Resumen Curso Introductorio

Objetivo

El participante aprende a plantear soluciones lógicas a distintos problemas mediante la creación de programas.

Objetivos Específicos

1. El participante aprende a obtener, eficientemente y por medios propios, ayuda para la resolución de problemas.
2. El participante obtiene las bases prácticas de la programación estructurada.

Las tres formas de aprender en {Hack}

1. Atendiendo a lo que dicen los mentores
2. Buscando en internet para ampliar lo visto en clase (autodidacta)
3. Preguntando las dudas a los mentores (luego de investigar, probar y armar el caso)

Valores clave para el éxito a partir de {Hack}

- **Comunicación**, con los compañeros, con la administración de {Hack} y, en especial, con los mentores. Es la clave para un mayor aprendizaje y aprovechamiento de los conocimientos de los mentores.
- **Ser autodidacta**, tener el espíritu y el entusiasmo para explorar y profundizar en el aprendizaje de las tecnologías, y no limitarse a lo aprendido en clases.

Desarrollar estos valores permitirán a los cursantes tener éxito, no solo en {Hack}, sino en todas las actividades emprendidas luego del curso, especialmente, una vez dentro del mercado laboral.

Índice

- Objetivo
- Objetivos Específicos
- Las tres formas de aprender en {Hack}
- Valores clave para el éxito a partir de {Hack}
- Índice
- Algoritmos y Programación
  - Algoritmo
  - Algunas herramientas para la representación de algoritmos
  - Análisis del problema y diseño de algoritmos
  - Pseudocódigo
  - Corridas en frío - Verificación de algoritmos
  - Programación
  - Programa
  - Lenguaje de programación
  - Tipos de datos
  - Tipos predefinidos
  - Operaciones con datos
  - Operadores aritméticos
  - Operadores relacionales
  - Operadores lógicos
  - Expresiones
  - Jerarquía o Precedencia de operadores
  - Asociación de operadores
  - Sentencias
  - Sintaxis
  - Palabras reservadas
  - Variables
  - Reglas básicas para la asignación de nombres a variables
  - Asignación de un dato a una variable
  - Conversión de datos (type casting)
    - 1.- Conversión implícita
    - 2.- Conversión explícita
  - Mostrar datos al usuario por consola (Ruby)
  - Pedir datos al usuario por consola (Ruby)
  - Comentarios
    - Comentario de solo una línea
    - Comentario de varias líneas
- Teorema del programa estructurado
  - Las 3 estructuras de control de la programación estructurada
- Secuencia
- Estructuras de selección
  - Condicional - If… then
  - Condicional con opción por defecto - If… then… else

- Condicional anidado - If… then… elsif...else
- Case

Estructuras iterativas
- Iterador While
- Iterador Repeat
- Iterador For

Arreglos
- Declarar arreglos
- Acceder a valores de arreglos
- Modificar los valores de un arreglo
- Agregar nuevos valores a un arreglo
- Obtener y retirar el último valor de un arreglo
- Obtener y retirar el primer valor de un arreglo
- Saber si un arreglo está vacío
- Saber la cantidad de elementos de un arreglo (longitud)

Arreglos Bidimensionales (matrices)
- Declarar una matriz
- Llenar automáticamente una matriz nueva
- Acceder a valores de una matriz
- Modificar valores de una matriz
- Mostrar todos los valores de una matriz

Procedimientos y Funciones
- Definir una función

Recursividad
- Funciones recursivas
- Función Sumatoria recursiva
- Función Factorial recursiva
- Sucesión de Fibonacci

Archivos
- Crear un nuevo archivo
- Abrir un archivo existente
- Leer de un archivo
- Escribir en un archivo
- Borrar y renombrar un archivo

Algoritmos y Programación

Algoritmo

Es un conjunto de instrucciones o reglas bien definidas, claras, ordenadas y precisas que permite realizar una actividad. Ej. sacar dinero de un cajero, calcular el factorial de un número, hacer una torta, etc.

