Introducción.

Tema 1

Fundamentos de Análisis de Algoritmos

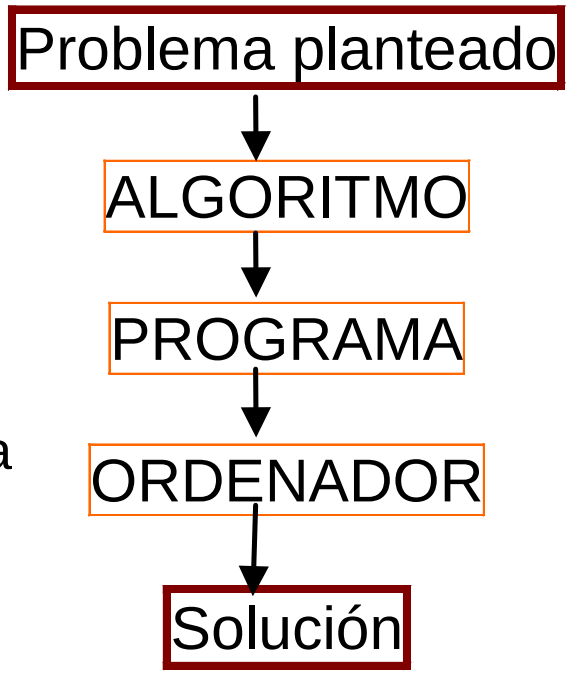
TEMA 1. Introducción.

1. Resolución de problemas.
2. Concepto de algoritmo.
   - 2.1. Definición.
   - 2.2. Propiedades.
3. Clasificación de problemas.
4. Algorítmica.
5. Aspectos a considerar de los algoritmos: diseño y estudio de su eficiencia.
6. Representación de algoritmos.
   - 6.1. Representación de algoritmos: Pseudocódigo y Diagramas de flujo.
   - 6.2. Elementos de un algoritmo: Variables, Constantes, Expresiones, Sentencias.
   - 6.3. Datos: Definición y Tipos de datos.
   - 6.4. Sentencias:
     - 6.4. 1. Asignación
     - 6.4. 2. Entrada/salida
     - 6.4. 3. Estructuras de control: condicional, bucles, funciones.
   - 6.5. Subalgoritmos.
   - 6.6. Resumen.

1. Resolución de problemas.

- La resolución de un problema mediante un ordenador conlleva, a grandes rasgos, la realización de los siguientes pasos:

1. Entender el problema
2. Análisis del problema
3. Diseñar un algoritmo para el problema
4. Expresar el algoritmo como un programa
5. Ejecutar el programa correctamente



2. Resolución de problemas.

- **Concepto de Algoritmo.** Serie de operaciones elementales, detalladas, no ambiguas y ordenadas que ejecutadas una a una conducen a la resolución de un cierto problema. Cada una de esas operaciones elementales puede ser considerada una instrucción.
- **Ejemplo.** Supongamos que el problema consiste en obtener las soluciones de una cierta ecuación de segundo grado: $\mathbf{ax^2 + bx + c = 0}$
- Un posible algoritmo para resolver el problema es el siguiente: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

1. Paso 1. Elevar al cuadrado b.
2. Paso 2. Realizar el producto 4*a*c.
3. Paso 3. Restar las cantidades obtenidas en los pasos 1 y 2.
4. Paso 4. Calcular la raíz cuadrada del resultado del paso 3.
5. Paso 5. Realizar el producto 2*a.
6. Paso 6. Restar b del resultado del paso 4.
7. Paso 7. Dividir el resultado del paso 6 por la cantidad obtenida en el paso 5. Esa cantidad es la primera raíz.
8. Paso 8. Sumar el resultado del paso 4 con b y cambiar el signo al resultado.
9. Paso 9. Dividir el resultado del paso 8 por la cantidad obtenida en el paso 5. Esa cantidad es la segunda raíz.

1. Resolución de problemas.

- A la hora de plantear el algoritmo para la resolución de un determinado problema, se contemplan en informática dos campos de estudio teórico importantes:
  - **Computabilidad.**

    Diremos que un problema es computable si admite **solución algorítmica**, es decir, si se puede establecer un algoritmo para su resolución. La teoría de la computabilidad intenta establecer qué problemas son computables, o sea, cuáles son abordables mediante un ordenador.
  - **Complejidad.**

    En los problemas computables es interesante estimar el **orden de magnitud de los recursos computacionales** que requieren los distintos algoritmos que puedan resolverlos. El estudio de la complejidad se encarga de estimar este orden de magnitud. Los recursos computacionales más importantes contemplados a la hora de procesar un algoritmo son:

    - **tiempo** de procesamiento y
    - **requisitos de dispositivos** (memoria, disco, etc.).

1. Resolución de problemas.

- Un problema puede ser teóricamente computable pero con un algoritmo de una complejidad tan alta que sea irrealizable un una máquina actual.
- El estudio de este paso es vital para la eficaz resolución de un determinado problema: plantear un buen algoritmo mejorará el rendimiento del sistema.

- Un problema tiene **solución algorítmica** si además de que el algoritmo existe, su tiempo de ejecución es razonablemente corto. Por ejemplo, es posible diseñar un algoritmo para jugar ajedrez que triunfe siempre pero el espacio de búsqueda se ha estimado en 1000^{40} tableros por lo que puede tardarse miles de años en tomar una decisión.
  - Se considera que si un problema tiene una solución que toma años en computar, dicha solución no existe.
- Considérese ahora el problema de ordenar un conjunto de valores, si el conjunto tiene 2 elementos es más fácil resolverlo que si tiene 20, análogamente un algoritmo que resuelva el problema tardará más tiempo mientras más grande sea el conjunto y requerirá una cantidad de memoria mayor para almacenar los elementos del conjunto.

2. Concepto de algoritmo. 2.1. Definición de algoritmo.

- **Definición 1**: Un algoritmo es un conjunto finito de instrucciones no ambiguas y efectivas que indican cómo resolver un problema, producen al menos una salida, reciben cero o mas entradas y, para ejecutarse, necesitan una cantidad finita de recursos.
  - Una instrucción es no ambigua cuando la acción a ejecutar esta perfectamente definida.
    - x = log(0) o x = (10 o 27)/3 no pueden formar parte de un algoritmo.
  - Una instrucción es efectiva cuando se puede ejecutar en un intervalo finito de tiempo.
    - x= 2+8 o mensaje=Une('Corre','Caminos') son efectivas.
    - x ← cardinalidad(numeros naturales) no es efectiva
- **Definición 2**: Secuencia **ordenada de pasos (instrucciones) exentos de ambigüedad** y **determinísticos** tal que al llevarse a cabo con fidelidad dan como resultado que se realice la tarea para el que se ha diseñado en un **tiempo finito** (se obtiene la **solución** del problema planteado)

➢ Los algoritmos que vamos a considerar son determinísticos, esto es, para unos mismos datos de entrada, producen siempre una salida y además es la misma. Existen también algoritmos no determinísticos o aleatorios.

2. Concepto de algoritmo. 2.2. Propiedades.

Propiedades de un ALGORITMO

- **Finitud:** Siempre acaban después de un número finito de etapas
- **Precisión:** Cada etapa está debe estar definida de forma precisa y las acciones a realizar rigurosamente especificadas para cada caso.
- **Entrada:** Necesita unos datos de entrada
- **Salida:** Siempre produce una salida
- **Efectividad:** Todas las acciones que hay que realizar deben ser tan básicas como para que se puedan realizar exactamente y en un tiempo finito.

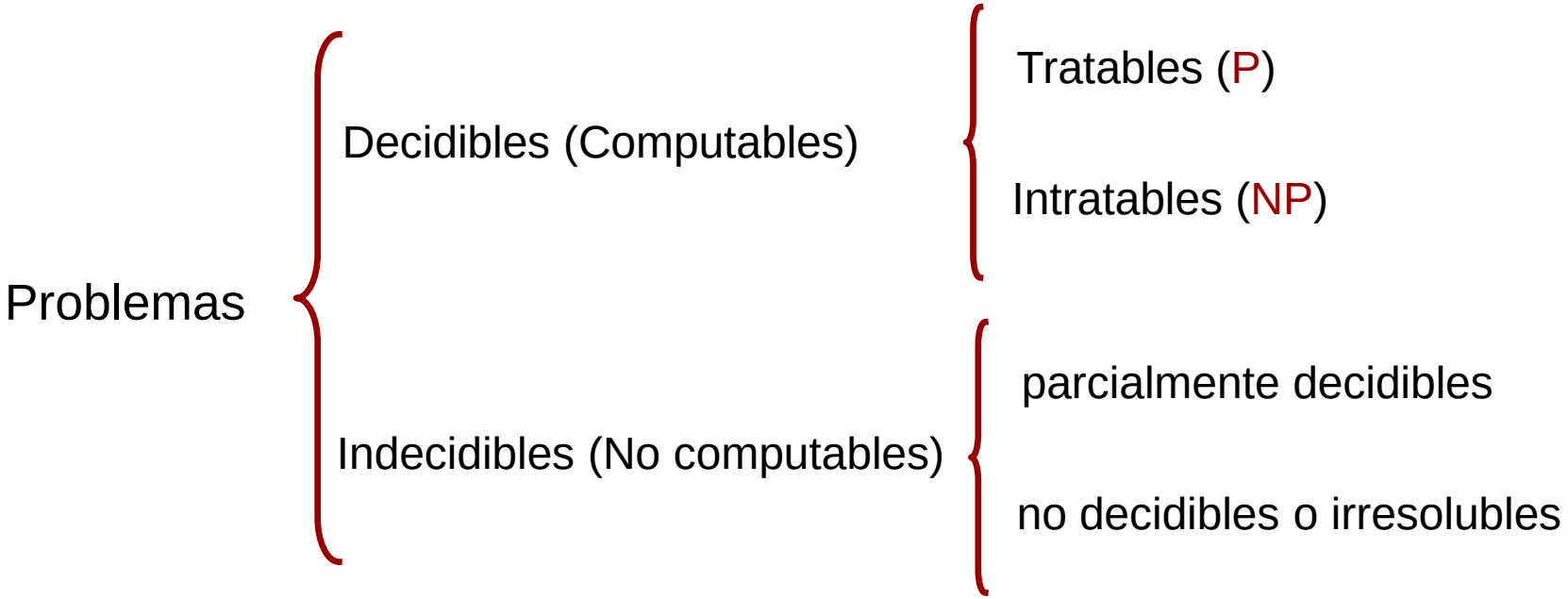
3. Clasificación de problemas.

