Tema 7

Programación

La utilización de modelos matemáticos implica, en mayor o menor medida, escribir un programa que implemente el modelo en cuestión. Se trata de una tarea cuya dificultad dependerá de la complejidad del modelo pero que, lógicamente, requiere una fase inicial de aprendizaje.

Los ejemplos que van a verse en este curso se han desarrollado en lenguaje S que se ejecutarán con el programa R. Se trata de un entorno integrado para trabajar con el lenguaje S.

7.1 Introducción a la computación

Un ordenador (*computador*) puede definirse como una máquina programable para el tratamiento de información, es decir, una máquina para realizar **cómputos** (determinación indirecta de una cantidad mediante el cálculo de ciertos datos. Un programa es la descripción detallada de los **algoritmo** necesarios para llevar acabo un computo en un lenguaje, intermedio entre el lenguaje humano y el lenguaje de la máquina, denominado **lenguaje de programación**.

Un **algoritmo** es un método general de resolución de todos los problemas del mismo tipo en el que se detalla paso a paso lo que de be hacerse. Cada uno de estos pasos constituye una **sentencia** o **instrucción**. Una **instrucción** es una combinación de palabras, variables, constantes y símbolos que, obedeciendo a la sintaxis propia del lenguaje de programación utilizado, indican al ordenador la acción que debe ejecutar.

Un **programa** es la expresión de un algoritmo en un lenguaje de programación entendible por el ordenador.

A la hora de realizar algún tipo de cómputo en un ordenador, lo normal es utilizar un programa ya desarrollado (una **aplicación**) de propósito general. El problema surge cuando estas aplicaciones no hacen exactamente lo que queremos que hagan. En ese caso la solución puede ser programar el ordenador para que realice la tarea. De hecho la mayor parte de las aplicaciones informáticas incluyen algún tipo de lenguaje de programación para aumentar su flexibilidad.

7.2 Lenguajes de programación

Desde los años cincuenta se han desarrollado múltiples lenguajes de programación. Los lenguajes de primera generación eran los ensambladores, lenguajes de bajo nivel que simplemente renombran instrucciones de código máquina en un formato más legible por humanos.

Una segunda generación esta constituida por lenguajes imperativos en los que la unidad de trabajo del programa es la instrucción (Fortran, Cobol o Basic). Cada instrucción representa varias instrucciones en lenguaje

máquina. La evolución de estos da lugar a lenguajes imperativos de tercera generación (Algol, Pascal, Modula-2, C).

En una cuarta generación aparecen lenguajes aún de más alto nivel orientados a aplicaciones concretas, como SQL para el acceso a bases de datos. Las instrucciones que manejan estos lenguajes manipulan objetos no se basan en instrucciones que mane

En la quinta generación aparecen lenguajes orientados al procesamiento de lenguaje natural (Perl o AWK) y lenguajes orientados a objetos (C++ o Java). En estos últimos lo importante no es la instrucción sino el objeto definido como un tipo de dato más las funciones que puede ejecutar.

Derivados de los lenguajes orientados a objetos, aparecen los lenguajes visuales, capaces de generar interfaces gráficas (Visual-Basic, Visual-C, Tcl-Tk)

Otra diferencia importante es entre lenguajes compilados e interpretados. En los primeros se sigue toda la secuencia de fases presentada anteriormente (C, Pascal, Fortran, Cobol). En los segundos no hay compilación ni enlace, el programa es leído por un **intérprete** que lee cada orden, la traduce a lenguaje máquina y la interpreta. Lógicamente, debido a este proceso los programas son más lentos y requieren más recursos del ordenador. La ventaja es que todo el proceso de compilación se simplifica (Basic, Awk, Java, Tcl-Tk).

Aunque existen varios lenguajes en realidad todos responden más o menos a las mismas ideas y elementos básicos. A partir de aquí se utilizará para desarrollar los ejemplos y hacer las prácticas el lenguaje de programación **S** que puede utilizarse para programar aplicaciones dentro del programa **R**. La ventaja fundamental de utilizar S como lenguaje de programación es que es un lenguaje de altísimo nivel que incluye funciones para la entrada, salida y representación de datos que exigirían muchas lineas de código en otros lenguajes de programación. Por otro lado al ser un lenguaje orientado al análisis de datos, resulta muy adecuado para trabajar en modelización.