Los algoritmos,

- **Se usan para resolver problemas** -> solucionan algo
- **No son ambiguos** -> siempre tienen el mismo comportamiento
- **Son una abstracción** -> hay que abstraerse; es un proceso creativo

Algunas herramientas para la representación de algoritmos

- Lenguaje natural
- Diagramas de flujo (representación gráfica)
- Pseudocódigo (mezcla informal entre lenguaje natural y lenguaje de programación)
- Lenguajes de programación

Análisis del problema y diseño de algoritmos

1. Leer detalladamente el planteamiento y entender el problema (varias veces si es necesario) -> **Qué se pide**
2. Plantear la solución con lógica/matemática (la obvia) -> **Qué hace**
3. Identificar las partes (primera abstracción) -> **Qué recibe, qué devuelve y cómo obtenerlo**
4. Escribir el algoritmo estructurado (abstracción más detallada) -> **Usando la programación estructurada. El papel es muy buena idea**
5. Verificar el algoritmo -> **Hacer corridas en frío con casos borde de prueba**
6. Implementación - **Escribir el algoritmo en un lenguaje de programación y corregir**

Pseudocódigo

El objetivo del pseudocódigo es permitir que el programador se centre en lo aspectos lógicos de la solución, evitando las reglas de sintaxis de un lenguaje de programación.

Corridas en frío - Verificación de algoritmos

Una vez que se ha terminado de escribir un algoritmo, es necesario comprobar que realiza las tareas para las que se ha diseñado y produce el resultado correcto y esperado.

El modo de comprobar un algoritmo es mediante su ejecución manual, anotando en una hoja de papel las modificaciones que se producen en las variables más significativas durante su ejecución. Esta prueba manual es conocida como **corrida en frío.**

Programación

Es la escritura de instrucciones correctas para que sean interpretadas por una computadora.

Programa

Un programa de computadora es un conjunto de instrucciones que se introducen en la máquina y se utilizan para conseguir que la computadora produzca un resultado específico.

Lenguaje de programación

Son lenguajes formales que **permiten escribir programas** informáticos mediante **códigos fuentes**. Estos códigos, una vez escritos, serán traducidos a instrucciones que el computador puede realizar (lenguaje máquina), mediante un **traductor de lenguaje**, lo que permitirá la ejecución del programa en dicho computador.

Los lenguajes de programación ofrecen una serie de elementos que pueden ser usados para la escritura de programas. Algunos de estos elementos son:

- Símbolos
    - Operadores, etc.
- Reglas sintácticas
    - Sintaxis
    - Palabras reservadas
- Reglas semánticas
    - Tipos de datos
    - Expresiones
    - Estructuras de control

La creación de un programa se planifica inicialmente en papel, mediante la creación de **algoritmos**, que son un **conjunto de instrucciones** o reglas bien definidas, claras, ordenadas y precisas que permite **realizar una actividad**. Luego, estos algoritmos se introducen a la computadora mediante su escritura en código fuente en algún lenguaje de programación.

Tipos de datos

Es la propiedad de los datos que indica al computador y/o al programador, qué clase de datos se está manipulando. En otras palabras, son un conjunto de valores y las operaciones sobre estos.

Tipos predefinidos

- Entero
- Real
- Boolean (Lógico)
- Cadenas de texto o String
- Caracter

Operaciones con datos

Los operadores son símbolos que indican que debe realizarse una operación entre dos (2) valores (datos). Existen tres (3) tipos básicos de operadores aplicables a tipos de datos de diversa naturaleza:

Operadores aritméticos

Operador	nombre	ejemplo	significado
+	suma	*a + b*	*a* más *b*
-	resta	*a - b*	*a* menos *b*
*	multiplicación	*a * b*	*a* por *b*
/	división	*a / b*	*a* entre *b*
%	residuo	*a % 5*	*a* módulo *b*
**	potenciación	*a ** b*	*a* elevado a la *b*

