3.7.2 EJERCICIOS

1.- Construir utilizando Apply una función que teniendo como argumentos una lista de números calcule su media geométrica.

2.-Utilizando Map y Apply construir una función que se llame Laplaciano , que tenga como argumentos una función f y una lista de variables y que calcule el Laplaciano de f.

Nota: El laplaciano de una función real de varias variables f es: $\Delta f = \frac{\partial^2 f}{\partial x_1^2} + \ldots\ldots + \frac{\partial^2 f}{\partial x_n^2}$

CAPÍTULO 4
PROGRAMACIÓN BASADA EN REGLAS DE TRANSFORMACIÓN

En este capítulo exponemos los fundamentos de la programación por reglas de transformación. Este modo de programar consiste en la definición de objetos y de sus reglas de transformación. Así el objeto deja de ser un tipo de dato, para convertirse en el dato junto con las operaciones que con el se pueden realizar. Esto no nos ha de resultar extraño, es algo habitual en matemáticas.

Esta tendencia es la que ultimamente se sigue en programación y a este estilo de programar se le denomina "programación con orientación al objeto". Resulta especialmente útil cuando se tratan de programar matemáticas simbólicas. Veremos algunos ejemplos donde esta forma de programar resulta particularmente eficaz (definición de producto exterior de tensores, derivada, integral ...).

Este estilo se basa: por un lado en la construcción de modelos para expresiones y por otro en la definición de reglas de transformación para estos modelos (reglas locales y reglas globales)

4.1.MODELOS

Un modelo en *Mathematica* es una plantilla ("pattern" en inglés) que representa a un tipo concreto de expresión.
Ya nos hemos encontrado con un ejemplo: en la definición de una función
F[x_]:=cuerpo , x_ representa a cualquier expresión llamada x. Es fundamental el símbolo "_", que expresa que x_ puede sustituirse por cualquier expresión.
Aunque los modelos se utilizan fundamentalmente en la definición de funciones, existen otras situaciones donde podemos encontrarlos:

• Se pueden usar para encontrar la posición de todas las expresiones de un tipo particular.

Position[{1,x,x^2,x^3},x^_}]

• Se pueden encontrar en sustituciones locales

f[a]+f[b]/.f[x_]->x^2
indica como deben sustituirse expresiones del tipo f[algo]

■ Debemos tener en cuenta que para la adaptación a modelos *Mathematica* recurre a la forma dada por FullForm.

• Por ejemplo:

1/x tiene como forma completa Power[-1,x] y por tanto se adapta al modelo x^_ y no al modelo x_/y_ que tiene como forma interna Times[x,Power[-1,y]]

■ Para ver si una expresión se adapta a un modelo o no se utiliza

MatchQ[exp,modelo] devuelve True si expresión se adapta a modelo y False en caso contrario

por ejemplo:

```
MatchQ[1 / x, x^_]
True

MatchQ[1 / x, x_ / y_]
False
```

■ 4.1.1.CONSTRUYENDO MODELOS PARA EXPRESIONES

■ 4.1.1.1. ALGUNOS MODELOS SENCILLOS

■ A continuación vemos la sintaxis para la construcción de modelos.

_ representa cualquier expresión.
x_ representa cualquier expresión llamada x.
x:modelo representa una expresión llamada x verificando modelo.

• Por ejemplo, x:_^_ significa que x es una expresión del tipo algo elevado a algo.

x_h representa cualquier expresión llamada x con cabecera h.

Las cabeceras más comunes son: Integer, Rational, Real, Complex, List, Symbol, o cualquier otra que nosotros construyamos.

• Por ejemplo, un modelo para una lista sería

```
x_List
```

```
x_List
```

```
MatchQ[{a, b, c}, x_List]
```

```
True
```

```
MatchQ[c, x_List]
```

```
False
```

•Un modelo para los enteros sería x_Integer

```
MatchQ[3, x_Integer]
```

```
True
```

```
MatchQ[3., x_Integer]
```

```
False
```

4.1.1.2. PONIENDO CONDICIONES A MODELOS

modelo/;condición representa un modelo de la forma modelo y que verifica condición (condición es una expresión booleana).

