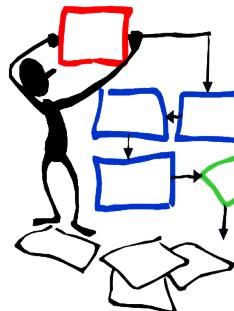




Instituto Politécnico Nacional

Escuela Superior de Cómputo



Algoritmia y programación estructurada

Tema 01: Algoritmia y diagramas de flujo

M. en C. Edgardo Adrián Franco Martínez
<http://www.eafranco.com>
edfrancom@ipn.mx
[@edfrancom](https://twitter.com/edfrancom) [@edgardoадrianfranco](https://facebook.com/edgardo.adrian.franco)



Contenido

- Algoritmia
- ¿Qué es un algoritmo?
- Capacidades de una computadora
- Métodos algorítmicos
- Diagrama de flujo
- Símbolos utilizados en los diagramas de flujo
- Reglas para la construcción de diagramas de flujo
- Ejemplos



Algoritmia

- Área de estudio cuyo objeto de estudio son los **algoritmos**.
- **En computación un algoritmo permite modelar** la serie de pasos necesarios a realizar para poder resolver una parte o la totalidad de un problema computacional.
- Al hablar de un **problema computacional** se hace referencia a la necesidad de dada cierta información alcanzar un resultado buscado sabiendo que existe una o un **conjunto de soluciones capaces de ser encontradas** utilizando las **capacidades de procesamiento de una computadora**.



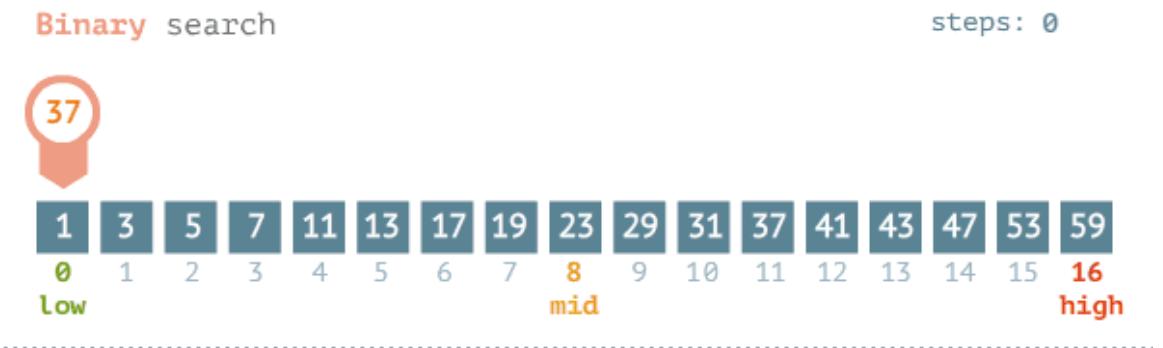
¿Qué es un algoritmo?

- Es un **conjunto ordenado y finito de operaciones** que permite hallar la solución de un problema.
- Podemos decir que un algoritmo es una "receta", ya que si se sigue de manera correcta encuentran un resultado en un tiempo acotado.
 - P.g. escribe un algoritmo para agradar a tu maestro de programación.



¿Qué es un algoritmo computacional?

- Es un **conjunto ordenado y finito de operaciones inherentes a un cómputo** que permite hallar la **solución de un problema**.



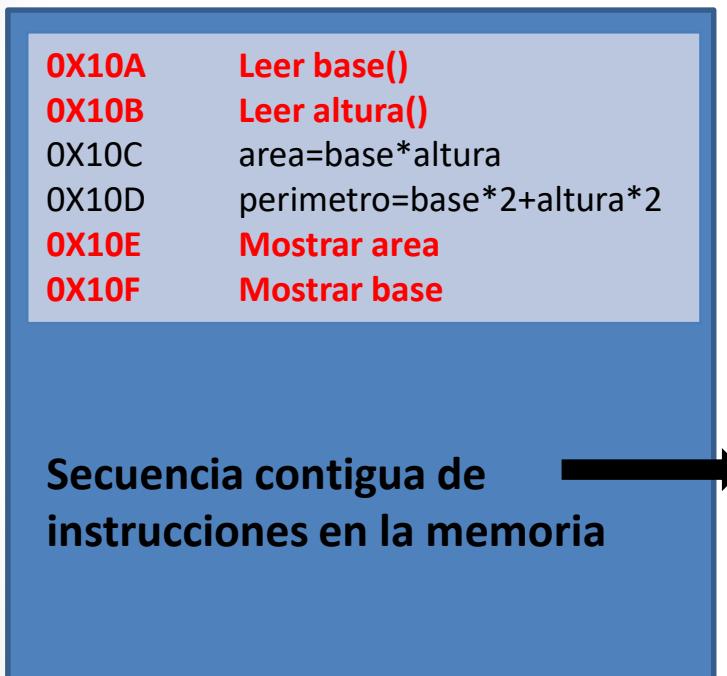
¿Cuáles son las capacidades de una computadora?

Al pensar en la idea de crear algoritmos a ser ejecutados mediante una computadora es necesario entender cuales son las capacidades de una computadora de manera general:

1. Procesar instrucciones en secuencia (*Las instrucciones se colocan secuencialmente en la memoria y se ejecutan en ese orden*)
2. Recibir o mostrar datos (*Una computadora cuenta dispositivos para la entrada y salida de información capaz de ser colocada en la memoria para leerlos o tomarlos y mostrarlos*)
3. Saltar a otras instrucciones (*Continuar una secuencia de instrucciones en otra dirección de memoria no continua*)
4. Almacenar, editar o eliminar datos en la memoria (*Guardar, editar, mover y eliminar valores en una o más direcciones de memoria*)
5. Operar con números almacenados en la memoria (*Una computadora cuenta con la posibilidad de realizar operaciones numéricas en base 2 con las variables colocadas en la memoria*)



1. Procesar instrucciones en secuencia (*Las instrucciones se colocan secuencialmente en la memoria y se ejecutan en ese orden*)
2. Recibir o mostrar datos (*Una computadora cuenta dispositivos para la entrada y salida de información capaz de ser colocada en la memoria para leerlos o tomarlos y mostrarlos*)



```

00001A1E 4D 4B      LDR
00001A20 E3 58      LDR
00001A22 1B 68      LDR
00001A24 18 46      MOV
00001A26 FF F7 52 EA BLX
00001A2A 03 46      MOV
00001A2C 18 46      MOV
00001A2E 4A 4B      LDR
00001A30 7B 44      ADD
00001A32 19 46      MOV
00001A34 00 F8 34 FB BL
00001A38 48 4B      LDR
00001A3A E3 58      LDR
00001A3C 1B 68      LDR
00001A3E 18 46      MOV
00001A40 FF F7 44 EA BLX
00001A44 02 46      MOV
00001A46 07 F5 43 63 ADD.W
00001A4A 18 46      MOV
00001A4C 19 46      MOV
00001A4E 4F F8 02 02 MOV.W
00001A52 4F F8 0A 03 MOV.W
00001A56 00 F8 77 FB BL
00001A5A 03 46      MOV
00001A5C 00 2B      CMP
R3, =(stdout_ptr - 0xC000)
R3, [R4,R3] ; stdout
R3, [R3]
R0, R3 ; stream
fileno
R3, R0
R0, R3
R3, -(a1ChangeDisplay - 0x1)
R3, PC ; "1 ) Change displ
R1, R3
print
R3, =(stdin_ptr - 0xC000)
R3, [R4,R3] ; stdin
R3, [R3]
R0, R3 ; stream
fileno
R2, R0
R3, R7, #0xC30
R0, R2
R1, R3
R2, #2
R3, #0xA
read
R3, R0
R3, #0
  
```

ASM	HEX
mov eax,0x4	b8 04 00 00 00
mov ebx,0x1	bb 01 00 00 00
mov ecx,0x80490a0	b9 a0 90 04 08
mov edx,0x26	ba 26 00 00 00
int 0x80	cd 80
mov eax,0x1	b8 01 00 00 00
xor ebx,ebx	31 db
int 0x80	cd 80



3. Saltar a otras instrucciones (Continuar una secuencia de instrucciones en otra dirección de memoria no continua)

0X10A	Leer base()
0X10B	Leer altura()
0X10C	area=base*altura
0X10D	perimetro=base*2+altura*2
0X10E	Mostrar area
0X10F	Mostrar base
0X110	Salta 0X10A

	1	2	3	4
1	* = \$6000			
2	inicio			
3	lda #\$00			
4	sta \$2c6			
5	lda #\$0F			
6	sta \$2c5			
7	lda #'H			
8	jsr \$f2b0			
9	lda #'0			
10	jsr \$f2b0			
11	lda #'L			
12	jsr \$f2b0			
13	lda #'A			
14	jsr \$f2b0			
15	lda #155 ;Es un RETURN			
16	jsr \$f2b0			
17	jmp inicio			
18	*=\$2e0			
19	.word inicio			



4. Almacenar, editar o eliminar datos en la memoria (*Guardar, editar, mover y eliminar valores en una o más direcciones de memoria*)
5. Operar con números almacenados en la memoria (*Una computadora cuenta con la posibilidad de realizar **operaciones numéricas en base 2** con las variables colocadas en la memoria*)

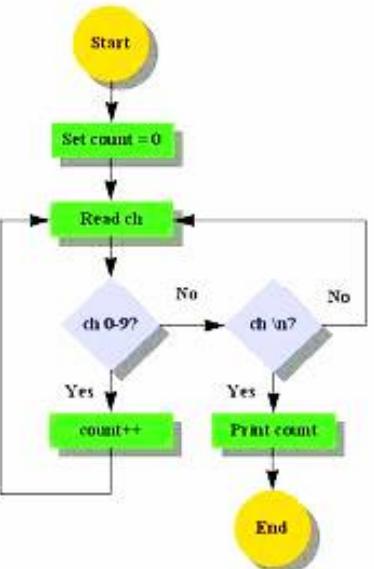
0X10A	Leer base()	~ 0x200 = Entrada
0X10B	Leer altura()	~ 0x203 = Entrada
0X10C	area=base*altura	~ 0x206 = 0x200 * 0x203
0X10D	perimetro=base*2+altura*2	~ 0x208 = 0x002 ~ 0x20A = 0x200 * 0x208 ~ 0x20B = 0x200 * 0x203 ~ 0x20C = 0x20A + 0x20B
0X10E	Mostrar área	~ Salida = 0x206
0X10F	Mostrar base	~ Salida = 0x206
0X110	Salta 0X10A	



Definición formal de algoritmo

“Es un **conjunto de instrucciones o reglas bien definidas, ordenadas y finitas** que permite realizar una actividad mediante pasos no ambiguos y efectivos que no generen dudas a quien lo ejecute”.

- Dados un **estado inicial** y una entrada, siguiendo los pasos sucesivos se llega a un **estado final** y se **obtiene una solución**.



```
Inicio
    aprobados ← 0
    reprobados ← 0
    resultado ← 0
    estudiantes ← 1
    Mientras estudiantes <= 10
        Leer resultado
        Si resultado == 1 entonces
            aprobados ← aprobados + 1
        Sino
            reprobados ← reprobados + 1
        FinSi
        estudiantes ← estudiantes + 1
    FinMientras
    Mostrar aprobados
    Mostrar reprobados
    Si aprobados > 8 entonces
        Mostrar "Aumentar la colegiatura"
    FinSi
Fin
```

Diagrama de flujo

- Existen distintas formas gráficas de representar un algoritmo, el **diagrama de flujo** fue una de las primeras empleadas.
- Un **diagrama de flujo** se utiliza símbolos con significados bien definidos que representan los pasos del algoritmo, y representan el flujo de ejecución mediante flechas que conectan los puntos de inicio y de término.

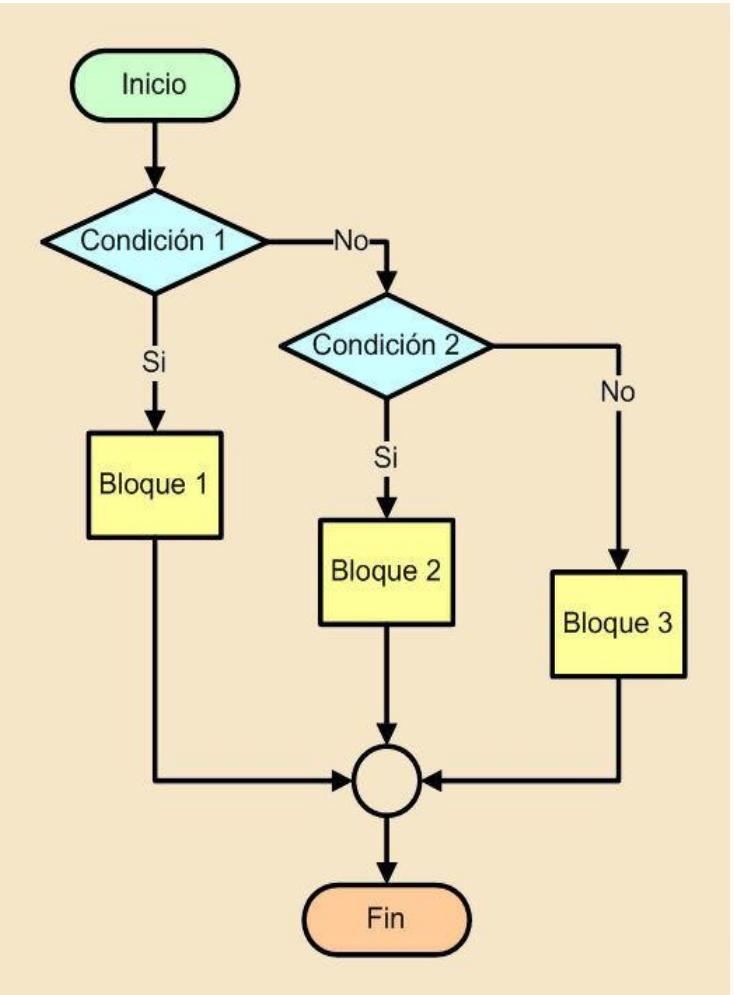
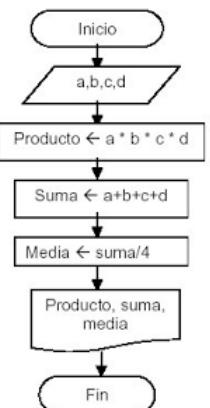
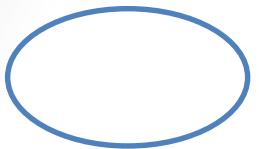


Diagrama de flujo

- Es la representación gráfica de un algoritmo
- Muestra los pasos o procesos a seguir para alcanzar la solución de un problema
- Utilizan símbolos (cajas) estándar y tienen los pasos del algoritmo escritos en estas cajas unidas por flechas
- La secuencia del algoritmo esta determinado por el flujo de la flechas



Símbolos utilizados en los diagramas de flujo



Inicio y Fin



Almacenamiento/
Salida de datos



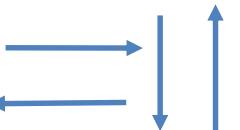
Lectura/Entrada
de datos



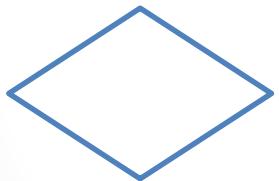
Proceso



Conector



Flujo del
diagrama



Decisión



Decisión múltiple



Operadores típicamente utilizados

+	Sumar
-	Menos
*	Multiplicación
/	División
=	Equivalencia
>	Mayor que
<	Menor que
≥	Mayor o igual que
≤	Menor o igual que
<> o !=	Diferente de
← o → o =	Asignación