Operadores relacionales

Operador	nombre	ejemplo	significado
<	menor que	*a<b*	*a* es menor que *b*
>	mayor que	*a>b*	*a* es mayor que *b*
==	igual a	*a==b*	*a* es igual a *b*
!=	distinto a	*a!=b*	*a* no es igual a *b*
<=	menor que o igual a	*a<=5*	*a* es menor que o igual a *b*
>=	mayor que o igual a	*a>=b*	*a* es menor que o igual a *b*

Operadores lógicos

Operador	nombre	ejemplo	significado
&&	“Y” lógico	*a && b*	*a* y *b*
||	“O” lógico	*a || b*	*a* o *b*
^	“O” exclusivo (XOR)	*a ^ b*	*a b*
!	negado	*!a*	*El negado de a*

Expresiones

Son la forma básica de **representar un cómputo** mediante la **combinación** de variables, constantes y/o funciones que el lenguaje interpreta de acuerdo a su precedencia y asociación.

Una expresión es una representación de un valor

Jerarquía o Precedencia de operadores

Es la **jerarquía** que indica el **orden de evaluación** de los operadores en las expresiones. Ej:

4 + 5 - 3 / 2 * 2 - 1 = 6

La precedencia de operadores aritméticos de la mayoría de los lenguajes de programación, puede recordarse mnemotécnicamente con el acrónimo:

PEMDRAS

---------->

1. Paréntesis
2. Exponencial
3. Multiplicación, División y Residuo
4. Adición y Sustracción

Los paréntesis se usan para anular la precedencia regular y forzar prioridades de operadores en una expresión. Ej:

4 + 5 - 3 / (2 * 2) - 1 = 8
4 + (5 - 3) / 2 * 2 - 1 = 5
4 + 5 - (3 / 2) * 2 - 1 = 6

Asociación de operadores

Es el orden en que se evalúan los operadores de igual precedencia ubicados consecutivamente dentro de una expresión. La mayoría de los lenguajes de programación implementan asociación por la izquierda (de izquierda a derecha). Ej:

4 + 5 - 3 - 2 - 2 + 1 = 3

Sentencias

Son **instrucciones** escritas en un lenguaje de programación de alto nivel que **ordenan a la computadora** a que **realice un una acción** determinada. Un programa está formado por una secuencia de una o más sentencias.

Sintaxis

Es el conjunto de reglas que hay que seguir para que el compilador de un lenguaje reconozca un código como válido.

Palabras reservadas

Son palabras que pertenecen a la especificación de un lenguaje y tienen un significado gramatical especial, por lo que su uso dentro de un programa queda restringido. Dicho de otra forma, una palabra reservada no puede usarse como nombre de variable.

Cada lenguaje tiene su propia **sintaxis** y cuenta con sus propias **palabras reservadas**.

Variables

Una variable es un espacio en la memoria principal del computador (volátil) que se asocia a un nombre simbólico. Ese espacio contiene una cantidad de información, conocida o desconocida, es decir, un valor.

Reglas básicas para la asignación de nombres a variables

- debe sugerir lo que representa
- no puede coincidir con palabras reservadas del lenguaje
- no debe exceder un tamaño de 32 caracteres
- comenzará siempre por un caracter alfabético o el símbolo "_", y los siguientes podrán ser letras y/o dígitos

Asignación de un dato a una variable

La asignación de un dato a una variable, se realiza de la siguiente forma:

```
mi_variable = [ valor ]
```

Por ejemplo, para asignar el valor *7* a la variable *cantidad*, se escribirá:

```
cantidad = 7
```

Conversión de datos (type casting)

Es la transformación de un tipo de dato en otro. Esto se hace para tomar las ventajas que pueda ofrecer el tipo de dato a que se va a convertir. Existen dos tipos de conversión:

1.- Conversión implícita

Este tipo de conversión lo realiza el compilador, ya que no hay pérdida de datos. Por ejemplo, si se pasa un *int* (tipo entero) a *float* (tipo decimal).

