Capítulo 9

Recursividad

Índice General

> **Ley de Hofstadter Modificada:** *El tiempo necesario para estudiar este capítulo es más del que se espera, aún cuando se tenga en cuenta la ley de Hofstadter Modificada*

9.1. ¿Qué es la Recursividad?.

El concepto de Recursividad es muy amplio (por ejemplo, se encuentra reflejado cuando se leen relatos que describen relatos ("*La Historia Interminable*", "*Las Mil y Una Noches*"), cuando se ven películas que hacen referencia a otras películas ("*La Rosa Púrpura del Cairo*", "*La Mujer del Teniente Francés*", "*La Noche Americana*"), o jugando con las muñecas rusas en cuyo interior hay otras muñecas (o, cuando se realizan comentarios entre paréntesis (¡dentro de otros comentarios entre paréntesis!))). Es muy habitual en la literatura y en los juegos matemáticos que la recursividad se aproxime mucho a una paradoja. Un ejemplo de esta situación ocurre en la extendida, incorrecta y, un tanto divertida, siguiente definición de recursividad[1]:

Definición (Recursividad) ver *Recursividad*.

Esta definición conduce a una regresión infinita. Nunca se sabrá lo que es la recursividad ateniéndose a ella. Para que una definición recursiva sea correcta, jamás debe conducir a una regresión

[1] Douglas Hofstadter, "Escher, Gödel, Bach: un Eterno y Grácil Bucle".

infinita ni a una paradoja. Esto es así porque una definición recursiva nunca define un concepto en función de si mismo, sino siempre en función de interpretaciones más simples de dicho concepto.

Una de las formas más comunes bajo la que aparece la recursividad en la vida cotidiana es cuando se posterga la finalización de una tarea, con el objeto de ocuparse de otra más sencilla del mismo género. Por ejemplo, durante la búsqueda de una palabra en el diccionario: en su definición puede aparecer otra palabra cuyo significado se desconoce y que, por lo tanto, obliga a buscar esa nueva palabra en el diccionario. Se podría seguir con este proceso hasta que se diese con una definición en la que se sepa el significado de cada una de las palabras que la componen. Con esta información se "vuelve" a la definición de la palabra anterior, a la palabra anterior a la anterior, y así, hasta que se llega a la primera palabra, la que dio origen a todo este proceso, y que ya se puede interpretar correctamente.

Desde un punto de vista más formal, se pueden encontrar muchos ejemplos de definiciones recursivas en un gran número de problemas matemáticos.

Ejemplo:
Sea la definición recursiva de la suma de dos números naturales a *y* b

$$suma(a,b) = \begin{cases} a & si\ b = 0, \\ sig(suma(a, b-1)) & en\ otro\ caso \end{cases} .$$

donde sig(b) *es el número natural siguiente a* b*. Según esto, para calcular* suma(3, 2) *se realizarían los siguientes pasos:*

suma(3,2) = sig(suma(3,1) = sig(sig(suma(3,0))) = sig(sig(3)) = sig(4) = 5.

9.1.1. El Concepto de Recursividad en Algorítmica.

Un algoritmo se dice que es **recursivo** cuando *tiene la capacidad de llamarse a si mismo o de llamar a otro algoritmo desde el cual se puede volver a invocar al primero*. En cualquier caso, se garantiza que el número de llamadas recursivas que se realizan es un número finito (es decir, en algún momento ha de finalizar la ejecución del algoritmo). Para ello, cada llamada recursiva resuelve un problema de menor tamaño, uno tras otro, hasta llegar a un problema cuya resolución sea directa. Se puede decir, por lo tanto, que un algoritmo recursivo consta de una *condición de parada*, que cuando es cierta permite resolver el problema directamente, y de la *llamada recursiva*, que resuelve el problema cuando la solución no es directa.

Para ilustrar el concepto de recursividad en un algoritmo, se aceptará, por ahora, la idea intuitiva de que realizar una llamada recursiva sería equivalente a volver a escribir de nuevo el algoritmo recursivo invocado. Por ejemplo, sea el siguiente algoritmo recursivo que resuelve el problema de la suma de dos números naturales, visto en el ejemplo anterior:

```
void Suma (int a, int b, int *s){
    if (b==0)  /* condición de parada */
       *s=a;
    else {
       Suma(a,b-1,s);
       *s=*s+1;
   }
}
```

En este algoritmo recursivo, si se quiere sumar `a` y 0, el resultado es inmediato, `a`, pero si se quiere sumar `a` y `b`, `b` > 0, entonces el problema es equivalente a calcular la suma de `a` y de `b-1` y al resultado añadirle 1.