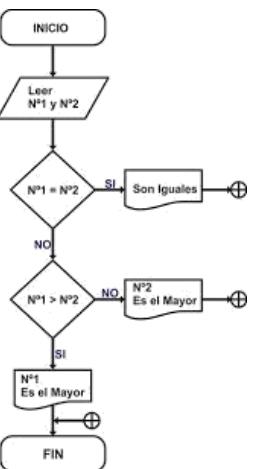
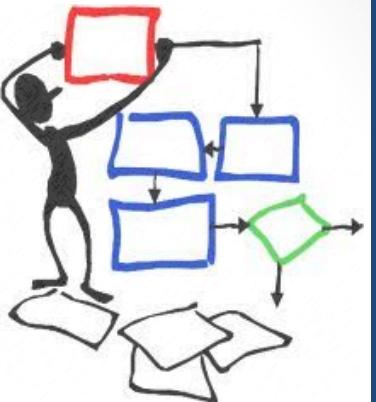
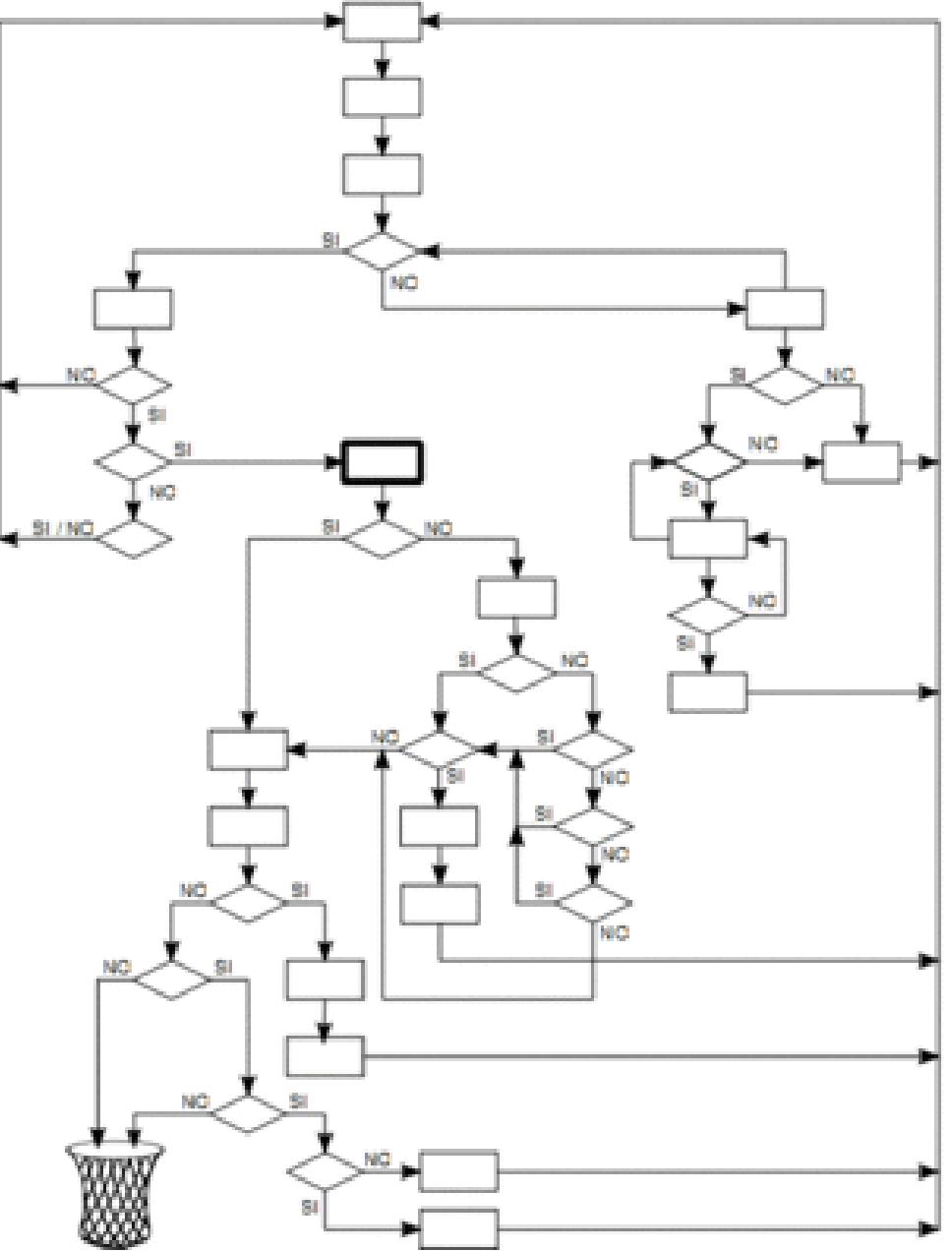


«La definición de datos se da por supuesta, principalmente para variables sencillas»

- # Reglas para la construcción de diagramas de flujo
1. Todo diagrama de flujo debe tener un inicio y un fin
 2. Las líneas utilizadas para indicar la dirección del flujo del diagrama deber ser rectas: verticales u horizontales
 3. Todas las líneas utilizadas para indicar la dirección del flujo del diagrama deben estar conectadas mediante algún símbolo
 4. El diagrama de flujo debe construirse de arriba hacia abajo y de izquierda a derecha
 5. La notación utilizada en el diagrama de flujo debe ser independiente del lenguaje de programación
 6. Si la construcción del diagrama de flujo requiere más de una hoja se deben utilizar los conectores adecuados

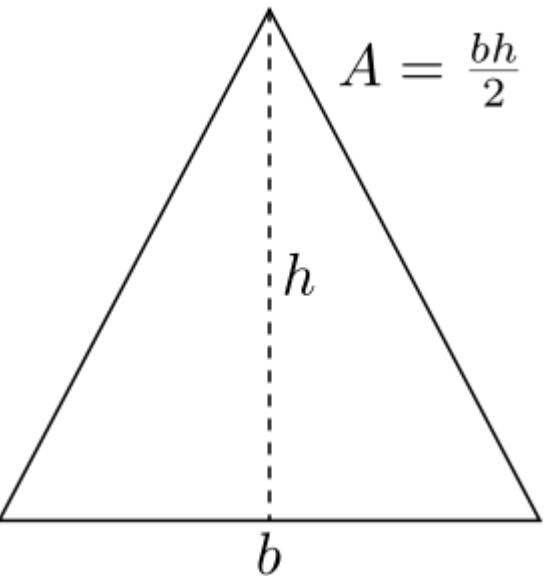


Algoritmia y programación estructurada
01: Algoritmia y diagramas de flujo
Prof. Edgardo Adrián Franco Martínez

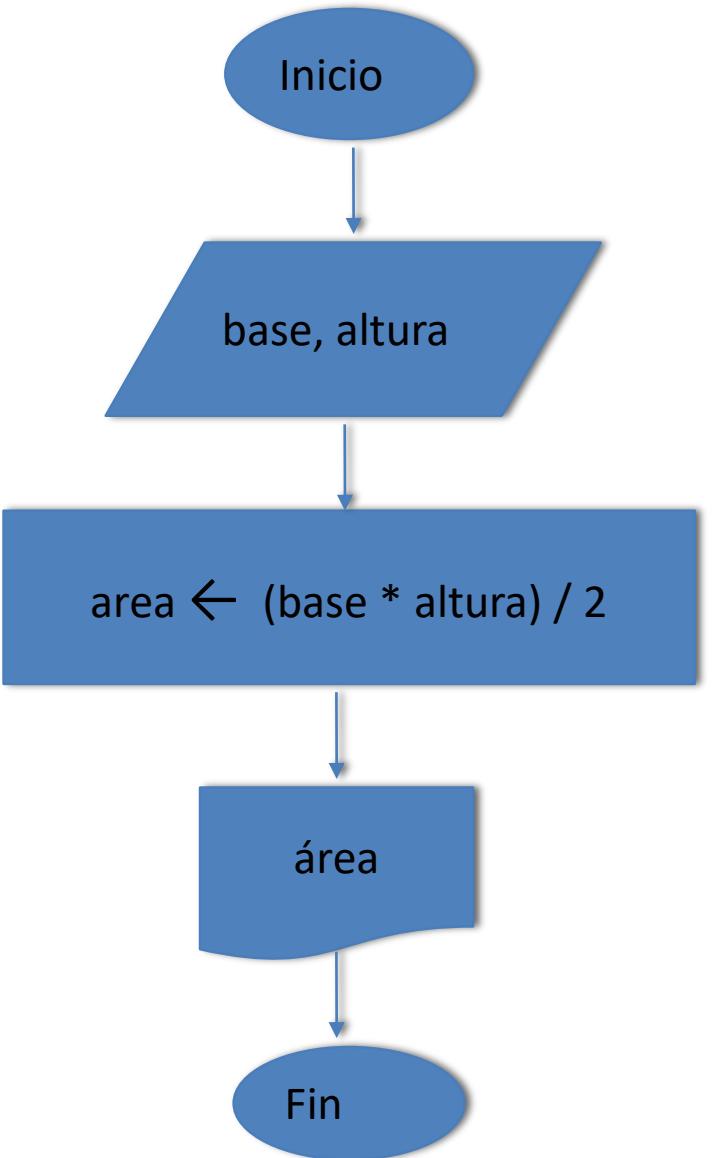


Ejemplo 01: Área de un triángulo

- Describa un algoritmo mediante un diagrama de flujo para calcular el área de un triangulo.
- Se recibe como entrada la base y la altura.



Ejemplo 01: Área de un triángulo



Ejemplo 02: Ingresos de empleado

- Construir un algoritmo que, al recibir como entrada una clave de un empleado y los seis primeros sueldos mensuales del año de este, calcule el ingreso total semestral y el promedio mensual para el empleado, finalmente se imprimirá su clave, el ingreso total y el promedio mensual.



Clave
Empleado

Sueldo 1

Sueldo 2

Sueldo 3

Sueldo 4

Sueldo 5

Sueldo 6

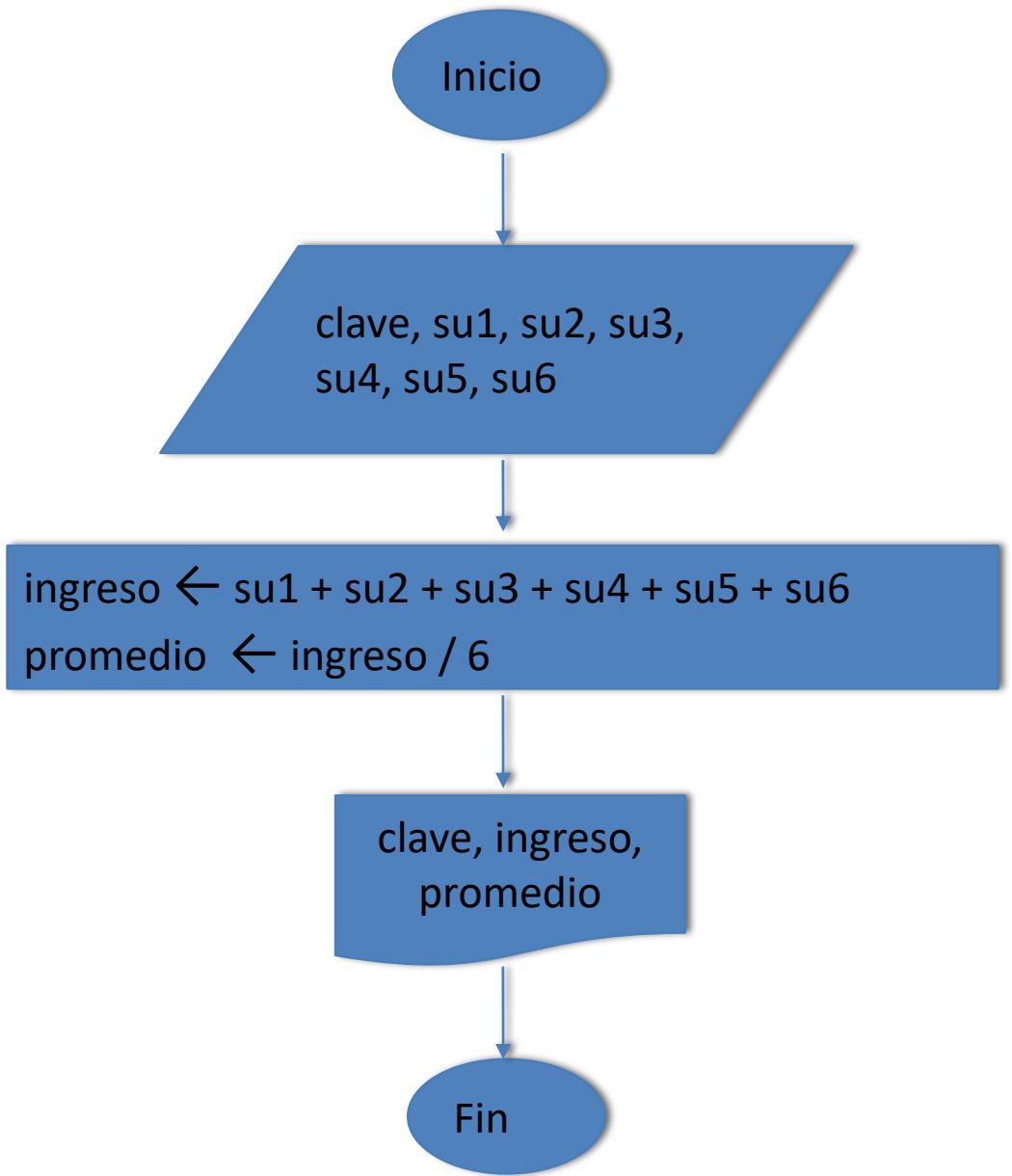


Clave
Empleado

Ingreso total

Promedio mensual

Ejemplo 02: Ingresos de empleado

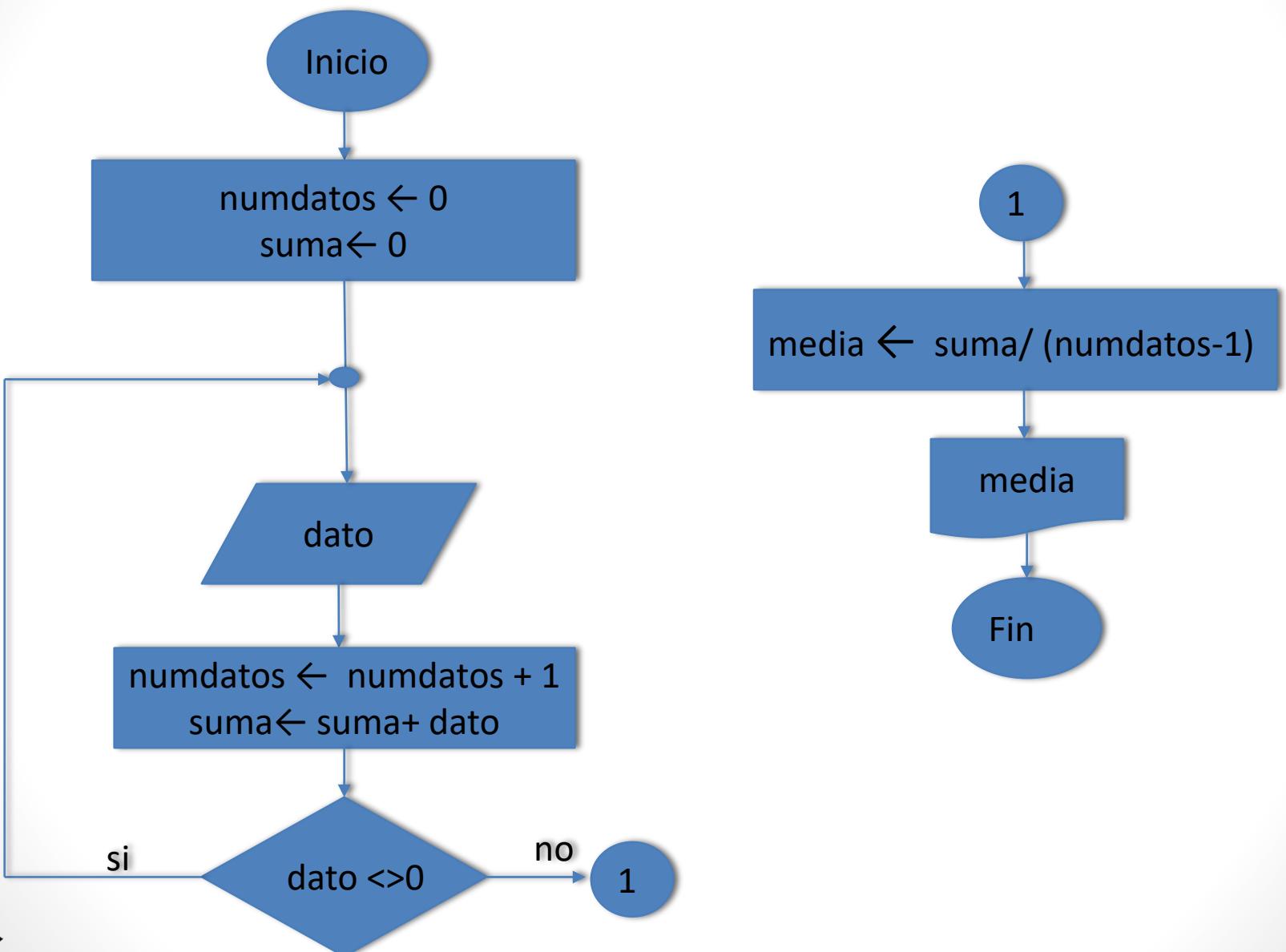


Ejemplo 03: Media de una serie de números

- Calcular la media de una serie de números positivos, suponiendo que los datos se introducen uno a uno.
- Un valor de cero como entrada indicará que se ha alcanzado el final de la serie de números positivos.