2.- Conversión explícita

En este caso, el compilador no es capaz de realizarla por sí solo y por ello debe definirse explícitamente en el programa. Por ejemplo, la conversión de un *string* (tipo texto) a *int* (tipo entero).

Mostrar datos al usuario por consola (Ruby)

```
puts( v )
```

Pedir datos al usuario por consola (Ruby)

```
v = gets.chomp
```

Comentarios

Permiten colocar información dirigida a lectores humanos dentro del código fuente que será absolutamente ignorada por el intérprete del lenguaje. Se usan, principalmente, para describir partes del código para facilitar su entendimiento. Se recomienda su uso cada vez que sea necesario y con moderación.

Comentario de solo una línea

```
# Este texto no será procesado de ninguna forma por el intérprete.
mi_variable = 5           # Este también es un comentario válido
```

Comentario de varias líneas

```
=begin
Este texto será completamente
ignorado por el intérprete
del lenguaje
=end
```

Teorema del programa estructurado

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine sólo tres estructuras lógicas, también llamadas estructuras de control.

Las 3 estructuras de control de la programación estructurada

a. Secuencia
b. Selección
c. Iteración

Estas 3 estructuras controlan el flujo de ejecución de las instrucciones del programa.

Secuencia

La estructura secuencial realiza o **ejecuta** las sentencias del programa, **instrucción por instrucción**, de **forma secuencial**, de modo que la salida de una es la entrada de la otra, hasta el final del proceso. Es una estructura, generalmente, **implícita**, pues **todos los lenguajes** ejecutan las instrucciones línea a línea por defecto. Para poder alterar (o desviar) la secuencia de ejecución de las instrucciones (sentencias) de un programa, se debe hacer uso de las otras dos estructuras de control del lenguaje: estructuras de selección y estructuras de iteración.

Estructuras de selección

Condicional - If… then

```
i = rand(10)
if i > 5 then
   puts “#{i} es mayor a 5”
end
```

Condicional con opción por defecto - If… then… else

```
i = rand(10)
if i >= 5 then
   puts “#{i} es mayor a 5”
else
   puts “#{i} es menor o igual a 5”
end
```

Condicional anidado - If… then… elsif...else

```
i = rand(20)
if i > 15 then
   puts “#{i} es mayor a 15”
elsif i >= 10 then
   puts “#{i} está entre 10 y 15”
else
   puts “#{i} es menor a 10”
end
```

Case

```
n = rand(4)
case n
   when 0 then
      puts "Hoy comes carne"
   when 1 then
      puts "Hoy comes pollo"
   when 2 then
      puts "Hoy comes pescado"
   else
      puts "Hoy comes huevos"
end

### Otro ejemplo del uso del Case
n = rand(4)
opc = case
   when n == 0 then
      "Carne"
   when n == 1 then
      "Pollo"
   when n == 2 then
      "Pescado"
   else
      "Huevos"
end
puts opc
```

Estructuras iterativas

Iterador While

```
i = 0
while i < 5 do
   puts i
   i += 1
end
```

El iterador While se utiliza cuando se desea **evaluar la condición** de parada **antes de ejecutar** las instrucciones de su bloque.

Iterador Repeat

```
i = 1
loop do
    puts i
    i += 1
    break if (i > 8)
end
```

El iterador Repeat se utiliza cuando se desea **ejecutar las instrucciones** de su bloque **al menos una vez** antes de **evaluar** la condición de parada.

Iterador For

```
for i in 1..8 do
    puts i
end
```

El iterador For se utiliza cuando **se conoce el número de iteraciones** en que se desea ejecutar su bloque de instrucciones.