¿Qué ocurre al realizar una llamada a este algoritmo, por ejemplo `Suma(3, 2, &s)`?.

```
int main(){

    int s;
    ...

    ...
    /* entre otras operaciones, se hace la llamada siguiente */
    Suma(3, 2, &s);
    ...
}
```

El efecto es la ejecución del código de `Suma`, sabiendo que `a` recibirá el valor `3`, `b` el valor `2` y sobre `s`, que se pasa por referencia, se espera recibir el valor correspondiente a la suma de ambos:

```
void Suma (int a, int b, int *s){
    if (b==0)      /* condición de parada FALSA*/
       *s=a;
    else {         /* b==2, b!=0 */
       Suma(a,b-1,s);     /* Se ejecuta Suma(3,1,s) */
       *s=*s+1;   /* al resultado se le sumará 1 */
   }
}
```

Ejecutar esta nueva llamada recursiva sería equivalente a reescribir, en la línea en la que se realiza la llamada, todo el algoritmo de nuevo (¡ojo!, con los valores de los parámetros actualizados),

```
void Suma (int a, int b, int *s){
    if (b==0)      /* condición de parada FALSA*/
       *s=a;
    else {         /* b==2, b!=0 */
       if (b==0)       /* condición de parada FALSA*/
          *s=a;
       else {          /* b==1, b!=0 */
          Suma(a,b-1,s);      /* Se ejecuta Suma(3,0,s) */
          *s=*s+1;    /* al resultado se le sumará 1 */
       }
       *s=*s+1;    /* al resultado se le sumará 1 */
   }
}
```

Como el valor de b es 1, se vuelve a realizar otra llamada recursiva, Suma(3, 0, s),

```
void Suma (int a, int b, int *s){
    if (b==0)  /* condición de parada FALSA*/
       *s=a;
    else {     /* b==2, b!=0 */
       if (b==0)      /* condición de parada FALSA*/
          *s=a;
       else {         /* b==1, b!=0 */
          if (b==0)            /* condición de parada CIERTA*/
             *s=a;             /* s toma el valor 3 */
          else {               /* b==0*/
             Suma(a,b-1,s);    /* NO SE EJECUTA */
             *s=*s+1;          /* NO SE EJECUTA */
          }                    /* FIN Llamada Suma(3,0,s), s=3 */
          *s=*s+1;     /* al resultado se le sumará 1 */
       }               /* FIN Llamada Suma(3,1,s), s=4 */
       *s=*s+1; /* al resultado se le sumará 1 */
   }            /* FIN Llamada Suma(3,2,s), s=5 */
}
```

A la vista de la exposición, se puede observar que las llamadas recursivas se abren y se cierran como un telescopio.

Al seguir esta definición matemática de la suma, puede ser más simple expresarla mediante un algoritmo tipado (función). Así, el ejemplo anterior puede reescribirse como algoritmo tipado recursivo de la forma,

```
int sumaf (int a, int b){
    int s;

    if (b==0)
        s=a;
    else
        s=sumaf(a,b-1)+1;

    return s;
}
```

Al compararlo con la otra versión, se debe hacer notar que para resolver la expresión

```
s = sumaf(a,b-1) + 1;
```

primero se debe resolver la llamada recursiva (sumaf(a,b-1)) y, después, al valor devuelto se le añadirá 1. En ese sentido, el comportamiento de ambas versiones es equivalente.

9.1.2. Ejemplos de Algoritmos Recursivos.

Cualquier función matemática puede ser expresada de forma recursiva. Algunos ejemplos clásicos de definiciones recursivas son las del cálculo del factorial, el cálculo del máximo común divisor de dos números (según el método de Euclides) o la sucesión de Fibonacci.

Cálculo del factorial: La definición recursiva de la función que calcula el factorial de un número natural `n` es la siguiente:

$$factorial(n) = \begin{cases} 1 & si\ n = 0, \\ n * factorial(n-1) & en\ otro\ caso \end{cases} .$$

A partir de esta definición recursiva se deriva el siguiente algoritmo recursivo,

```
int fact(int n){
    int f;

    if (n==0)  /* condición de parada */
        f = 1;
    else
        f = n*fact(n-1); /* llamada recursiva */

    return f;
}
```

Este algoritmo se puede analizar leyendo las instrucciones en orden: si $n = 0$ el factorial de `n` es 1, en otro caso hay que calcular el factorial de `n-1`. Tras resolverse la llamada, para obtener el valor del factorial de `n` a partir del factorial de `n-1`, basta con multiplicarlo por el valor de `n`.

