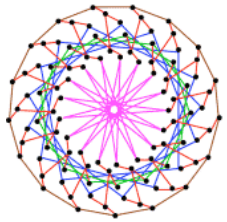
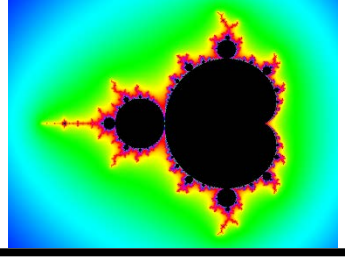
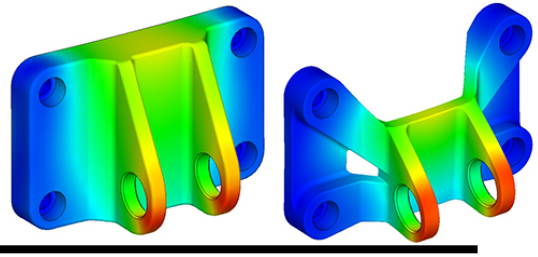




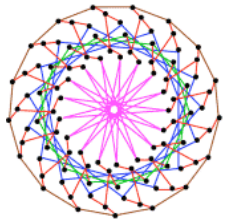
# Introducción a la Computación


Pedro Corcuera

Dpto. Matemática Aplicada y
Ciencias de la Computación

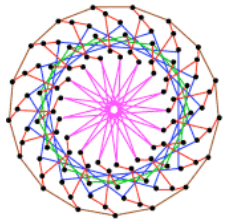
**Universidad de Cantabria**

**corcuerp@unican.es**



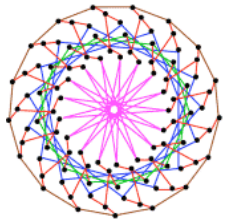
# Indice

- Computación
- Algoritmos
- Lenguajes de programación
- Técnicas generales de resolución de problemas.

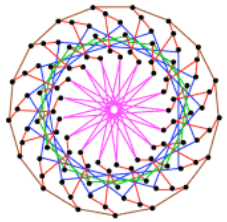


# Objetivos

- Introducir los conceptos básicos de computación, algoritmos, complejidad computacional, lenguajes de programación y técnicas generales de resolución de problemas.

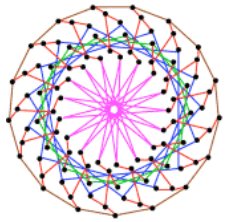


# Computación



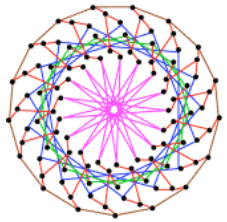
# Definición

- **Computación**: tiene su origen en el vocablo en latín *computatio*. Actividad orientada al estudio y análisis de métodos, técnicas, procesos, desarrollos (algoritmos) con el fin de almacenar, procesar, transmitir y hacer uso de información y datos.
- Herramienta: computadora. Formato: digital.
- *Nota: Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes. Science is not about tools. It is about how we use them and what we find out when we do.*



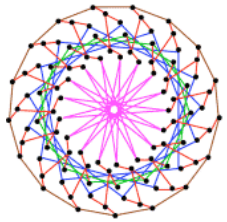
# Áreas de la Computación

- Arquitectura de computadoras (Computer engineering)
- Ingeniería de programación (Software engineering)
- Ciencias de la computación (Computer science)
- Sistemas de Información (Information systems)
- Tecnologías de la información (Information technology)



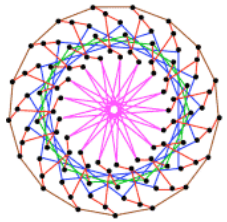
# Tipos de Computación

- Computación personal
  - Escritorio, portátiles, móviles. Se utiliza lenguajes de programación para el desarrollo de aplicaciones.
  - Virtualización. Permite crear, a través de software, de una versión virtual de algún recurso tecnológico (p.e. sistema operativo, dispositivo de almacenamiento).
- Computación grid, distribuída, cluster
- Computación en la nube (Cloud Computing): paradigma que permite ofrecer servicios de computación a través de una red (mayor uso Internet)



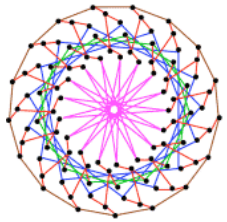
# Modelo de computación

- Un modelo de computación define un conjunto de operaciones permitidas usadas en el cómputo y sus respectivos costos.
- Permite analizar los recursos de cómputo requeridos: **tiempo de ejecución y espacio de memoria**, así como las limitaciones de algoritmos o computadores. Ejemplos: *máquinas de Turing*, funciones recursivas.
- El modelo de computación explica cómo el comportamiento del sistema entero es el resultado del comportamiento de cada uno de sus componentes.



