</td></tr><tr><td>V</td><td>F</td><td>F</td></tr><tr><td>F</td><td>V</td><td>F</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table>	p	q	p Y q	V	V	V	V	F	F	F	V	F	F	F	F	<table><tr><td>p</td><td>q</td><td>p O q</td></tr><tr><td>V</td><td>V</td><td>V</td></tr><tr><td>V</td><td>F</td><td>V</td></tr><tr><td>F</td><td>V</td><td>V</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table>	p	q	p O q	V	V	V	V	F	V	F	V	V	F	F	F
p	NO p																																					
V	F																																					
F	V																																					
p	q	p Y q																																				
V	V	V																																				
V	F	F																																				
F	V	F																																				
F	F	F																																				
p	q	p O q																																				
V	V	V																																				
V	F	V																																				
F	V	V																																				
F	F	F																																				

5. Diseñar un algoritmo que dados dos números cualquiera, calcule el resto de la división entre ambos, finalmente imprima los números dados y el resultado. Realizar la prueba de escritorio con los siguientes valores de lectura: dividendo=39 y divisor=11.

HELP:

<pre> Algoritmo FuncionMOD Definir a Como Entero; Definir b Como Entero; Definir valorMod Como Entero; a<-2021; b<-12; valorMod<- a MOD b; Escribir a," MOD ", b, " = ", valorMod; FinAlgoritmo </pre>	<pre> *** Ejecución Iniciada. *** 2021 MOD 12 = 5 *** Ejecución Finalizada. *** </pre>
---	--

6. Diseñar un algoritmo para simular tirar dos dados y sumar las dos caras resultantes. Mostrar los números que salieron y su suma. Realizar la prueba de escritorio suponiendo que el 1° dado arroja un 5 y el segundo un 6.

HELP:

```
Algoritmo TirarUnDado
  Definir dado Como Entero;
  dado<-Aleatorio(1, 6);
  Escribir "Salió un: ", dado;
FinAlgoritmo
```

```
*** Ejecución Iniciada. ***
Salió un: 4
*** Ejecución Finalizada. ***
```

Estructuras de decisión o bifurcación

Vamos a agregar un poco más de funcionalidad a nuestro algoritmo: queremos que nos informe si la suma total obtenida es par o impar, por medio de un mensaje de texto. Así, si el total de la suma es par, el algoritmo deberá mostrar la frase, por ejemplo:

“El total de la suma es 12 y es par.”

y en caso contrario:

“El total de la suma es 9 y es impar.”

Determinar si un número es par es lo mismo que determinar si es divisible por dos. Sabemos que un número es divisible por otro cuando el resto de dicha división es cero. Así $15 / 2 = 7$ con resto 1. Esto implica que 15 no es divisible por dos y por lo tanto, es impar.

Aquí nos encontramos con un problema: ¿Cómo hacemos para realizar esta cuenta que nos parece tan sencilla a mano, en un algoritmo?. No podemos expresar cualquier operación de cualquier manera en un algoritmo, sino que deberemos seguir las reglas especificadas por el lenguaje que estemos utilizando para representarlo, en nuestro caso, pseudocódigo.

En el apunte “Sintaxis Pseudocódigo”, en el apartado “Funciones” se encuentran listados los operadores matemáticos disponibles en el lenguaje. Entre ellos, encontramos el operador MOD, que hace precisamente lo que necesitamos: Devuelve el resto de la división entre dos números.

Ahora sólo necesitamos algún mecanismo que le permita al algoritmo **decidir** qué mensaje corresponde escribir basándose en el resultado de la suma. El lenguaje nos ofrece a tal fin lo que llamamos una estructura de decisión o bifurcación. Decimos que son de **decisión** porque sirven para que el algoritmo pueda decidir entre varias acciones o grupos de acciones a ejecutar. Por otro lado, también decimos que son estructuras de bifurcación porque podemos visualizar al flujo de ejecución del algoritmo encontrándose ante una bifurcación de su camino normal. Qué camino tome dependerá de la condición que se defina en la bifurcación.

Imaginemos una bifurcación en un camino como la siguiente:



En nuestro caso, imaginemos que tomaremos el camino de la derecha si la suma es par, y el de la izquierda en caso contrario. Hacia la derecha se encuentran las acciones a ejecutar en caso de que la suma sea par (mostrar la frase “El total de la suma es... y es par”) y hacia la izquierda las acciones a ejecutar en caso contrario (mostrar la frase “El total de la suma es... y es impar”) Existen estructuras de bifurcación más complejas, que permiten más de dos caminos a seguir, pero no las veremos en este curso.

Vamos a escribir nuestro algoritmo para poder ver todo esto paso a paso:

Algoritmo SumaDosNumeros_V3

```
//Calcular la suma de los números ingresados por el usuario
// e indicar si la suma es par o impar
definir sumando1 como Entero;
definir sumando2 como Entero;
definir total como Entero;
definir resto como Entero;

Escribir "Ingrese sumando 1:";
Leer sumando1;

Escribir "Ingrese sumando 2:";
Leer sumando2;

total <- sumando1 + sumando2;

resto <- total mod 2;

//si resto = 0 significa que es divisible por 2 y por lo tanto, par.
//en caso contrario (resto distinto de 0) no es divisible por 2
//y es impar.
si (resto = 0)
    escribir "El total de la suma es ", total, " y es par";
SiNo
    escribir "El total de la suma es ", total, " y es impar";
FinSi
```

FinAlgoritmo

La primera parte de nuestro algoritmo es similar a [SumaDosNúmeros_V2](#). Sólo se agrega la declaración de una variable numérica más, resto, que servirá para almacenar el resto de la división del total por dos. A continuación pedimos los números a sumar y realizamos la suma. Las verdaderas diferencias comienzan con la sentencia

```
resto <- total mod 2;
```

en la que realizamos la operación MOD entre total y 2. Como ya dijimos, realizará la división entre ambos y en la variable resto quedará almacenado el resto de la división, como su nombre lo indica.