Algoritmo de Euclides: El siguiente ejemplo consiste en el cálculo del máximo común divisor de dos números enteros `a` y `b`, utilizando el método de Euclides[2]:

$$mcd(a,b) = \begin{cases} a & si\ a = b, \\ mcd(a, b-a) & si\ a < b, \\ mcd(a-b, b) & si\ a > b \end{cases} .$$

Para calcular el MCD de 22 y 16 según este método, el proceso sería:

> *"Como 22 > 16, entonces el problema de calcular mcd(22, 16) es equivalente a calcular mcd(22-16,16) = mcd(6,16). Al ser 6 < 16, este es equivalente a calcular mcd(6,10). De nuevo, 6 < 10, por lo tanto se calcula mcd (6, 4). El mcd(6, 4) es el mismo que el mcd(2, 4), que es igual al mcd(2, 2), que es 2 ya que a=b es la condición de parada."*

Este método se puede expresar en forma de algoritmo recursivo:

[2]Primer algoritmo conocido; se estima que fue formulado en el año 300 A.C.

```
int mcd (int a, int b){
    int m;

    if (a==b) /* condición de parada */
        m=a;
    else
        if (a<b)
           m=mcd(a,b-a); /* llamada recursiva */
        else
           m=mcd(a-b,b); /* llamada recursiva */

    return m;
}
```

Cálculo del término n de la sucesión de Fibonacci: El último ejemplo que se va a tratar es el caso del cálculo del término n-ésimo de la sucesión de Fibonacci. Esta sucesión está definida por medio de la siguiente regla recursiva:

$$fibonacci(n) = \begin{cases} 1 & si\ n = 1, \\ 1 & si\ n = 2, \\ fibonacci(n-1) + fibonacci(n-2) & si\ n > 2 \end{cases} .$$

La escritura de un algoritmo recursivo para este caso queda de la siguiente forma:

```
int fibon (int n){
    int f;

    if (n==1)  /* condición de parada */
        f=1;
    else
        if (n==2)  /* condición de parada */
            f=1;
        else
            f=fibon(n-1)+fibon(n-2); /* 2 llamadas recursivas */

    return f;
}
```

9.2. Tipos de Recursividad.

De acuerdo a la definición de recursividad vista en la subsección 10.1.2, se desprende que, en función del modo en que se realicen las llamadas recursivas, hay dos tipos de recursividad:

Recursividad Directa: Un algoritmo recursivo se llama a sí mismo.

Recursividad Indirecta: Un algoritmo recursivo llama a otro algoritmo desde el cual se vuelve a llamar al primero.

Todos lo ejemplos vistos, lo son de recursividad directa. Un ejemplo de recursividad indirecta sería el siguiente, que también resuelve el problema de la suma de dos números enteros a y b,

```
void Suma2 (int a, int b, int *s){
    if (b==0)
       *s = a;
    else
       Sig(a, b-1, s);
}
```

```
void Sig (int a, int b, int *s){
    Suma2(a, b, s);
    *s = *s + 1;
}
```

Además, es posible clasificar el tipo de recursividad atendiendo al resto de acciones que quedan por ejecutar después de realizar la llamada recursiva; así, se distingue:

Recursividad Final: Al finalizar la llamada recursiva no hay más acciones que ejecutar.

Recursividad No Final: Quedan acciones por ejecutar después de que finalice la llamada recursiva.

El algoritmo `fact` es un caso de recursividad no final. El siguiente ejemplo, muestra un algoritmo recursivo, con recursividad final:

Algoritmo recursivo que acumula en una variable s*, cuyo valor inicial es 0, la suma de todos los números enteros naturales comprendidos entre dos números dados* i *y* n*:*

```
void sumatorio(int n, int *s, int i){
     if (i<n){
         *s = *s + i;
         sumatorio(n, s, i+1);
     }
}
```

Se deja como problema determinar de todos los algoritmos expuestos en este tema cuáles son ejemplos de recursividad final/no final.

9.3. La Recursividad en el Computador.

Hasta el momento se ha visto como se expresa una tarea recursiva mediante un algoritmo, pero ¿cómo se realiza esa tarea recursiva cuando se programa en un computador?.