- Estudio/clasificación de problemas.
  - **Década 30:** Problemas Computables y No Computable.

    ¿Cuáles son los problemas abordables por un informático?.

    - Ejemplo de problema no abordable: Programa que tenga como entrada otro programa y sus datos de entrada y que responda en la salida a si el programa terminará algún día de hacer el cálculo. Es imposible resolver este problema, se conoce como "**Problema de la parada**"
  - **Década 50:** Complejidad de los problemas computables (búsqueda de algoritmos más eficaces). Todos los problemas NO presentan la misma dificultad.
    - Ejemplo: **Problema del viajante de comercio** que tiene que visitar en un **tiempo mínimo** un número finito de ciudades. Si tratamos de abordar el problema intuitivamente, buscando todas las rutas posibles y eligiendo luego la más corta, para n ciudades, tenemos **n-n!** rutas posibles. Es un número inabordable, de ahí la necesidad de buscar algoritmos más eficaces.
  - **Década 70:** Clasificación de los problemas computables: **P y NP.**
    - **Problemas P**: Problemas para los que se conoce un algoritmo que lo resuelve en un tiempo polinomial, asumiendo que es un tiempo eficiente.
    - **Problemas NP**: Problemas no determinísticos. Sólo se pueden resolver de manera polinomial realizando una etapa aleatoria.

3. Clasificación de problemas.



3. Clasificación de problemas.

- **Problemas computables, problemas decidibles o resolubles**, aquellos para los que existe un algoritmo, es decir, se detiene tanto cuando se acepta la entrada como cuando no.
  - Tratables, conocidos también como **problemas P**, problemas para los cuales existe un algoritmo de complejidad polimonial.
  - Intratables, a los problemas no tratables se les llama también **problemas NP** (de orden no determinístico polinomial), problemas para los que no se conoce ningún algoritmo de complejidad polimonial.. Ejemplos:
    - Agente Viajero.
    - Caminos Hamiltonianos. Encontrar un camino en un grafo que pasa una sola vez por cada nodo. O(n!).
    - Caminos Eulerianos. Encontrar un camino en un grafo que pasa una sola vez por cada arco.
- **Problemas no computables, problemas indecidibles o iresolubles.**
  - Problemas parcialmente decidibles, aquellos problemas para los que se detiene cuando se acepta la entrada y corre indefinidamente para algunas entradas que no deben ser aceptadas.
  - Problemas indecidibles o irresolubles, aquellos que cuando se debe aceptar la entrada, es posible que continúe ejecutándose indefinidamente.

3. Clasificación de problemas.

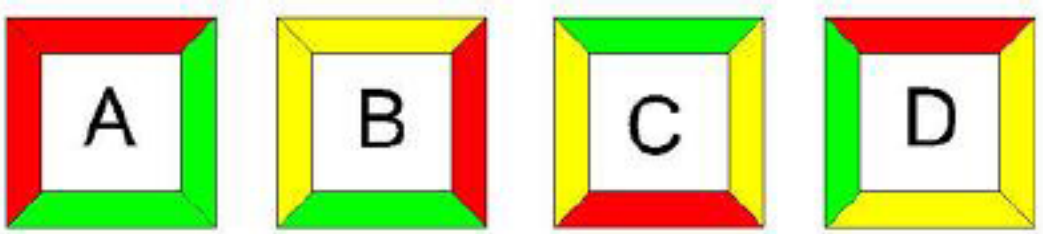
Límites de la computación

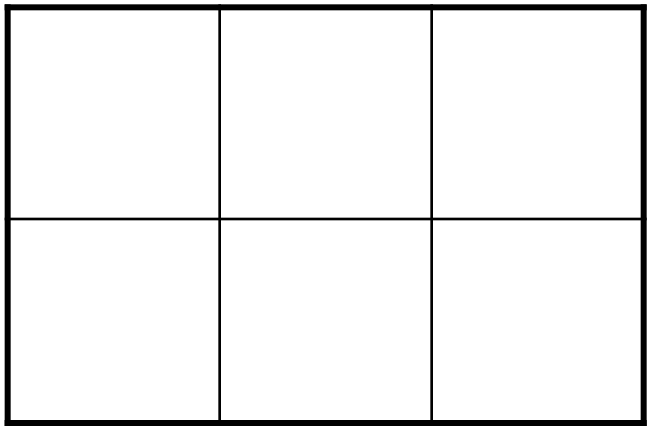
- Newell, 1958 “En un plazo de 15 años los computadores podrán hacer todas las tareas que hacen los humanos”
- Revista TIME, 1984

  “Pongamos el tipo de software adecuado en un computador y hará lo que queramos. Puede haber límites en lo que podamos hacer con las máquinas, pero no lo hay en lo que podemos hacer con el software”
- Con el poder computacional de los modelos que existen actualmente, hay problemas que aun no se pueden resolver

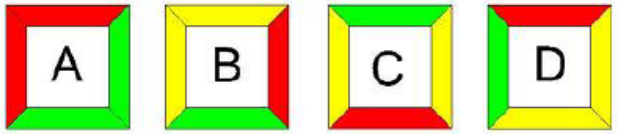
3. Clasificación de problemas. Ejemplo 1: teselación.

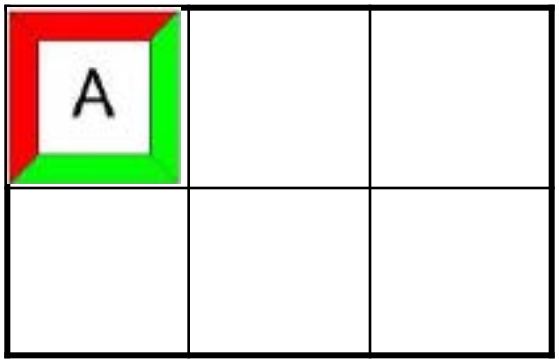
- **El problema de la teselación.** El problema consiste en: dada cualquier superficie finita de cualquier tamaño ¿puede recubrirse usando teselas de forma que cada dos teselas adyacentes tengan el mismo color en los lados que se tocan?
- Considere un conjunto finito de teselas, cada una con un color dado en sus lados

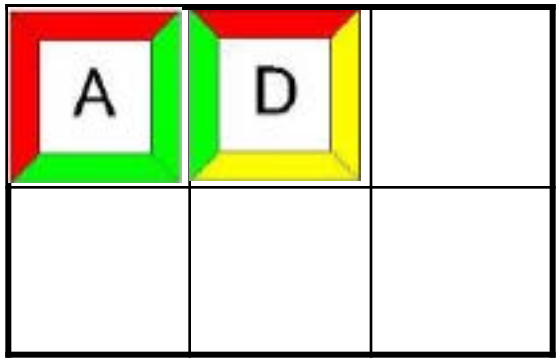


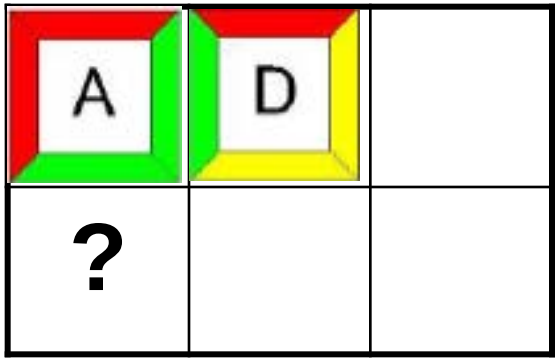


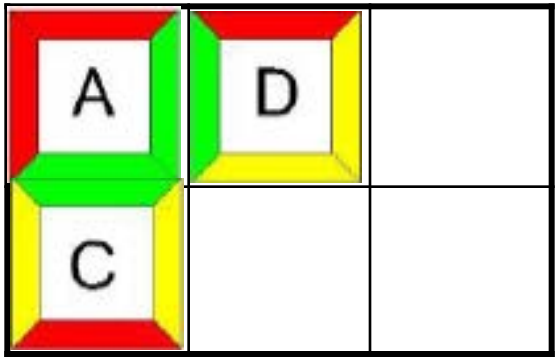
3. Clasificación de problemas.