La desventaja de R es que, precisamente por ser un lenguaje de muy alto nivel, se vuelve muy lento cuando el programa se complica o cuando el volumen de datos implicado es muy alto. No es un buen lenguaje para hacer grandes programas, pero resulta muy adecuado para pequeñas aplicaciones.

7.3 Fases en el desarrollo de un programa

Existentes una serie de etapas desde el reconocimiento de la existencia de un problema, susceptible de ser resuelto por un programa hecho por nosotros, y la solución de aquel mediante la ejecución del programa:

- **Análisis del problema**
- **Desarrollo del algoritmo** mediante esquemas o pseudocódigo.
- **Codificación** en un lenguaje de programación concreto.
- **Edición**, transcripción del código al ordenador (código fuente).
- **Prueba de ejecución** y corrección de los errores de ejecución (el programa no se ejecuta correctamente) o de lógica (el programa produce resultados erróneos).
- **Utilización**

Este esquema es válido para la programación con lenguajes interpretados (que suele ser lo habitual en modelización a pequeña escala) como puede ser BASIC, awk, python, Perl o el lenguaje S. En caso de que se trabaje con un lenguaje compilado (C, C++, Java, etc.) habría que introducir las fases de compilación y enlazado antes de la ejecución.

7.4 Estructura de un programa

Un programa consiste en una secuencia de instrucciones que pueden agruparse en:

- **Entrada de datos**. Conjunto de instrucciones que toman datos de una unidad de entrada, los depositan en la memoria RAM. En esta etapa se definen también los **tipos de datos** con los que va a trabajar el programa.
- **Proceso**. Conjunto de instrucciones que resuelven el problema a partir de los datos de entrada y almacenan el resultado en memoria RAM. Describen el algoritmo del programa.
- **Salida**. Conjunto de instrucciones que vuelcan los resultados, a un fichero, pantalla o impresora, con un formato definido por el usuario.

En un programa *profesional* resulta a menudo difícil distinguir unas fases de otras, sin embargo es una buena práctica separar claramente estas etapas cuando se empieza a programar.

7.5 Elementos de programación

7.5.1 Variables y operadores

Todos los programas manipulan datos que se guardan en una o varias posiciones de memoria. Como, evidentemente, sería muy complejo hacer referencia directa a las posiciones de memoria, se les asignan variables cuyo nombre es más sencillo e intuitivo.

Por ejemplo si queremos que una variable contenga el día del mes, podemos escribir en el programa: $dia = 3$. El ordenador interpretará que queremos reservar una posición de memoria que contenga el número 3 y a la que nos referiremos durante el desarrollo del programa como a.

El número de posiciones de memoria que ocupe una variable dependerá de su tipo. Algunos lenguajes exigen que se definan todas las variables (junto con su tipo de dato) antes de utilizarlas.

Las variables pueden manipularse mediante operadores aritméticos o lógicos $(+, -, *, /, \&\&, ||, =, != , >, <, >=, <=)$ que combinan dos variables para producir un resultado que se almacena en otra variable $a = b + c$. También es posible modificar variables $a = a + 5$.

Otra posibilidad es almacenar cadenas de caracteres en una variable. Para indicarle al ordenador que toda la cadena es un sólo objeto, esta debe entrecomillarse $a ='' Hola''$

7.5.2 Entrada y salida

Puesto que el objetivo fundamental de un programa de ordenador es manipular unos datos de entrada para producir unos datos de salida. Es necesario disponer de algún mecanismo para introducir información desde fichero o teclado y sacarla a la pantalla o a otro fichero.