Para responder a esta pregunta se necesita conocer la definición y fundamentos de una estructura de datos denominada *pila*.

Una **pila** es un *estructura de datos en la que se pueden añadir o eliminar elementos por un único extremo* que recibe el nombre de *cima*. Las eliminaciones se realizan en orden inverso a las inserciones. En cada momento sólo se puede acceder al elemento situado en la cima de la pila. Un ejemplo del funcionamiento de una pila es el servicio de bandejas de una cafetería: cada cliente que precisa de una bandeja coge la primera del montón (la que está en la "cima") y cuando la deja de usar se vuelve a colocar encima del bloque actual (y se convierte en la nueva "cima").

Esta estructura de datos, pila, se emplea muy frecuentemente cuando se ejecutan programas en un computador. Cuando desde un programa -es decir, cuando desde el algoritmo ya expresado en un lenguaje de programación concreto- se hacen llamadas a otro programa, el programa desde el que se hace la llamada queda interrumpido para que pueda comenzar la ejecución del segundo; una vez que éste finaliza, se retoma la ejecución del primer programa. Para recordar el instante y el estado del entorno del programa original el computador utiliza una pila: almacena en ella esta información en el momento de la llamada y, una vez se ha resuelto, la recupera para poder continuar la ejecución.

El esquema es el mismo en el caso de programar un algoritmo recursivo. En este caso el programa que se llama para ejecución es el mismo, con lo que en cada llamada recursiva se crea un conjunto diferente de copias del estado del entorno, variables internas y parámetros, y se almacenan en la pila.

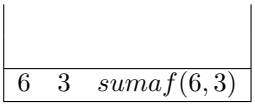
Las llamadas recursivas cesan cuando en alguna se cumple la condición de parada. En ese momento, se elimina de la cima de la pila la copia del entorno de la última llamada, con lo que pasa a la cima la información sobre la penúltima, y se devuelve el control al punto desde el que se realizó esta penúltima llamada recursiva. Esta secuencia se repetiría tantas veces como llamadas recursivas se hayan hecho, hasta haber resuelto la primera.

Para ilustrar este mecanismo de almacenamiento y recuperación de la información se tomará como ejemplo una llamada a la función recursiva `sumaf`, que se vio en la subsección 10.1.2. Para simplificar el ejemplo sólo se representarán los parámetros de cada llamada. Al ser una función, se almacenarán en la pila los valores de los parámetros de entrada y el nombre del algoritmo (que realiza el papel de parámetro de salida implícito).

Supóngase que en un programa se hace la siguiente asignación:

```
x = sumaf(6,3);
```

Esta primera llamada provoca que en la pila se almacenen los valores de $a = 6$ *y* $b = 3$*, y que se reserve espacio para devolver un valor sobre* `sumaf(6,3)`.



Pero este algoritmo es recursivo, y provocará a su vez otra llamada, esta vez con $a = 6$ *y* $b = 2$*. Esta información se almacenará en la pila, además de reservar espacio para devolver el valor de* `sumaf(6,2)`.

6	2	$sumaf(6,2)$
6	3	$sumaf(6,3)$

Se realizan nuevas llamadas recursivas de forma similar hasta que se llega a una que hará cierta la condición de parada ($b = 0$*),*

6	0	$sumaf(6,0)$
6	1	$sumaf(6,1)$
6	2	$sumaf(6,2)$
6	3	$sumaf(6,3)$

En ese momento ya se puede resolver directamente el problema, por lo que no se realizan nuevas llamadas recursivas. Se le asigna un valor a `sumaf(6,0)`*, que será* 6 *(ya que si* $b = 0$*, entonces* `sumaf(a,b)=a`*) y comienza el proceso de vaciado de la pila, puesto que se puede continuar con las ejecuciones interrumpidas y se podrá ir asignando en cada llamada a* $sumaf$ *el valor obtenido más* 1*:*

6	0	**6**
6	1	$sumaf(6,1)$
6	2	$sumaf(6,2)$
6	3	$sumaf(6,3)$

$\longrightarrow$

6	1	(6+1=)***7***
6	2	$sumaf(6,2)$
6	3	$sumaf(6,3)$

$\longrightarrow$

6	2	(7+1=)***8***
6	3	$sumaf(6,3)$

$\longrightarrow$

6	3	(8+1=)***9***

Una vez resuelta la primera llamada, se procedería a realizar la asignación del valor 9 *sobre la variable* x*.*

9.3.1. Utilidad de la Recursividad.

El objetivo de este apartado es discutir cuáles son las ventajas y cuáles los inconvenientes de utilizar un algoritmo recursivo frente a uno iterativo.