• Por ejemplo un modelo para los enteros positivos sería x_Integer/;x>o

```
MatchQ[3, x_Integer /; x > 0]
```

```
True
```

```
MatchQ[-3, x_Integer /; x > 0]
```

```
False
```

Algunas de las condiciones más habituales son:

• **NumberQ[exp]** testea si exp es un número o no.
• **IntegerQ[exp]** testea si exp es un número entero o no.
• **EvenQ[exp]** testea si exp es número par o no.
• **OddQ[exp]** testea si exp es un número impar o no.
• **PrimeQ[exp**] testea si expr es un número primo o no.
• **PolynomialQ[exp,{x1,..}**] testea si exp es un polinomio en x1,x2,...
• **VectorQ[exp**] testea si exp es un vector o no.
• **MatrixQ[exp]** testea si exp es una lista que representa a una matriz.
• **VectorQ[exp,NumberQ], MatrixQ[exp,NumberQ]** testean si exp son vectores o matrices cuyos elementos son números.
•**VectorQ[exp,test], MatrixQ[exp,test]** testean si exp son vectores o matrices para los que test da True.

modelo?test representa a un modelo de la forma modelo y para el que test da True
(test es una función pura o el nombre de una función).

• Usando ? en vez de /;condición se pueden conseguir definiciones más sútiles.

•Por ejemplo:

```
{x_Integer, y_Integer}?(Function[v, v.v > 4])
```

representa una lista de dos enteros en las que el módulo al cuadrado del vector es mayor que 4.

• O este otro:

```
x_?NumberQ
```

que representa a aquellas expresiones que son números. Notar que test se introduce en forma de función pura.

4.1.1.3. MODELOS REPRESENTANDO ALTERNATIVAS

modelo1 | modelo2 representa a una expresión del tipo 1 o del tipo2.

Con este modelo solucionamos el problema de referirnos a 1,x,x^2,...con un único modelo: 1 | x | x^_

```
MatchQ[x, 1 | x | x^_]
```

```
True
```

```
MatchQ[x^2, 1 | x | x^_]
```

```
True
```

■ 4.1.1.4. MODELOS QUE REPRESENTAN UN NÚMERO VARIABLE DE ARGUMENTOS

x__ representa a uno o más argumentos.
x__h representa a uno o más argumentos con cabecera h.
x___ representa a cero o más argumentos
x___h representa a cero o más argumentos con cabecera h

•Por ejemplo

```
MatchQ[, x___]
```

```
True
```

■ 4.1.1.5. VALORES POR DEFECTO EN ARGUMENTOS

x:valor representa un modelo x que si es omitido tiene como valor por defecto valor

Mathematica tiene algunos valores por defecto construidos, por ejemplo:

x_. que tiene valor por defecto distinto dependiendo de donde se use.

• En x_+y_. , y tiene como valor por defecto 0 mientras que en x_* y_. tiene valor por defecto 1.

• El uso de valores por defecto es especialmente útil en la definición de funciones con valores opcionales.
Por ejemplo:

```
F[x_ + y_.] := x * y
```

```
F[2]
```

```
0
```

```
F[2 + x]
```

```
2 x
```

Así, para F[2] obtenemos el valor 0, ya que y por defecto vale cero

■ 4.1.1.6. MODELOS QUE ESPECIFICAN UNA MISMA EXPRESIÓN REPETIDA

exp.. representa a una expresión repetida una ó más veces
exp... representa a una expresión repetida cero ó más veces

Por ejemplo a.. representa a cualquier expresión del tipo a ó a,a ó a,a,a etc..

■ 4.1.1.7. UN EJEMPLO

Encontrar un modelo para los números complejos con partes real e imaginaria números enteros

Es útil obtener primero la forma completa de una expresión que se adapte al modelo.

```
FullForm[3 + 5 I]
```

```
Complex[3, 5]
```

El modelo podría ser

```
Complex[x_Integer, y_Integer]
```

Comprobemoslo

```
MatchQ[3 + 5 I, Complex[x_Integer, y_Integer]]
```

```
True
```

```
MatchQ[3. + 5 I, Complex[x_Integer, y_Integer]]
```

```
False
```

4.1.1.8. EJERCICIOS

Construir modelos para los siguientes tipos de expresiones:

- Un número real de cualquier tipo
- Un número racional o entero
- Una lista que no contiene sublistas
- Un vector de números
- Una matriz de dos columnas
- Un polinomio en el que los coeficientes que aparecen sean todos 2
- Una matriz con ceros debajo de la diagonal
- Un número complejo del primer cuadrante

4.2. REGLAS DE TRANSFORMACION LOCALES

4.2.1. SINTAXIS

La sintaxis más simple para una regla de transformación es:

exp/.lhs->rhs que sustituye en exp lhs por rhs.