Ejemplo 03: Media de una serie de números

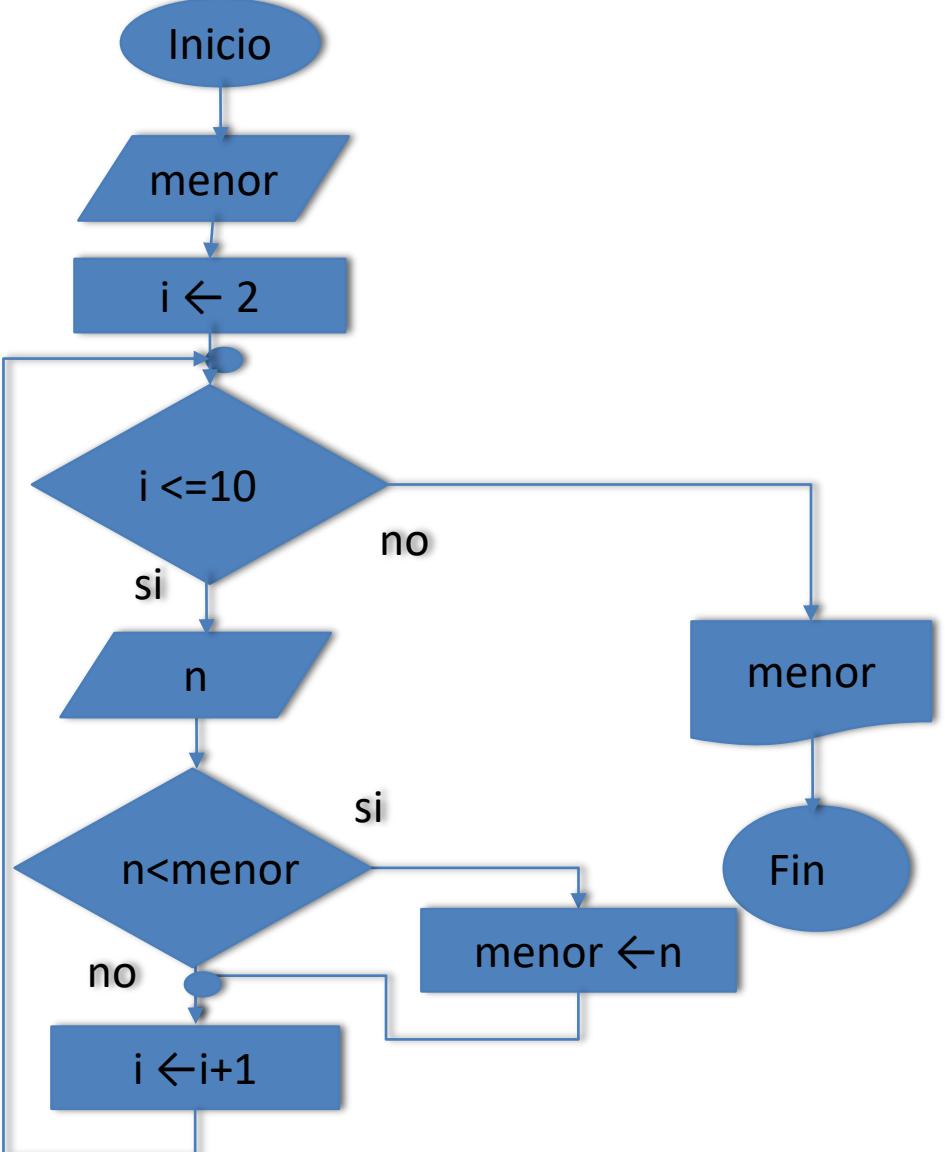


Ejemplo 04: Mínimo de 10 números

- Determinar el menor valor de una serie de 10 números positivos, suponiendo que los datos se introducen uno a uno.



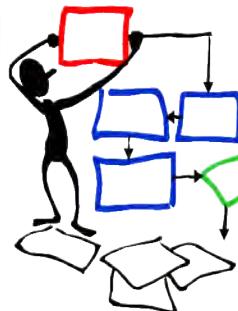
Ejemplo 04: Mínimo de 10 números





Instituto Politécnico Nacional

Escuela Superior de Cómputo



Algoritmia y programación estructurada

Tema 02: Algoritmia y pseudocódigo

M. en C. Edgardo Adrián Franco Martínez
<http://www.eafranco.com>
edfrancom@ipn.mx
[@edfrancom](https://twitter.com/edfrancom) [f edgaroadrianfrancocom](https://facebook.com/edgaroadrianfrancocom)



Contenido

- Pseudocódigo
- Ejemplo 01
- Ejemplo 02
- Ejemplo 03
- Ejemplo 04
- Representación de Algoritmos en Pseudocódigo
 - Asignación
 - Variables
 - Estructuras de control de flujo
 - Anidamiento



Pseudocódigo

- El **pseudocódigo** es una **descripción de alto nivel** de un **algoritmo** que emplea una mezcla de **lenguaje natural** con algunas **convenciones sintácticas** propias de lenguajes de programación, a usar (es un supuesto lenguaje) .
- Es utilizado para describir algoritmos de manera formal en libros y publicaciones científicas, y como producto intermedio durante el desarrollo de un algoritmo.
- El **pseudocódigo** está pensado para **facilitar a las personas el entendimiento de un algoritmo**, y por lo tanto puede omitir detalles irrelevantes que son necesarios en una implementación.



- Programadores diferentes suelen utilizar **convenciones distintas**, que pueden estar basadas en la sintaxis de lenguajes de programación concretos. Sin embargo, el **pseudocódigo en general es comprensible sin necesidad de conocer o utilizar un entorno de programación específico**, y es a la vez **suficientemente estructurado** para que su implementación se pueda hacer directamente a partir de él.
- Es independiente del lenguaje de programación.
- La definición de datos se da por supuesta, principalmente para variables sencillas, pero si se emplea variable más complejas, por ejemplo pilas, colas, vectores, etc., se pueden definir en la cabecera del algoritmo.



Programa Encender lámpara
 si (*lámpara enchufada*) entonces
 si (*lámpara encendida*) entonces
 problema solucionado
 si no
 si (*foco quemado*) entonces
 reemplazar foco
 si no
 comprar nueva lámpara
 fin si
 fin si
 si no
 enchufar lámpara
 si (*lámpara encendida*) entonces
 problema solucionado
 si no
 si (*foco quemado*) entonces
 reemplazar foco
 si no
 comprar nueva lámpara
 fin si
 fin si
 fin si
 Fin Programa

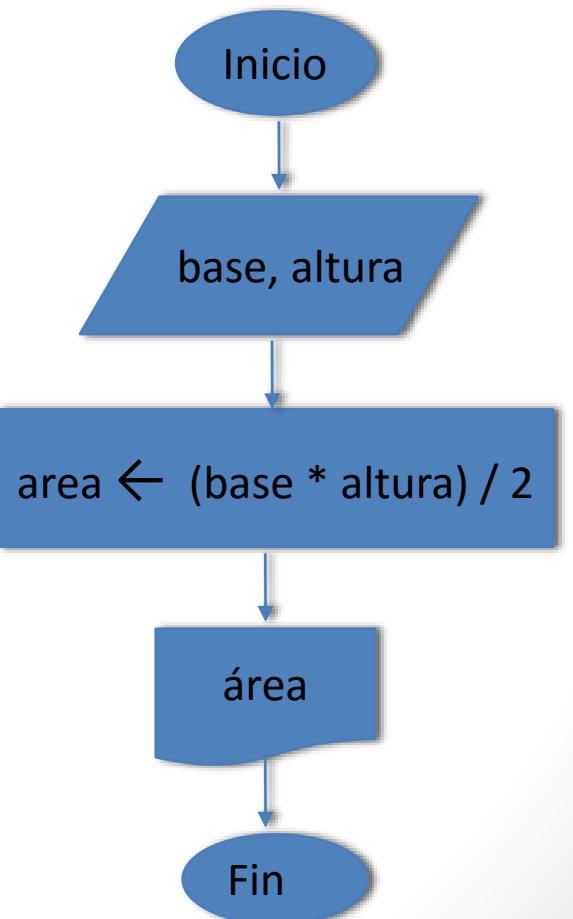


Ejemplo 01

- Calcular el área de un triangulo. Se recibe como entrada la base y la altura.

```

1 Algoritmo EJEMPLO_01
2   Leer base, altura;
3   area<- (base*altura) / 2 ;
4   Escribir area;
5 FinAlgoritmo
```



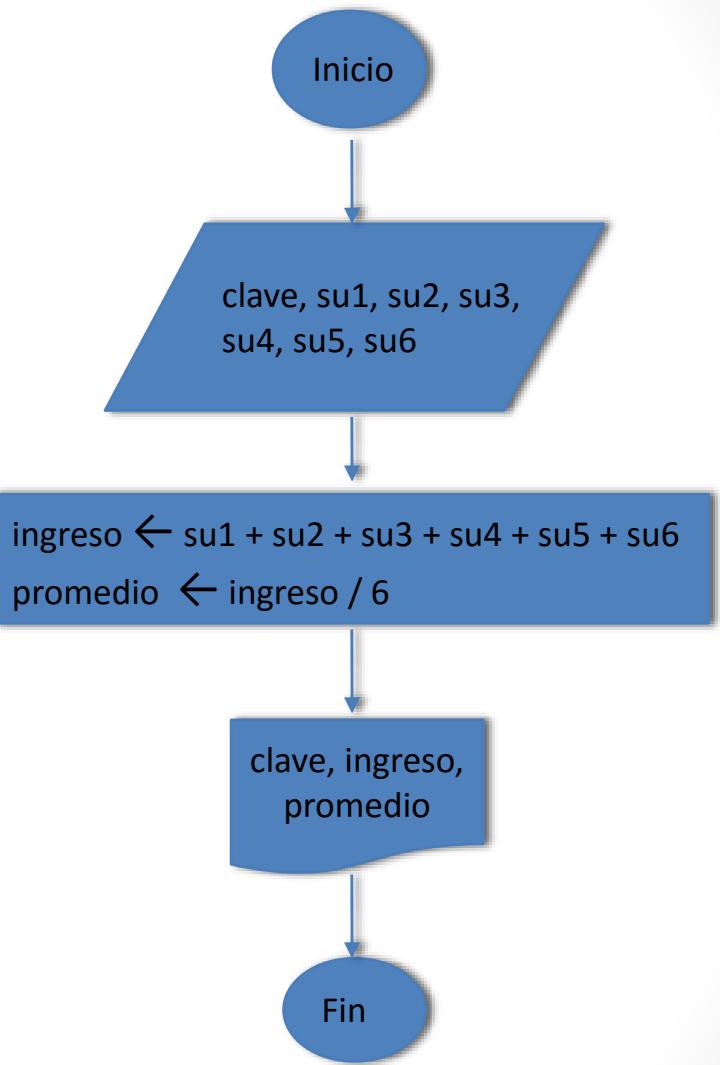
Ejemplo 02

- Construir un algoritmo que, al recibir como entrada una clave de un empleado y los seis primeros sueldos del año de este, calcule el ingreso total semestral y el promedio mensual para el empleado, finalmente se imprimirá su clave, el ingreso total y el promedio mensual.



Ejemplo 02

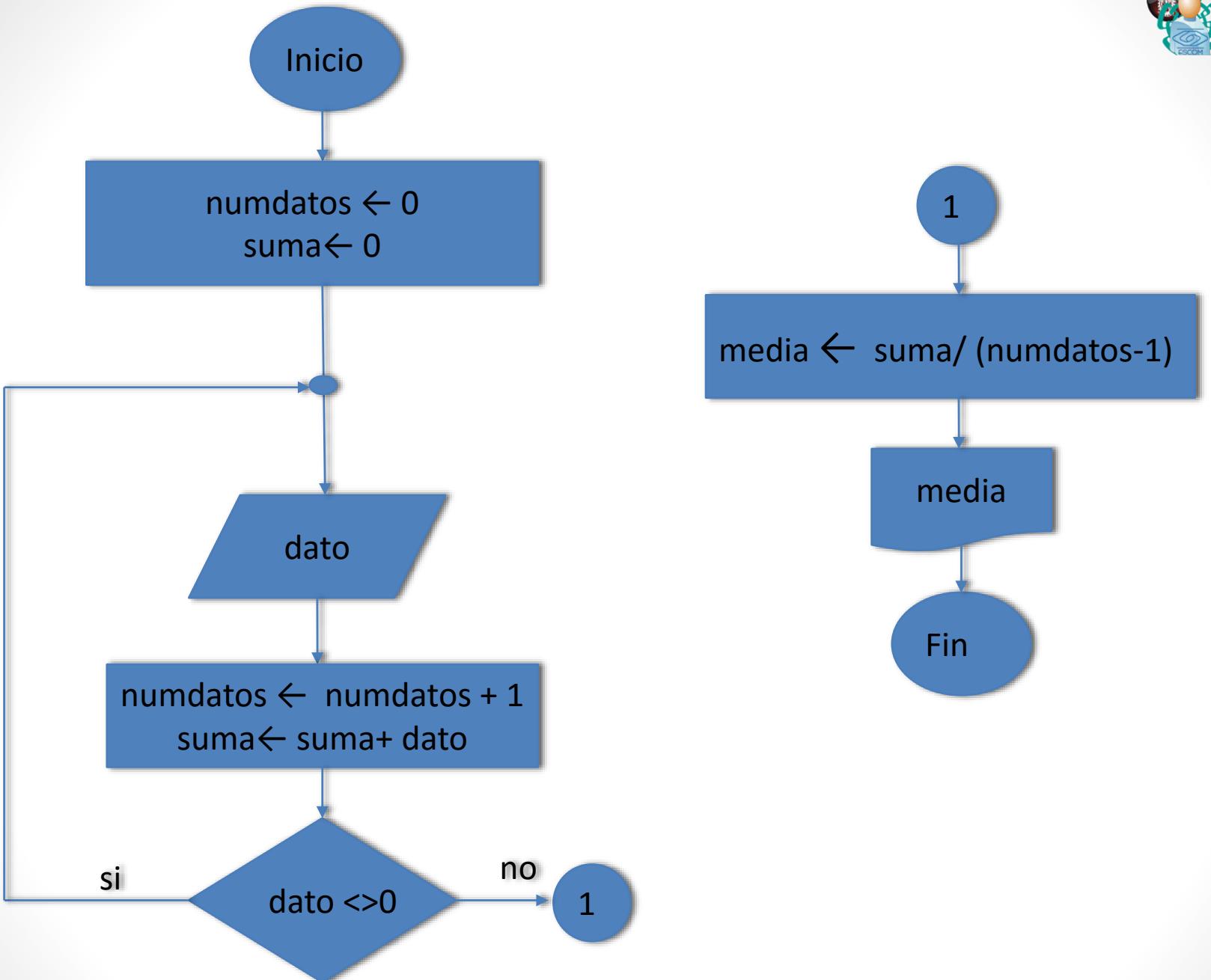
```
1 Algoritmo EJEMPLO_02
2   Leer clave,su1,su2,su3,su4,su5,su6;
3   ingreso<-su1+su2+su3+su4+su5+su6;
4   promedio<-ingreso/6;
5   Escribir clave,ingreso,promedio;
6 FinAlgoritmo
```



Ejemplo 03

- Calcular la media de una serie de números positivos, suponiendo que los datos se introducen uno a uno.
 - Un valor de cero como entrada indicará que se ha alcanzado el final de la serie de números positivos.





```
1  Algoritmo EJEMPLO_03
2      numdatos=0;
3      suma=0;
4  Repetir
5      Leer dato;
6      numdatos<-numdatos+1;
7      suma<-suma+dato;
8  Hasta Que dato=0;
9      media<-suma/(numdatos-1);
10 Escribir media;
11 FinAlgoritmo
```



Ejemplo 04

- Ordenamiento por Selección

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Buscar el mínimo elemento de la lista

Intercambiarlo con el primero

Buscar el siguiente mínimo en el resto de la lista

Intercambiarlo con el segundo

Y en general:

 Buscar el mínimo elemento entre una posición i y el final de la lista

 Intercambiar el mínimo con el elemento de la posición i .



```

//Ordenamiento por selección
i=1
mientras i<=n-1 hacer
    minimo = i;
    j=i+1
    mientras j<=n
        si A[j] < A[minimo] entonces
            minimo = j
        fin si
        j=j+1
    fin mientras
    //intercambiar(A[i], A[minimo])
    aux<-A[minimo]
    A[minimo]<-A[i]
    A[i]<-aux
    i=i+1
fin mientras

```

Algoritmo Selección (Mientras)

Algoritmo Selección (Para)

```

//Ordenamiento por selección
para i=1 hasta n-1
    minimo = i;
    para j=i+1 hasta n
        si A[j] < A[minimo] entonces
            minimo = j
        fin si
    fin para
    //intercambiar(A[i], A[minimo])
    aux<-A[minimo]
    A[minimo]<-A[i]
    A[i]<-aux
fin para