Una vez más insistimos con esto: Los nombres de las variables deben reflejar qué representa el valor que almacenan. Esto redundará en código más prolijo, legible y con menores posibilidades de que contenga algún error.

El grupo de sentencias que sigue representa la bifurcación propiamente dicha:

```
si (resto = 0)
    escribir "El total de la suma es ", total, " y es par";
SiNo
    escribir "El total de la suma es ", total, " y es impar";
FinSi
```

Comencemos por partes:

```
si (resto = 0)
```

la sentencia SI nos indica que nos encontramos ante una bifurcación: "Si ocurre esto, entonces debe hacerse aquello" diríamos en lenguaje coloquial. En nuestro caso, la forma más sencilla de leerla es "si resto es igual a cero, entonces escribir...". El signo igual es el operador de igualdad: Devuelve verdadero cuando los valores o variables a ambos lados del mismo son iguales.

7 = 2 devuelve falso

7 = 7 devuelve verdadero

7 = 3 + 4 devuelve verdadero

Si resto vale 0 cuando llegamos a la bifurcación, entonces resto = 0 devolverá verdadero. Es como si nos preguntáramos "¿Resto es igual a 0? Si nos contestamos que sí (verdadero) ejecutamos las acciones a continuación, en nuestro caso

```
escribir "El total de la suma es ",total, " y es par"
```

Las acciones a ejecutar son las que se encuentran a continuación de la sentencia SI y hasta una sentencia SINO. La sentencia FINSI indica el fin de la bifurcación. La sentencia SINO indica el segundo camino de la bifurcación, el que tomaremos si la respuesta a la pregunta ¿Resto es igual a 0? es no, es decir, falso. En este caso, las sentencias a ejecutar son las que siguen a la sentencia SINO y hasta la sentencia FINSI:

```
escribir "El total de la suma es ",total, " y es impar";
```

Algo para notar: En nuestro caso sólo tenemos una sentencia, pero pueden existir cuantas sean necesarias. Sólo hay que recordar que se ejecutarán si la condición es verdadera aquellas que se encuentren entre la sentencia SI y SINO y las que se encuentran entre las sentencias SINO y FINSI si es falsa.

La pregunta que nos hacemos para decidir qué camino tomar en la bifurcación decimos que es la **condición** de la misma. Así es como diremos que las acciones a ejecutar en una estructura de bifurcación dependerán del valor de verdad de su condición. Una vez terminada la ejecución de cualquiera de los caminos de la bifurcación, la ejecución del algoritmo seguirá con las sentencias que se encuentren a continuación de la instrucción FINSI.

Hagamos dos pruebas de escritorio: Una para el caso en que la suma resulte impar y otro para el caso contrario:

Paso	sumando1	sumando2	total	resto	Entrada / Salida	Comentarios
1	-	-	-	-	<Ingrese Sumando 1:	-
2	10	-	-	-	>10	-
3	10	-	-	-	<Ingrese Sumando 2:	-
4	10	5	-	-	>5	-
5	10	5	15	-	-	total<-sumando1 + sumando2
6	10	5	15	1	-	resto<-total MOD 2
7	10	5	15	1	-	resto=0? -> falso
8	10	5	15		<El total de la suma es: 15 y es impar	-

Con PseInt:

```

*** Ejecución Iniciada. ***
Ingrese sumando 1:
> 10
Ingrese sumando 2:
> 5
El total de la suma es 15 y es impar
*** Ejecución Finalizada. ***
  
```

Con respecto a la versión anterior (V2), se agregaron dos pasos más a la prueba de escritorio: El paso donde calculamos el resto de la división entre 15 y 2, que nos da resultado 1 y lo almacenamos en la variable resto, y la bifurcación propiamente dicha, donde preguntamos si resto vale 0. Como esta condición devuelve falso, sólo ejecutamos la sentencia luego de la cláusula SINO.

Una vez ejecutada la misma, continuamos con la sentencia FINSI (las sentencias SINO y FINSI no generan pasos en la prueba de escritorio), y a continuación de la misma encontramos FIN_ALGORITMO, que ya sabemos que nos indica el fin del algoritmo y tampoco genera un paso en la prueba.

Ahora hagamos el caso en el que la suma sí es par:

Paso	sumando1	sumando2	total	resto	Entrada / Salida	Comentarios
1	-	-	-	-	<Ingrese Sumando 1:	-
2	7	-	-	-	>7	-
3	7	-	-	-	<Ingrese Sumando 2:	-
4	7	3	-	-	>3	-
5	7	3	10	-	-	total<-sumando1 + sumando2
6	7	3	10	0	-	resto<-total MOD 2
7	7	3	10	0		resto=0? -> verdadero
8	7	3	10		<El total de la suma es: 10 y es par	-

Con PseInt:

```

*** Ejecución Iniciada. ***
Ingrese sumando 1:
> 7
Ingrese sumando 2:
> 3
El total de la suma es 10 y es par
*** Ejecución Finalizada. ***

```

☐ No cerrar esta ventana ☐ Siempre visible Reiniciar

Existe otra variante de bifurcación: una que no tiene definido el camino en que la condición es falsa. En este caso, no se emplea la cláusula SINO y sólo se ejecutan las acciones a continuación de las cláusula SI. Si la condición resultara falsa, se continuaría el flujo de ejecución con las instrucciones que aparecen luego de la cláusula FINSI.