Otra sintaxis equivalente para lo anterior es:

ReplaceAll[exp,lhs->rhs]

Esta forma básica admite algunas modificaciones:

■ Podemos aplicar varias reglas con exp/.{lhs1->rhs1,lhs2->rhs2,...} las reglas se aplican de forma sucesiva y ordenada, y cuando se aplica la primera de ellas, ya no se aplican las demás.

• Por ejemplo:

```
{x^2, x^3, x^4} /. {x^3 -> u, x^(n_) -> p[n]}
{p[2], u, p[4]}
```

obtenemos como resultado {p[2],u,p[4]}. Esto es porque la regla para x^3 se aplica la primera, y por tanto la regla para x^_n no se aplica.

■ Con exp/.lhs->rhs sólo aplicamos la regla a cada parte de la expresión una vez.

• Por ejemplo si escribimos

```
x + a /. {x -> a, a -> b}
a + b
```

no obtenemos b+b como cabría esperar,sino a+b por que la regla se aplica una sola vez. Si queremos que la regla se aplique sucesivamente hasta que la expresión deje de cambiar tenemos que escribir // en vez de /.

exp//.lhs-> rhs que sustituye en exp lhs por rhs hasta que la expresión deje de cambiar

Otra sintaxis equivalente para lo anterior es:

ReplaceRepeated[exp,lhs->rhs]

Hay que tener cuidado cuando se aplicamos reglas circulares (por ejemplo x->x+1) porque podemos entrar en un proceso iterativo infinito. ReplaceRepeated admite como tercer argumento opcional MaxIteration cuyo valor por defecto es 65536.

• Ejemplo:

```
x^2 //. {x -> x + 1}
ReplaceRepeated::rrlim : Exiting after x^2 scanned 65536 times.
(65 536 + x)^2
```

4.3. FUNCIONES

Las funciones son las reglas de transformación globales, en el sentido de que se aplican a lo largo de todo el libro de notas o de una sesión, a diferencia de las reglas de transformación locales que hemos visto en la sección anterior, que sólo se aplican sobre la expresión a la que van acompañando.

La forma más sencilla de definir una función es

f[x_]:=cuerpo de la función

donde expresiones del tipo de "x" se transforman según cuerpo. Sin embargo , podemos obtener definiciones más sútiles: eligiendo modelos adecuados podemos conseguir que f sólo se aplique a determinados tipos de expresiones, podemos definir atributos para f, como que sea asociativa o conmutativa..., o podemos definir funciones con muchas opciones tipo Plot....

■ 4.3.1. ATRIBUTOS DE UNA FUNCIÓN

Cuando definimos una función tenemos la posibilidad de asociarle atributos tales como que sea asociativa, conmutativa...

Algunas instrucciones útiles para manejar atributos son las siguientes:

Attributes[f] devuelve los atributos de f
Attributes[f]={atr1,atr2,...} definimos el espacio de atributos de f
SetAttributes[f,atr] añade a f el atributo atr.
ClearAttributes[f,atr] quita a f el atributo atr.

Algunos de los atributos más comunes para funciones son:

Ordeless , una función conmutativa.
Flat , una función asociativa.
OneIdentity, f cuando se aplica sobre un solo argumento es la identidad. Funciones como Plus, Times ó Dot tienen este atributo.
Listable, al aplicarla sobre listas se aplica sobre cada elemento de la lista
Constant, todas las derivadas de f son cero.
Protected, los valores de f no pueden ser cambiados.
Locked, los atributos de f no pueden ser cambiados.
Read Protected, los valores de f no pueden ser leidos.