Ya se ha visto como **awk** está especialmente concebido para leer ficheros sin necesidad de funciones especiales. Sin embargo en algunos casos se hace necesario combinar varios ficheros de entrada en un sólo programa. Hay dos opciones:

- Leerlos uno detrás del otro. Para ello es necesario darle al programa algún método para distinguir un fichero del siguiente.
- Leerlos con la función $getline$, por ejemplo $linea = getline <'' fichero.txt''$

%d	Número entero
%nd	Número entero formateado a n caracteres
%f	Número real
%$m.n$f	Número real con n decimales formateado a m caracteres
%s	Cadena de caracteres

La función básica de salida en awk es $print$, más sofisticada es $printf$ donde la f final significa *formateada*, es decir permite determinar exactamente que forma tendrá la linea de salid a. Por ejemplo ...

```
awk '{printf("Año=%d Enero=%.1f Julio=%.1f\n"),  $1,$2,$8}' precmurcia.txt
```

printf es una función y por tanto sus argumentos se sitúan entre paréntesis. El primer argumento es el formato entrecomillado, posteriormente se incluyen como argumentos las variables que se van a escribir. En el formato se puede incluir cualquier conjunto de caracteres más combinaciones especiales de caracteres que se inician con el carácter % y que indican como se van a imprimir el resto de los argumentos de la función:

```
\n
```

significa salto de linea.

7.5.3 Estructuras de control

En la gran mayoría de los programas, el proceso de los datos requiere ejecutar alguna tarea repetidas veces o decidir realizar una tarea u otra en función de los valores de alguna variable. Las instrucciones para llevar a cabo estas acciones constituyen las estructuras de control.

Todas las estructuras de control comienzan con una palabra clave como **if** o **while**

Bucles

Existen dos tipos fundamentales de bucles, los bucles *FOR* y *WHILE*, aunque en realidad son intercambiables. La orden

```
for (i=1;i<=10;i=i+1){ printf(‘‘Número %d\n’’,i) }
```

comienza dando el valor 1 a la variable contador i, y ejecutará la orden entre llaves mientras se cumpla que $i <= 10$, en cada ejecución incrementará en 1 el valor de i.

Las ordenes

```
i=0
while (i<=10){i=i+1;printf(‘‘Número %d\n’’,i)}
```

hacen los mismo, sólo que ahora la variable contador se inicializa y se incrementa fuera de la orden.

Las instrucciones **break**, **continue** y **exit** alteran también el orden de ejecución. La primera fuerza al flujo del programa a salir de un bucle for o while y constinuar con la siguiente orden. La segunda vuelve a la orden inicial del bucle (e incrementa el contador en el caso del bucle FOR. La última sale totalmente del programa.

En este tipo de bucles se suele distinguir entre la **variable contador**, destinada a contener diferentes valores que se van incrementando o decrementando cada vez que el ordenador ejecuta la orden FOR y las **variables acumuladoras** que van a permitir almacenar valores que aumentan o disminuyen de forma no constante durante el proceso. El siguiente ejemplo permite calcular el factorial de un número (variable **n**) que se guardará en la variable acumuladora **fac**:

```
fac=1
for (i in 1:n){ fac = fac * i }

\subsubsection{Secuencias de números con S}

El lenguaje S incluye diversas funciones para generar secuencias de números co

\begin{verbatim}
sec=seq(1,10)
for i in seq{i=i+1;printf(‘‘Número %d\n’’,i)}
```

escribirá los números del 1 al 10

```
sec=seq(1,10,by=2)
for i in seq{i=i+1;printf(‘‘Número %d\n’’,i)}
```

escribirá sólo los impares.

Incluso podemos utilizar un bucle FOR con una secuencia arbitraria de números:

```
sec=c(23,45,32,23,9)
for i in seq{i=i+1;printf(‘‘Número %d\n’’,i)}
```

Toma de decisiones

En ocasiones se debe romper el flujo de un programa y ejecutar un grupo de instrucciones u otras en función de los valores que adopta una variable.