Práctica

1. Diseñar un algoritmo para que dado un número **a**, determine si es múltiplo de otro número **b**. Realizar la prueba de escritorio para los escenarios posibles: Escenario 1: que a sea múltiplo de b. Escenario 2: que a NO sea múltiplo de b.
2. Escribir el algoritmo para que dado un año determinar si es bisiesto. Realizar la prueba de escritorio para Escenario 1: Bisiestos para los años: 1996, 2004, 2000, 1600; Escenario 2: NO bisiestos para los años: 1700, 1800, 1900, 2100.

Serán bisiestos los años divisibles por 4, exceptuando los que son divisibles por 100 y **no** divisibles por 400.

3. Escribir un algoritmo, que lea las variables **a** y **b** y ordene el contenido de las mismas en forma ascendente, de manera que cuando imprima **a** y **b** en **a** estará el valor menor, por lo tanto si **a** es mayor que **b** se deberá intercambiar el contenido de las variables. Realizar la prueba de escritorio para: Escenario 1: a=23 y b=37; Escenario 1: a=91 y b=43
4. Diseñar un algoritmo para simular tirar una moneda. Mostrar CARA o CECA. Realizar la prueba de escritorio para los dos escenarios posibles.

HELP:

```

Algoritmo AleatorioMoneda
  Definir moneda Como Entero;
  moneda <- Aleatorio(0,1);
  Escribir "Salió--> ", moneda;
FinAlgoritmo

```

```

*** Ejecución Iniciada. ***
Salió--> 0
*** Ejecución Finalizada. ***

```

5. Diseñar un algoritmo para simular tirar un dado. Mostrar los mensajes PAR o IMPAR según la cara obtenida. Realizar la prueba de escritorio para los seis escenarios posibles.

6. Diseñar un algoritmo dado un número (**a**) determine **si es múltiplo de 3 y de 5**. Realizar la prueba de escritorio para: Escenario 1: a=25; Escenario 2: a=27; Escenario 3: a=43 Escenario 4: a=105
7. Diseñar un algoritmo dado un número (**a**) determine **si es múltiplo de 2 pero no de 5**. Realizar la prueba de escritorio para: Escenario 1: a=15; Escenario 2: a=30; Escenario 3: a=17 Escenario 4: a=34
8. Diseñar un algoritmo dado un número (**a**) determine **si es múltiplo de 3 o de 7**. Realizar la prueba de escritorio para: Escenario 1: a=27; Escenario 2: a=28; Escenario 3: a=31 Escenario 4: a=84
9. Dados **a, b y c** escribir el algoritmo que imprima **a, b y c**, teniendo en cuenta previamente ordenar en forma ascendente el contenido de las variables.
10. Escribir el algoritmo para emitir un pre-ticket dado el **precio** de un artículo y la cantidad de **unidades**. Si la cantidad de artículos supera las 4 unidades se aplicará el 5% de descuento.

Completar la prueba de escritorio :

Esc.	precio	unidades	Entrada / Salida	(unidade>4)	subTotal	descuento	total	comentario
1	25	6	<precio? >25 <unidades? >6	V				subTotal <-precio*unidad descuento<-subTotal *0.05 total<-subTotal-descuento
2	32	3						

11. Escribir el algoritmo para emitir un pre-ticket dado el precio de un artículo y la cantidad de unidades. Calcular el descuento (3X2) si cada tres unidades de compra tendrá un descuento del importe de una unidad; por ejemplo: Si compra 7 latas de tomates a \$ 30.- el subtotal es \$210.- el Descuento (3x2) es \$60 y el total pre-ticket: \$150.-

Completar la prueba de escritorio :

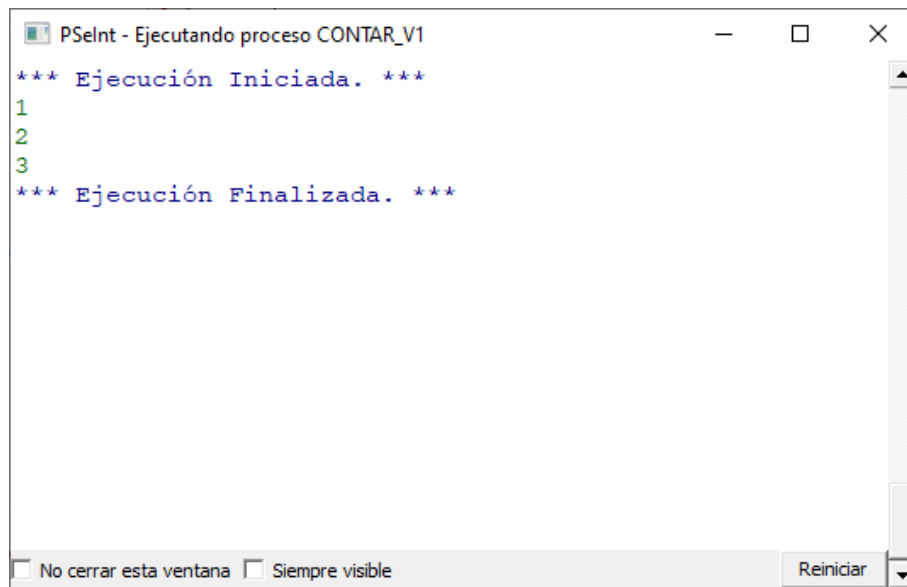
Esc.	precio	unidades	Entrada / Salida	unGratis	subTotal	descuento	total	comentario
1	30	6	<precio? >30 <unidades? >6					subTotal<-precio*unidad unGratis<-TRUNC(unidades/3) descuento<-unGratis *precio total<-subTotal-descuento
2	30	11	<precio? >30 <unidades? >11					