Atributos de funciones

• Cuando tenemos una función borramos sus valores con Clear[f]. Sí además queremos borrar sus atributos lo haremos con ClearAll[f].
• Todas las funciones del núcleo del *Mathematica* tienen el atributo **Protected,** para que sus definiciones no se puedan cambiar
• Cuando en *Mathematica* pedimos información de una función con ?función obtenemos información sobre la sintaxis y el uso de la función, si preguntamos un poco más con ??función obtenemos además información sobre los atributos y opciones de la función

Veamos un ejemplo:

```
? Integrate
```

```
Integrate[f, x] gives the indefinite integral of f with respect to x.
  Integrate[f, {x, xmin, xmax}] gives the definite integral of f with
  respect to x from xmin to xmax. Integrate[f, {x, xmin, xmax}, {y, ymin,
  ymax}] gives a multiple definite integral of f with respect to x and y.
```

```
?? Integrate
```

```
Integrate[f, x] gives the indefinite integral of f with respect to x.
  Integrate[f, {x, xmin, xmax}] gives the definite integral of f with
  respect to x from xmin to xmax. Integrate[f, {x, xmin, xmax}, {y, ymin,
  ymax}] gives a multiple definite integral of f with respect to x and y.
```

```
Attributes[Integrate] = {Protected, ReadProtected}
```

```
Options[Integrate] =
 {Assumptions -> {}, GenerateConditions -> Automatic, PrincipalValue -> False}
```

?? Plot

Info3353072165-6495363

```
Plot[f, {x, xmin, xmax}] generates a plot of f as a function of x from xmin to xmax.
   Plot[{f1, f2, ... }, {x, xmin, xmax}] plots several functions fi. More...
```

Info3353072165-6495363

```
Attributes[Plot] = {HoldAll, Protected}
```

```
Options[Plot] = {AspectRatio → 1/GoldenRatio, Axes → Automatic, AxesLabel → None,
  AxesOrigin → Automatic, AxesStyle → Automatic, Background → Automatic,
  ColorOutput → Automatic, Compiled → True, DefaultColor → Automatic,
  DefaultFont :→ $DefaultFont, DisplayFunction :→ $DisplayFunction, Epilog → {},
  FormatType :→ $FormatType, Frame → False, FrameLabel → None, FrameStyle → Automatic,
  FrameTicks → Automatic, GridLines → None, ImageSize → Automatic,
  MaxBend → 10., PlotDivision → 30., PlotLabel → None, PlotPoints → 25,
  PlotRange → Automatic, PlotRegion → Automatic, PlotStyle → Automatic,
  Prolog → {}, RotateLabel → True, TextStyle :→ $TextStyle, Ticks → Automatic}
```

?? Plus

Info3446822943-5563330

$x+y+z$ represents a sum of terms. ≫

Info3446822943-5563330

```
Attributes[Plus] = {Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
```

```
Default[Plus] := 0
```

?? Times

Info3446822998-7118680

$x*y*z$, $x\times y\times z$ or $x\,y\,z$ represents a product of terms. ≫

Info3446822998-7118680

```
Attributes[Times] = {Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
```

```
Default[Times] := 1
```

?? D

Info3446823030-4378398

D[f, x] gives the partial derivative $\partial f/\partial x$.
D[f, {x, n}] gives the multiple derivative $\partial^n f/\partial x^n$.
D[f, x, y, ...] differentiates f successively with respect to x, y, ...
D[f, {{x_1, x_2, ...}}] for a scalar f gives the vector derivative $(\partial f/\partial x_1, \partial f/\partial x_2, ...)$.
D[f, {*array*}] gives a tensor derivative. ≫

Info3446823030-4378398

```
Attributes[D] = {Protected, ReadProtected}
```

```
Options[D] := {NonConstants → {}}
```

■ 4.3.2. FUNCIONES RECURSIVAS

Muchas veces es útil programar con funciones recursivas (funciones del tipo el factorial de un número entero). En estos casos el programa resulta más rápido, si hacemos que la función recuerde los valores que ya ha calculado. Esto lo conseguimos con f[x_]:=f[x]=cuerpo

• Ejemplo: Calculamos los términos de la sucesión de Fibonacci.

```
ClearAll[f];
f[0] = 1;
f[1] = 1;
f[n_] := f[n] = f[n - 1] + f[n - 2]
```

```
f[2]
```

```
2
```

```
? f
```