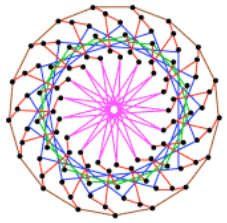
# Programas soporte para computación

- Sistemas operativos
  - Ordenadores personales: MS Windows, Linux, OS X
  - Móviles, tabletas: Android, iOS, Windows Mobile
- Lenguajes de programación
  - Compilados: Fortran, C, C++, C#, Visual Basic
  - Interpretados: Java (JVM), JavaScript, Python, Matlab, Mathematica
- Entornos integrados de desarrollo (IDE)
  - Microsoft Visual Studio, Xcode, Eclipse, NetBeans

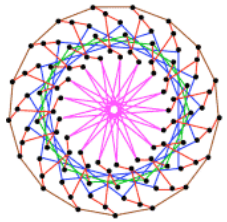


# Sistema Operativo

- Un sistema operativo es un programa (o conjunto de programas) de control que actúa como interfaz entre el usuario y el ordenador con objeto de facilitar su uso de manera eficiente.
- Objetivos y funciones:
  - Control
  - Facilidad
  - Eficiencia
- Ejemplo de uso de un SO con comandos: Disk Operating System

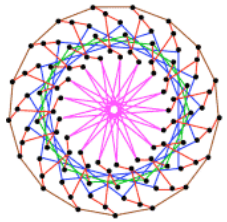


# Algoritmos



# Definición

- **Algoritmo** es una secuencia ordenada de instrucciones que resuelve un problema concreto
- Ejemplos:
  - Algoritmo de la media aritmética de N valores.
  - Algoritmo para la resolución de una ecuación de segundo grado.
- Niveles de detalle de los algoritmos:
  - Alto nivel: no se dan detalles.
  - Bajo nivel: muchos detalles.



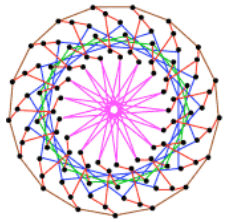
# Propiedades

## Necesarias o básicas:

- Corrección (sin errores).
- Validez (resuelve el problema pedido)
- Precisión (no puede haber ambigüedad).
- Repetitividad (en las mismas condiciones, al ejecutarlo, siempre se obtiene el mismo resultado).
- Finitud (termina en algún momento). Número finito de órdenes no implica finitud.
- Eficiencia (lo hace en un tiempo aceptable)

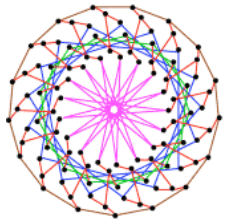
## Deseables:

- Generalidad
- Fácil de usar
- Robustez



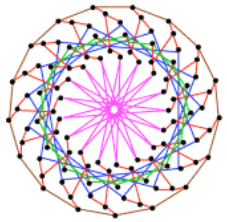
# Algoritmos: Métodos de Representación

- Verbal
- Diagramas de flujo
- Diagramas de Bloques, Cajas o de Nassi-Shneiderman
- Gráficos
- Pseudocódigo
- Representaciones algebraicas (fórmulas y expresiones)



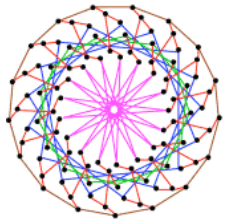
# Modificación de algoritmos

- *Generalización y extensibilidad*: proceso de aplicar el algoritmo a más casos y de incluir más casos dentro del algoritmo
- *Robustez*: proceso de hacer un algoritmo mas fiable o robusto (se recupera de errores), anticipando errores de entrada u otras dificultades.



# Algoritmos alternativos y equivalentes

- Pueden haber muchas formas de llevar a cabo un algoritmo.
- En esos casos la elección se basa en la eficiencia (memoria y velocidad).
- El *análisis de algoritmos* estudia la cantidad de recursos que demanda la ejecución de un algoritmo.
- Preocupa más el tiempo de ejecución de un algoritmo: *Complejidad del algoritmo*

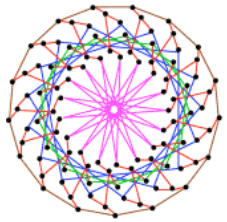


# Programación estructurada

- Método para construir algoritmos a partir de un número pequeño de bloques básicos.
- Formas fundamentales:
  - *Secuencia*: indica secuencia temporal lineal de las acciones a realizarse.

    A
    B
  - *Selección*: especifica una condición que determina la acción a realizarse.

```
if C
        D
else
        E
```



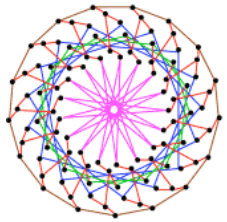
# Programación estructurada

– *Repetición*: indica que una o más acciones deben repetirse un determinado número de veces.

```
while G do
        H
```

– *Invocación*: corresponde al grupo de acciones agrupadas bajo un nombre.

Calcula_promedio



# Pseudocódigo

- Lectura o entrada de datos

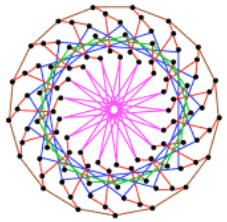
  Input

- Repetición

  **while** expr
    instrucción
  **endwhile**

  **for** i = 1 to m
    instrucción
  endfor

  do
    instrucción
  **while** expr



# Pseudocódigo

- Decisión

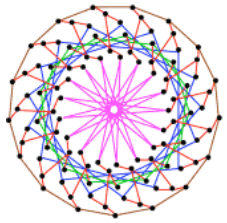
  **if** expr

    instrucción

  endif

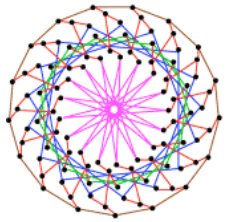
- Escritura o salida de datos

  Output



# Análisis de Algoritmos: Complejidad

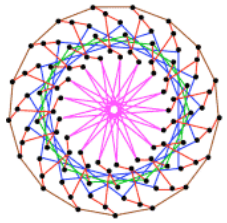
- Para comparar algoritmos se pueden estudiar desde dos puntos de vista:
  - el tiempo que consume un algoritmo para resolver un problema (complejidad temporal) ⟵ más interés
  - la memoria que necesita el algoritmo (complejidad espacial).
- Para analizar la complejidad se cuentan los pasos del algoritmo en función del tamaño de los datos y se expresa en unidades de tiempo utilizando la notación asíntotica “O- Grande” (complejidad en el peor caso).