```



Representación de Algoritmos en Pseudocódigo

- **Asignación**

```
x←y  
y→x
```

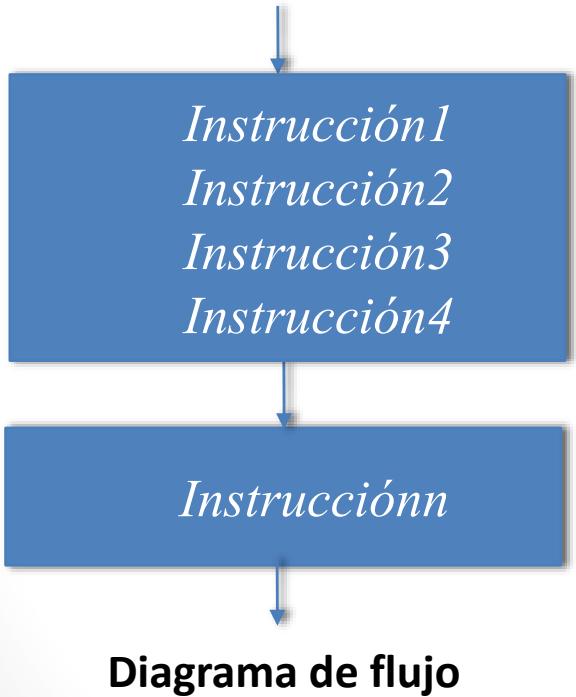
- **Variables**

```
volumen←Π r2h  
resultado←sin(a)
```



- **Estructuras de control de flujo**

- **Secuencial**



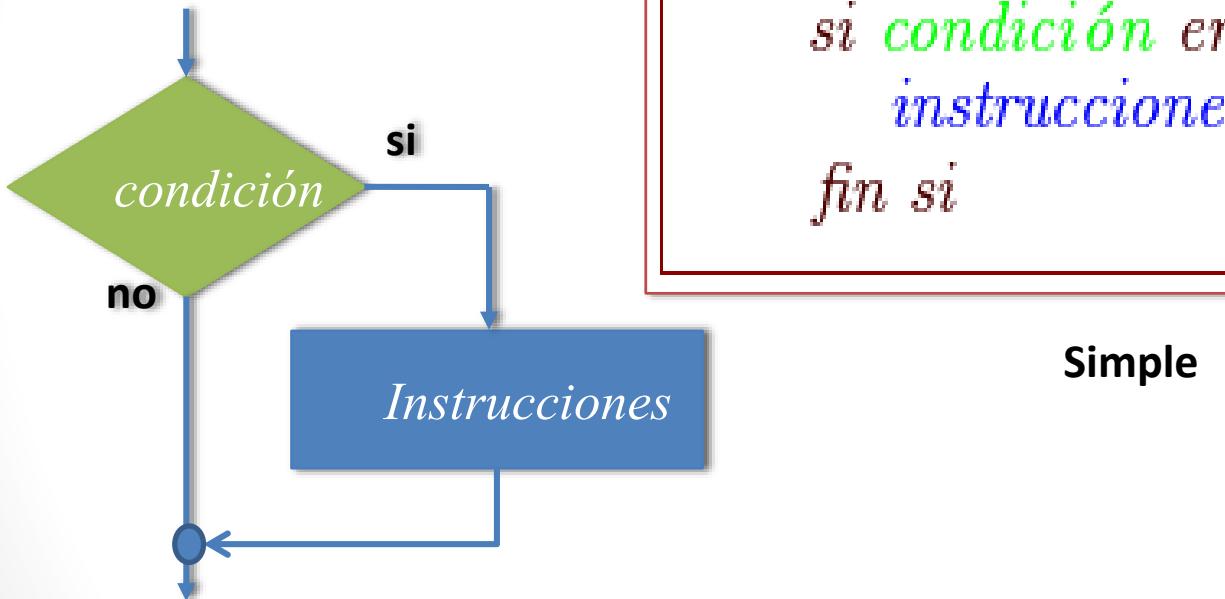
instrucción₁
instrucción₂
instrucción₃
...
instrucción_n

Pseudocódigo



• Estructuras de control de flujo

• Selectiva



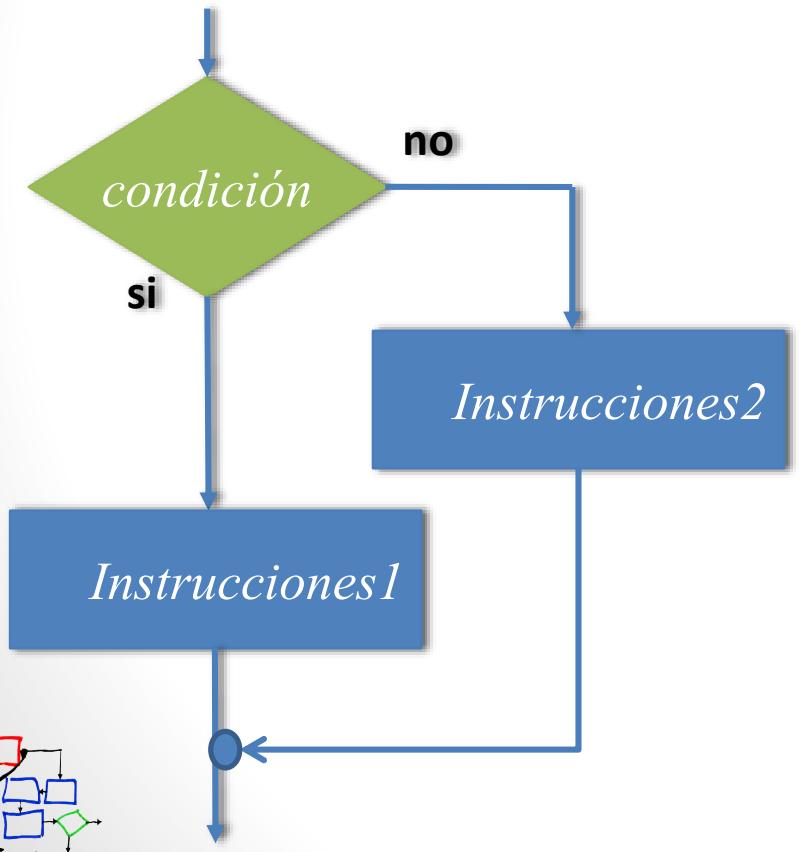
*si condición entonces
instrucciones
fin si*

Simple



- # Estructuras de control de flujo

- ## Selectiva

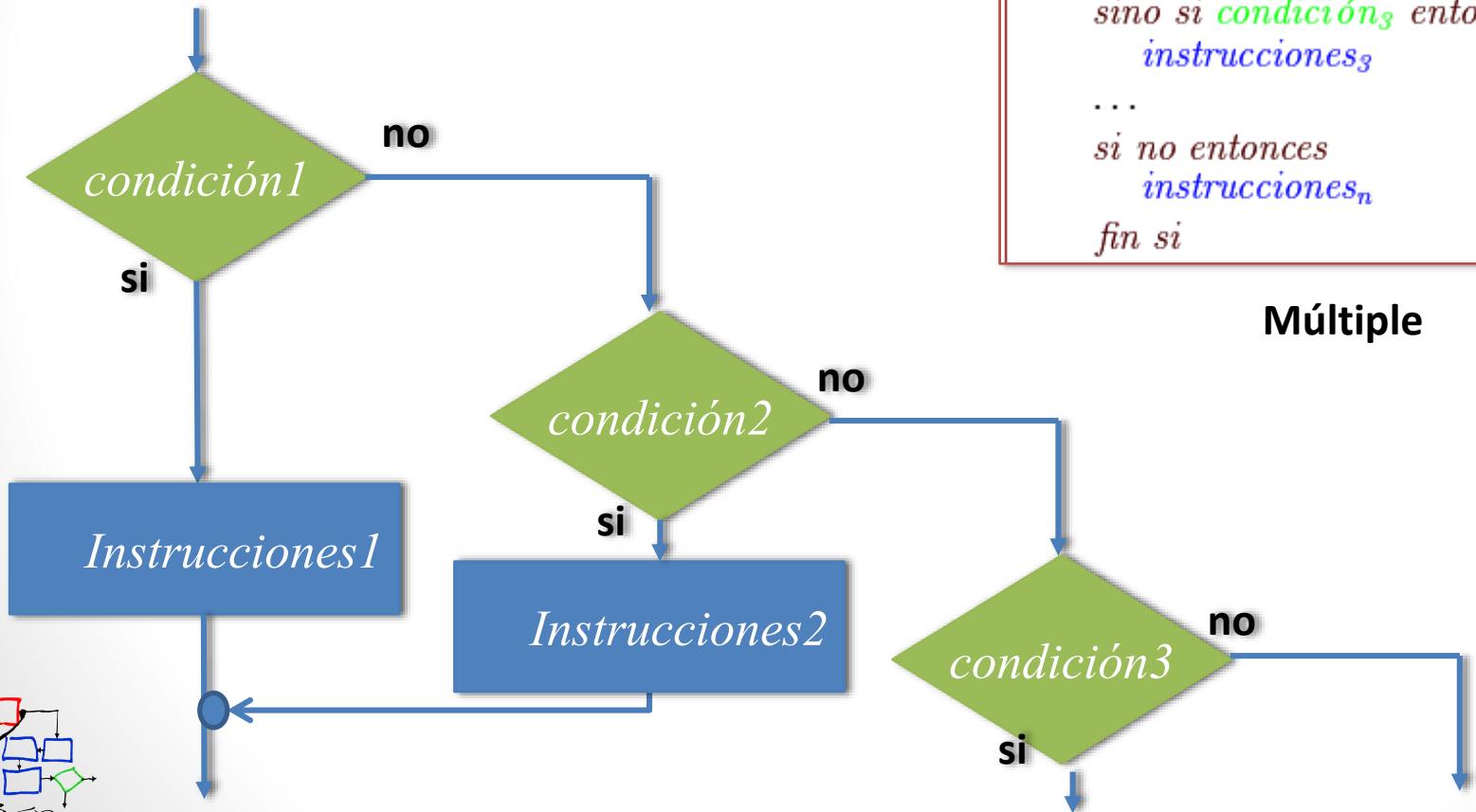


*si condición entonces
 instrucciones₁
 si no entonces
 instrucciones₂
 fin si*

Doble

- **Estructuras de control de flujo**

- **Selectiva**

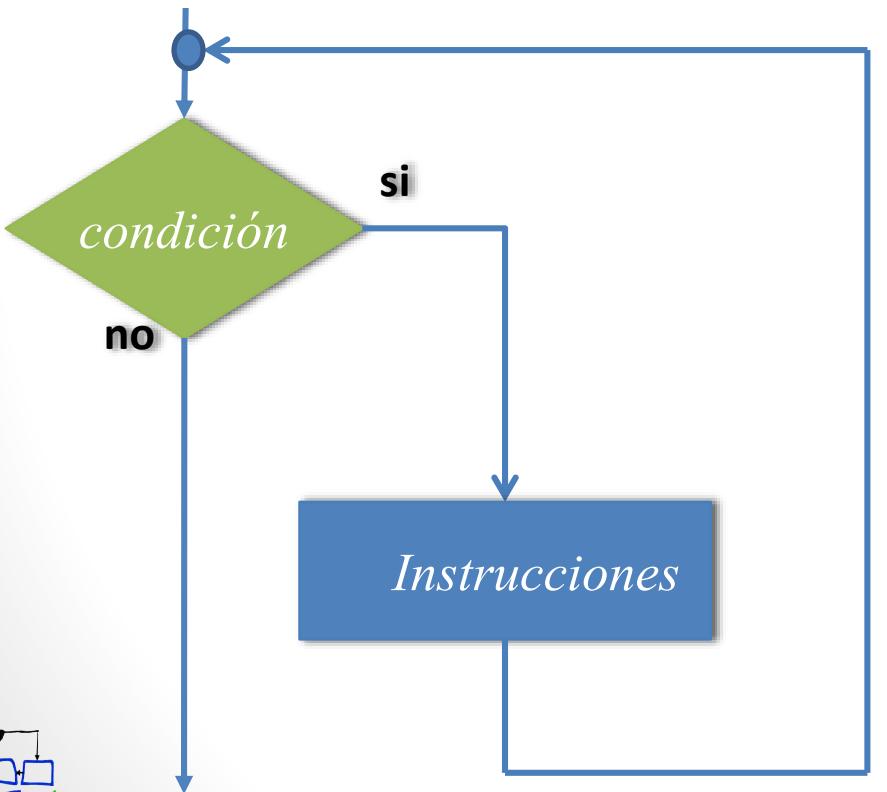


*si condición₁ entonces
 instrucciones₁
 sino si condición₂ entonces
 instrucciones₂
 sino si condición₃ entonces
 instrucciones₃
 ...
 si no entonces
 instrucciones_n
 fin si*

Múltiple

- **Estructuras de control de flujo**

- **Iterativa**

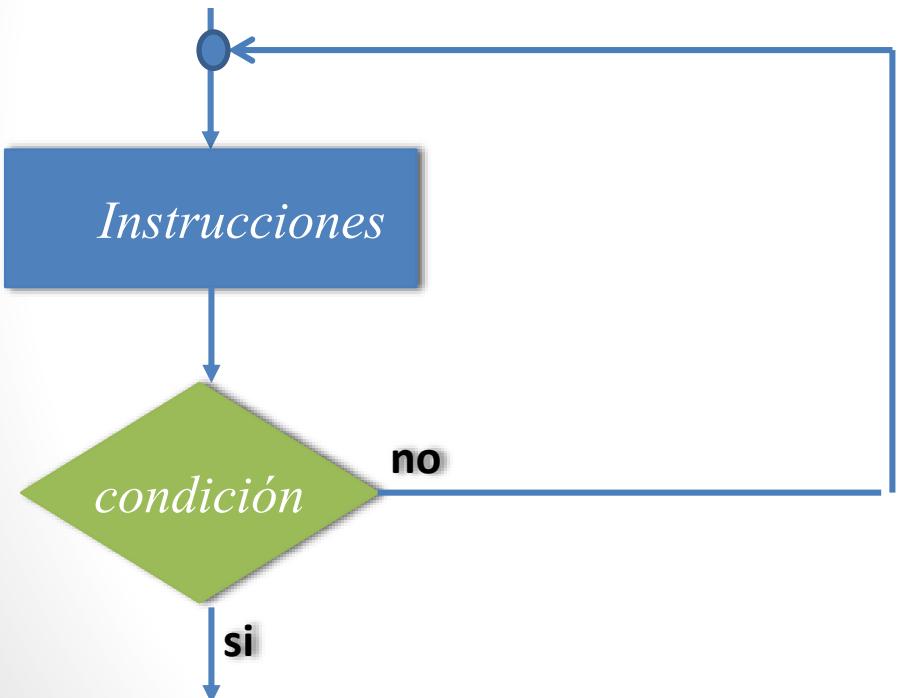


*mientras condición hacer
instrucciones
fin mientras*

Mientras

• Estructuras de control de flujo

• Iterativa



*repetir
instrucciones
hasta que condición*

Repetir

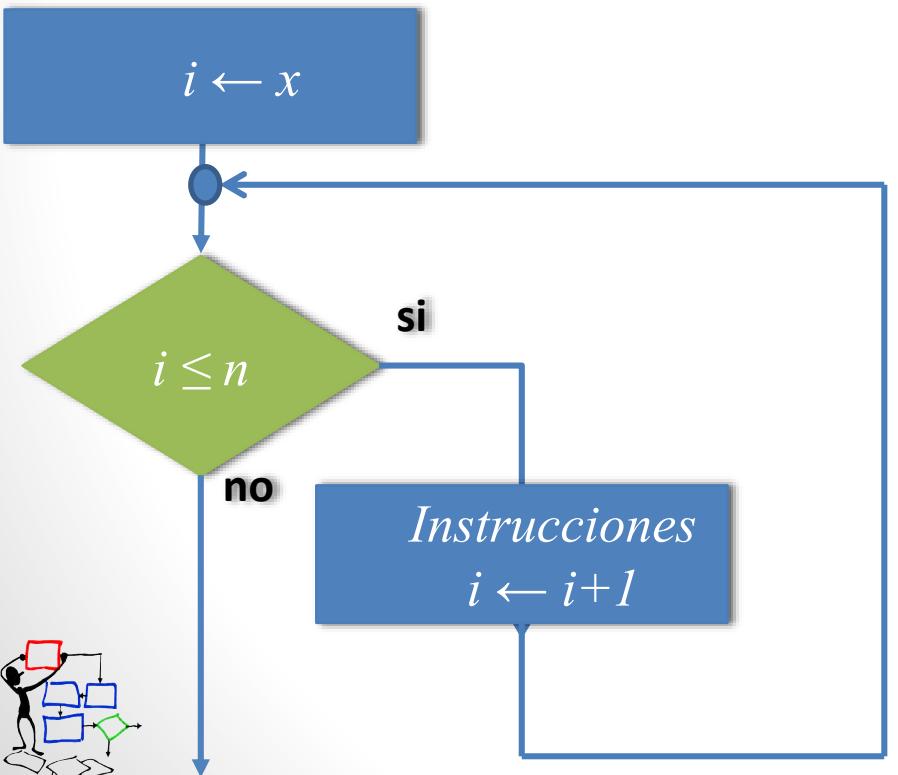
*instrucciones
mientras \neg (condición) hacer
instrucciones
fin mientras*