Arreglos

Es un tipo de datos estructurado que está formado de una **colección** finita y ordenada de **datos** del **mismo tipo**. Es la estructura natural para modelar listas de elementos iguales. El tipo de acceso a los datos de un arreglo es el **acceso directo**, mediante el uso de un índice para cada elemento del arreglo que nos da su posición relativa, es decir, podemos acceder a cualquier elemento del arreglo sin tener que consultar elementos anteriores o posteriores. En la mayoría de los lenguajes de programación, el primer valor del arreglo es accedido con el índice 0, es decir, las posiciones de los valores de los arreglos comienzan a enumerarse desde el cero.

Declarar arreglos

```
# Declaración de un array en Ruby (declaración formal)
mi_arreglo = Array.new    # Crea un arreglo vacío
mi_array4 = Array.new(5)  # Crea el arreglo [nil, nil, nil, nil, nil]

# Declaración de un array en Ruby (declaración literal)
mi_array = []                    # Crea un arreglo vacío
mi_array2 = [3, 4, 7, 12, 0]     # Crea un arreglo con los valores indicados

# Declaración de un array en Ruby (otras formas de declaración)
mi_array3 = (0..5).to_a          # Crea el arreglo [1, 2, 3, 4, 5]
```

Acceder a valores de arreglos

```
# Crea un arreglo nuevo (declaración literal)
mi_array = [3, 4, 7, 12, 0]

puts mi_array[1]                 # Muestra el valor de la 2da casilla: 4

# Recorre todo el arreglo mostrando el valor de cada posición
for i in 0..mi_array.length - 1 do
  puts mi_array[i]
end
```

Modificar los valores de un arreglo

```
mi_array = [3, 4, 7, 12, 0]      # Crea el arreglo

# Sobreescribe el valor de la 3ra casilla
mi_array[2] = 0
=> [3, 4, 0, 12, 0]

# Cambia todos los valores del arreglo por valores aleatorios
for i in 0..mi_array.length - 1 do
  mi_array[i] = rand(20)
end
=> [16, 11, 9, 4, 13]
```

Agregar nuevos valores a un arreglo

```
mi_array = [1, 2]

mi_array[1] = 0           # Sustituye (sobreescribe) la 2da posición del arreglo
=> [1, 0]

mi_array[3] = 6           # Pone en nil la 3ra posición del arreglo y agrega el
=> [1, 0, nil, 6]         # valor 6 en la 4ta posición

mi_array.push( 5 )        # Agrega el valor 5 al final del arreglo
=> [1, 0, nil, 6, 5]
```

```
mi_array << 8             # Agrega el valor 8 al final del arreglo
=> [1, 0, nil, 6, 5, 8]

mi_array << 3 << 7        # Agrega los valores 3 y 7 al final del arreglo
=> [1, 0, nil, 6, 5, 8, 3, 7]
```

Obtener y retirar el último valor de un arreglo

```
mi_array = [3, 4, 1, 12, 7]

valor = mi_array.pop()    # valor= 7  =>  mi_array= [3, 4, 1, 12]
```

Obtener y retirar el primer valor de un arreglo

```
mi_array = [3, 4, 1, 12, 7]

valor = mi_array.shift()  # valor= 3  =>  mi_array= [4, 1, 12, 7]
```

Saber si un arreglo está vacío

```
mi_array = []

mi_array.empty?           # empty? devuelve false si el arreglo tiene elemento(s)
=> true
```

Saber la cantidad de elementos de un arreglo (longitud)

```
mi_array.length           # La longitud del arreglo es cero
=> 0


mi_array.size             # El tamaño del arreglo es cero
=> 0
```

Arreglos Bidimensionales (matrices)

Un **arreglo bidimensional** o **matriz** es una estructura de datos de un mismo tipo ordenados secuencialmente en forma de tabla mediante filas y columnas. Son un caso especial de arreglos **multidimensionales** en el que cada valor del arreglo que representa la matriz es, a su vez, un arreglo que contiene datos del mismo tipo. Por lo general, las matrices poseen filas de igual cantidad de elementos, lo que les permite representar un rectángulo (matriz rectangular) o un cuadrado (matriz cuadrada), pero puede que no siempre sea así.