# Análisis de Algoritmos: Complejidad

Problema: Buscar el mayor valor en una lista de números desordenados (array)

```
Algoritmo: (n = número de elementos)
1   max = s1
2   i = 2
3   while i <= n
4     if si > max then
5               max = si
6      i = i + 1
7   endwhile
```



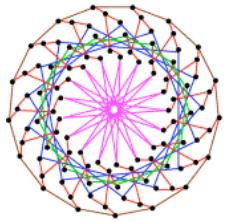
# Análisis de Algoritmos: Complejidad

Número de operaciones realizadas (unid):

| Línea | Operaciones | Tiempo |
|---|---|---|
| 1 | indexado y asignación | 2 |
| 2 | asignación | 1 |
| 3 | comparación | 1 |
| 4,5,6 | 2 indexado, comparación, 2 asignación, suma | 6 |

**Tiempo total**:

$$t(n) = 2 + 1 + (n - 1) + 6 \cdot (n - 1) = 3 + 7 \cdot (n - 1) = 7n - 4$$



# Notación asintótica

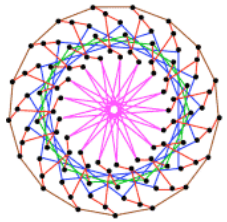
- Es útil concentrarse en la tasa de crecimiento del tiempo de ejecución t como función del tamaño de la entrada *n*.
- Se usa la notación **Big O** (**O grande** cota superior al ritmo de crecimiento de un algoritmo):

  Sean f(*n*) y g(*n*) funciones no negativas, f(*n*) es O(g(*n*)) si hay un valor c > 0 y

  $n_0 \geq 1$ tal que $f(n) \leq cg(n)$ para $n \geq n_0$
- Se dice que f(*n*) **es de orden** g(*n*)

  Ej: $7n - 4$ es O(*n*) si c=7 y $n_0 = 1$
- Generalmente para cualquier polinomio

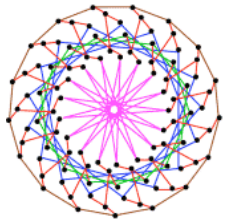
  $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$ es $O(n^k)$



# Eficiencia de un algoritmo

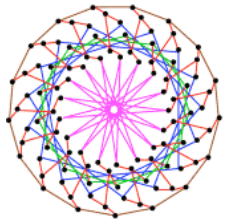
- Definición de eficiencia: Un algoritmo es eficiente si tiene un tiempo de ejecución polinómico.
- Tiempos de ejecución (redondeados) de diferentes complejidades de algoritmos en entradas de tamaños crecientes, para un procesador que realiza un *millón de instrucciones de alto nivel por segundo*. Si el tiempo de ejecución > $10^{25}$ años, se considera que el algoritmo toma un tiempo "very long".

| | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $1.5^n$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| $n = 10$ | < 1 sec | < 1 sec | < 1 sec | < 1 sec | < 1 sec | < 1 sec | 4 sec |
| $n = 30$ | < 1 sec | < 1 sec | < 1 sec | < 1 sec | < 1 sec | 18 min | $10^{25}$ years |
| $n = 50$ | < 1 sec | < 1 sec | < 1 sec | < 1 sec | 11 min | 36 years | very long |
| $n = 100$ | < 1 sec | < 1 sec | < 1 sec | 1 sec | 12,892 years | $10^{17}$ years | very long |
| $n = 1{,}000$ | < 1 sec | < 1 sec | 1 sec | 18 min | very long | very long | very long |
| $n = 10{,}000$ | < 1 sec | < 1 sec | 2 min | 12 days | very long | very long | very long |
| $n = 100{,}000$ | < 1 sec | 2 sec | 3 hours | 32 years | very long | very long | very long |
| $n = 1{,}000{,}000$ | 1 sec | 20 sec | 12 days | 31,710 years | very long | very long | very long |



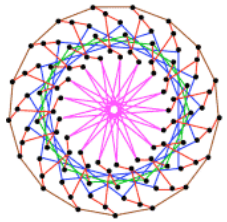
# Desarrollo de algoritmos

- Análisis del problema
  - Dominio del problema
  - Modelo
- Diseño del algoritmo
  - Refinamiento sucesivo
  - Top down o botton up
- Análisis del algoritmo
  - Cuánto tarda en dar una solución? Se puede modificar para aumentar la eficiencia?
  - Análisis de la Complejidad
- Verificación del algoritmo
  - Comprobar que es correcto



# Algoritmos iterativos

- Muchos algoritmos se basan en *ciclos o bucles*, es decir en la ejecución de una serie de pasos repetitivos.
- **Iteración** significa hacer algo de forma repetida.



# Recursión

- **Recursión** mecanismo de repetición que expresa una solución como función de la solución de un problema de menor tamaño.

  *Ejemplo*: Suma de una lista de números ($a_i$, i=1,..,n).