Mientras ≈ Repetir



- # Estructuras de control de flujo

- ## Iterativa



*para $i \leftarrow x$ hasta n hacer
 instrucciones
 fin para*

Para

*$i \leftarrow x$
 mientras $i \leq n$ hacer
 instrucciones
 $i \leftarrow i + 1$
 fin mientras*

Para → mientras

• Anidamiento

```

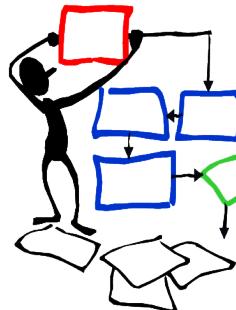
procedimiento Ordenar (L)
  ↑ //Comentario : L = (L1,L2,...,Ln) es una lista con n elementos//
  k ← 0
  repetir
    ↑ intercambio ← falso
    k ← k + 1
    para i ← 1 hasta n - k hacer
      ↑ si Li > Li+1 entonces
      ↑   ↑ intercambiar (Li,Li+1)
      ↑   intercambio ← verdadero
      ↑   fin si
      ↓ fin para
    ↓ hasta que intercambio = falso
  fin procedimiento
  
```





Instituto Politécnico Nacional

Escuela Superior de Cómputo



Algoritmia y programación estructurada

Tema 03: Resolución de problemas computables

M. en C. Edgardo Adrián Franco Martínez

<http://www.eafranco.com>

edfrancom@ipn.mx

[@edfrancom](https://twitter.com/edfrancom) [@edgardoадrianfranco](https://facebook.com/edgardo.adrian.franco)



Contenido

- Recordando la razón de ser de las computadoras
 - ¿Qué es un programa de computadora?
 - ¿Qué es un algoritmos computacional?
 - ¿Qué tipos de instrucciones puede realizar una computadora?
- Resolución de problemas
 - ¿Qué es un problema?
- Recomendación para lograr resolver un problema mediante un programa computacional
 - El camino al éxito
 - Errores típicos al buscar una solución computable
- Como representar una solución
 - Una instrucción computable tras otra
 - Una instrucción u otra según algún criterio
 - Una instrucción o muchas mientras algo suceda



Recordando la razón de ser de las Computadoras

- Debido a su gran velocidad para realizar cálculos, almacenamiento, procesamiento y recuperación de información de manera precisa podemos decir que:

“La razón de ser de una computadora es poder resolver problemas capaces de ser modelados y representados en datos coherentes y ordenados (información**) procesables para una computadora, apoyándose de su gran velocidad para tratar la información y su capacidad de seguir una serie de pasos programados con anterioridad”.**



¿Dónde se plasma la solución a un problema a ser resuelto por una computadora?



En un programa de computadora



¿Qué es un Programa de Computadora?

- Un **programa de computadora** es una serie de **instrucciones** a ser realizadas por la computadora **para darle una solución a un problema** la cuáles han sido **descritas por un programador** bajo un **lenguaje de programación**.



¿Cuál es la esencia de un programa de computadora?

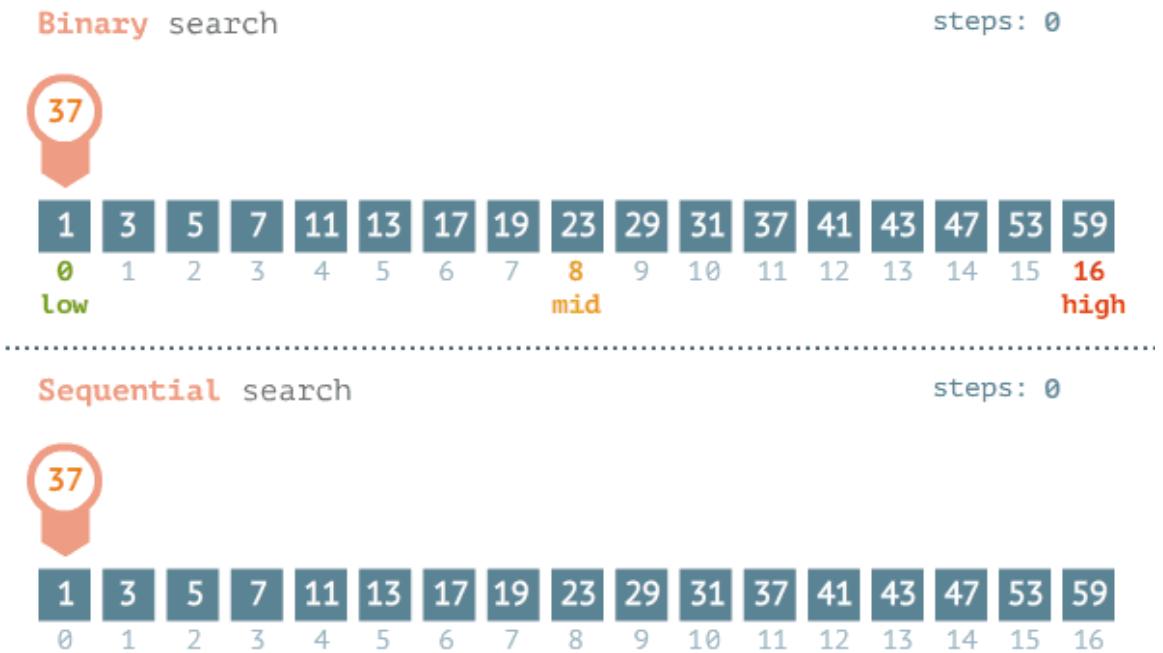
```
var scrollHeight = element.clientHeight + 0.02 ^ window
element.scrollHeight);
window.scroll(0, scrollHeight);
```

Uno o varios algoritmos plasmados en el

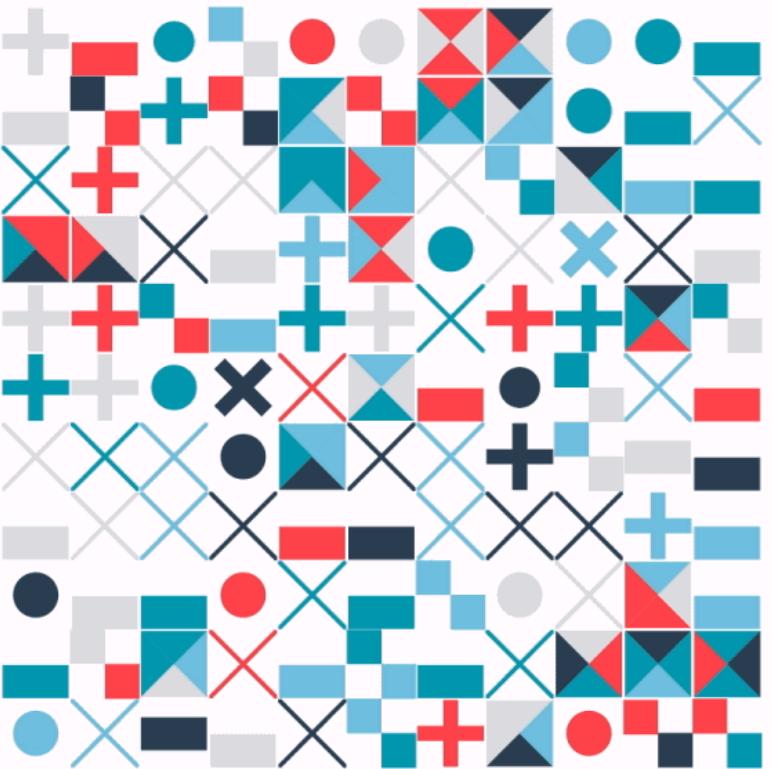


¿Qué es un algoritmo computacional?

- Es un **conjunto ordenado y finito de operaciones inherentes a un cómputo** que permite hallar la **solución de un problema**.



¿Qué tipos de instrucciones puede realizar una computadora?



1. Instrucciones en secuencia
2. Recibir o mostrar datos (Entrada / Salida)
3. Saltar a otras instrucciones
4. Almacenar, editar y eliminar datos en la memoria
5. Operar matemáticamente con los datos



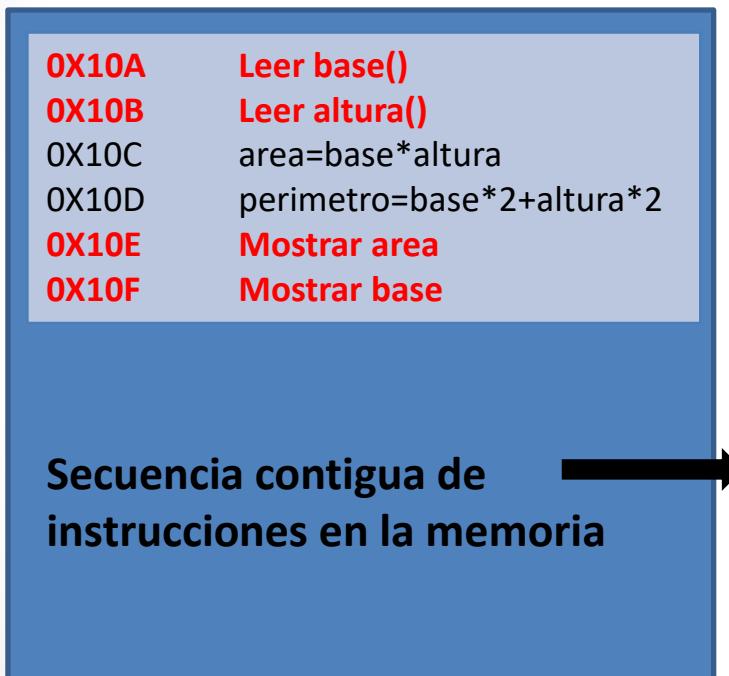
¿Qué tipos de instrucciones puede realizar una computadora?

Al pensar en la idea de crear algoritmos a ser ejecutados mediante una computadora es necesario entender cuales son las instrucciones capaces de realizar una computadora, por lo que podemos decir de manera general que estas son las capacidades de procesamiento de una computadora:

- 1. Procesar instrucciones en secuencia** (*Las instrucciones se colocan secuencialmente en la memoria y se ejecutan en ese orden*)
- 2. Recibir o mostrar datos** (*Una computadora cuenta dispositivos para la entrada y salida de información capaz de ser colocada en la memoria para leerlos o tomarlos y mostrarlos*)
- 3. Saltar a otras instrucciones** (*Continuar una secuencia de instrucciones en otra dirección de memoria no continua*)
- 4. Almacenar, editar o eliminar datos en la memoria** (*Guardar, editar, mover y eliminar valores en una o más direcciones de memoria*)
- 5. Operar con números** almacenados en la memoria (*Una computadora cuenta con la posibilidad de realizar operaciones numéricas en base 2 con las variables colocadas en la memoria*)



1. Procesar instrucciones en secuencia (*Las instrucciones se colocan secuencialmente en la memoria y se ejecutan en ese orden*)
2. Recibir o mostrar datos (*Una computadora cuenta dispositivos para la entrada y salida de información capaz de ser colocada en la memoria para leerlos o tomarlos y mostrarlos*)



00001A1E 4D 4B	LDR	R3, -(stdout_ptr - 0xC000)
00001A20 E3 58	LDR	R3, [R4,R3] ; stdout
00001A22 1B 68	LDR	R3, [R3]
00001A24 18 46	MOV	R0, R3 ; stream
00001A26 FF F7 52 EA	BLX	fileno
00001A2A 03 46	MOU	R3, R0
00001A2C 18 46	MOU	R0, R3
00001A2E 4A 4B	LDR	R3, -(a1ChangeDisplay - 0x1)
00001A30 7B 44	ADD	R3, PC ; "1" Change displ
00001A32 19 46	MOU	R1, R3
00001A34 00 F8 34 FB	BL	print
00001A38 48 4B	LDR	R3, -(stdin_ptr - 0xC000)
00001A3A E3 58	LDR	R3, [R4,R3] ; stdin
00001A3C 1B 68	LDR	R3, [R3]
00001A3E 18 46	MOV	R0, R3 ; stream
00001A40 FF F7 44 EA	BLX	fileno
00001A44 02 46	MOU	R2, R0
00001A46 07 F5 43 63	ADD.W	R3, R7, #0xC30
00001A4A 18 46	MOU	R0, R2
00001A4C 19 46	MOU	R1, R3
00001A4E 4F F8 02 02	MOU.W	R2, #2
00001A52 4F F8 0A 03	MOU.W	R3, #0xA
00001A56 00 F8 77 FB	BL	read
00001A5A 03 46	MOU	R3, R0
00001A5C 00 2B	CMP	R3, #0

ASM	HEX
mov eax,0x4	b8 04 00 00 00
mov ebx,0x1	bb 01 00 00 00
mov ecx,0x80490a0	b9 a0 90 04 08
mov edx,0x26	ba 26 00 00 00
int 0x80	cd 80
mov eax,0x1	b8 01 00 00 00
xor ebx,ebx	31 db
int 0x80	cd 80



3. Saltar a otras instrucciones (Continuar una secuencia de instrucciones en otra dirección de memoria no continua)

0X10A	Leer base()
0X10B	Leer altura()
0X10C	area=base*altura
0X10D	perimetro=base*2+altura*2
0X10E	Mostrar area
0X10F	Mostrar base
0X110	Salta 0X10A

	1	2	3	4
1	* = \$6000			
2	inicio			
3	lda #\$00			
4	sta \$2c6			
5	lda #\$0F			
6	sta \$2c5			
7	lda #'H			
8	jsr \$f2b0			
9	lda #'0			
10	jsr \$f2b0			
11	lda #'L			
12	jsr \$f2b0			
13	lda #'A			
14	jsr \$f2b0			
15	lda #155 ;Es un RETURN			
16	jsr \$f2b0			
17	jmp inicio			
18	*=\$2e0			
19	.word inicio			



4. Almacenar, editar o eliminar datos en la memoria (*Guardar, editar, mover y eliminar valores en una o más direcciones de memoria*)
5. Operar con números almacenados en la memoria (*Una computadora cuenta con la posibilidad de realizar operaciones numéricas en base 2 con las variables colocadas en la memoria*)

0X10A	Leer base()	~ 0x200 = Entrada
0X10B	Leer altura()	~ 0x203 = Entrada
0X10C	area=base*altura	~ 0x206 = 0x200 * 0x203
0X10D	perimetro=base*2+altura*2	~ 0x208 = 0x002
		~ 0x20A = 0x200 * 0x208
		~ 0x20B = 0x200 * 0x203
		~ 0x20C = 0x20A + 0x20B
0X10E	Mostrar área	~ Salida = 0x206
0X10F	Mostrar base	~ Salida = 0x206
0X110	Salta 0X10A	

```
int x = 79;  
x = x + 1;
```

START



Resolución de problemas

- La razón de ser de un programa computacional es modelar las soluciones a problemas modelados en un contexto informático (procesamiento de la información) por medio de instrucciones computables, para lograr este objetivo es necesario crear abstracciones del problema y con base en una filosofía de programación (paradigma) y un lenguaje de programación plasmar una solución a cada abstracción realizada.

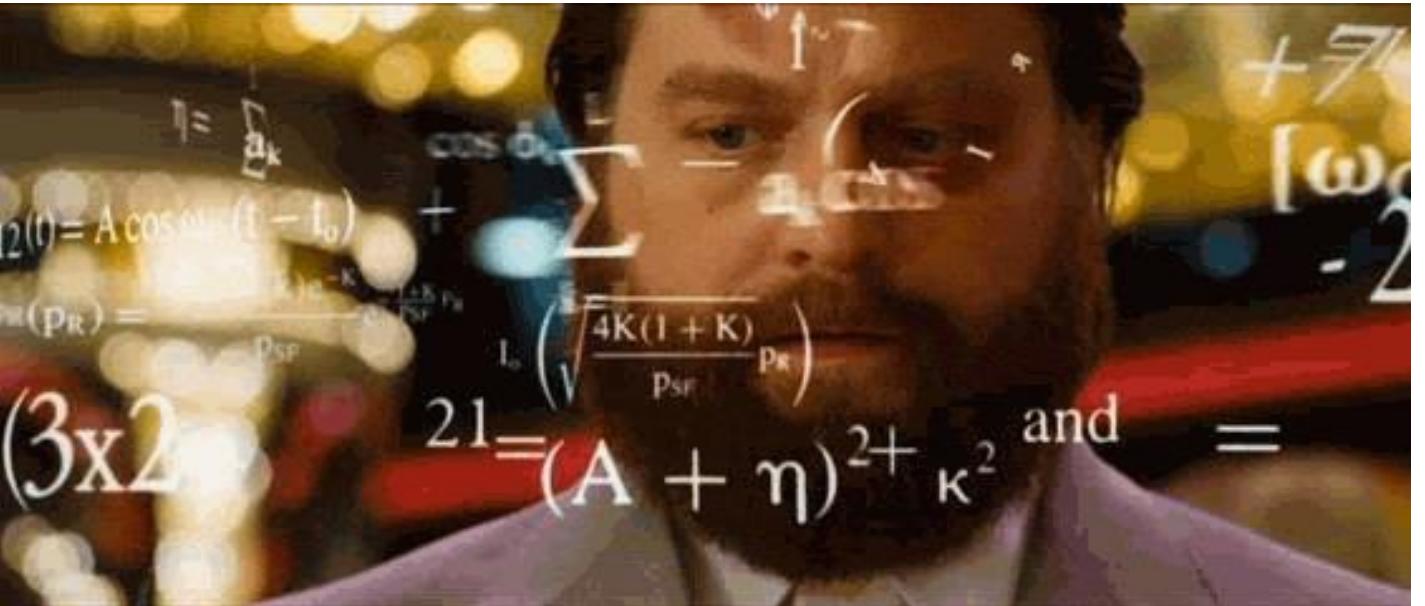


Abstracción: **Acto mental** en el que conceptualmente se aísla un objeto o una propiedad de un objeto para su análisis y entendimiento.