Para acceder a un elemento de una matriz deben especificarse el nombre del arreglo que lo representa, seguido de dos índices: **fila** y **columna**, encerrados entre corchetes de la forma “[fila][columna]”. Es importante recordar que el índice debe ser un valor de tipo **entero**

comprendido **entre 0** y el número de elementos de cada fila menos uno (**longitud - 1**) y que **primero** se especifica la **fila** y luego la **columna**, siguiendo exactamente la misma regla pero con la longitud de las columnas. De esta forma, el algoritmo para recorrer la matriz podrá recorrer todos sus elementos correctamente. Es muy usual recorrer las matrices haciendo uso de dos (2) iteradores anidados: uno para moverse a través de las filas y otro para moverse a través de las columnas.

Declarar una matriz

```
# Declaración de una matriz en Ruby (declaración formal)
mi_matriz = Array.new            # Crea un arreglo vacío
mi_arreglo = Array.new           # Crea un arreglo vacío
mi_matriz[0] = mi_arreglo        # Representa una matriz 0x0 (vacía)
mi_matriz.push(mi_arreglo)       # Equivalente a la línea anterior (no usar ambas)

# Declaración de una matriz en Ruby (declaración literal)
mi_matriz = [[ 0, 1, 2],
             [ 3, 4, 5],
             [ 6, 7, 8],
             [ 9, 10, 11]]       # Crea una matriz 4x3 con los valores indicados

mi_matriz2 = [[3, 4], [7, 12]]   # Crea una matriz 2x2 con los valores indicados

mi_matriz3 = [[7, 9, 8], [10, 1], [3]] # Crea una matriz no cuadrada
```

Llenar automáticamente una matriz nueva

```
# Crea una matriz 6x3 de números aleatorios entre el 0 y el 9 (forma 1)
matriz = []
for i in 0..5 do
   aux = []
      for j in 0..2 do
         aux[j] = rand(10)
      end
   matriz[i] = aux
end

# Crea una matriz 6x3 de números aleatorios entre el 0 y el 9 (forma 2)
matriz = []
for i in 0..5 do
   matriz[i] = []
      for j in 0..2 do
         matriz[i][j] = rand(10)
      end
end
```

Acceder a valores de una matriz

```
# Declaración de una matriz en Ruby (declaración literal)
mi_matriz = [[ 0, 1, 2],
             [ 3, 4, 5],
```

```
            [ 6, 7, 8],
            [ 9, 10, 11]]      # Crea una matriz 4x3 con los valores indicados

puts mi_matriz[1][2]           # Muestra el valor 5 (2da fila, 3ra columna)
```

Modificar valores de una matriz

```
# Modifica el valor de la 3ra fila 1ra columna de la matriz
mi_matriz[2][0]

# Recorre toda la matriz cambiando cada valor por un valor aleatorio
for i in 0..mi_matriz.length - 1 do
  for j in 0..mi_matriz[i].length - 1 do
    mi_matriz[i][j] = rand(20)
  end
end
```

Mostrar todos los valores de una matriz

```
# Recorre toda la matriz mostrando cada valor
for i in 0..mi_matriz.length - 1 do
  for j in 0..mi_matriz[i].length - 1 do
    puts mi_matriz[i][j]
  end
end
```

Procedimientos y Funciones

Los procedimientos y funciones **permiten la organización y la reutilización de código**. En lugar de escribir largas secciones de código, por lo general, repetido, es posible definir bloques de instrucciones lógicamente independientes y con una única finalidad, que luego pueden ser invocados cuando se necesiten, sin tener que reescribirlo una y otra vez. Los procedimientos y funciones, también conocidos en el paradigma orientado a objetos como métodos, son simples de escribir y usar: sólo necesitan ser declarados con un nombre para poder ser usados.