Info3446823205-9270646

```
Global`f
```

Info3446823205-9270646

```
f[0] = 1

f[1] = 1

f[2] = 2

f[n_] := f[n] = f[n - 1] + f[n - 2]
```

```
f[5]
```

```
8
```

```
? f
```

Info3446823236-4148172

```
Global`f
```

Info3446823236-4148172

```
f[0] = 1

f[1] = 1

f[2] = 2

f[3] = 3

f[4] = 5

f[5] = 8

f[n_] := f[n] = f[n - 1] + f[n - 2]
```

Hay sin embargo incovenientes en hacer recordar los valores ya calculados. Es más rápido encontrar un valor particular, pero esto ocupa más espacio de memoria para almacenarlos todos. En general deberiamos definir funciones que recuerden valores ya calculados sólo cuando el número total de valores que puede tomar la función es relativamente pequeño o el "gasto" de recalcularlos es grande.

4.3.3. MODIFICANDO LAS FUNCIONES DEL NÚCLEO

Las funciones del núcleo tienen los atributos Protect, Locked, para que los valores y los atributos de f no se puedan cambiar. Pero a veces es interesante modificar alguna función del núcleo para que se reconozcan nuevas definiciones o valores en argumentos que nos interesen.

■ Ejemplo: Queremos dar definiciones para el logaritmo del producto, del cociente ,...

```
Unprotect[Log];
Log[x_ * y_] := Log[x] + Log[y];
Log[x_ / y_] := Log[x] - Log[y];
Log[x_^y_] := y * Log[x];
Protect[Log]
```

```
{Log}
```

```
?? Log
```

Info3446823603-3506127

Log[*z*] gives the natural logarithm of *z* (logarithm to base *e*).
Log[*b*, *z*] gives the logarithm to base *b*. ≫

Info3446823603-3506127

```
Attributes[Log] = {Listable, NumericFunction, Protected}
```

$$\text{Log}\left[\frac{x\_}{y\_}\right] := \text{Log}[x] - \text{Log}[y]$$

```
Log[x_ y_] := Log[x] + Log[y]
```

$$\text{Log}[x\_^{y\_}] := y\,\text{Log}[x]$$

```
Log[x^3 / (z^4 * y^2)]
```

```
3 Log[x] - 2 Log[y] - 4 Log[z]
```

■ 4.3.4. EL ORDEN EN LAS DEFINICIONES

En ocasiones una función f no viene definida de una sóla vez, sino que damos varias definiciones o reglas que se aplican sobre diferentes tipos de expresiones. Estas definiciones que hacen referencia a f se almacenan todas juntas, y cuando a lo largo de la sesión evaluamos una expresión que contenga a f se aplican sucesivamente. Para aplicar las definiciones *Mathematica* sigue el principio de aplicar primero las más especificas y después las más generales. Esto permite que las definiciones particulares no sean sombreadas por las más generales.

• Veamos un ejemplo: Vamos a definir la función factorial

```
f[1] = 1;
f[n_] := n * f[n - 1]
```

En este caso es claro cuál es la regla general y cuál la particular. Si queremos calcular f[3], intenta primero aplicar la regla f[1]=1 y si esto no es posible aplica la regla f[n_]:=n*f[n-1]

• Además Mathematica almacena junto con f, la información en que f aparece como cabecera, es decir, si preguntamos por f con ?f

```
? f
```

```
Global`f
```

```
f[0] = 1
```

```
f[1] = 1
```

```
f[2] = 2
```

```
f[3] = 3
```

```
f[n_] := n * f[n - 1]
```

f deja de ser solamente un objeto global, para ser ese objeto junto con las operaciones que con él se pueden realizar.

• En otros casos en que no está claro que regla es más general que otra *Mathematica* las aplica en el orden en el que han sido dadas y aplica antes las reglas que nosotros le damos que las reglas del núcleo.

■ 4.3.4.OPCIONES. PARAMETROS OPCIONALES

El uso de parametros opcionales permite obtener definiciones más compactas. Ya hemos visto como usarlos, existe sin embargo una construcción estandar para la construcción de opciones en una función,especialmente útil cuando el número de opciones es grande. De esta forma es como están definidas las opciones para la función Plot y la mayoria de las funciones del núcleo que tienen opciones.