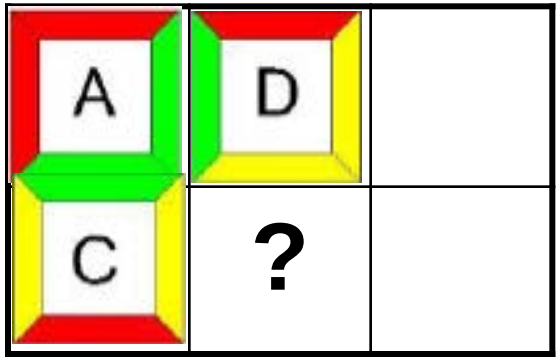


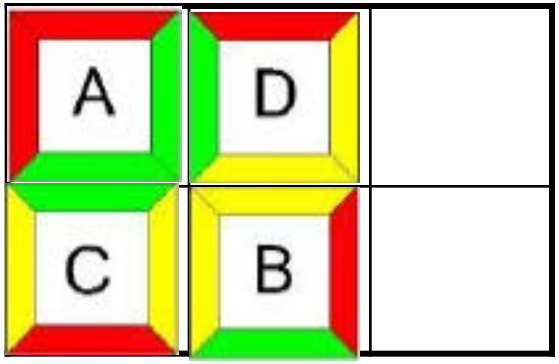


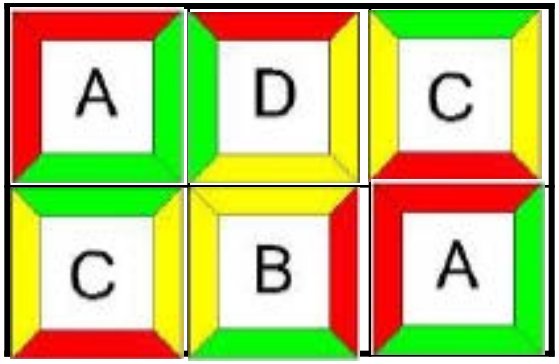




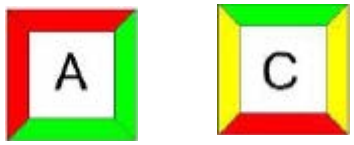


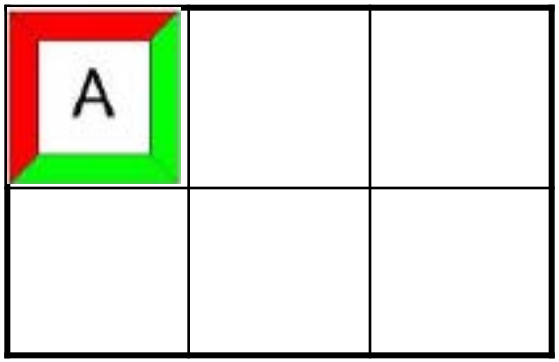


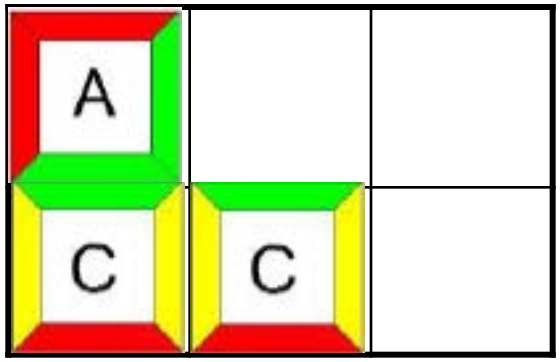


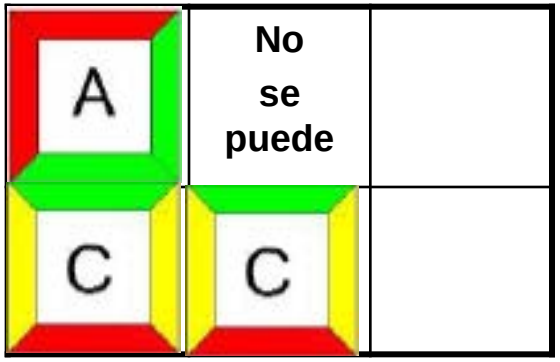


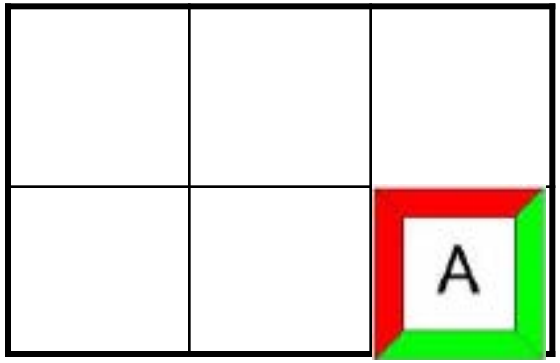
3. Clasificación de problemas.

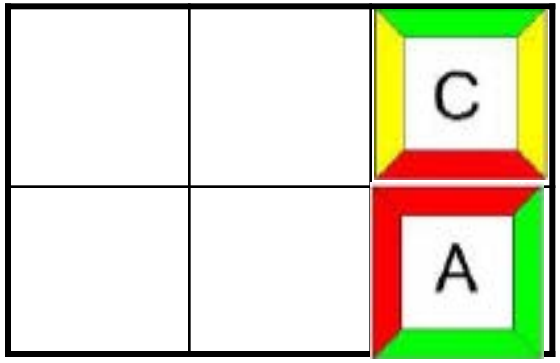


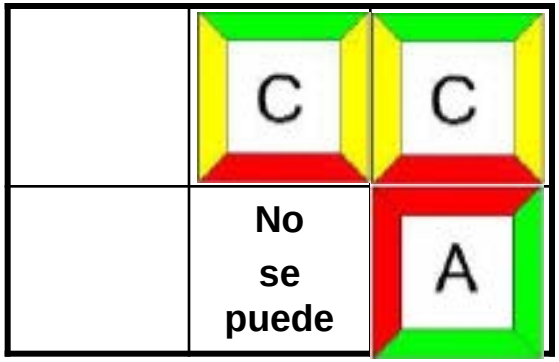


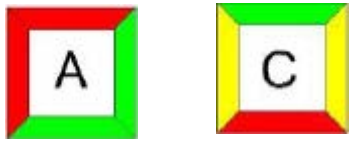


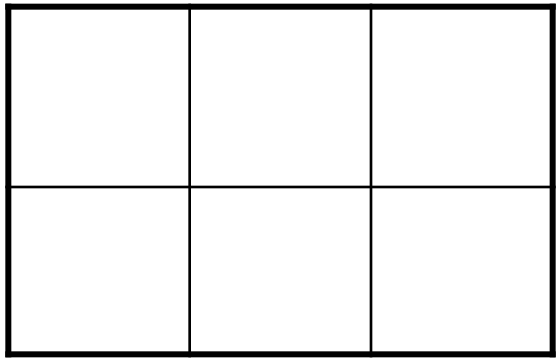












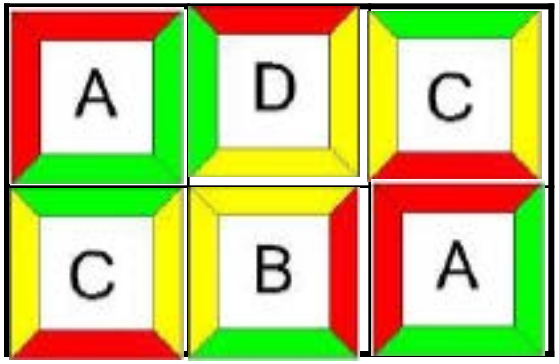
En este caso particular

NO hay solución

3. Clasificación de problemas. Ejemplo 1: teselación.

- **El problema de la teselación**

- Es imposible diseñar un algoritmo que resuelva el **problema de la teselación**



- Es un problema **no decidible o no resoluble.**

3. Clasificación de problemas. Ejemplo 2: serpiente.

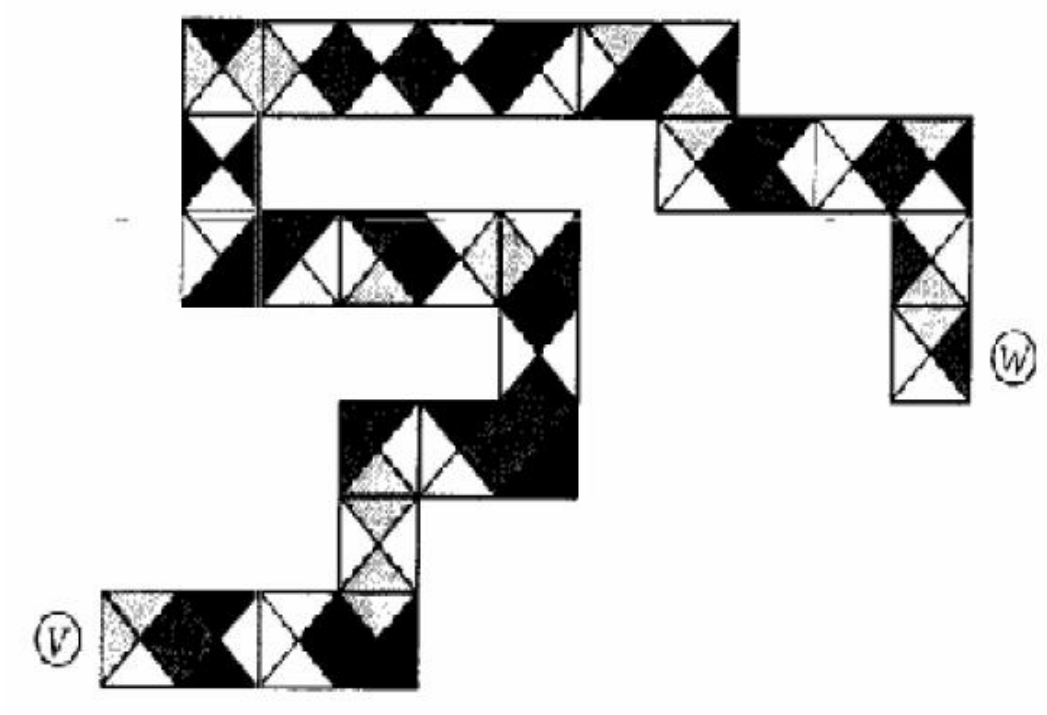
- **El problema de la serpiente**. Dados dos puntos, V y W, del plano ¿es posible unirlos entre sí por teselas que cumplan la restricción de lados adyacentes iguales?