Estructuras de iteración (Bucles)

Las estructuras de iteración nos permiten ejecutar una acción más de una vez; el verbo iterar significa, precisamente, repetir. Si quisiéramos escribir un algoritmo para contar de 1 a 3, podría ocurrirnos escribirlo de la siguiente manera:

```
Algoritmo Contar_V1
  Escribir 1;
  Escribir 2;
  Escribir 3;
FinAlgoritmo
```

¿Funciona? Si funciona:



¿Está bien? No parece estar bien. Tiene un problema no menor: el algoritmo no es escalable. ¿Qué quiere decir escalable? que no es simple modificarlo para que haga lo mismo pero ampliando los límites. En nuestro caso, ¿Qué deberíamos hacer para que el algoritmo cuente hasta 1000? Con lo que sabemos hasta ahora nos veríamos obligados a escribir, al estilo de Bart Simpson castigado en el pizarrón, 1000 sentencias Escribir, una para cada número del 1 al 1000. Debe existir una manera más simple y menos trabajosa de resolver el problema.

Nota: En general, cuando notamos que en un algoritmo repetimos varias veces las misma sentencias, es señal de que muy posiblemente nuestro algoritmo no esté bien diseñado. Seguramente la repetición podrá eliminarse empleando una estructura de iteración, agregando variables o por medio de otras técnicas que no veremos en este curso.

Estructura Para

La estructura Para nos permite indicar al algoritmo que hay una sentencia o un grupo de sentencias que queremos ejecutar un número determinado de veces. A tal fin mantiene una variable contadora que es la que contiene un valor numérico que indica en qué iteración nos encontramos, y nos permite definir el valor de inicio de la cuenta, el valor final (hasta cuánto contamos) y de a cuánto contamos (de a 1, de a 2, etc). Vamos a emplearla en nuestro algoritmo contador para ver cómo funciona:

```

Algoritmo Contar_V2
    // algoritmo para demostrar estructura Para.
    // cuenta desde 1 hasta el número provisto por el usuario

    definir contarHasta como Entero; // hasta cuánto contamos
    definir contador como Entero; // Contador del ciclo Para

    Escribir "Hola! Hasta cuánto contamos?";
    Leer contarHasta;

    Para contador<-1 hasta contarHasta con paso 1
        Escribir contador;
    FinPara

FinAlgoritmo

```

Las partes de la sentencia **Para** son las siguientes: luego de la palabra reservada **Para** tenemos una asignación: contador<-1. Por medio de la misma se indica el valor inicial de la variable contador (en este caso la llamamos contador, pero puede tener cualquier otro nombre de variable legal) que será desde donde comenzará a contar. A continuación sigue la palabra reservada **Hasta** que sirve para indicar el límite superior de la iteración (hasta cuánto contamos). Seguidamente, encontramos la expresión **con** **paso** que indica de a cuánto se incrementa el contador del bucle. Esta sentencia puede omitirse cuando el paso es 1, como en nuestro caso. No la hemos omitido para poder demostrar la sintaxis de la estructura **Para** completa. Las sentencias que siguen a continuación y hasta la sentencia **FinPara** deberán ejecutarse en cada iteración del ciclo.

La estructura **Para** funciona de la siguiente manera: Cuando nos la encontramos, lo que debemos hacer es inicializar la variable que empleemos como contador con el valor indicado (en nuestro caso 1). Continuamos con la sentencia siguiente, en nuestro caso **Escribir** contador. una vez ejecutada, seguimos con la siguiente sentencia que en este caso es **FinPara**. Al encontrar **FinPara** lo que debemos hacer es volver a la sentencia **Para**, es decir, “saltamos” nuevamente al inicio de la estructura **Para** (decimos también al inicio del “bucle”; se denomina así porque podemos imaginarnos la ejecución como un rulo que recorremos en cada iteración) y a partir de allí, cada vez que volvamos a la sentencia **Para** incrementaremos la variable contadora en el valor indicado luego de **con** **paso**, en este ejemplo, 1. Verificamos que no hayamos alcanzado el límite de la iteración y si no es así, ingresamos nuevamente, ejecutamos **Escribir** contador y llegamos una vez más a **Fin Para**. Saltamos nuevamente a **Para**, incrementamos y verificamos y continuamos iterando hasta que contador>contarHasta. Cuando esto último ocurre, salimos del bucle **Para** y continuamos con las sentencias que se encuentran luego de **FinPara**.