Tal y como se acaba de ver, el empleo de algoritmos recursivos tiene asociado un mayor empleo de memoria (uso de la pila) y un mayor tiempo de ejecución (tiempo que se emplea en apilar/desapilar) que sus equivalentes soluciones algorítmicas no recursivas.

Otra cuestión a tener en cuenta es que, al igual que en los algoritmos no recursivos, hay distintas descripciones recursivas para solucionar un mismo problema. Cada uno de estos algoritmos realizará mayor o menor número de operaciones si se comparan entre sí. En el caso de los algoritmos recursivos, hay que ser especialmente cuidadoso con esto, puesto que fácilmente se pueden obtener algoritmos con un gran número de llamadas, que se traduzcan en programas muy lentos. Tómese,

por ejemplo, el algoritmo visto anteriormente para calcular la sucesión de Fibonacci; si se desea calcular el sexto término de la sucesión ¿cuántas llamadas recursivas hay que hacer?.

$$fibon(6) = \begin{cases} fibon(5) = \begin{cases} fibon(4) = \begin{cases} fibon(3) = \begin{cases} fibon(2) \\ + \\ fibon(1) \end{cases} \\ + \\ fibon(2) \end{cases} \\ + \\ fibon(3) = \begin{cases} fibon(2) \\ + \\ fibon(1) \end{cases} \end{cases} \\ + \\ fibon(4) = \begin{cases} fibon(3) = \begin{cases} fibon(2) \\ + \\ fibon(1) \end{cases} \\ + \\ fibon(2) \end{cases} \end{cases}$$

En este caso, además de gastar más memoria y de tardar más tiempo en ejecutarse que una versión no recursiva debido al tiempo empleado en apilar y desapilar, se calcularían dos veces `fibon(4)`, tres veces `fibon(3)`, cinco veces `fibon(2)` y tres veces `fibon(1)`, por lo que, en total, se resuelven 13 subproblemas. Si, por ejemplo, se quisiera calcular `fibon(15)`, serían *1.217* los subproblemas a resolver y `fibon(30)` supone resolver *1.664.078* subproblemas. Esto es debido a que el número de llamadas para calcular el término n-ésimo de la sucesión de Fibonacci, siguiendo este método, sigue una fórmula exponencial. En cambio es posible resolver este mismo problema mediante el siguiente algoritmo iterativo, en el que sólo se calcula una vez cada subproblema:

```
int fibonNoRec(int n){
    int ultimo, penultimo, i, f;

    ultimo = 1;
    penultimo = 1;
    f = 1;
    for (i=3; i<=n; i=i+1){
         f = ultimo + penultimo;
         penultimo = ultimo;
         ultimo = f;
    }

    return f;
}
```

Evidentemente se puede usar la misma idea general de este algoritmo no recursivo para escribir uno recursivo, en el que cada subproblema se calcule solamente una vez:

```
int fibonacci(int n){
    int f;

    if (n==1)
       f = 1;
    else
       if (n==2)
          f = 1;
       else
          fibon2(2,n,1,1,&f);

    return f;
}
```

```
void fibon2 (int i, int n, int f2, int f1, int *f){
     if (i==n)
         *f = f2;
     else
         fibon2(i+1,n,f2+f1,f2,f);
}
```

Pero, como se puede observar, se pierde legibilidad: resulta más díficil entender qué hace este nuevo algoritmo recursivo respecto de la versión recursiva anterior.

A la vista de las definiciones y ejemplos expuestos, puede parecer que el estudio de la recursividad en Algorítmica no tiene más que un interés anecdótico. Pero eso realmente no es así. Es cierto que en el empleo de la recursividad hay que tener un especial cuidado para que no se solucionen mucha veces los mismos subproblemas, y también es cierto que, en general, los algoritmos recursivos son algo más lentos en ejecución que sus equivalentes versiones no recursivas.

No obstante, el empleo de la recursividad para exponer soluciones algorítmicas presenta muchas ventajas:

expresividad es decir, para muchos problemas la descripción recursiva suele ser más elegante y sencilla de escribir (lo que mejora la legibilidad), mientras que su descripción no recursiva puede resultar más difícil de realizar,

facilidad de análisis ya que permite el uso de fórmulas de recurrencias para analizar el coste en operaciones de los algoritmos, y

corrección ya que la definición recursiva de un algoritmo permite utilizar el mecanismo de inducción matemática para demostrar su corrección formal.