- Considerar las siguientes teselas:

(1)

(2)

(3)



- Es un problema **no decidible o no resoluble**

3. Clasificación de problemas. Ejemplo 3: MCD

- **El problema del MCD.** Calcular el Máximo Común Divisor entre dos números enteros positivos.

```
funcion MCD (a; b);
   comienza
      r[0] ← a;
      r[1]← b;
      i ← 1;
      mientras (r[i] ≠ 0) hacer
            r[i+1] ← r[i-1] mod r[i];
            i ← i+1;
      fmientras;
      MCD (a; b)= r[i-1];
    termina.
```

Ejemplo: MCD(60, 48)=12:

i=0	r	60	48			
		0	1	2	3	
i=1	r	60	48	12		60/48=cociente 1,resto 12
		0	1	2	3	
i=2	r	60	48	12	0	
		0	1	2	3	48/12=cociente 4, resto 0
i=3	r	60	48	12	0	i=3 r[2]=12
		0	1	2	3	

- Es posible diseñar un algoritmo que resuelva el **problema del MCD.**
- Es un problema **decidible o resoluble, computable.**

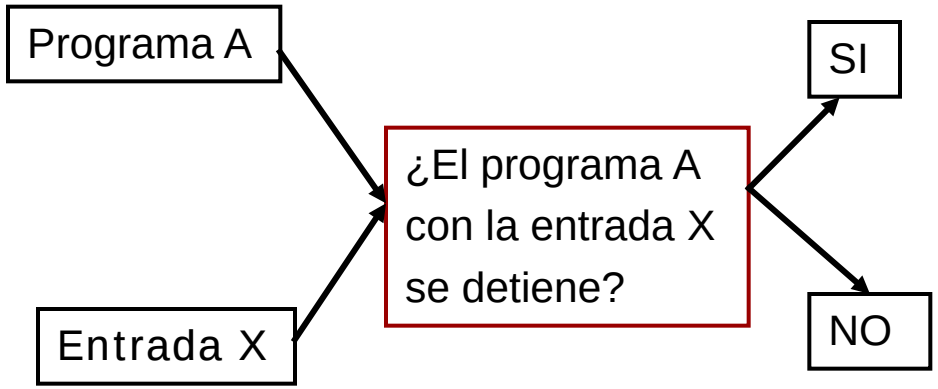
3. Clasificación de problemas. Ejemplo 4: Parada.

- **El problema de la parada de Lagarias, 1985**
  - Dado el siguiente algoritmo:

```
procedimiento Parada;
  comienza
      Leer(X);
      Mientras (X ≠1) hacer
            Si EsPar(X) entonces
                  X ← X/2;
            otro
                  X ← 3*X+1;
            fsi
      fmientras
  termina
```

- ¿se detiene o no para cualquier X?
- Si el valor inicial de X es 7, tomaría los siguientes valores:

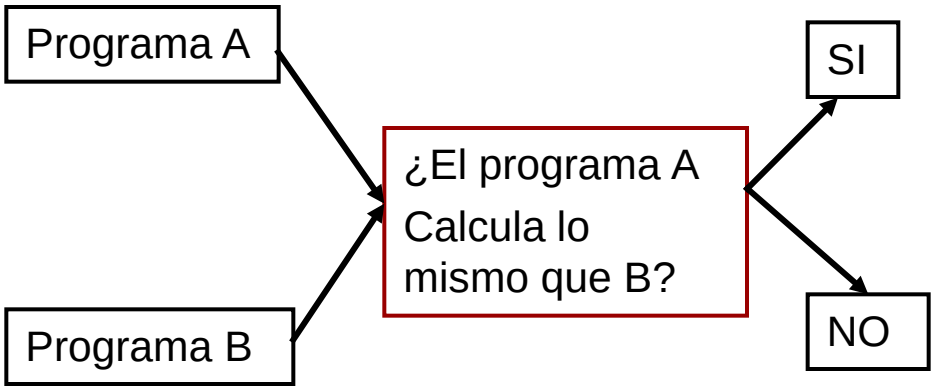
  7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1
- No hay manera de decidir, en general, si la ejecución se detiene o no



- El **problema de la parada** es **no decidible o no resoluble**

3. Clasificación de problemas. Ejemplo 5. Equivalencia.

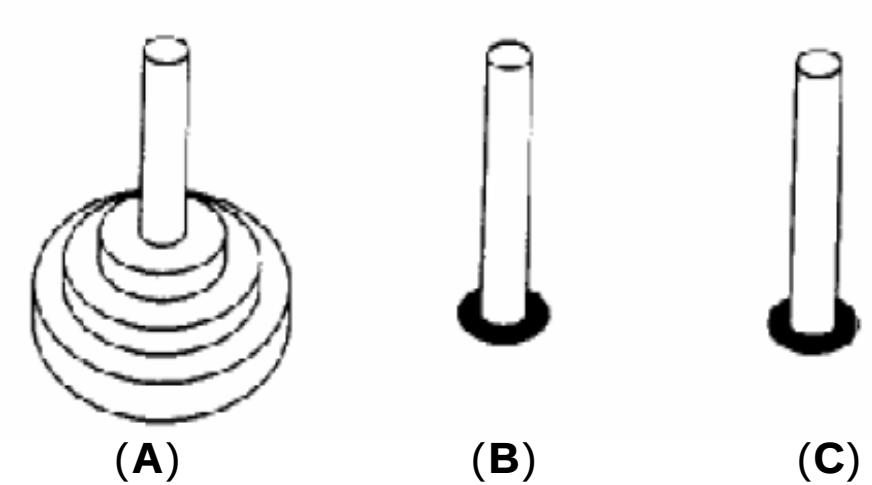
- **El problema de la equivalencia**
  - Dados dos programas A y B, ¿calculan lo mismo?



- El **problema de la equivalencia** es **no decidible o no resoluble**

3. Clasificación de problemas. Ejemplo 5. Torres de Hanoi.

- **El problema de las torres de Hanoi**. El problema consiste en: Mover los N anillos de la torre A a la torre B o C, usando la otra como ayuda, pero sin que haya un disco de mayor diámetro sobre otro de menor.



N	Operaciones
10	1.024
20	1.048.576
50	1.125.899.906.842.624

- La cota mínima de complejidad para este problema es $2^{N.}$
- Un ordenador capaz de hacer 1 millón de operaciones por segundo, tardaría 1 ms en resolver el juego con 10 anillos y casi 36 años si se colocan 50 anillos.
- El problema de las **torres de Hanoi** es **decidible intratable.**

3. Clasificación de problemas. Ejemplos.

Problemas decidibles o resolubles (Computables)	Tratables (P)	Problema del MCD
	Duros o Intratables (NP)	Problema de las torres de Hanoi

Problemas no decidibles o no resolubles (no Computables)	Parcialmente decidibles	Problemas de la parada, de la teselación
	No decidibles o Irresolubles	Problema de la verificación

4. Algorítmica.

- Algorítmica: estudio de los algoritmos. Incluye diferentes e importantes áreas de investigación y docencia.

La Algorítmica considera:

1. La construcción
2. La expresión
3. La validación
4. El análisis, y
5. Estudio empírico

4. Algorítmica.

1. **La construcción de algoritmos**
   - Creación de algoritmos: parte creativa y sistemática.
   - El acto de crear un algoritmo no se puede dominar si no se conocen a la perfección las técnicas de diseño de los algoritmos
   - El área de la construcción de algoritmos engloba el estudio de los métodos que, facilitando esa tarea, se han demostrado en la práctica mas útiles.
2. **La expresión de algoritmos**
   - Los algoritmos deben expresarse de forma precisa, clara e independientes de un lenguaje de programación en concreto.**(Diagramas de flujo, pseudocódigo).**
3. **Validación de algoritmos**
   - Validación: Demostrar que las respuestas que da son correctas para todas las posibles entradas
   - Único método válido: demostración formal.
   - En muchos casos no es factible: pruebas empíricas o parciales.