Hagamos la prueba de escritorio con un escenario de contarHasta=3 para ver cómo funciona:

Paso	contarHasta	contador	Entrada / Salida	Comentarios
	-	-	<"Hola! Hasta cuánto contamos?"	
	3	-	>3	Leer contarHasta
	3	1		Sentencia Para. Inicializamos contador<-1
	3	1	<1	Escribir contador
	3	1		FinPara. Volvemos a Para
	3	2		Incrementamos contador (contador<-contador+1).Como contador<=contarHasta, seguimos en el bucle
	3	2	<2	Escribir contador
	3	2		FinPara. Volvemos a Para
	3	3		Incrementamos contador

				(contador<-contador+1).Como contador<=contarHasta, seguimos en el bucle
	3	3	<3	Escribir contador
	3	3		FinPara. Volvemos a Para
	3	4		Incrementamos contador (contador<-contador+1).Como contador>contarHasta, salimos del el bucle y continuamos con las sentencias luego de FinPara

Vemos que cada vez que pasamos por la sentencia Para realizamos dos tareas: incrementar y verificar si llegamos al final del ciclo (ciclo, bucle, loop o iteración se emplean como sinónimos). Si no llegamos al final, volvemos a ejecutar las sentencias que haya entre Para y FinPara. Cuando deje de cumplirse la condición de iteración (en este caso, contador<=contarHasta), seguiremos con las sentencias a continuación de FinPara.

Ahora vamos a ver un ejemplo con varias acciones dentro del ciclo Para: Vamos a contar, pero esta vez sólo los números pares o múltiplos de 7 y , además de contar, vamos a pedir que el algoritmo nos devuelva la suma de todos los números contados.

Ya sabemos que un número es múltiplo de otro cuando al dividirlo por el mismo da resto 0. (Ver SumaDosNumeros_V3). El algoritmo involucra una estructura de iteración y dos bifurcaciones.

Algoritmo Contar_V3

```
// algoritmo para demostrar estructura Para.
// cuenta desde 1 hasta el número provisto por el usuario
// sólo los pares o múltiplos de 7

definir contarHasta como Entero; // hasta cuánto contamos
definir contador como Entero; // Contador del ciclo Para
definir resto como Entero;
definir acumulador como Entero;

//inicializamos
acumulador<-0;

Escribir "Hola! Hasta cuánto contamos pares y múltiplos de 7?";
Leer contarHasta;

Para contador<-1 hasta contarHasta
    // verificamos si es múltiplo de 2 (par)
    resto<-contador mod 2;
    Si resto=0
        Escribir contador;
        acumulador<-acumulador+contador;
    SiNo
        // Es impar. Verificamos si es múltiplo de 7
        resto<-contador mod 7;
        Si resto=0 // esta estructura Si no tiene cláusula SiNo
            escribir contador;
            acumulador<-acumulador+contador;
        FinSi
    FinSi
FinPara

Escribir "La suma de los números contados es ",acumulador;

FinAlgoritmo
```


La rama verdadera de la primera bifurcación (la que verifica si son pares) sólo se limita a escribir el valor del contador. En la rama falsa se verifica si se trata de un múltiplo impar de 7 (los múltiplos pares de 7 ya se escribieron en la rama verdadera) y si es así, lo escribe. En ambas ramas verdaderas (múltiplos de 2 y de 7) se procede a sumar el valor contado a la variable acumulador (los acumulamos). De este modo obtenemos la suma de todos los números escritos, para mostrarla finalmente luego de la sentencia FinPara (al terminar de iterar).

Este patrón de iterar y acumular se utiliza a menudo. El acumulador puede ser, como en este caso de suma (donde acumulamos por medio de la suma, como en este caso) o de producto (donde se acumula multiplicando). En el primer caso, el acumulador se inicializa en 0 ya que el 0 es el elemento neutro para la suma (si sumamos 0 a cualquier número obtenemos el mismo número) y en el segundo es 1, porque éste es el elemento neutro para el producto.

Práctica

1. Realizar la prueba de escritorio para el algoritmo Contar_V3 con un escenario en el que contarHasta=15.
2. Realizar un algoritmo que reciba un número base y otro exponente y devuelva base a la potencia de exponente sin emplear la función de elevación a potencia del pseudocódigo (a^b). Realizar la prueba de escritorio. Ayuda: hay un acumulador de producto involucrado.
3. Realizar un algoritmo que pida dos años y devuelva los años bisiestos entre los mismos. Realizar la prueba de escritorio para un escenario entre los años 2020 y 2035.
4. El factorial de un número natural n , denotado como $n!$, es el producto de ese número por todos sus antecesores. es decir, el factorial de 4, $4!=4*3*2*1=24$. Escribir un algoritmo que pida el valor de n y nos devuelva $n!$. Además, por definición, $!0=1$. Realizar la prueba de escritorio con $n=5$. Ayuda: $4*3*2*1=1*2*3*4$. Hay un acumulador de producto involucrado.
5. Realizar un algoritmo para tirar n veces un dado, obtener la cantidad de veces que salió una cara con número par y cuantas con número impar.
6. Realizar un algoritmo para simular tirar una moneda n veces y luego calcular la probabilidad salga una cara al arrojar una moneda con la siguiente fórmula: $P(cara) = \frac{cantCara}{n}$ y la probabilidad salga una seca $P(seca) = \frac{cantSeca}{n}$.
7. Escribir en pseudocódigo el algoritmo para generar la sucesión de Fibonacci hasta n primeros términos. Realizar la prueba de escritorio para $n=10$.