  Sum($a_i$, i=1,..,n)= $a_n$+ Sum($a_i$, i=1,..,n-1)

  si i=1 Sum = $a_1$

  Input $a_i$, i=1,..,n

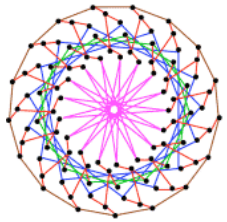
  SumaListaRec(n)

  **if** n=1 **then** sum = $a_1$

  **else** sum = $a_n$ + SumaListaRec(n-1)

  **endif**

  **return** sum

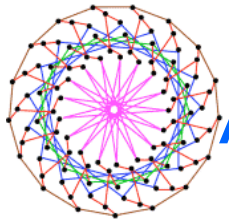


# Algoritmos recursivos

- Son algoritmos que expresan la solución de un problema en términos de una llamada a sí mismo (llamada recursiva o recurrente)
- Ejemplo típico: Factorial (n!) de un número

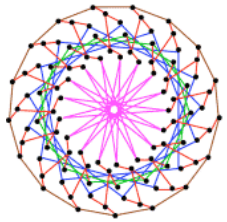
$$n! = \begin{cases} 1 & si\ n = 0 \\ n \cdot (n-1)! & si\ n > 0 \end{cases}$$

- Son más ineficientes que los iterativos pero más simples y elegantes
- Todo algoritmo recursivo tiene su equivalente iterativo



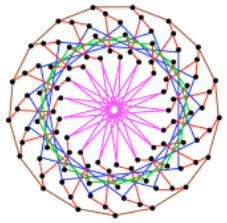
# Algoritmos recursivos – definición y diseño

- Un método recursivo es un método que se llama a sí mismo dentro del cuerpo del método.
- Para diseñar correctamente un algoritmo recursivo, es necesario:
  - Establecer correctamente la ley de recurrencia.
  - Definir el procedimiento de finalización del algoritmo recursivo (normalmente con el valor o valores iniciales).



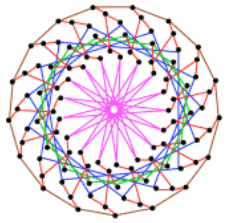
# Algoritmos recursivos – Verificación

- Para verificar funciones recursivas se aplica el método de las tres preguntas:
  - *pregunta Caso-Base*: Hay una salida no recursiva de la función, y la rutina funciona correctamente para este caso "base"?
  - *pregunta Llamador-Más Pequeño*: Cada llamada recursiva a la función se refiere a un caso más pequeño del problema original?
  - *pregunta Caso-General*: Suponiendo que las llamadas recursivas funcionan correctamente, funciona correctamente toda la función?



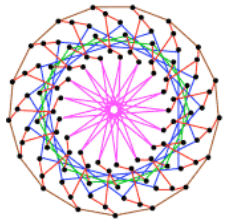
# Paradigmas de programación

- "*Un **paradigma de programación** indica un método de realizar cómputos y la manera en que se deben estructurar y organizar las tareas que debe llevar a cabo un programa* "
- Los paradigmas fundamentales están asociados a determinados **modelos de cómputo**.
- También se asocian a un determinado **estilo de programación**
- Los lenguajes de programación suelen implementar, a menudo de forma parcial, **varios** paradigmas.



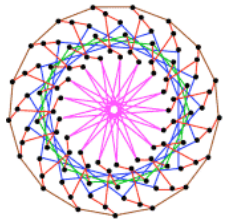
# Tipos de paradigmas de programación

- Los **paradigmas fundamentales** están basados en diferentes **modelos de cómputo** y por lo tanto afectan a las construcciones más básicas de un programa.
- La división principal reside en el enfoque **imperativo** (indicar el **cómo** se debe calcular) y el enfoque **declarativo** (indicar el **qué** se debe calcular).
  - El enfoque declarativo tiene varias ramas diferenciadas: el paradigma **funcional**, el paradigma **lógico**, la programación **reactiva** y los lenguajes **descriptivos**.



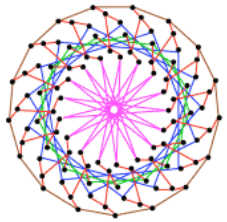
# Tipos de paradigmas de programación

- Otros paradigmas se centran en la estructura y organización de los programas, y son compatibles con los fundamentales:
  - Ejemplos: Programación estructurada, modular, **orientada a objetos**, **orientada a eventos**, programación genérica.
- Por último, existen paradigmas asociados a la **concurrencia** y a los **sistemas de tipado**.



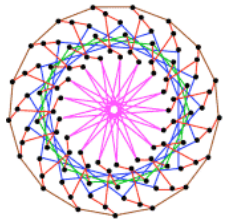
# Paradigma Imperativo

- Describe **cómo** debe realizarse el cálculo, no el porqué.
- Un cómputo consiste en una serie de sentencias, ejecutadas según un control de flujo **explícito**, que **modifican el estado** del programa.
- Las variables son **celdas de memoria** que contienen datos (o referencias), pueden ser modificadas, y representan el **estado** del programa.
- La sentencia principal es la **asignación**.



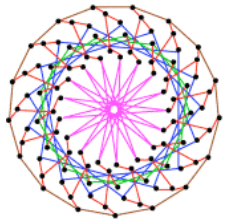
# Paradigma Imperativo

- Es el estándar ‘de facto’.
  - Asociados al paradigma imperativo se encuentran los paradigmas **procedural**, **modular**, y la programación **estructurada**.
  - El lenguaje representativo sería FORTRAN, junto con COBOL, BASIC, PASCAL, C, ADA.
  - También lo implementan Java, C++, C#, Eiffel, Python, ...