4. Algorítmica.

4. Análisis de algoritmos

- Análisis de algoritmos: Determinar los recursos (tiempo, espacio) que consume un algoritmo en la resolución de un problema
- Permite comparar algoritmos con criterios cuantitativos.
- Elección de un algoritmo entre varios para resolver un problema.

5. Estudio empírico

- Se trata de generar e implementar el algoritmo mediante un programa: codificación, prueba y validación (con depuración en su caso)
- Evaluación empírica del tiempo y espacio que requiere el algoritmo implementado para resolver problemas de distinto tamaño.

Estudio de la eficiencia de los algoritmos.

Elección de un Algoritmo

- En la práctica no solo queremos algoritmos, sino que queremos buenos algoritmos en algún sentido propio de cada usuario.
- A menudo tendremos varios algoritmos para un mismo problema, y deberemos decidir cual es el mejor, o cual es el que tenemos que escoger según algún criterio, para resolverlo.
- Esto nos centra el estudio en el campo del **Análisis de los Algoritmos**.

Problema Central: Determinar ciertas características que sirvan para evaluar su rendimiento.

- Un criterio de bondad es la longitud del tiempo consumido para ejecutar el algoritmo; esto puede expresarse en términos del número de veces que se ejecuta cada etapa.

Estudio de la eficiencia de los algoritmos.

Modelos para el análisis de la eficiencia:

Enfoque Empírico. **Análisis experimental o "a posteriori"** (se verá en la prácticas de laboratorio). Consiste en ejecutar casos de prueba, haciendo medidas para:

- una máquina concreta,
- un lenguaje concreto,
- un compilador concreto y,
- datos concretos.

Enfoque teórico. **Análisis teórico o "a priori"** (se verá en las clases teóricas). Consiste en obtener una expresión que indique el comportamiento del algoritmo en función de los parámetros que influyan. Interesante porque:

- La predicción del coste puede evitar una implementación posiblemente laboriosa.
- Es aplicable en la **etapa de diseño** de los algoritmos, constituyendo uno de los factores fundamentales a tener en cuenta.

Enfoque Híbrido.

DISEÑO DE ALGORITMOS.

- El diseño de algoritmos se refiere a la búsqueda de métodos o procedimientos, secuencias finitas de instrucciones adecuadas al dispositivo que disponemos, que permitan resolver el problema.
- Se necesitan técnicas de diseño, que corresponden a los patrones fundamentales sobre los que se construyen los algoritmos que resuelven un gran número de problemas, para la Construcción de Algoritmos Eficaces y Eficientes.

6. Representación de algoritmos.

- Algoritmo: es una secuencia finita de operaciones que resuelve un problema en un tiempo finito. Sus características son:
  - Finito: debe tener un número finito de pasos, por lo que debe estar limitado tanto en tiempo de realización como por el número de pasos que realiza
  - Definido: para los mismos datos de entrada obtienen los mismos de salida
  - Preciso: debe indicarse el orden de realización de cada paso
  - **Independiente del lenguaje de programación**
- Adicionalmente a la experimentación conviene disponer de un enfoque analítico que:
  - Tome en consideración todas las posibles entradas.
  - Permita evaluar la eficiencia de dos algoritmos de forma independiente del hardware y software.
  - Se pueda realizar estudiando una representación de alto nivel del algoritmo sin necesidad de implementarlo.

6.1. Representación de algoritmos: Pseudocódigo y Diagramas de flujo.

- Pseudocódigo.
- Diagramas de flujo.

algoritmo nombre del algoritmo (parámetros opcionalmente)

entrada descripción de los datos de entrada al algoritmo

salida descripción de los datos de salida del algoritmo

variables lista de variables usadas separadas por comas

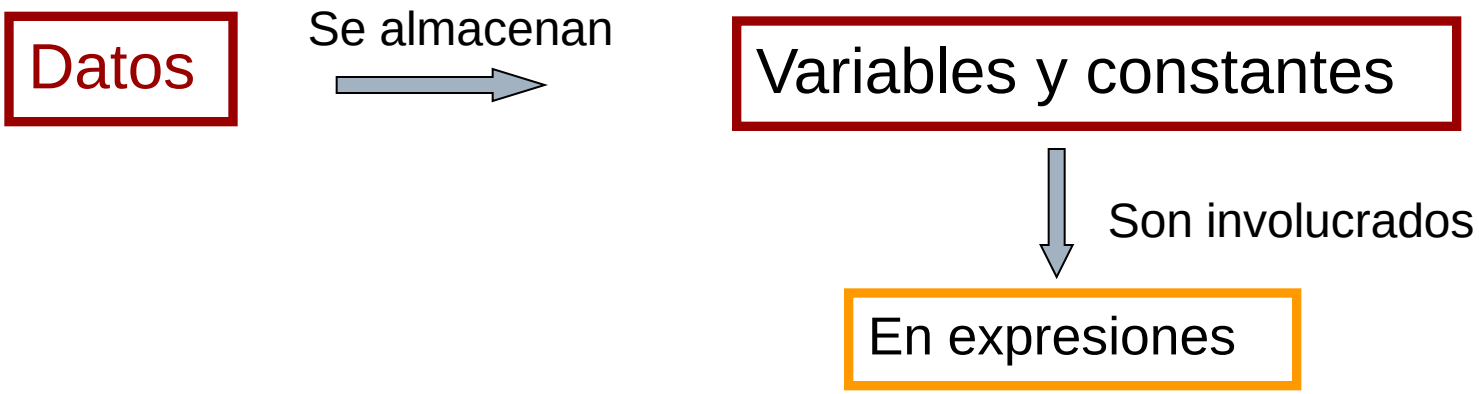
comienza

cuerpo del algoritmo (sentencias)

termina

6.2. Elementos de un algoritmo.

- Un algoritmo consta de ***datos*** y de ***sentencias***



Las ***sentencias*** describen las acciones que pueden ser ejecutadas, en general realizan cálculos, entradas/salidas y control de flujo del algoritmo

- ***Variable***: elemento del algoritmo que posee un valor, conocido por un nombre o identificador y que pertenece a un tipo de dato definido al inicio del algoritmo
  - Debe ser declarada antes de usarse !!!!!!!!
  - En un algoritmo la declaración consta de una sentencia que especifica: *el tipo de dato*, *su nombre* y *un valor inicial* en algunas ocasiones
- ***Constante:*** los elementos del algoritmo que no cambian de valor a lo largo del algoritmo
  - Las constantes deben ser inicializadas de acuerdo con el tipo de dato al que pertenecen
- ***Expresión:*** es una combinación de variables, constantes, valores constantes, operadores y funciones especiales que, en cada momento, al evaluarla tiene un valor concreto
  - Las expresiones más representativas son
    - las numéricas y
    - las lógicas

- Las expresiones **numéricas** tienen como resultado datos numéricos

Los operadores: ^, Signo: (+,-), *,/, +, -

Orden de precedencia de mayor a menor

  - Las operaciones entre paréntesis se evalúan primero
  - Las operaciones aritméticas se evalúan según el orden de prioridad

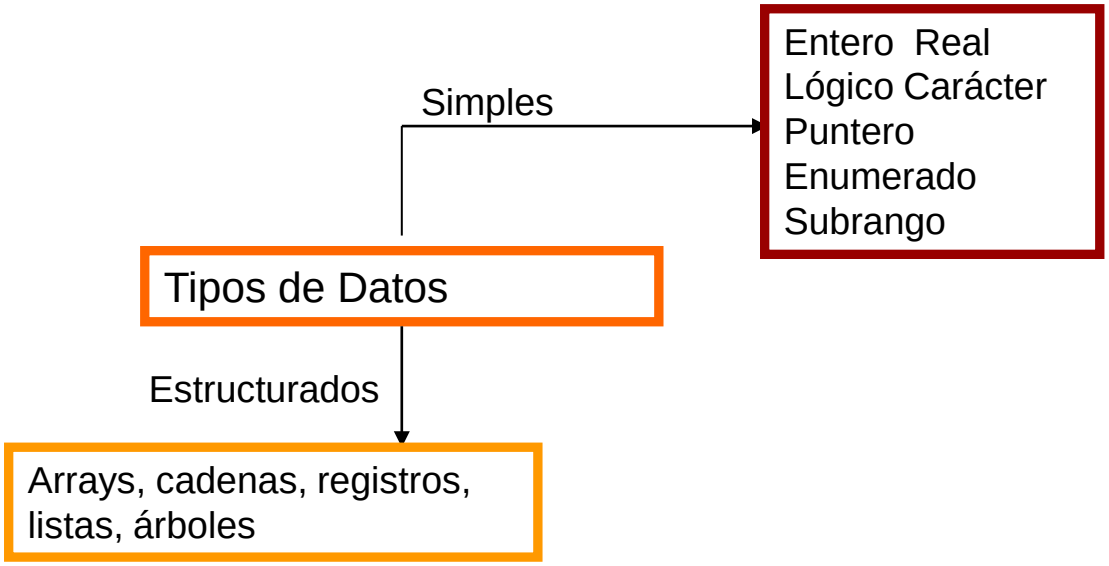
- Las expresiones **lógicas** son las que ofrecen como resultado después de su operación un valor lógico.
  - Los operadores lógicos que involucran son: AND, OR,NOT y los relacionales: <, >, ==, <=, >=, ~=