Sucesión de Fibonacci

TECtv La Señal de la Ciencia - Grandes temas de la matemática: Capítulo 4: Fibonacci. Dr Adrián Paenza <https://www.youtube.com/watch?v=0d4o57l3rn4>

La sucesión de Fibonacci 1, 1, 2, 3, 5, 8, 13, 21, 34, ... se define por la relación de recurrencia $c=a+b$, con condiciones iniciales $a=1$ y $b=1$.

Estructura Mientras

A veces ocurre que el algoritmo indica que hay que realizar una tarea hasta que o mientras se cumpla una condición determinada. Un ejemplo del primer caso, siguiendo con nuestro algoritmo para cocinar una torta de chocolate, sería la instrucción “Calentar el horno hasta los 180 grados”. ¿Cómo le explicamos a alguien que no sabe nada de cocina, pero que sí sabe leer el medidor de temperatura y darse cuenta de cuándo llegó a 180 grados cómo debe hacer para llevar adelante esta acción?

Nota aparte: Es necesario recordar que las máquinas para las que escribimos nuestros algoritmos no entienden nada más allá de las instrucciones existentes en el pseudocódigo o lenguaje de programación. Sólo de esta forma podremos descomponer una acción que a nosotros nos resulta intuitiva, natural y sobre todo muy sencilla, en una secuencia de pasos lo suficientemente detallada para que la máquina pueda llevarla a cabo.

Podríamos explicar en lenguaje natural cómo se controla la temperatura del horno paso por paso de la siguiente manera:

Encender el horno.

Leer la temperatura.

Mientras la temperatura sea menor a 180° esperar 5 minutos y leer la temperatura nuevamente.

Finalmente, avisar que el horno se encuentra a 180°

La parte que nos interesa se encuentra en la sentencia mientras. Lo que estamos diciendo es que las acciones esperar 5 minutos y mirar la temperatura deberán repetirse **mientras** la temperatura sea menor que 180 grados.

Este tipo de estructuras, en las que una serie de acciones se repiten, se denominan de iteración. En este caso, decimos que el algoritmo itera mientras la temperatura sea menor que 180°. Que la temperatura sea mayor o igual a 180° es la condición de salida o terminación de la iteración.

Para que quede un poco más claro, vamos a escribirlo de la siguiente manera:

Encender el horno.

Leer la temperatura.

Mientras la temperatura sea menor a 180°

 Esperar 5 minutos

 Leer la temperatura.

FinMientras

Avisar que el horno se encuentra a 180°

Esta forma más formal de escribirlo permite explicar mejor cómo funciona la estructura: todas aquellas acciones que se encuentren entre las sentencias **Mientras** y **FinMientras**, se repetirán MIENTRAS la condición (en este caso que la temperatura sea menor a 180°) sea VERDADERA. Por ejemplo, encendemos el horno y vemos que la temperatura es 30°. Como es menor que 180°, “entramos” en el bucle. Lo siguiente que hacemos es esperar cinco minutos, y al cabo de ese plazo volvemos a leer la temperatura, digamos que su valor es 70°. La sentencia siguiente es **FinMientras**, y al alcanzarla tenemos que volver a la sentencia **Mientras**; es decir, volvemos al principio del bucle. Dijimos que la temperatura ahora es de 70°, y como es menor que 180°, volvemos a ejecutar las acciones esperar 5 minutos y leer la temperatura. Ahora vemos que es de 120°, iteramos nuevamente, y como $120^\circ < 180^\circ$, ejecutamos esperar 5 minutos y leemos la temperatura nuevamente. Esta vez es de 185°. Volvemos a la sentencia **Mientras** y como $185^\circ > 180^\circ$, “salimos” del bucle; esto significa que continuamos la ejecución con la sentencia que se encuentra luego del **FinMientras**, es decir, “Avisar que el horno...”

Ahora hagamos un ejemplo que podamos implementar en pseudocódigo. Vamos a realizar un juego sencillo, en el que la máquina “piense” un número de 1 a 100 y nosotros debamos adivinarlo.

```

Algoritmo AdivinaElNumero_V1
    // Implementación del juego "adivina el número"
    // La máquina elige un número de 1 a 100
    // el usuario elige un número y lo ingresa
    // la máquina informa si el número ingresado es el elegido o
    // si es mayor o menor que el mismo
    // el proceso se repite hasta que se haya adivinado el número

    definir numeroSecreto como Entero;
    definir numeroUsuario como Entero;

    Escribir "Adiviná el número que estoy pensando...";

    // definimos el numero secreto
    numeroSecreto <- Aleatorio(1,100);

    Escribir "Qué número te parece que es?";
    Leer numeroUsuario;

    Mientras numeroUsuario != numeroSecreto
        Si numeroUsuario > numeroSecreto
            escribir numeroUsuario, " es mayor que el número que pensé";
        SiNo
            escribir numeroUsuario, " es menor que el número que pensé";
        FinSi

        Escribir "Probá de nuevo!";
        Leer numeroUsuario;
    FinMientras

    escribir "Adivinaste! El número que pensé es ",numeroSecreto;

FinAlgoritmo

```

La sentencia **Aleatorio(1,100)** nos devuelve un número al azar en el intervalo que le pasemos como parámetros, en nuestro caso los números 1 y 100. Cada vez que se ejecute **Aleatorio** nos devolverá un número distinto.