*La abstracción depende totalmente de los conocimientos del sujeto



- ¿Qué es un problema?



¿Qué es un problema?

- **Un problema es una cuestión discutible que hay que resolver o a la que se busca una explicación, solución o dato.**



(15)



El camino al éxito

- Una de los mayores deseos de toda persona es **poder resolver los problemas a los que se enfrenta** ser capaz de dar **las soluciones a los problemas** siempre será gratificante.
- El éxito se da al realizar buenas **abstracciones del problema** y con base en una **filosofía clara (forma de ver la vida)** se puede **plasmar una solución** a cada abstracción y llevarla a cabo para determinar si hemos logrado resolver la cuestión planteada.

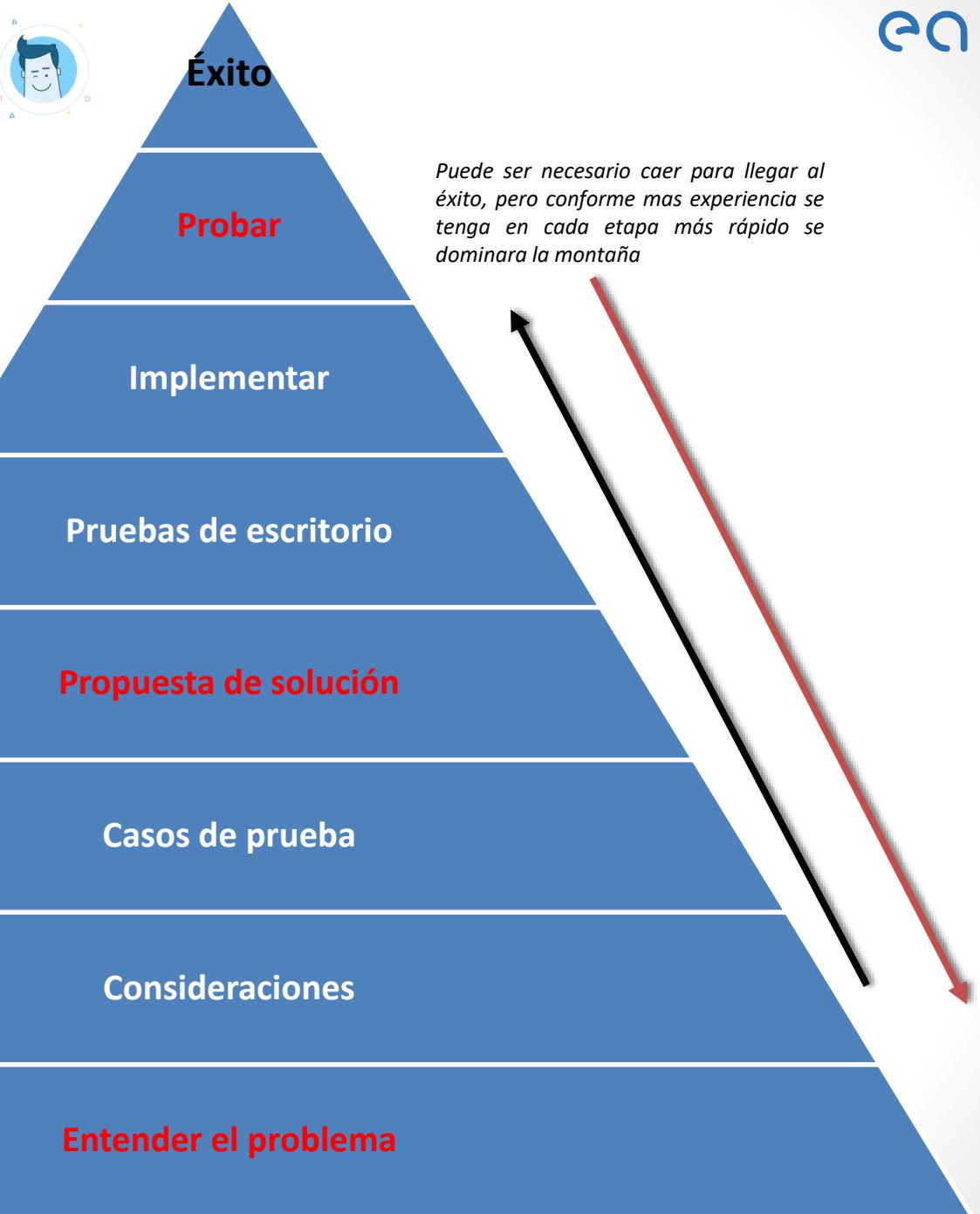


(16)



*Y esto no es coaching, pero se parece <https://es.wikihow.com/resolver-problemas>

Basándose en la experiencia de personas que han dado solución a muchos problemas computables se recomienda



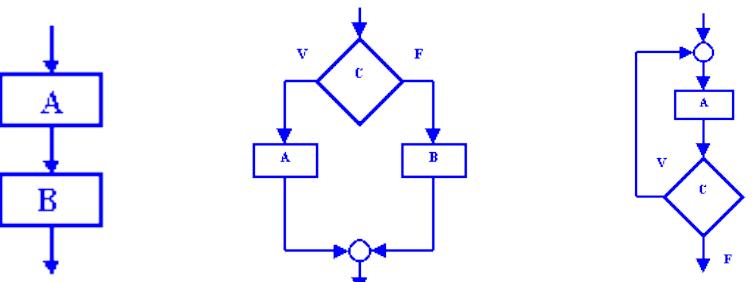
Errores típicos al buscar una solución computable

- 1. No entender completamente el problema:** Puede llevarnos a convertir el problema en algo que no es.
- 2. Tratar de ver el problema exactamente igual que un resuelto anteriormente:** A veces problemas resueltos previamente pueden servirnos para dar solución a uno nuevo, pero nunca serán iguales al 100% por lo que debe tenerse cuidado en distinguir las diferencias.
- 3. Buscar una solución 100% perfecta y optima desde el inicio:** No será posible desde la idea inicial ser optimo en la solución, esta se perfecciona conforme se va formalizando su implementación.
- 4. No probar con nuevas instancias del problema:** Tratar de dar solución a los ejemplos sencillos únicamente y no plantear nuevas instancias del problema puede caer en encontrar una solución parcial al problema.
- 5. No considerar las posibilidades de la computadora o las limitantes de las tecnologías** donde se implementara la solución desde el inicio puede complicar mucho la búsqueda o su implementación.
- 6. No considerar soluciones optimas o paradigmas de solución** de autores relevantes o tratar de inventar el hilo negro: El **desconocimiento de algoritmos** produce trabajar doble en la búsqueda de soluciones a un problema o llegar a resultados poco eficientes.



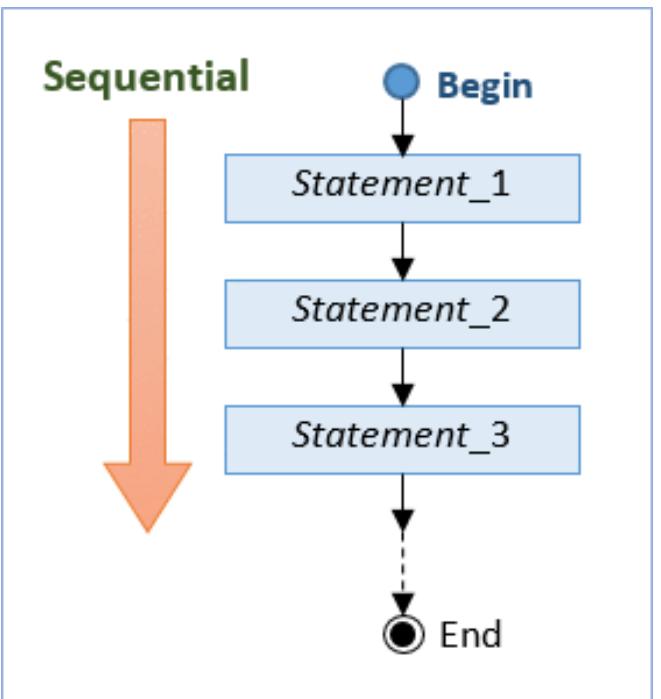
Como representar un solución

- En esencia considerando las posibilidades y la arquitectura básica de una computadora podemos decir que una solución computable a un problema debe considerar integrar algoritmos con objetivos claramente definidos. Cada uno de ellos pueden ser descritos como un grupo de instrucciones que pueden llevarse a cabo bajo tres estructuras básicas de flujo en la computadora:
 1. Una instrucción computable tras otra
 2. Una instrucción u otra según algún criterio
 3. Una instrucción o muchas mientras algo suceda



Una instrucción computable tras otra

- La secuencia simple de instrucciones es la estructura de programación más natural, pues consiste solamente en la colocación de las instrucciones en un orden secuencial, lógica y preciso. Trabajando aisladamente, las instrucciones se ejecutan una única vez, una después de la otra, sin cambio en el flujo de ejecución.

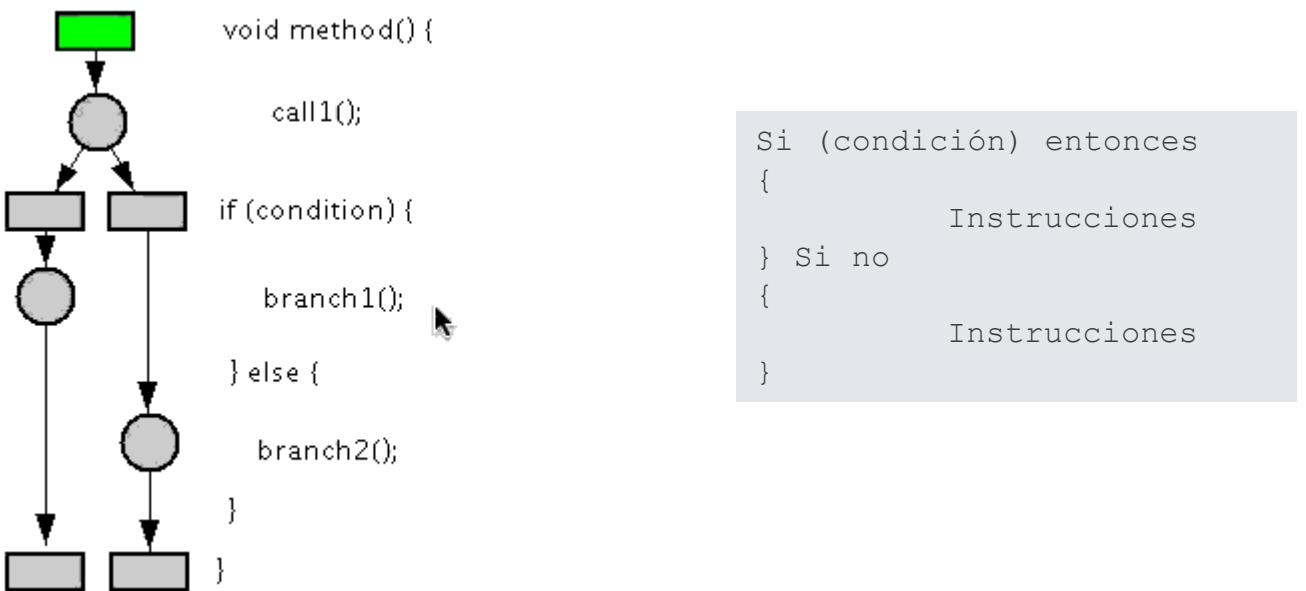


```
Inicio
Instrucción 1
Instrucción 2
Instrucción 3
. . .
Instrucción N Fin
```



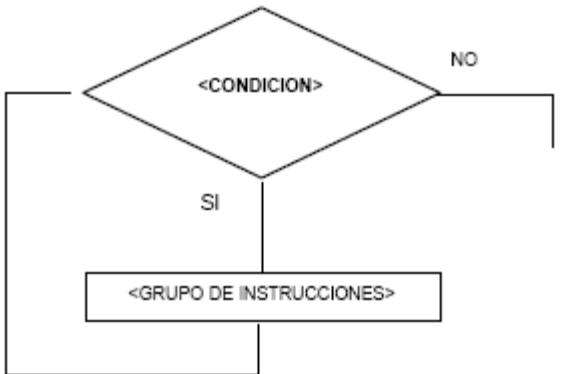
Una instrucción u otra según algún criterio

- Las estructuras selectivas son sentencias de programación que nos permiten elegir entre dos o más opciones o caminos de instrucciones. La elección se hace mediante la evaluación de un criterio.



Una instrucción o muchas mientras algo suceda

- Las estructuras selectivas son sentencias de programación que nos permiten elegir entre dos a más opciones o caminos de instrucciones. La elección se hace mediante la evaluación de un criterio.



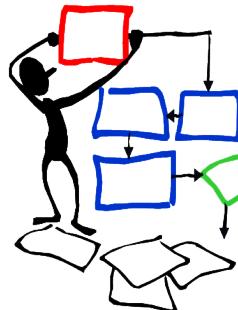
```
Mientras (condición) hacer
{
    Instrucciones
}
```





Instituto Politécnico Nacional

Escuela Superior de Cómputo



Algoritmia y programación estructurada

Tema 04: Recursividad

M. en C. Edgardo Adrián Franco Martínez

<http://www.eafranco.com>

edfrancom@ipn.mx

[@edfrancom](https://twitter.com/edfrancom) [edgaroadrianfrancocom](https://facebook.com/edgaroadrianfrancocom)



Contenido

- Introducción
 - Recursión
 - Recursividad
 - Programación recursiva
 - Ejemplo: Factorial
 - ¿Por qué escribir programas recursivos?
 - ¿Cómo escribir una función en forma recursiva?
 - ¿Qué pasa si se hace una llamada recursiva que no termina?
 - ¿Cuándo usar recursividad?
 - ¿Cuándo NO usar recursividad?
- 
- Recursión vs. Iteración

Introducción

- La recursión es una forma de pensar soluciones computables las cuales están en términos de si mismas. O desarrollar un conjunto de espacios de búsqueda de un problema desarrollando todos los caminos posibles de manera recursiva.