NOT, AND, OR, operadores relacionales

Orden de precedencia

6.3. Datos.

- ***Dato*** es una información relativa a un objeto que es manipulable por el ordenador, que posee un valor y que es conocido en un programa o algoritmo por un nombre o identificador del dato.
- El identificador indica una dirección de memoria, y es el nombre por el que se conoce a ese dato.
- Existen datos ***elementales*** y ***estructurados***



6.3. Datos. Tipos de datos simples.

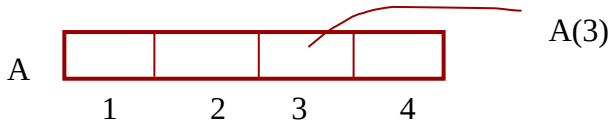
- Enteros: Representan números positivos o negativos sin decimales, en el rango $[-2^{n-1}, 2^{n-1}-1]$
- Reales: Almacenan un valor de la forma $N=M*B^E$
- Carácter: Representan elementos individuales de un conjunto finito de caracteres.
- Lógico o booleano: Solo pueden tener dos posibles valores verdadero o falso.
- Enumerado: este tipo de dato requiere que el programador defina el rango de valores que puede tomar
- Subrango: este tipo de dato se define a partir del tipo de dato entero, carácter o enumerado, con solo decir que el tipo de dato definido podrá tomar un conjunto de valores limitado del original
- Puntero: es aquel cuyo valor es la dirección en memoria de otro dato

6.3. Datos. Tipos de datos estructurados.

- Una estructura es ***estática*** cuando el tamaño en memoria ocupado se define antes de la ejecución del programa y no puede modificarse durante la ejecución.
- Una estructura ***dinámica*** es aquella en la que no se define a priori su tamaño en memoria
- Un conjunto ***homogéneo*** es aquel que está formado por datos del mismo tipo, y es **ordenado** si se puede acceder a cada uno de sus elementos usando un identificador.

6.3. Datos. Tipos de datos estructurados.

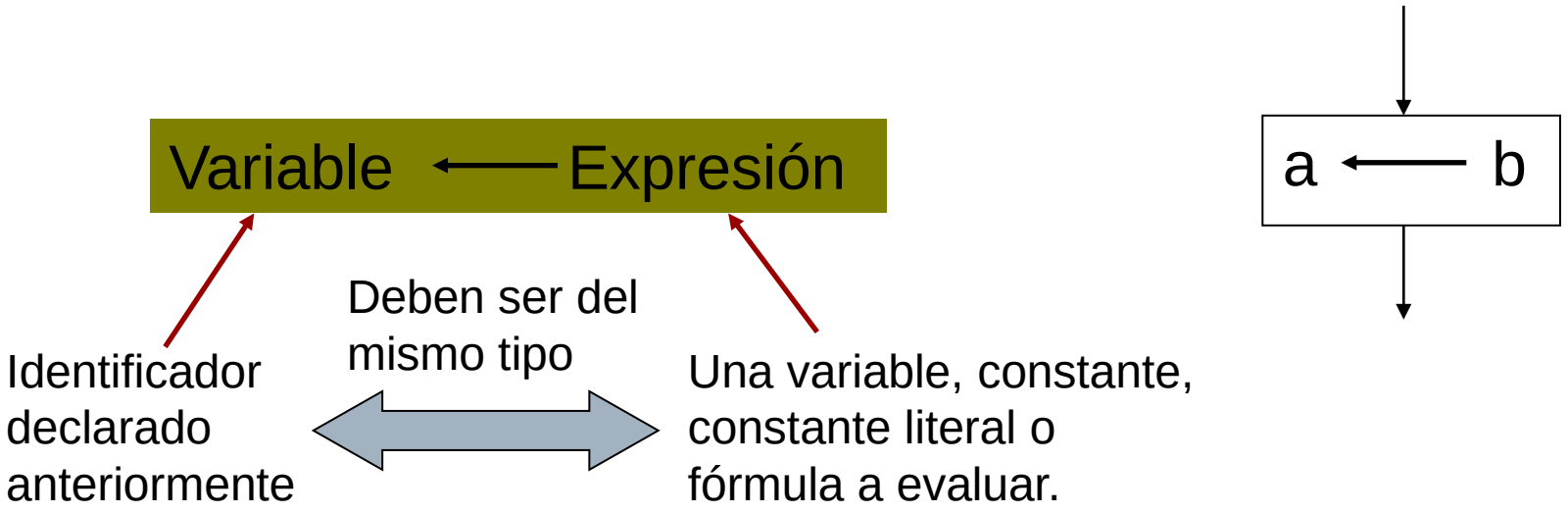
- ***Array*** es una estructura de datos homogénea, estática y ordenada, formada por una cantidad fija de datos de un mismo tipo, cada uno tiene asociado uno o más índices que determinan la posición del dato en el array.



- ***Cadenas de caracteres*** está formada por una secuencia de caracteres en un orden determinado, por lo tanto es una estructura homogénea, estática y de acceso por posición
- ***Registros***: formada por varios elementos o campos que se refieren a una misma entidad, es heterogénea, estática y de acceso por nombre
- ***Lista***: es una estructura de datos homogénea, dinámica y de acceso por clave. Se constituye por una cantidad no prefijada de registros, con al menos dos campos, uno de los cuales sirve para localizar al siguiente elemento de la lista
- ***Árbol***: es una estructura de datos homogénea y dinámica que ordena los elementos que la integran en forma de árbol, usando nodos y subárboles

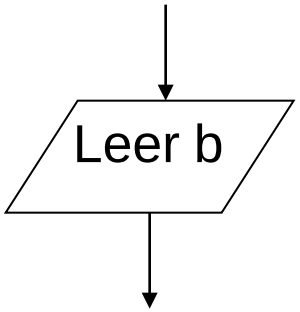
6.4. Sentencias.

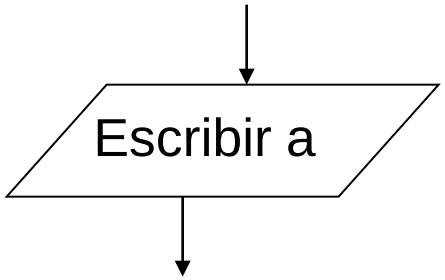
- Sentencias: Describen lo que debe hacer el algoritmo. Varios tipos:
  - **Asignación**: almacena un valor en una variable o constante. Se representa con el operador = ó el símbolo ←



6.4. Sentencias.

- **Entrada/salida**: Pueden almacenarse de tres formas: asociados con constantes, asignados a una variable o una sentencia de lectura.
  - Entrada: **Leer(** lista de variables separadas por comas)
  - Salida*:* **Escribir(** lista de variables o expresiones separadas por comas)





- En C++

cin >> identificador_var;

cout << expresión;

6.4. Sentencias.

- En C++:
  - Sentencias de asignación:
    - A=23
    - b = 48*A+b
    - .....
  - Sentencias de entada /salida:
    - Entrada: cin >> var1 >> var2;
    - Salida: cout << “La cantidad es :\n” << var1;
    - .....

6.4. Sentencias.

- Sentencias de control de flujo del algoritmo:
  - Secuenciales: todas las instrucciones se ejecutan una detrás de otra:
    - Ejemplo: calcular el área de un triangulo
  - Selectivas (bifurcaciones): se evalúa una expresión lógica o relacional, y en función de su resultado se selecciona cual de las posibles opciones se toma:
    1. Condicionales.
    2. Alternativas.
    - Ejemplo: dados dos números imprime el mayor de ellos

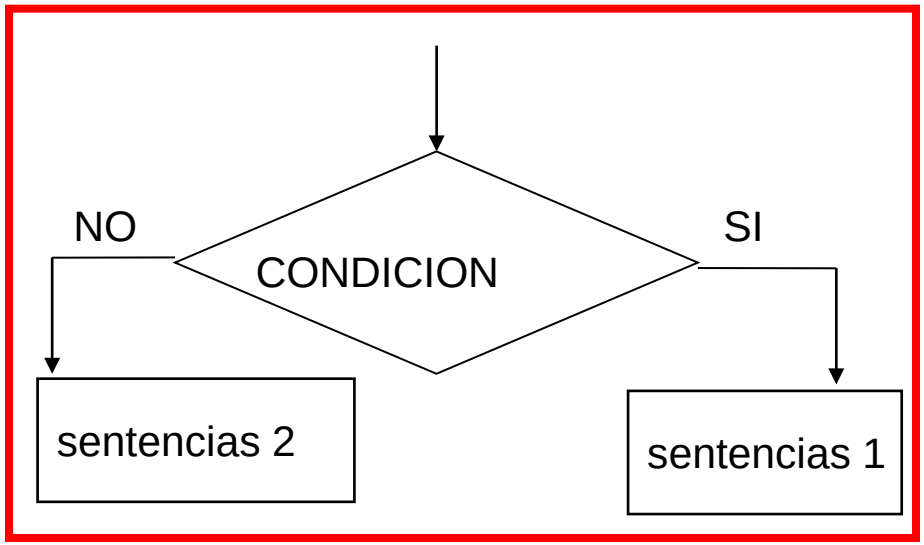
6.4. Sentencias.