Lo importante para recordar aquí es que todo lo que haya entre **Mientras** y **FinMientras** se ejecutará cada vez que la condición `numeroUsuario != numeroSecreto` sea verdadera. `!=` es el operador de desigualdad del pseudocódigo y devuelve VERDADERO cuando los números que se comparan son distintos. Cada vez que la ejecución del algoritmo alcanza la sentencia **FinMientras**, vuelve ("salta") a la sentencia **Mientras** y vuelve a preguntar el valor de verdad de la condición. Se puede pensar como una bifurcación **Si** que se ejecuta repetidamente, y del que sólo salimos cuando la condición es falsa.

Vamos a realizar la prueba de escritorio de nuestro algoritmo:

Paso	numeroSecreto	numeroUsuario	Entrada / Salida	Comentarios
1	-	-	<Adiviná el número que estoy pensando...	
2	50	-	-	numeroSecreto <- Aleatorio
3	50	-	<Qué número te parece que es?	
4	50	10	>10	

5	50	10	-	10!=50 es VERDADERO
6	50	10	-	10>50 es FALSO
7	50	10	<10 Es menor que...	
8	50	10	<Probá de nuevo!	
9	50	70	>70	
10	50	70	-	vuelve a cláusula mientras
11	50	70	-	70!=50 es VERDADERO
12	50	70		70>50 es VERDADERO
13	50	70	<70 es mayor que...	
14	50	70	<Probá de nuevo!	
15	50	50	>50	
16	50	50	-	vuelve a cláusula mientras
17	50	50	-	50!=50 es FALSO. Sale de mientras, sigue después de FinMientras
18	50	50	<Adivinaste! El número que pensé es 50	

Con PseInt:

```

*** Ejecución Iniciada. ***
Adiviná el número que estoy pensando...
Qué número te parece que es?
> 10
10 es menor que el número que pensé
Probá de nuevo!
> 70
70 es mayor que el número que pensé
Probá de nuevo!
> 50
Adivinaste! El número que pensé es 50
*** Ejecución Finalizada. ***

```

Mientras predicado sea verdadero (simple o compuesto)

Si bien nuestro algoritmo funciona bien, tiene la desventaja de que hasta que no adivinemos el número, nos va a dejar seguir jugando. A fin de hacerlo más difícil, vamos a ponerle un límite de 10 intentos. Hasta ahora definimos el ciclo **Mientras** para que itere siempre que el número del usuario sea distinto del número secreto. Necesitamos además decirle que deje de iterar cuando hayamos realizado 10 intentos. Es decir, la condición es "Iterar mientras el número del usuario sea distinto al secreto **Y** la cantidad de intentos sea menor que 10"

Podemos ver que estamos componiendo dos proposiciones en una:

- 1) el número del usuario sea distinto al secreto
- 2) la cantidad de intentos sea menor que 10

Lo que une ambas proposiciones es la conjunción **Y**. **Y** es también un operador lógico que devuelve verdadero siempre que las dos proposiciones que la componen sean verdaderas. Si ambas son falsas, el operador **Y** devuelve FALSO. Si una es verdadera y la otra falsa, **Y** devuelve FALSO. Sólo devolverá VERDADERO cuando

ambas sean verdaderas. El operador lógico **Y** también es conocido por su nombre en inglés, AND. En PseInt, el operador lógico es **Y**.

Ahora sólo nos queda crear la segunda condición para poder usar **Y**. Veamos cómo quedaría el código de nuestro algoritmo:

```

Algoritmo AdivinaElNumero_V2
    // Implementación del juego "adivina el número"
    // La máquina elige un número de 1 a 100
    // el usuario elige un número y lo ingresa
    // la máquina informa si el número ingresado es el elegido o si es mayor o menor
    // El proceso se repite hasta que se haya adivinado el número
    // o se se haya alcanzado el máximo de intentos
    definir numeroSecreto como Entero;
    definir numeroUsuario como Entero;
    definir intentos como Entero;
    definir maximoIntentos como Entero; // la mayor cantidad de intentos

    maximoIntentos<-10;

    Escribir "Adiviná el número que estoy pensando en ",maximoIntentos," intentos o menos...";

    // definimos el numero secreto
    numeroSecreto <- Aleatorio(1,100);

    Escribir "Qué número te parece que es?";
    Leer numeroUsuario;

    // inicializamos la cuenta de intentos
    intentos<-1;

    Mientras numeroUsuario != numeroSecreto Y intentos < maximoIntentos
        intentos<-intentos+1;

        Si numeroUsuario > numeroSecreto
            escribir numeroUsuario, " es mayor que el número que pensé.";
        SiNo
            escribir numeroUsuario, " es menor que el número que pensé.";
        FinSi

        Escribir "Probá de nuevo!";
        Leer numeroUsuario;

    FinMientras

    si intentos = maximoIntentos
        escribir "Perdiste! no adivinaste en ",maximoIntentos," intentos o menos";
    sino
        escribir "Adivinaste en ",intentos," intentos!";
    FinSi

    escribir "El número que pensé es ",numeroSecreto;

FinAlgoritmo

```

A fin de poder llevar la cuenta de los intentos realizados, definimos una nueva variable Entero donde iremos acumulándola. En cada iteración del ciclo **Mientras**, sumaremos 1 a su valor. Así, en la primera iteración valdrá 1, en la segunda 2, etc. Cuando su valor llegue a 10, la proposición `intentos < maximoIntentos` del ciclo mientras será falsa y por lo tanto toda la condición (está compuesta con Y, y por lo tanto, ambas proposiciones deben ser verdaderas para que la compuesta también lo sea), provocando que la ejecución continúe luego de la sentencia **FinMientras**.