# Computabilidad

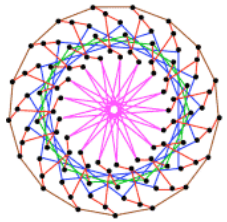
- **Algoritmo**: Procedimiento sistemático que permite resolver un problema en un número finito de pasos, cada uno de ellos especificado de manera efectiva y sin ambigüedad.
- **Función computable**: Aquella que puede ser calculada mediante un dispositivo, dado un tiempo y espacio de almacenamiento ilimitado (pero finito)
- No importa la eficiencia, sino la posibilidad de ser calculada.



# Computabilidad

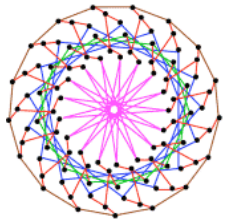
- Hay problemas que *no son computables o intratables*: ningún computador pueden resolverlos sin entrar en ciclos infinitos (nunca paran) para ciertas entradas.
- Un ejemplo de problema intratable es el problema del vendedor viajero:

  un vendedor tiene que visitar un cierto número de ciudades para lo que tiene un presupuesto fijo y conoce lo que le cuestan los viajes entre todas las ciudades. La cuestión es si puede encontrar un recorrido, que no exceda el presupuesto que tiene, que parta y termine en la ciudad en que vive y que visite una sola vez cada una de las ciudades requeridas.
- Existen técnicas que permiten obtener soluciones parciales o aproximadas para problemas no computables.



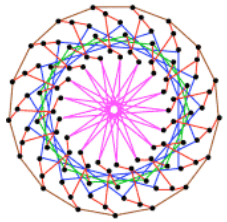
# Técnicas de diseño de algoritmos

- Algoritmos voraces (greedy): seleccionan los elementos más prometedores del conjunto de candidatos hasta encontrar una solución. En la mayoría de los casos la solución no es óptima.
- Divide y vencerás: dividen el problema en subconjuntos disjuntos obteniendo una solución de cada uno de ellos para después unirlas, logrando así la solución al problema completo.
- Programación dinámica: intenta resolver problemas disminuyendo su coste computacional aumentando el coste espacial.
- Vuelta Atrás (Backtracking): se construye el espacio de soluciones del problema en un árbol que se examina completamente, almacenando las soluciones menos costosas.



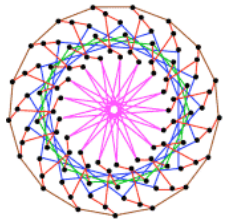
# Técnicas de diseño de algoritmos

- Ramificación y acotación: se basa en la construcción de las soluciones al problema mediante un árbol implícito que se recorre de forma controlada encontrando las mejores soluciones.
- Metaheurísticas: encuentran soluciones aproximadas (no óptimas) a problemas basándose en un conocimiento anterior (a veces llamado experiencia) de los mismos.
- Algoritmos determinísticos: El comportamiento del algoritmo es lineal: cada paso del algoritmo tiene únicamente un paso sucesor y otro ancesor.
- Algoritmos probabilísticos: algunos de los pasos de este tipo de algoritmos están en función de valores pseudoaleatorios

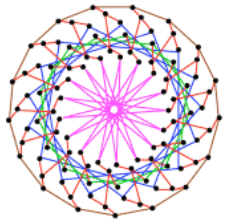


# Técnicas de diseño de algoritmos

- Algoritmos no determinísticos: El comportamiento del algoritmo tiene forma de árbol y a cada paso del algoritmo puede bifurcarse a cualquier número de pasos inmediatamente posteriores, además todas las ramas se ejecutan simultáneamente.
- Algoritmos paralelos: permiten la división de un problema en subproblemas de forma que se puedan ejecutar de forma simultánea en varios procesadores.