“Para entender la recursividad lo primero que hay que entender es la recursividad”

Describelle a un ciego la siguiente imagen

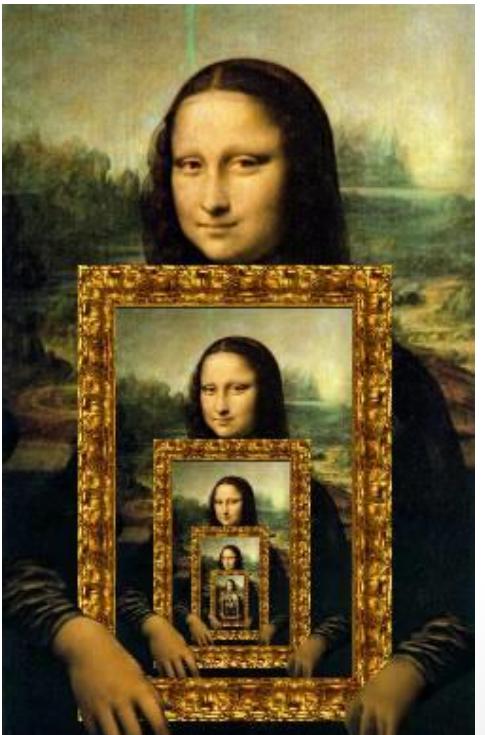


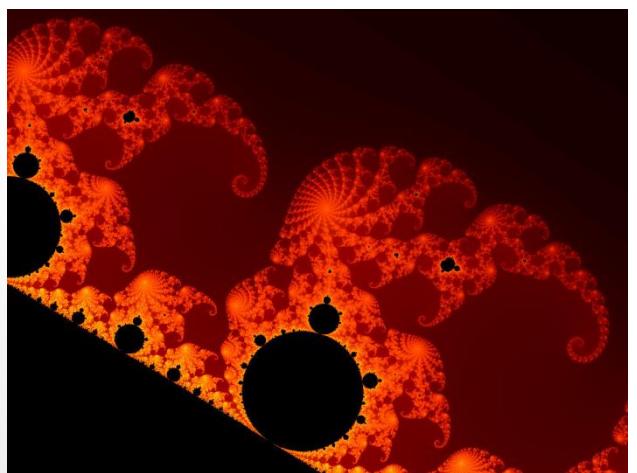
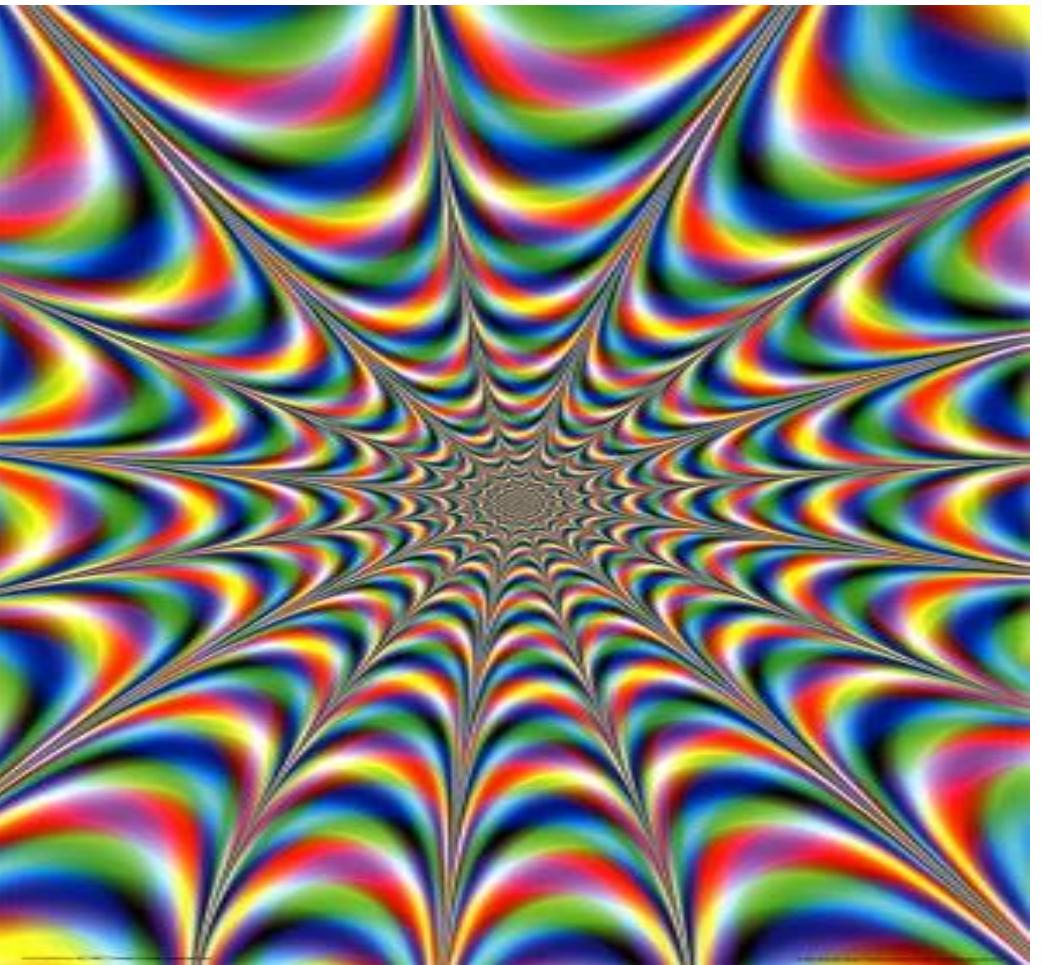
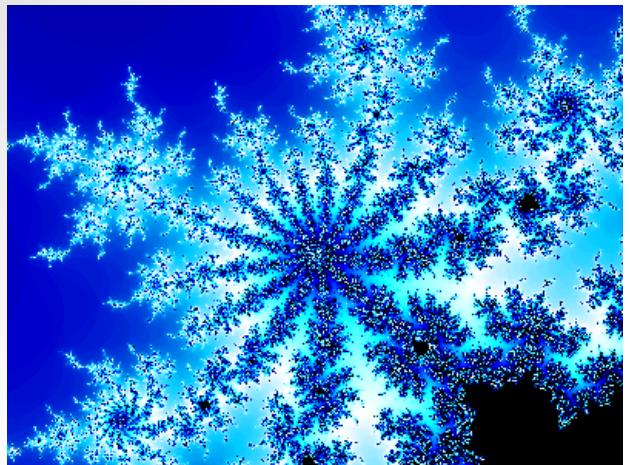
(3)



Recursión

- La **recursión** o **recursividad** es un concepto amplio, con muchas variantes, y difícil de precisar con pocas palabras.





- La Matrushka es una artesanía tradicional rusa. Es una muñeca de madera que contiene otra muñeca más pequeña dentro de sí. Esta muñeca, también contiene otra muñeca dentro. Y así, una dentro de otra.



[6]



Recursividad

- La **recursividad** es un concepto fundamental en matemáticas y en computación.
- Es una **alternativa** diferente para **implementar** estructuras de repetición (**iteración**).
- Se puede usar en toda situación en la cual la solución pueda ser expresada como una secuencia de movimientos, pasos o transformaciones gobernadas por un conjunto de **reglas no ambiguas**.



- La recursividad es un recurso muy poderoso que **permite expresar soluciones simples y naturales a ciertos tipos de problemas.** Es importante considerar que no todos los problemas son naturalmente recursivos.
- Un **objeto recursivo** es aquel que **aparece en la definición de si mismo**, así como el que *se llama a sí mismo*.



Programación recursiva

- Las funciones recursivas se componen de:
 - **Paso base:** una solución simple para un caso particular (*puede haber más de un paso base*). La secuenciación, iteración condicional y selección son estructuras válidas de control que pueden ser consideradas como enunciados.
 - **Paso recursivo:** una solución que involucra volver a utilizar la función original, con parámetros que se acercan más al paso base. Los pasos que sigue el paso recursivo son los siguientes:
 - El procedimiento se llama a sí mismo.
 - El problema se resuelve, resolviendo el mismo problema pero de tamaño menor.
 - La manera en la cual el tamaño del problema disminuye asegura que el paso base eventualmente se alcanzará.



- Asimismo, puede definirse un programa en términos recursivos, como una serie de pasos básicos, o **pasos base** (*condición de parada*), y uno más **pasos recursivo**, donde vuelve a llamarse al programa.
- En un programa recursivo, esta serie de casos recursivos debe ser finita, terminando con al menos un paso base.
- Es decir, a cada paso recursivo se reduce el número de pasos que hay que dar para terminar, llegando un momento en el que no se verifica la condición de paso a la recursividad.
- Ni el **paso base** ni el **paso recursivo** son necesariamente únicos.



Recursividad (Ejemplo: Factorial)

- Escribe un programa que calcule el **factorial (!) de un entero no negativo** :
- El factorial de n esta dado por:

$$n! = \prod_{i=1}^n i$$

$$n! = 1 \times 2 \times 3 \times 4 \times \dots \times (n - 1) \times n$$

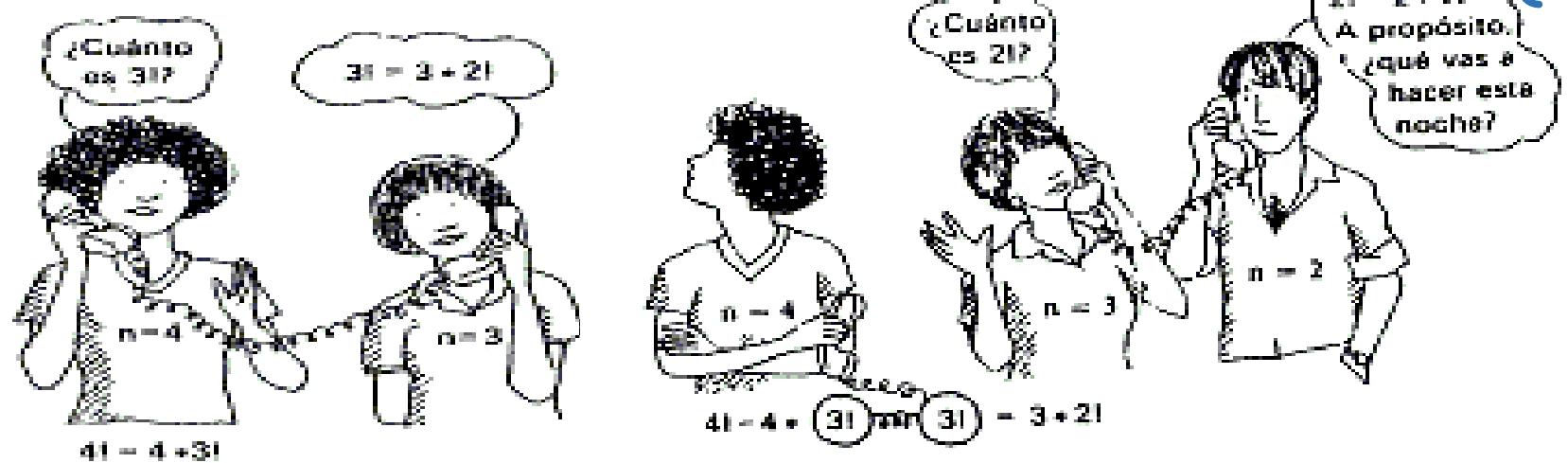


- Definición recursiva del factorial:

$$n! = \begin{cases} 1 & \rightarrow \text{si } n = 0 \\ (n - 1)! \cdot n & \rightarrow \text{si } n \geq 1 \end{cases}$$

- $0! = 1$
- $1! = 1$
- $2! = 2 \quad \Rightarrow \quad 2! = 2 * 1!$
- $3! = 6 \quad \Rightarrow \quad 3! = 3 * 2!$
- $4! = 24 \quad \Rightarrow \quad 4! = 4 * 3!$
- $5! = 120 \quad \Rightarrow \quad 5! = 5 * 4!$

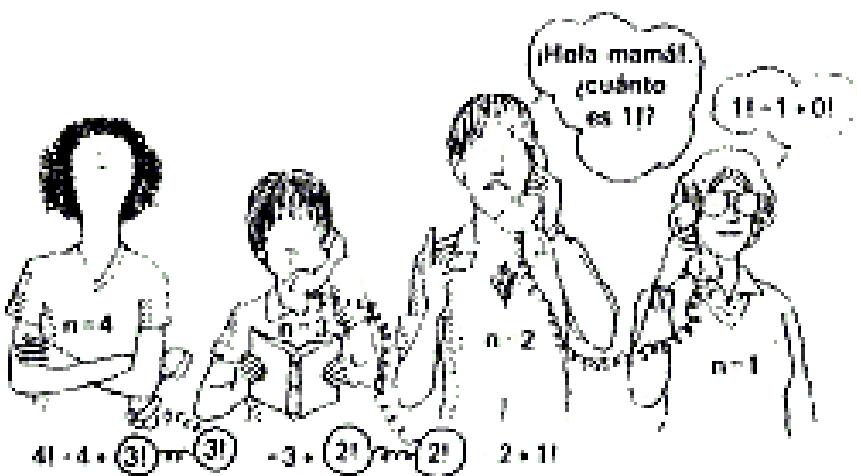


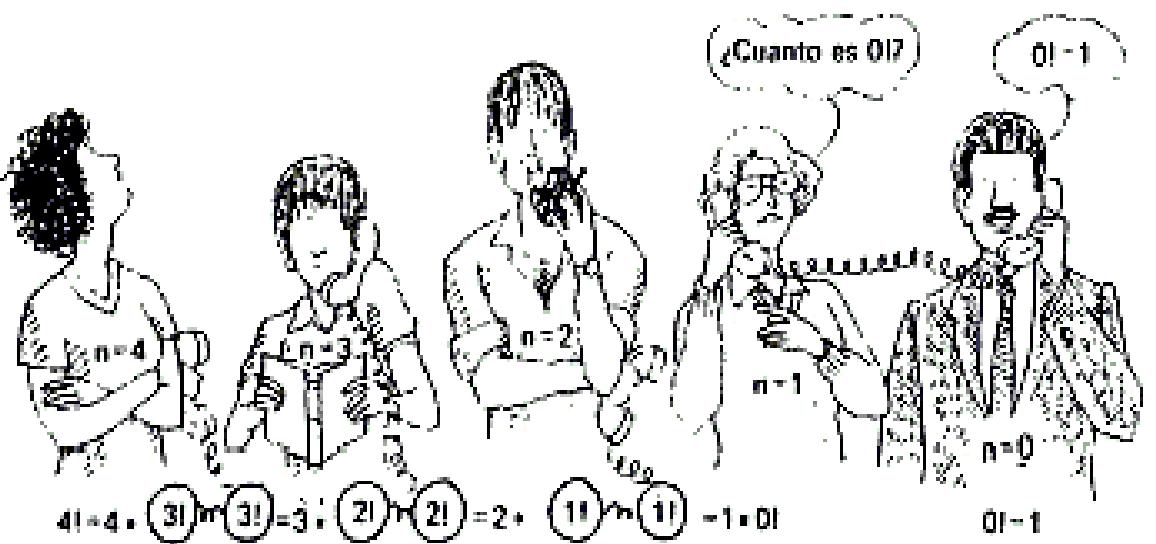


$$4! = 4 * 3! \quad n = 4$$

$$n = 3$$

$$n = 2$$





• Secuencia de factoriales.

- $0! = 1$
- $1! = 1 \rightarrow = 1 * 1 = 1 * 0!$
- $2! = 2 \rightarrow = 2 * 1 = 2 * 1!$
- $3! = 6 \rightarrow = 3 * 2 = 3 * 2!$
- $4! = 24 \rightarrow = 4 * 6 = 4 * 3!$
- $5! = 120 \rightarrow = 5 * 24 = 5 * 4!$
- ...
- $N! = \dots = N * (N - 1)!$



Secuencia que toma el factorial

$$N! = \begin{cases} 1 & \text{si } N = 0 \text{ (base)} \\ N * (N - 1)! & \text{si } N > 0 \text{ (recursión)} \end{cases}$$

- Un razonamiento recursivo tiene dos partes: la base y la regla recursiva de construcción. La base no es recursiva y es el punto tanto de partida como de terminación de la definición.



Recursividad (Ejemplo Factorial “Solución”)

- Dado un entero no negativo x, regresar el factorial de x fact:

//Entrada: n entero no negativo.

//Salida: entero.

```
int fact (int n)
{
    if (n == 0)
        return 1;
    else
        return fact(n - 1) * n ;
}
```

Es importante determinar un caso base, es decir un punto en el cual existe una condición por la cual no se requiera volver a llamar a la misma función.



Llamadas recursivas

fact (3)

3	?
---	---

n fact

fact (2)

2	?
---	---

n fact

fact (3)

3	?
---	---

n fact

fact (1)

1	?
---	---

n fact

fact (2)

2	?
---	---

n fact

fact (3)

3	?
---	---

n fact

Resultados de las llamadas recursivas

0	1
1	?
2	?
3	?

n fact

1	1
2	?
3	?

n fact

2	2
3	?

n fact

3	6
---	---

n fact



¿Por qué escribir programas recursivos?

- Son mas cercanos a la descripción matemática.
- Generalmente mas fáciles de analizar
- Se adaptan mejor a las estructuras de datos recursivas (*árboles, listas, etc.*).
- Los algoritmos recursivos ofrecen soluciones estructuradas, **modulares y elegantemente simples**.



¿Cómo escribir una función en forma recursiva?

```
<tipo_de_Regreso><nom_fnc> (<param>)
```

{

```
[declaración de variables]
```

```
[condición de salida]
```

```
[instrucciones]
```

```
[llamada a <nom_fnc> (<param>)]
```

```
return <resultado>
```

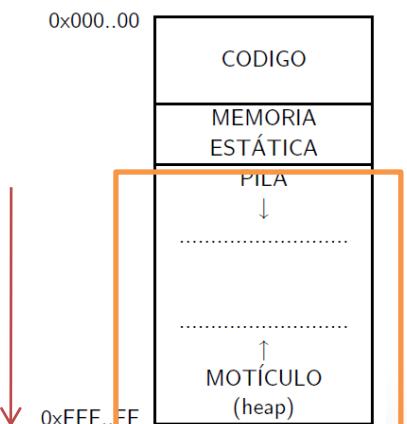
}



¿Qué pasa si se hace una llamada recursiva que no termina?

- Cada llamada recursiva almacena los parámetros que se pasaron al procedimiento, y otras variables necesarias para el correcto funcionamiento del programa. Por lo tanto si se produce una llamada recursiva infinita, esto es, que no termina nunca, llega un momento en que no quedará memoria para almacenar más datos, y en ese momento se abortará la ejecución del programa. Algunos S.O. modernos detectan la recursividad infinita de manera muy anticipada y evitan la ejecución de esta.
- Para probar esto se puede intentar hacer esta llamada en el programa factorial definido anteriormente:

factorial(-1);



Ejercicio de clase

- Considere la siguiente ecuación recurrente:

$$a_n = a_{n-1} + 2^n$$

$$a_0 = 1$$

- Diseña un algoritmo computacional recursivo que de la solución.