La prueba de escritorio de este algoritmo queda como ejercicio para los estudiantes. Sólo plantearemos el encabezado de la misma:

Paso	numeroSecreto	numeroUsuario	maximoIntentos	intentos	Entrada / Salida	Comentarios
------	---------------	---------------	----------------	----------	------------------	-------------

Práctica

1. Realizar la prueba de escritorio para el algoritmo AdivinaElNumero_V2 con un escenario en el que numeroSecreto=26 y no se adivina el número.
2. Diseñar un algoritmo que nos pida un número del 1 al 20 y luego comience a escribir números aleatorios hasta que salga el número ingresado, en cuyo caso deberá detenerse e informar cuántos números se escribieron. Hacer la prueba de escritorio para el escenario en que el número buscado sale en el quinto lugar.
3. Realizar un algoritmo que pida dos números, dividiendo y divisor, y que realice la división por el método de restas sucesivas. El algoritmo deberá informar el cociente y el resto de la división. El método de restas sucesivas se basa en restar al dividendo el divisor de manera sucesiva, hasta que el resultado sea menor que el divisor, en cuyo caso la división termina. El cociente es la cantidad de restas realizadas y el último resultado es el resto de la división. Ejemplo: Para dividendo=7 y divisor=2
 $7 - 2 = 5$
 $5 - 2 = 3$
 $3 - 2 = 1 \Rightarrow$ cociente=3 (tres restas) y resto=1 (último resultado).
 Realizar la prueba de escritorio para el escenario dividendo=22 y divisor=7
4. Diseñar un algoritmo para determinar si un número n es primo. Un número es primo cuando sólo es divisible por 1 y por sí mismo. Para determinar si un número n es primo, basta con verificar que no es divisible por ningún número que se encuentre entre 2 y \sqrt{n} . El algoritmo deberá pedir el número n y detenerse en cuanto verifique que n no es primo e informarlo con la frase "El número 24 es compuesto". Si no se verifica que n es compuesto, deberá informarse "El número 11 es primo", por ejemplo. Realizar la prueba de escritorio para: Escenario 1: n=82; Escenario 2: n=17.

¿Cuáles son los números primos?

Audiovisual de Doctor en Matemática Adrián Paenza:

<https://www.educ.ar/recursos/50654/capitulo-1-numeros-primos>

5. Dados a y b números enteros positivos escribir el algoritmo para calcular el MCD(a;b).

HELP

Algoritmo Euclidiano para calcular el MCD(a, b)

Se calcula el **resto** entre **a** y **b**, luego mientras que **r** sea distinto de cero asigna a la variable **a** el valor de **b**, la variable **b** el valor del resto y se vuelve **a** calcular **r**. Cuando la variable **r** es cero **b** es el MCD.

Prueba de escritorio para MCD (105 ; 30)

a	b	r	Entrada/Salida	(r!=0)	comentarios
105	30	15	<a? >105 <b? >30		r<- a MOD b
30	15	0		V	a<-b b<-r r<- a MOD b
			<MCD es 15	F	

ENLACES DE VIDEOS

TEMA	ENLACE
Introducción: ¿Qué es un algoritmo? ¿Qué es un dato? ¿Un algoritmo es un programa?	https://youtu.be/KY04cDU6fn0
Pseudocódigo: ¿Qué es? Pseint Configuración del perfil Errores de Sintaxis / Lógica	https://youtu.be/yobpGDbqn6Y
El primer algoritmo: Explicación SumaDosNumeros_V2	https://youtu.be/PWZq4WrmTYs
La primer prueba de escritorio: Explicación SumaDosNumeros_V2	https://youtu.be/0rMwgvJMIQo
Escenarios de Ejecución: Ingreso de valores por teclado ¿Qué son los escenarios?	https://youtu.be/7b4nHWKNQNI
Buenas prácticas de programación	https://youtu.be/TyJSuVAI4W0
Estructuras de decisión o bifurcación ¿Qué significa? Explicación SumaDosNumeros_V3	https://youtu.be/1pvPRnJxQO8
Estructuras de decisión o bifurcación Prueba de Escritorio: Par e Impar	https://youtu.be/d9yfSkg4JMA
Estructuras de iteración (Para) ¿Qué significa? Explicacion Contar_V2	https://youtu.be/1tLTsx5ZG3g
Estructuras de iteración (Para) Prueba de Escritorio	https://youtu.be/O9P91v3xfxc
Combinación de estructuras (PARA / SI) Explicación Contar_V3	https://youtu.be/ka7vcVZDcm0
Estructuras de iteración (Mientras) ¿Qué significa? Explicacion AdivinaElNumero_V1	https://youtu.be/-BhSYVknhDo
Estructuras de iteración (Mientras) Prueba de Escritorio	https://youtu.be/C6wMVNj0rIU
Mientras predicado sea verdadero ¿Qué significa? Simple o compuesto Explicacion AdivinaElNumero_V2	https://youtu.be/WQx85u28HGg
Mientras predicado sea verdadero Prueba de Escritorio	https://youtu.be/GZbBMQAex4o