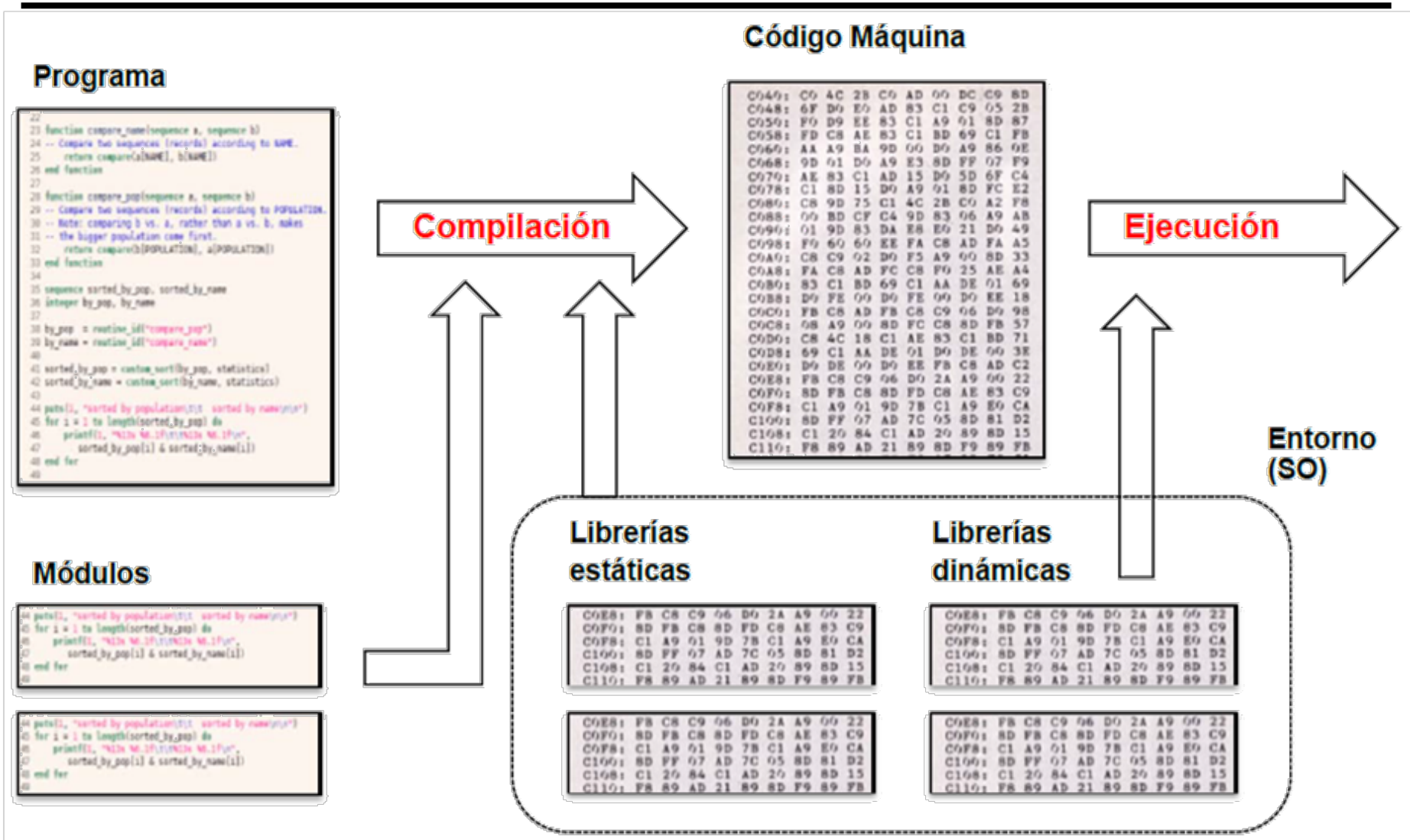
# Lenguajes de Programación

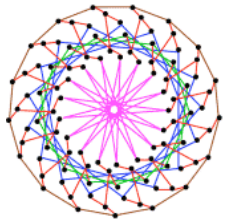


# Lenguajes de Programación

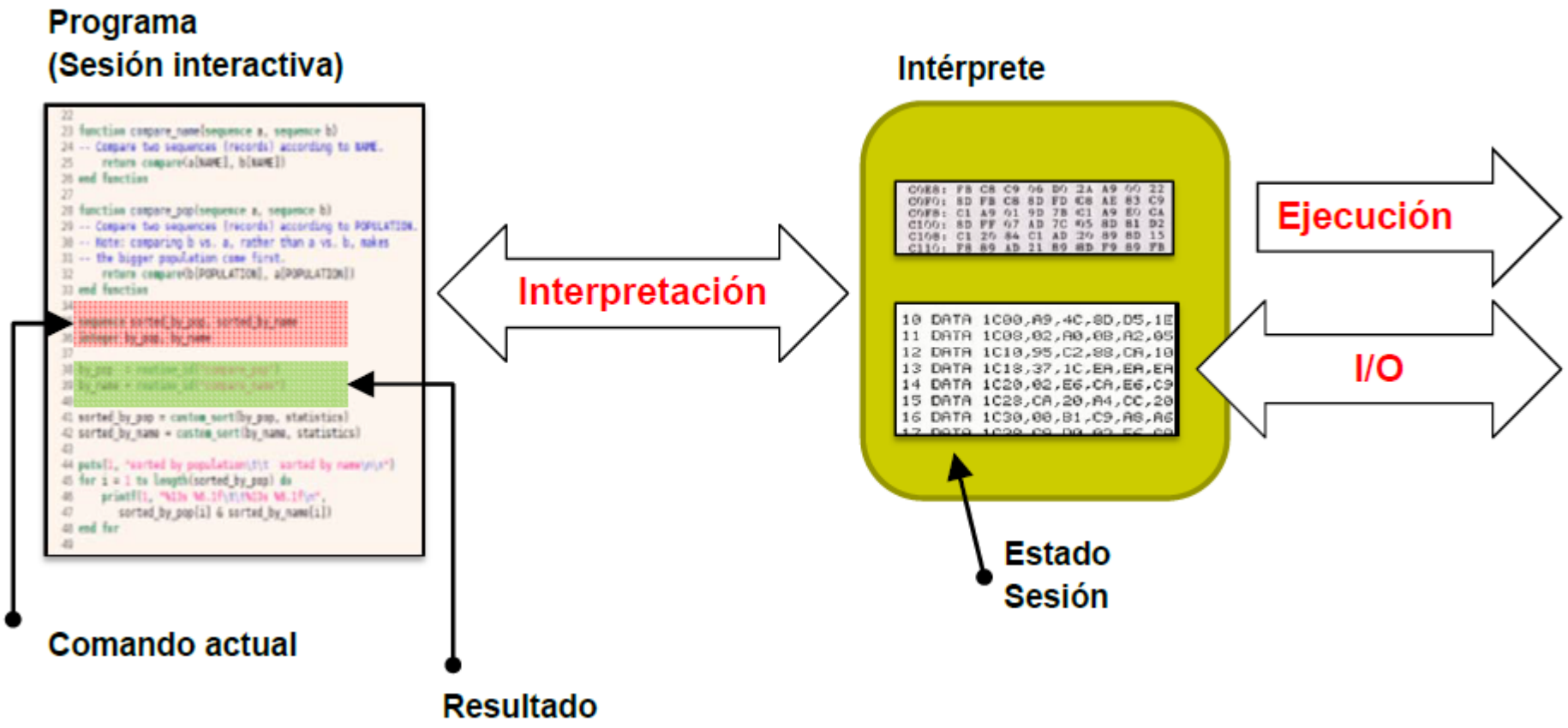
- Lenguaje artificial diseñado para expresar cómputos que pueden ser llevados a cabo por una máquina.
- Basado en un modelo de cómputo define un nivel de abstracción más elevado cercano al programador.
- Debe traducirse a un código que pueda entender el procesador: el **código máquina**.
- Modos de traducción:
  - Lenguaje Compilado
  - Lenguaje Interpretado (Entorno interactivo)
  - Lenguaje traducido a Código Intermedio (Java Bytecodes, .NET IDL, Python)