La construcción es la siguiente, que vamos a ver con un ejemplo :

■ Con

```
Options[f] = {option1 -> value1, option2 -> value2}
```

```
{option1 → value1, option2 → value2}
```

definimos las opciones para la función f. Es una función con dos opciones option1 y option2 que tienen como valores por defecto value1y value2 respectivamente.

■ Definimos entonces f como

```
f[x_, opt___] := k[x, option1 /. {opt} /. Options[f],
    option2 /. {opt} /. Options[f]]
```

```
f[x, option1 → 2]
```

```
k[x, 2, value2]
```

```
f[x, option1 → 2, option2 → 3]
```

```
k[x, 2, 3]
```

```
f[x]
```

```
k[x, value1, value2]
```

• Notar que opt va seguido de "___", esto nos permite especificar en este lugar cero o más argumentos. Si en opt especificamos option2->valor, el valor que usamos es valor,ya que la segunda regla al haber usado la primera no se aplica . Por ejemplo f[x,option2->4] da como resultado k[x,,value1,4]), en cambio si no especificamos nada la regla que se aplica es la segunda y option2 tiene como valor el valor por defecto value2

Este tipo de construcción es general.

■ 4.3.5. SUBVALORES Y SUPERVALORES DE FUNCIONES

Cuando hacemos una definición en la forma f[arg]:=rhs esta definición automáticamente se asocia con f. Definiciones de este tipo en que f aparece como cabecera se denominan **subvalores (downvalues)** de f.

Mathematica sin embargo soporta también **supervalores (upvalues)**, que permiten asociar definiciones con simbolos que no aparecen directamente como su cabecera.

■ Considerar por ejemplo una definición como Exp[g[x_]]:=cuerpo. Si no especificamos nada esta definición se asocia por defecto con Exp. Sin embargo sería deseable asociar esta definición con g, en vez de con Exp. Esta defínción sería un supervalor para g.

■ La manera de asociar supervalores es la siguiente:

f[g[arg],....]^:=cuerpo

De esta forma conseguimos asociar la definición a g, y en general a todas las cabeceras de los argumentos de f, en vez de a f.

■ 4.3.6. COMPILANDO FUNCIONES

Mathematica es un lenguaje de programación interpretado . Esto significa que cada vez que evaluamos una expresión , esta es traducida a código compilado.

Pero hay una manera de conseguir que las funciones que definamos sean más rápidas: compilarla. Entonces f se asocia con su código compilado y no es necesario hacer la traducción cada vez que llamamos a la función.

La instrucción para compilar expresiones es:

Compile[{x1,....,xn},exp]

Si utilizamos Compile para definir funciones, se supone que los datos sobre los que se aplica f son "float" o enteros o varibles logicas.

■ Ejemplo:

```
f = Function[{x}, x * Sin[x]]
```

```
Function[{x}, x Sin[x]]
```

```
fc = Compile[{x}, x * Sin[x]]
```

```
CompiledFunction[{x}, x Sin[x], -CompiledCode-]
```

```
Timing[f[2.5]]
```

```
{0. Second, 1.49618}
```

```
Timing[fc[2.5]]
```

```
{0. Second, 1.49618}
```

■ 4.3.7. EJERCICIOS

■ 1.- a) Obtener los atributos de : Plus, Times, Log, Plot
b) Obtener las opciones de Plot
c) Una sucesión está definida de la siguiente forma

```
a[0] = Sqrt[2];
a[n_] := Sqrt[1 + a[n - 1]]
```

Haz las modificaciones necesarias en la definición para que se "recuerden" los valores ya calculados. Dibujar los veinte primeros términos de la sucesión a[n].