El esquema básico de las instrucciones **if ... else** es:

```
if (condición) {instrucciones1} else {instrucciones2}}
\end{verbatin}

Donde instrucciones1 es el conjunto de ordenes que se ejecutan si condición se

\begin{verbatim}
if (x==3) y=27 else y=9

if (a>20){ printf(‘‘%d es mayor que 20\n’’,a)}
else{printf(‘‘%d no es mayor que 20\n’’,a)}
```

Si x es igual a 3 y será igual a 27, sino y será 9.

Si en lugar de estar formadas por una sola instrucción, **instrucciones1** e **instrucciones2** están compuestas por varias, es necesario construir un bloque con ellas poniendo corchetes a su alrededor, por ejemplo:

```
if (x==3) {y=27;z=3} else {y=9;z=4}
```

7.5.4 Funciones definidas por el usuario

Uno de los elementos básicos de cualquier lenguaje de programación son las funciones. Su objetivo es encapsular una serie de instrucciones que tienen consistencia por si mismas como *subprograma* y que son llamadas de forma habitual y siempre de la misma forma por el programa. Las funciones generan uno o más datos de salida a partir de unos datos de entrada.

Existen múltiples funciones ya disponibles en cualquier lenguaje de programación, en concreto el lenguaje S se caracteriza por un gran número de funciones de análisis de datos que ya has visto, sin embargo puede ocurrir que necesitemos una función no disponible, en ese caso podemos definirla y utilizarla en cualquier momento dentro de nuestro programa.

Cuando un programa se va haciendo más sofisticado, necesita llevar a cabo determinadas acciones varias veces. Pero no resulta una técnica eficiente de programación el repetir varias veces el mismo conjunto de órdenes.

En su lugar es preferible incluir estas órdenes como una *función definida por el usuario* con ello se consigue:

- Dividir el código en un número pequeño y manejable de partes
- Se verifica su correcto funcionamiento una sola vez y puede aplicarse tantas como se necesite
- Código reutilizable por otros programas

La forma de definir una función es:

```
nombre=function(lista\_de\_argumentos){
  instrucciones
}
```

nombre es el nombre de la función, con el cual será llamada en el programa. La *lista_de_argumentos* incluye todas las variables que necesita la función para ejecutarse y que son externas a ella. Las instrucciones procesan la lista de argumentos y generan un resultado que la función devuelve. En el siguiente ejemplo aparece la definición de una función que calcula el máximo de dos valores y una llamada a la misma

```
maximo=function(a,b){
   if (a>b){a}
   else{b}
}
c=maximo(3,4)
```

En la mayoría de los lenguajes de programación, el valor que devuelve una función se explicita con una orden concreta, generalmente **return**. En S el valor puede devolverse mediante una instrucción **return** pero si no se hace así, la función devuelve la última variable calculada o simplemente ejecutada en el seno de la función. En el anterior ejemplo, si a>b la última variable a que se hará referencia en la función es a y en caso contrario es b.

A continuación vemos otro ejemplo en el que una función, encargada de resolver ecuaciones de segundo grado, devuelve un vector con las dos soluciones:

```
e2g=function(a,b,c){
   if (a==0){-c/b}
   else{
        r=b^2-4*a*c;
        if (r<0){cat("Las soluciones son complejas\n");c(NA,NA)}
```

```
        else{
           s1=(-b+sqrt(r))/(2*a)
           s2=(-b-sqrt(r))/(2*a)
           c(s1,s2)
        }
   }
}
```

Esta función utiliza además dos instrucciones **if ... else** para determinar si el primer parámetro es cero, en cuyo caso se resuelve como una ecuación de primer grado y así se evita que la división por cero posterior de un error, y si las soluciones son complejas. En este último caso se da un mensaje de aviso y se devuelven dos NA.

Para llamarla bastará con ejecutar por ejemplo:

```
e2g(2,5,3)
```

El resultado será:

[1] -1.0 -1.5

Con el tiempo, se dispondrá de un elevado número de funciones que pueden agruparse en una librería para ser reutilizadas en diferentes programas. Una librería es un fichero que contiene las definiciones de las diferentes funciones y que se llama al comienzo de un programa con la orden:

```
source(‘‘libreria.R’’)
```

En R existen también librerías de funciones compiladas que se cargan con la función **library**