# Estrategias de traducción – Código compilado

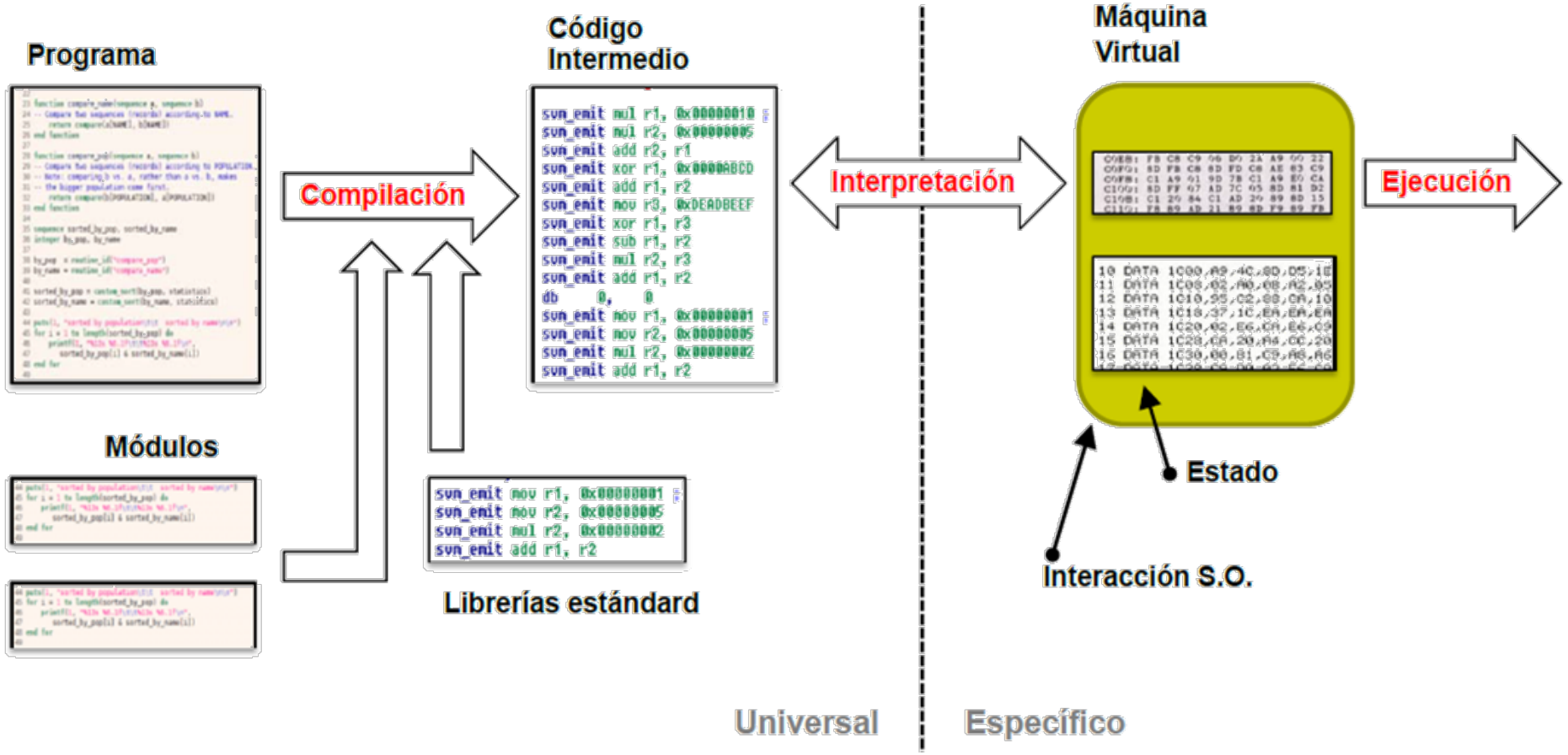


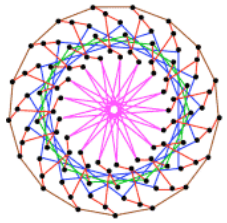


# Estrategias de traducción – Código interpretado



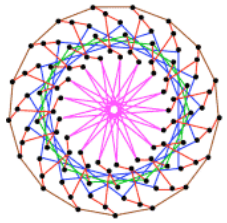
# Estrategias de traducción – Código intermedio





# Lenguajes de Programación

- History of programming languages
- Timeline of programming languages



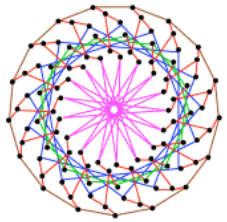
# Lenguajes de Programación - Sintaxis

- A "Hello, World!" program in Visual Basic.

```
Module Hello
   Sub Main()
      MsgBox("Hello, World!") ' Display message
   End Sub
End Module
```