■ 2.- Construir el operador derivada usando modelos y funciones. Observar el ejemplo adjunto de construcción de la integral.

```
Integral[y_ + z_, x_] := Integral[y, x] + Integral[z, x];
Integral[c_ * y_, x_] := c Integral[y, x] /; FreeQ[c, x];
Integral[c_, x_] := cx /; FreeQ[c, x];
Integral[x_^(n_.), x_] := x^(n + 1) / (n + 1) /; FreeQ[n, x] && n != 1;
Integral[1 / (a_. * x_ + b_.), x_] := Log[a x + b] / a /; FreeQ[{a, b}, x];
Integral[Exp[a_. * x_ + b_.], x_] := Exp[a x + b] / a /; FreeQ[{a, b}, x]
```

■3.- Definir una función que se llame exp y tenga las propiedades habituales de la función exponencial

```
Loga[0] = -∞;
Loga[1] = 0;
Loga[E^x_] := x /; Im[x] == 0;
Loga[a_ b_] := Loga[a] + Loga[b] /; Im[a] == 0 && Im[b] == 0;
Loga[a_^b_] := b Loga[a] /; Im[a] == 0 && Im[b] == 0;
Loga[z_] := Loga[Abs[z]] + I Arg[z]
```

```
General::spell1 : Possible spelling error: new
   symbol name "Loga" is similar to existing symbol "Log". More…
```

```
Loga[-I]
```

$$-\frac{i\,\pi}{2}$$

```
Loga[E^-I]
```

$$-i$$

```
Loga[E]
```

```
1
```

■4.-El algoritmo de ordenación rápida es un algoritmo de ordenación de listas en el que cada elemento es comparado con uno fijo de la lista, por ejemplo el primero. Si un elemento es menor que el primero se guarda en una lista que llamaremos lista1, y si no se guarda en otra que llamaremos lista2. Con cada una de esas listas se repite el proceso; asi sucesivamentehasta que en las particiones sucesivas sólo quede un elemento.

Programar el algoritmo anterior en una función recursiva que se llame Ordena , y que tenga como argumentos una lista de números

■ 5.-A continuación hemos definido la función ProdExt que da el producto hemisimétrico de tensores hemisimétricos. Explica cada instrucción razonadamente.

```
ProdExt[w[i__], w[j__]] :=
Signature[Join[List[i], List[j]]] *
Apply[w, Sort[Join[List[i], List[j]]]];
SetAttributes[ProdExt, Flat];
ProdExt[t1_ + t2_, t3_] := ProdExt[t1, t3] + ProdExt[t2, t3];
ProdExt[a_. x_w, b_. y_w] := Times[a, b, ProdExt[x, y]]
```

CAPÍTULO V: APLICACIONES AL CÁLCULO. ECUACIONES DIFERENCIALES Y SISTEMAS DE ECUACIONES DIFERENCIALES.

Se llama ecuación diferencial a una ecuación del tipo

$$F(x_1,\ldots,x_n,y_1,\ldots,y_m,\frac{\partial y_1}{\partial \mathrm{x1}},\ \ldots.\frac{\partial y_1}{\partial \mathrm{xn}},\ \ldots,\frac{\partial^{|\alpha|}}{\partial x_1^{\alpha_1}\ldots\partial x_n^{\alpha_n}})=0$$

en la que aparecen ligadas n variables independientes $x_1,\ldots,x_n$, m variables dependientes $y_1,\ldots,y_m$, y las derivadas parciales de las variables dependientes respecto de las n variables independientes. Resolver la ecuación es encontrar funciones $y_1=f_1(x_1,\ldots,x_n),\ldots,y_m=f_m(x_1,\ldots,x_n)$ tales que al ser sustituidas en la ecuación esta se verifique idénticamente.

Si sólo aparece una variable independiente diremos que la ecuación es una ecuación diferencial ordinaria (EDO u ODE en inglés). En caso contrario, diremos que la ecuación es una ecuación diferencial en derivadas parciales (EDP o PDE en inglés).

Llamaremos orden de la ecuación al orden de la mayor derivada que aparezca.

Mathematica puede resolver ecuaciones diferenciales y sistemas de ecuaciones diferenciales y dependiendo de su dificultad, se pueden obtener soluciones simbólicas o bien soluciones aproximadas con precisión arbitraria.

A continuación veremos algunos rudimentos sobre la resolución de ecuaciones diferenciales con Mathematica. Puedes encontrar más información en la ayuda y en particular en los tutoriales de la ayuda tutorial/DSolveOverview, tutorial/NDSolveOverview y tutorial/NumericalSolutionOfDifferentialEquations.