La diferencia básica entre los procedimientos y las funciones es que **los procedimientos no devuelven ningún valor**, mientras que **las funciones,**

obligatoriamente, deben devolver un valor, lo que las hace **equivalentes a un valor**. En el caso del lenguaje Ruby, no existen los procedimientos, sólo funciones.

Los métodos deben diseñarse de modo que tengan una **única funcionalidad** (principio de Responsabilidad Única) escrita genéricamente, a fin de que **funcionen para la mayor cantidad de casos posibles** (si no todos), por lo que pueden recibir valores que parametricen el computo y, en consecuencia, el resultado que devolverá el método. Estos valores (llamados **parámetros**) son dados al método al momento de ser invocados.

Definir una función

```
def nombre_funcion( param1, param2, …, paramN )
  .
  . # sentencias
  .
  return valor
end

# Ejemplos de funciones 1
# Devuelve un string con un saludo
def saluda_a( nombre )
  return “Hola “ + nombre
end

# Invocamos la función
puts saluda_a( “Juca” )   # Imprime “Hola Juca” en el terminal

# Ejemplos de funciones 2
# Devuelve la suma de los dos parámetros que recibe
def suma( val1, val2 )
  return val1 + val2
end

s = suma( 3, 5 )                       # La variable ‘s’ vale 8
s = suma( 6, 1 )                       # La variable ‘s’ vale 7
s = suma( suma( 9, 4 ), suma( 2, 3 ) ) # La variable ‘s’ vale 18



# Ejemplos de funciones 3
# No devuelve nada (procedimiento) ni recibe parámetros. Sólo imprime un string
def inutil
  put ““
end

# Se invoca el método
inutil
```

Recursividad

Funciones recursivas

Las funciones recursivas son funciones que se invocan a ellas mismas. Toda función recursiva debe tener 2 características obligatorias:

1. Tener una condición de parada, conocido como caso base.
2. Invocarse a ella misma, llamado caso recursivo.

Algunos ejemplos clásicos de recursividad son la función factorial, la sucesión de Fibonacci, el recorrido de listas y árboles binarios, etc.

Función Sumatoria recursiva

```
def sumat(n)
  if n > 0 # caso recursivo
    return n + sumat(n - 1)
  end

  return 0
end
```

Función Factorial recursiva

```
def fact(n)
  if n > 0 # caso recursivo
    return n * fact(n - 1)
  end

  return 1 #caso base
end

# Versión pedagógica
def fact(n)
  if n > 0 # caso recursivo
    return n * fact(n - 1)
  end

  if n == 0 # caso base
    return 1
  end
end
```

Sucesión de Fibonacci

```
# Suces. de Fibonacci: f(n) = f(n-1) + f(n-2) ^ f(1) = 1 ^ f(0) = 0
def fibonacci2( n )
  if n > 1 # caso recursivo
    return fibonacci2(n - 1) + fibonacci2(n - 2)
  end

  return n # caso base
end
# Versión pedagógica
def fibonacci n
  if n > 1 # caso recursivo
    return fibonacci(n - 1) + fibonacci(n - 2)
  end

  if n == 1 # caso base
    return 1
  end

  if n == 0 #caso base
    return 0
```

```
    end
end
```

Archivos

Un archivo es un conjunto de datos estructurados que se almacena en alguna unidad de memoria persistente referenciado por un identificador, y puede ser usado por programas. Se clasifican de diversas formas: por el tipo de datos que almacenan (archivo de texto, archivo binario, etc.), si pueden ejecutarse o no (archivo ejecutable y no ejecutable).

La gran mayoría de los lenguajes de programación ofrecen herramientas para la creación y manipulación de archivos.

Crear un nuevo archivo

Abrir un archivo existente

Leer de un archivo

Escribir en un archivo

Borrar y renombrar un archivo