1. Condicionales.

- **Pseudocódigo**

si *condición* **entonces**
sentencias 1
otro
sentencia 2
fsi

Diagrama de flujo



if *condición1*
sentencias 1
else if *condición2*
sentencias 2
……
else
sentencias n;

- En C++:

if *condición*
sentencias 1
else
sentencias 2

2. Alternativas o selectiva

- **Pseudocódigo**

```
en caso de que  expresión valga
        valor 1: bloque sentencias 1
         valor 2: bloque sentencias 2
         valor 3: bloque sentencias 3
         …
[en otro caso bloque sentencia x]
fin_caso
```

- En C++:

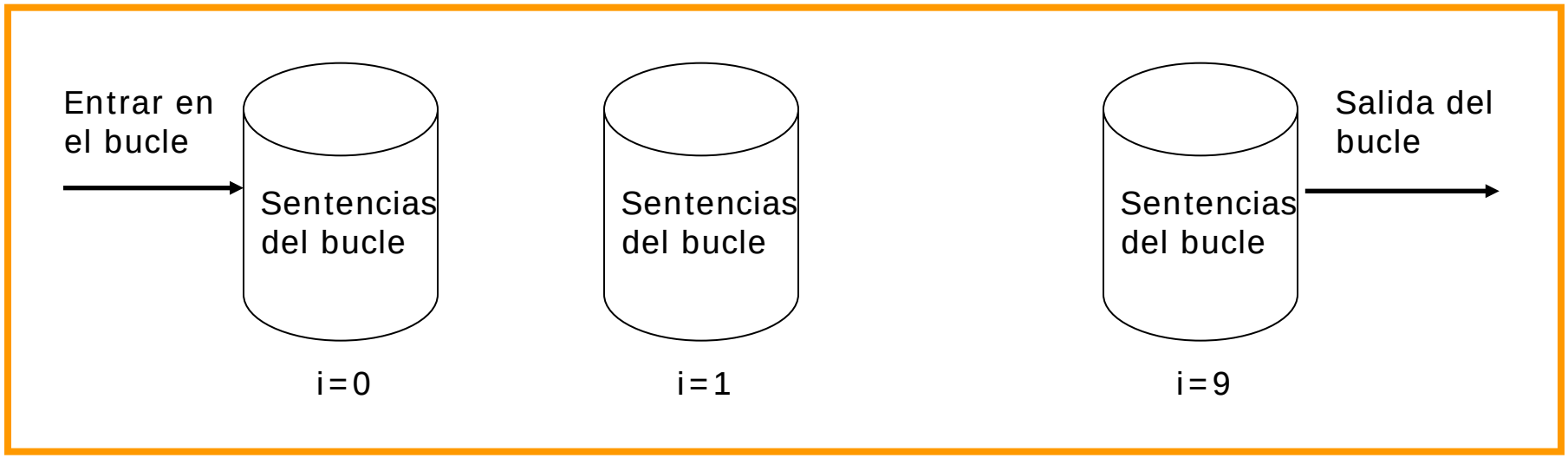
```
switch (variable o expresión) {
case valor1: sentencias1
        break;
case valor2: sentencias2
        break;
    . . .
        default: sentencias x
    }
```

6.4. Sentencias.

- **Iterativas (repetitivas**: cíclicas, bucles o lazos): Se utilizan para realizar varias veces el mismo conjunto de operaciones.

1. Ciclo determinista: bucle PARA

```
para i=0 hasta i=9 con incremento 1 hacer
    sentencias
fpara
```



6.4. Sentencias.

- En pseudocódigo:

para *contador* = *vi* **hasta** *vf* **con incremento** *n*

hacer

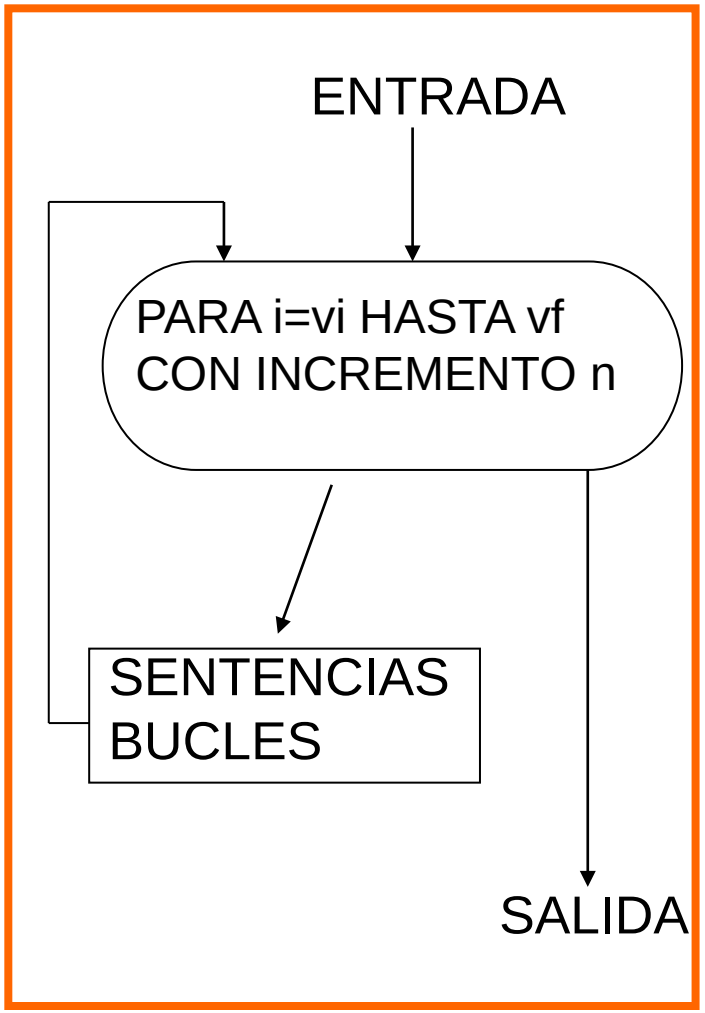
bloque de sentencias

fin_para

- Ejemplo: escribir los números pares del 2 al 50

6.4. Sentencias.

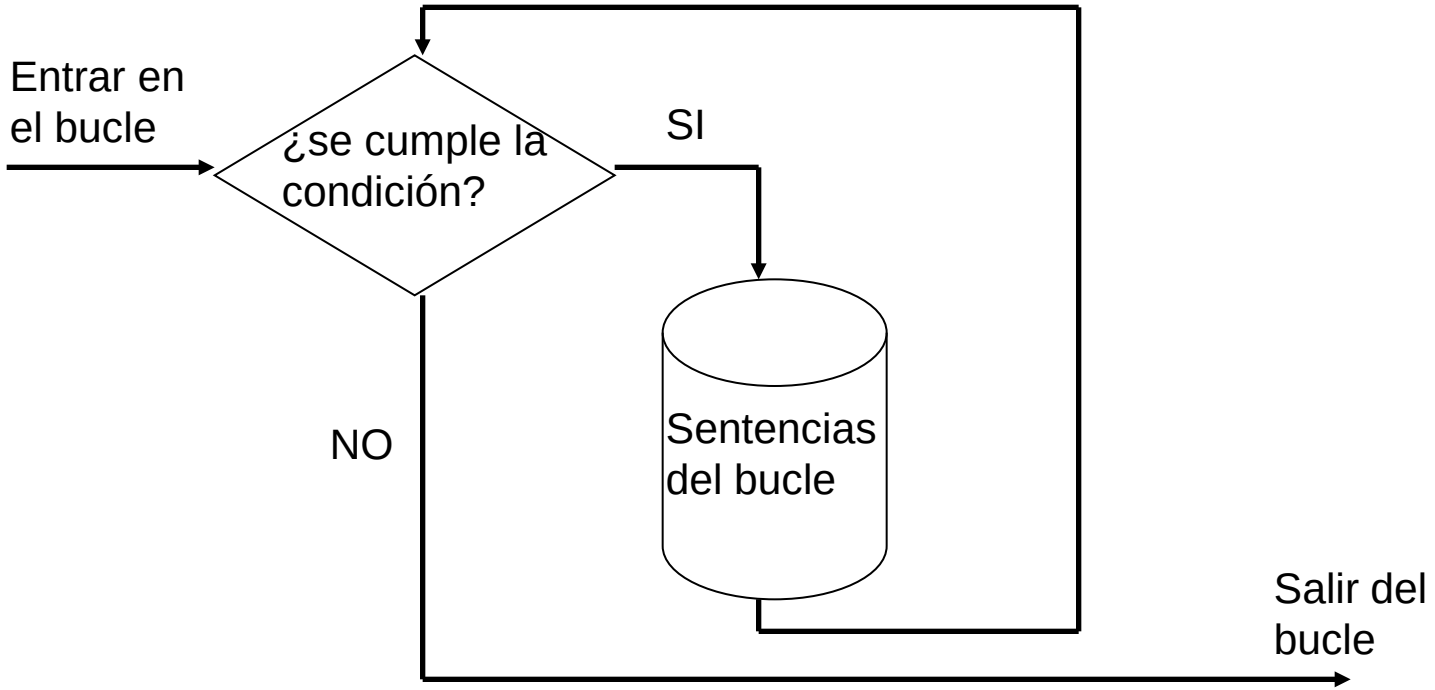
- En diagrama de flujo
- En C++:




```
for (inicialización ; condición ; incremento)
    sentencias;
```