¿Cuándo usar recursividad?

- Para empezar, algunos lenguajes de programación no admiten el uso de recursividad, como por ejemplo el ensamblador o el FORTRAN. Es obvio que en ese caso se requerirá una solución no recursiva (iterativa).
- Para simplificar el código.
- **Cuando el problema tiene por definición una solución recursiva.**
- Cuando es necesario navegar en una estructura de datos recursiva ejemplo: **listas y árboles**.



¿Cuándo NO usar recursividad?

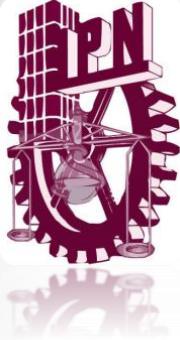
- **No se debe utilizar cuando la solución iterativa sea clara a simple vista.** Sin embargo, en otros casos, obtener una solución iterativa es mucho más complicado que una solución recursiva, y es entonces cuando se puede plantear la duda de si merece la pena **transformar la solución recursiva en otra iterativa**.
- Cuando los métodos usen arreglos o estructuras de datos con un **gran número de elementos (>miles)**.
- Cuando el método cambia de manera impredecible de campos.
- Cuando las iteraciones sean la mejor opción



Recursión vs. iteración

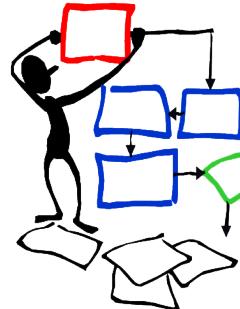
- **Repetición**
 - Iteración: ciclo explícito
 - Recursión: repetidas invocaciones a método
- **Terminación**
 - Iteración: el ciclo termina o la condición del ciclo falla
 - Recursión: se reconoce el paso base
- **En ambos casos podemos tener ciclos infinitos**
 - Considerar que resulta más positivo para cada problema la elección entre eficiencia (**iteración**) o una **buenas ingeniería de software**, La recursión resulta normalmente más natural.





Instituto Politécnico Nacional

Escuela Superior de Cómputo



Algoritmia y programación estructurada

Tema 05: El problema del ordenamiento

M. en C. Edgardo Adrián Franco Martínez

<http://www.eafranco.com>

edfrancom@ipn.mx

[@edfrancom](https://twitter.com/edfrancom) [@edgardoадrianfranco](https://facebook.com/edgardo.adrian.franco)



Contenido

- Introducción
- Ordenamiento Burbuja
 - Mejorando un poco la burbuja
- Ordenamiento por Inserción
- Ordenamiento por selección

(2)



Introducción

- Ordenar es simplemente colocar información de una manera especial basándonos en un criterio de ordenamiento.
- El propósito principal de un ordenamiento es el de facilitar las búsquedas de datos del conjunto ordenado. Un ordenamiento es conveniente usarlo cuando se requiere hacer una cantidad considerable de búsquedas y es importante el factor tiempo.



(3)



- El principal problema de la computación es el ordenamiento de la información, muchas de las operaciones que se realizan en una computadora dependen del orden de los datos y se facilitan mucho una vez los datos se encuentre con algún tipo de orden.
- Para poder ordenar una cantidad determinada de números almacenadas en un vector o matriz, existen distintos métodos (*algoritmos*) con distintas características y procedimientos para hacerse.



Ordenamiento Burbuja

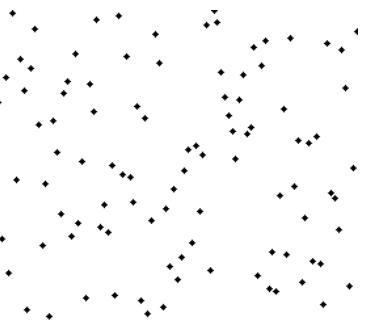
Burbuja Simple

- Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.
 - El método de la burbuja es uno de los mas simples, es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor o menor a otro, entonces los intercambia de posición.
 - Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo.



Burbuja Simple

6 5 3 1 8 7 2 4



```
Procedimiento BurbujaSimple(A,n)
    para i=0 hasta n-2 hacer
        para j=0 hasta (n-2)-i hacer
            si (A[j]>A[j+1]) entonces
                aux = A[j]
                A[j] = A[j+1]
                A[j+1] = aux
            fin si
        fin para
    fin para
fin Procedimiento
```

El arreglo A indexa desde 0 hasta n-1 -- A[0,1,..., n-1]



Una versión de Burbuja Optimizada

- Como al final de cada iteración el elemento mayor queda situado en su posición, ya no es necesario volverlo a comparar con ningún otro número, reduciendo así el número de comparaciones por iteración, además pude existir la posibilidad que realizar iteraciones de más si el arreglo ya fue ordenado totalmente.

Procedimiento BurbujaOptimizada (**A**, **n**)

```
cambios = "Sí"  
i=0  
Mientras i < n-1 && cambios != "No" hacer  
    cambios = "No"  
    Para j=0 hasta (n-2)-i hacer  
        Si(A[i] < A[j]) hacer  
            aux = A[j]  
            A[j] = A[i]  
            A[i] = aux  
            cambios = "Sí"  
        FinSi  
    FinPara  
    i = i+1  
FinMientras  
fin Procedimiento
```



El arreglo A indexa desde 0 hasta n-1 -- A[0,1,..., n-1]

Ordenamiento por inserción

6 5 3 1 8 7 2 4

- Es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria.
- Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento $k+1$ y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se **inserta** el elemento $k+1$ debiendo desplazarse los demás elementos.



Procedimiento Insercion(A, n)

{

```
    para i=0 hasta n-1 hacer
        j=i
        temp=A[i]
        mientras (j>0) && (temp<A[j-1]) hacer
            A[j]=A[j-1]
            j--
        fin mientras
        A[j]=temp
    fin para
fin Procedimiento
```

6 5 3 1 8 7 2 4

(9)

El arreglo A indexa desde 0 hasta n-1 -- A[0,1,..., n-1]



Ordenamiento por selección

8
5
2
6
9
3
1
4
0
7

- Se basa en buscar el mínimo elemento de la lista e intercambiarlo con el primero, después busca el siguiente mínimo en el resto de la lista y lo intercambia con el segundo, y así sucesivamente.
- **Algoritmo**
 - Buscar el mínimo elemento entre una posición i y el final de la lista Intercambiar el mínimo con el elemento de la posición i .



```
Procedimiento Seleccion(A,n)
    para k=0 hasta n-2 hacer
        p=k
        para i=k+1 hasta n-1 hacer
            si A[i]<A[p] entonces
                p=i
            fin si
        fin para
        temp = A[p]
        A[p] = A[k]
        A[k] = temp
    fin para
fin Procedimiento
```

							8
							5
							2
							6
							9
							3
							1
							4
							0
							7



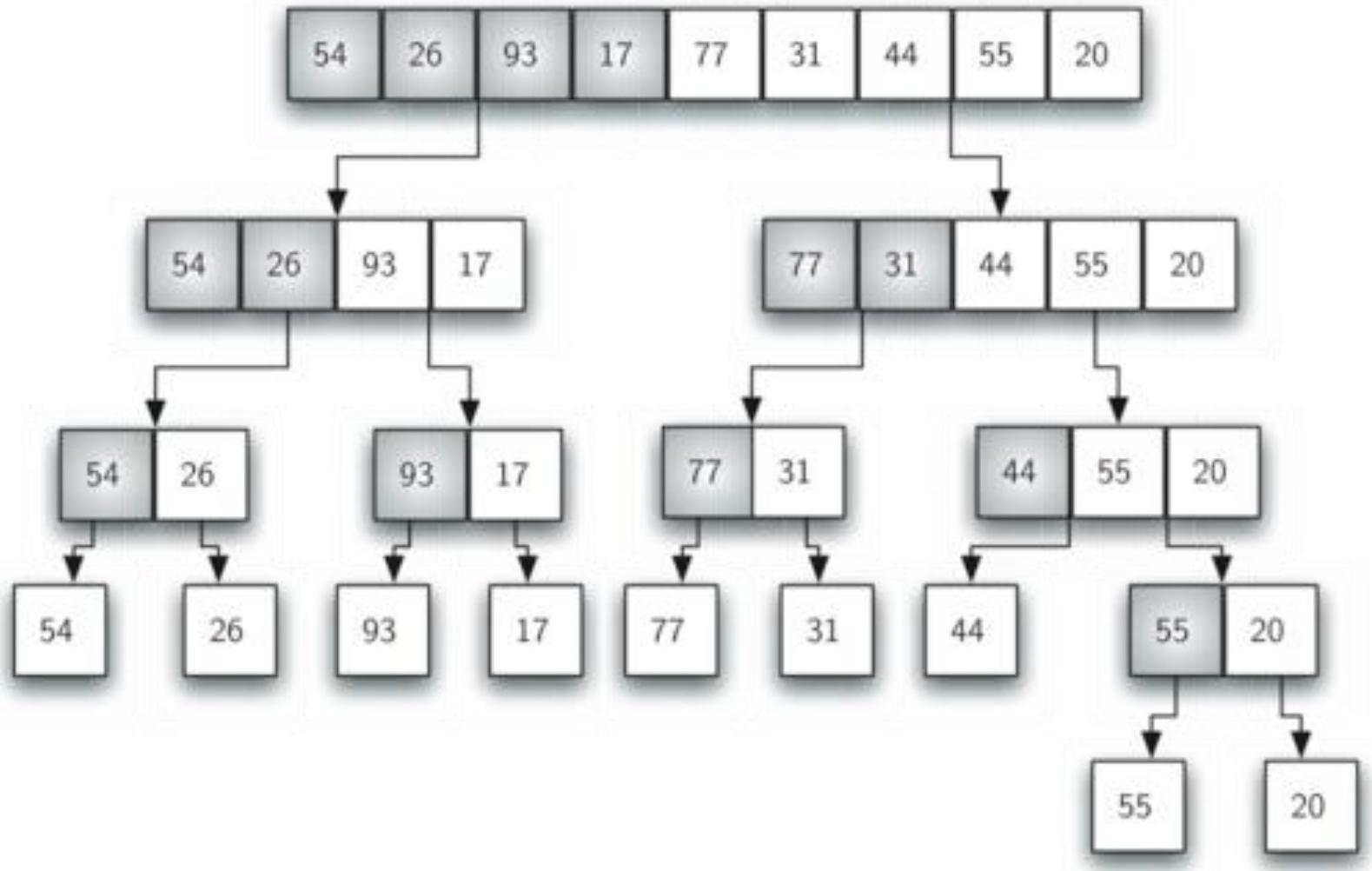
Ordenamiento por mezcla

- Se basa en dividir en dos conjuntos de elementos a ordenar hasta el caso mínimo posible que es tener 1 elemento, este ultimo ya esta ordenado por lo que podemos retornar mitades ordenadas y mezclarlas para lograr un ordenamiento final.

```
Merge-Sort(a, p, r)
{
    if ( p < r )
    {
        q = parteEntera((p+r)/2);
        Merge-Sort(a, p, q);
        Merge-Sort(a, q+1,r);
        Merge(a, p, q, r);
    }
}
```

6 5 3 1 8 7 2 4

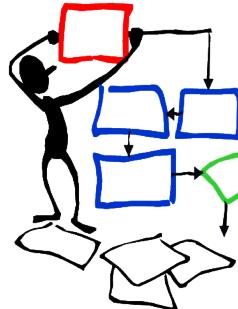






Instituto Politécnico Nacional

Escuela Superior de Cómputo



Algoritmia y programación estructurada

Tema 06: El problema de la Búsqueda

M. en C. Edgardo Adrián Franco Martínez

<http://www.eafranco.com>

edfrancom@ipn.mx

[@edfrancom](https://twitter.com/edfrancom) [edgaroadrianfrancocom](https://facebook.com/edgaroadrianfrancocom)



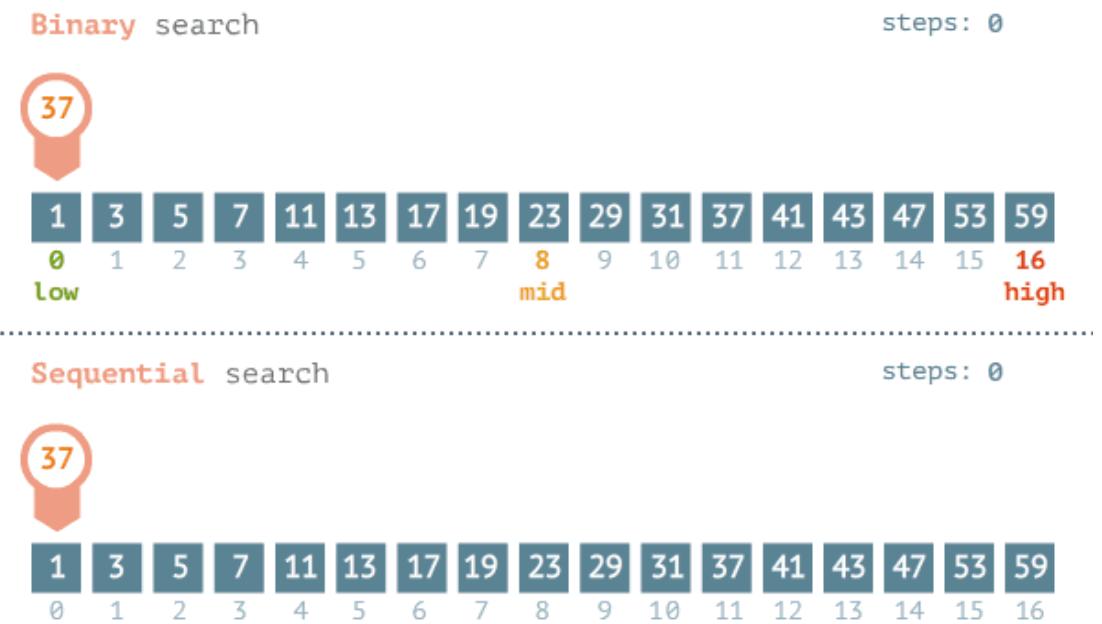
Contenido

- Introducción
- Búsqueda Lineal
- Búsqueda Binaria
- Búsqueda Binaria Recursiva



Introducción

- Un algoritmo de búsqueda es aquel que está diseñado para localizar un elemento con ciertas propiedades dentro de una estructura de datos; por ejemplo, ubicar el registro correspondiente a cierta persona en una base de datos, buscar un número en un arreglo de números, etc.

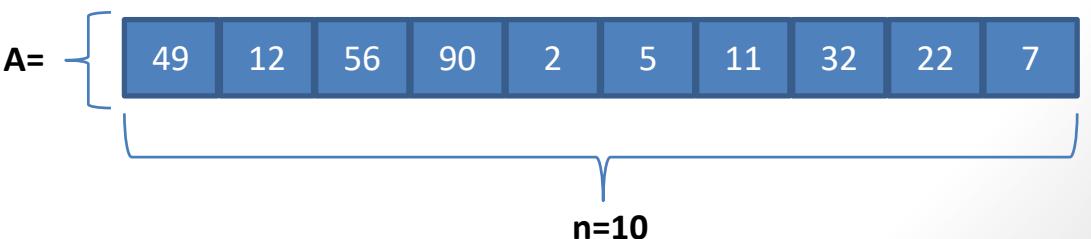


Búsqueda Lineal

- Una búsqueda lineal es un recorrido sobre los elementos comparando uno a uno hasta encontrarlo o descartar su aparición en el grupo de elementos donde se busca.

```
func BusquedaLineal(Valor,A,n)
{
    i=1;
    while(i<=n&&A[i]!=Valor)
    {
        i=i+1;
    }
    return i;
}
```

Valor=11



Búsqueda Binaria

- Una búsqueda binaria se basa en realizar una búsqueda con el antecedente de que los elementos se encuentran bajo un orden y se puede realizar una búsqueda dirigida.

```
BusquedaBinaria(A,n,dato)
    inferior=0;
    superior=n-1;
    while(inferior<=superior)
        centro=(superior+inferior)/2;
        if(A[centro]==dato)
            return centro;
        else if(dato < A[centro] )
            superior=centro-1;
        else
            inferior=centro+1;
    return -1;
```



Búsqueda Binaria Recursiva

- Una forma de visualizar mas simple la forma en la que opera la búsqueda binaria es su modelo recurrente.

```
int BusquedaBinaria(int num_buscado, int numeros[], int inicio, int centro, int final)
{
    if (inicio>final)
        return -1;
    else if (num_buscado == numeros[centro])
        return centro;
    else if (num_buscado < numeros[centro])
        return BusquedaBinaria(num_buscado,numeros,inicio,(int)((inicio+centro-1)/2),centro-1);
    else
        return BusquedaBinaria(num_buscado,numeros,centro+1,(int)((final+centro+1)/2),final);
}
```