- A "Hello, World!" script in JavaScript

```
<!DOCTYPE HTML><html><body><p>Header...</p>
   <script>
     alert('Hello, World!')
   </script>
   <p>...Footer</p></body>
</html>
```



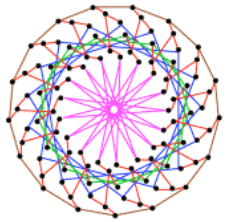
# Lenguajes de Programación - Sintaxis

- A "Hello, World!" program in **Matlab**

```
% Do it in command window
disp('Hello World!');
% Do it with GUI way
msgbox('Hello World!','Hello World!');
fprintf ( 1, '\n' );
fprintf ( 1, '  Hello, world!\n' );
```

- A "Hello, World!" program in **Java**

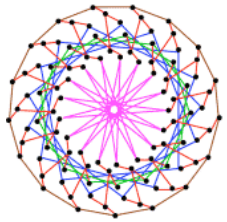
```
public class HelloWorld {
  public static void main(String[] args) {
      System.out.println("Hello, World");
  }
}
```



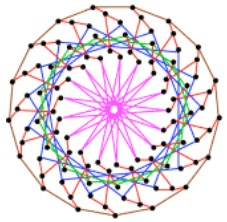
# Lenguajes de Programación - Sintaxis

- A "Hello, World!" program in Python

```
print('Hello World!')
```

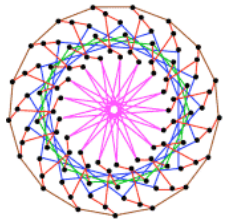


# Técnicas generales de resolución de problemas



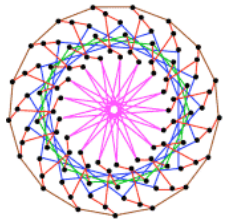
# Técnicas generales de resolución de problemas

- El *análisis de ingeniería* es un proceso sistemático para analizar y comprender los problemas que se encuentran en los diversos campos de la ingeniería.
- Para llevar a cabo este proceso de manera satisfactoria se debe estar familiarizado con técnicas generales de resolución de problemas.



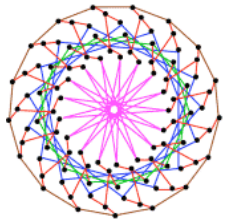
# Pasos para la resolución de problemas$_1$

- **Pensar** sobre el problema antes de intentar resolverlo.
- **Dibujar** un esquema del problema antes de empezar a resolverlo.
- **Identificar** el propósito general y los puntos clave.
- **Identificar** la información conocida (datos de entrada) y la información a calcularse (datos de salida).
- **Identificar** los principios de ingeniería básicos que pueden aplicarse.



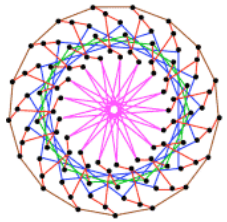
# Pasos para la resolución de problemas$_2$

- **Pensar** en la manera de resolver el problema antes de empezar la solución.
- **Desarrollar** la solución de manera ordenada y lógica.
- Después de obtener la solución, **pensar** en ella. Tiene sentido? →*Validación* de resultados
- Asegurarse que la **solución es clara y completa**.



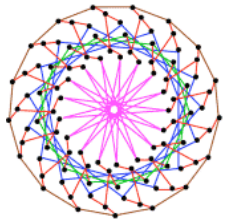
# Fundamentos de Ingeniería aplicables

- **Equilibrio**: La mayoría de problemas de estado estable (problemas en los que las cosas permanecen constantes respecto al tiempo) se basan en algún tipo de equilibrio. Ejm: Equilibrio de fuerzas, Equilibrio de flujo, Equilibrio químico
- **Leyes de Conservación**: Las más comunes son la conservación de masa y energía.
- **Fenómenos de cambio**: Tienen la forma de un potencial que cambia un flujo. Ejm: ley de Ohm, ley de Fourier.



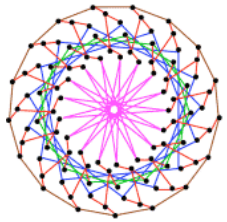
# Técnicas de solución matemática$_1$

- *Análisis de datos*. Técnicas estadísticas para obtener información a partir de un conjunto de datos.
- *Ajuste de curvas*. Trata de pasar una función que aproxime un conjunto de datos.
- *Interpolación*. Para obtener valores precisos de una variable dependiente a partir de un conjunto de datos tabulados.
- Solución de *ecuaciones algebraicas* simples y *lineales simultáneas*. Muchos problemas de ingeniería se representan como un conjunto de *eals*.
- *Evaluar derivadas e integrales*. En ingeniería se requiere calcular tasas de cambio y promedios.



# Técnicas de solución matemática$_2$

- *Optimización*. Seleccionar la mejor alternativa entre varias con un criterio óptimo.
- *Análisis económico de ingeniería*. Selección económica de alternativas.
- Solución de *ecuaciones diferenciales ordinarias y parciales*. Muchos problemas de ingeniería.
- *Representación y solución de problemas en grafos*. Típico en ingeniería.



# Procedimientos de solución matemática

- *Simbólicos*: soluciones que se expresan en términos algebraicos.
  - Mapple
  - Mathematica
  - Matlab (Symbolic toolbox)
  - MathCad
- *Numéricos*: soluciones expresadas en términos de números (o grafos).
  - Hojas de cálculo
  - Matlab
  - Lenguajes de programación (Fortran, C, Phyton, JavaScript)