6.4. Sentencias.

2. Ciclo condicional: se repiten las sentencias mientras se cumple una condición. Bucle **MIENTRAS**



6.4. Sentencias.

En pseudocódigo

mientras *condición* **hacer**

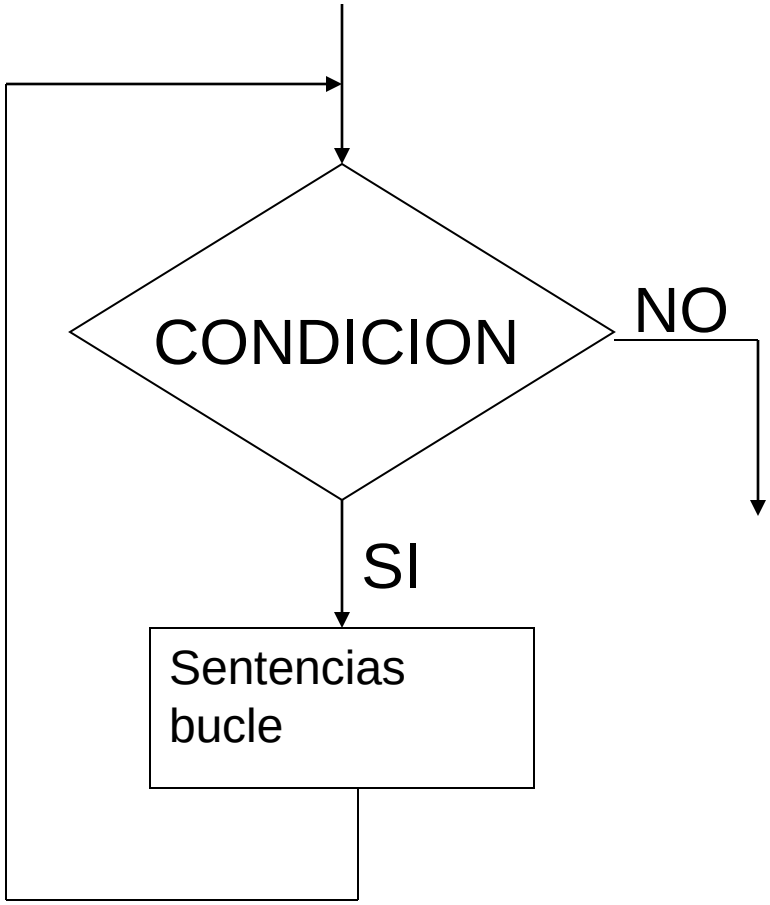
bloque de sentencias

fin_mientras

En C++:

```
while ( condición )
    Sentencias;
```

En diagrama de flujo



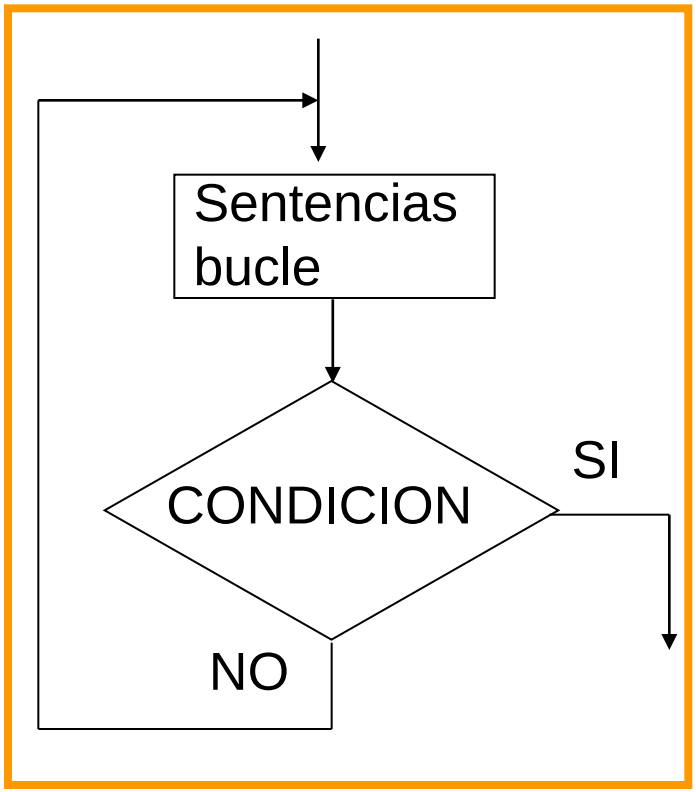
6.4. Sentencias.

- Ciclo condicional: se repiten las sentencias hasta que se cumple una condición. Bucle REPETIR

En pseudocódigo

```
repetir
        bloque de sentencias
hasta condición
```

En diagrama de flujo

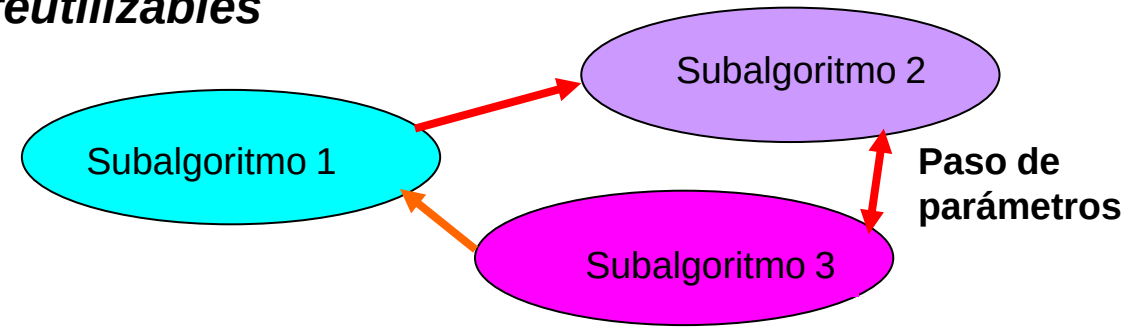


- En C++: La sentencia es ejecutada repetidamente mientras la condición resulte verdadera. Si no se especifica condición se asume que es **true**, y el bucle se ejecutará indefinidamente.

```
do
      Sentencias
while ( condición );
```

6.5. Subalgoritmos

- Un subalgoritmo es una parte de un algoritmo. Se utiliza para descomponer en varias partes un algoritmo que resuelve un problema complejo.
- Esta característica permite que los algoritmos sean ***simples***, ***modulares*** y ***reutilizables***



  - ***Parámetros* formales** son las variables que utiliza el algoritmo llamado para emitir o recibir datos a o desde el algoritmo llamante
  - ***Parámetros actuales*** son las variables, constantes o expresiones utilizadas por el algoritmo llamante.
  - El paso de parámetros se puede realizar por ***valor*** o por ***referencia***
  - Existen datos ***globales*** y datos ***locales***

6.5. Subalgoritmos

- Las ***funciones*** reciben valores de entrada y devuelven un valor que es el resultado de la función

  - **funcion** *nombre_funcion* (parm1, parm2,..)

- El algoritmo invoca a la función con un nombre y una lista de parámetros actuales
  - *nombre_funcion* (parm1, parm2,..)

- En C++:

```
tipo nombre([tipo v1, ..., tipo vN]) {
    [declaraciones;]
    sentencias;
    [return valor_devuelto;]
}
```

6.6. Resumen.

- El Pseudocódigo es una descripción de un algoritmo más estructurada que la verbal pero menos formal que la de un lenguaje de programación.
- El Pseudocódigo es una mezcla de lenguaje natural y conceptos de programación de ato nivel que describen las principales ideas que están en una implementación genérica de una estructura de datos o algoritmo.
- Ejemplo: hallar el elemento mayor de un array.

```
Algoritmo arrayMax(A, n; var currentMax;):
    Entrada: Un array A que almacena n enteros.
    Salida: El máximo elemento en A.
    comienza
        currentMax ← A[0]
        para  i← 1 hasta n -1 hacer
            si currentMax < A[i] entonces
                currentMax ← A[i]
            fsi
        fpara
        return currentMax
    Termina
```

- Pseudocódigo es la notación preferida para describir algoritmos.

6.6. Resumen.

- **NOTACIÓN:**
  - **Expresiones**: usa símbolos matemáticos standard para describir expresiones numéricas y booleanas
    - usa ← para asignación (“=” in C++)
    - usa **=** relación de igualdad (“==” in C++)
    - Indexación de arrays: **A[i]**
  - **Declaración de métodos**:
    - **Algoritmo** nombre(***param1, param2***) o
    - **funcion** nombre(***param1, param2***) o
    - **procedimiento** nombre(***param1, param2***)
  - **Bloques Programación**
    - Estructuras de decisión:
      - **si ... entonces ... [otro ...] …fsi**
    - Bucle mientras:
      - **mientras ... hacer…fmientras**
    - Bucle repetir:
      - **repetir... hasta**
    - Bucle-for:
      - **para ... hacer…fpara**
  - **Métodos**:
    - llamadas: nombre(args)
    - returns: **return** value

6.6. Resumen. Análisis de algoritmos.

- **Operaciones Primitivas:** se pueden identificar en el pseudocódigo instrucciones de bajo nivel independientes del lenguaje de programación.
- Ejemplos:
  - llamar un método y retornar de un método
  - operaciones aritméticas (e.g. suma)
  - comparación de dos números, etc.
- Inspeccionando el pseudocódigo se puede ***contar*** el número de operaciones primitivas ejecutadas por un algoritmo.