De hecho, estas tres ventajas han convertido a la recursividad en una gran herramienta de programación: el uso de la recursividad es frecuente en la descripción léxica y sintáctica de los lenguajes de programación (C, Python, Pascal, etc.), en el funcionamiento de las estructuras de control de otros lenguajes de programación orientados a la denominada *Inteligencia Artificial* (Lisp o Prolog). Además, muchas estructuras de datos se definen de forma muy simple recursivamente, como ocurre en el caso de las listas, colas, árboles, etc[3].

[3] Y, tal y como veréis en las asignaturas relacionadas con la definición e implementación de estructuras de datos, la definición formal de estas estructuras como *TAD*, Tipo Abstracto de Datos, es siempre recursiva.

Otro ejemplo típico de uso extensivo de la recursividad es el fijar algunos esquemas de programación para determinado tipo de problemas, como son los esquemas de *ramificación y poda* (branch & bound), *divide y vencerás* (divide & conquer) o *vuelta atrás* (backtracking)[4].

Esto puede provocar un conflicto a la hora de solucionar un problema, en cuanto a decidirnos entre solucionarlo mediante un algoritmo recursivo, con sus ventajas formales (legibilidad, análisis del coste y corrección), o mediante un algoritmo iterativo, con sus ventajas prácticas (velocidad de ejecución). La decisión a adoptar depende de cada caso. Pero las ventajas que aporta son tantas que muchos programadores han tratado de desarrollar soluciones iterativas a ciertas "elegantes" versiones recursivas. Fruto de este estudio se han desarrollado distintos métodos para poder convertir un algoritmo recursivo en iterativo sin perder la expresividad de la versión recursiva. Estos métodos, denominados *transformaciones recursivo-iterativas*, se apoyan en la estructura de control iterativa (`while`) y se suelen basar en la "simulación" de las llamadas recursivas, almacenando de forma iterativa tanto los parámetros como las variables intermedias en una pila, para que, posteriormente, una vez que se llega a la condición de salida del bucle (que coincidirá con la condición de parada recursiva), se puedan ir recuperando los datos almacenados.

Por último hay que comentar que el uso de la recursividad no dota de más potencia a nuestro lenguaje algorítmico, es decir, no permite resolver tareas que sin el empleo de la recursividad no se puedan resolver. De hecho, la existencia de las transformaciones recursivo-iterativas puede verse como una demostración de que cualquier problema que tenga una solución recursiva tiene también una solución iterativa. El planteamiento recíproco también es cierto, es decir, cualquier algoritmo iterativo puede ser transformado en recursivo.

9.4. Lecturas Recomendadas.

1. "Thinking Recursively". Eric S. Roberts. John Wiley & Sons. 1986.

   Está en inglés, pero es fácil de leer. Y los ejemplos están en Pascal (el libro tiene sus años). Pero os recomendamos que echéis un vistazo a los capítulos 1, 3 y 4. El capítulo 2 ha servido además como referencia para el próximo tema. Y los otros capítulos son una introducción simple al uso de la recursividad en IA, en esquemas algorítmicos, en estructuras de datos y en aplicaciones gráficas (fractales y demás).

2. Capítulo 8 del libro "Lisp. Introducción al Cálculo Simbólico". D. S. Touretzky.

   Una forma curiosa de presentar la idea de recursividad: en época de los alquimistas, los dragones servían como "computadoras". Martín es un aprendiz de alquimista que debe trabajar con un dragón perezoso. Y descubre que la recursividad es un buen truco para hacer trabajar al dragón.

[4] Estos esquemas se suelen aplicar a los problemas a los que se denominará *intratables* en el próximo tema, que son problemas cuyo coste computacional temporal se expresa mediante una función mayor que cualquier polinomio.

9.5. Problemas Propuestos.

1. Dado el algoritmo recursivo,

```
int Recursivo(int n) {
    int aux;

    if (n==0)
        aux=0;
    else
        aux=Recursivo(n-1)+(n*n);
    return aux;
}
```

   - Realizar una traza con n=5.
   - Explicar brevemente qué hace el algoritmo.

2. Realizad una traza del siguiente código, para saber qué hace el algoritmo `Recursivo`:

```
#define N 6
int Recursivo(int v[], int n);

int main() {
    int v[N]={1, 2, 3, 4, 5, 6};
    int x;

    x=Recursivo(v, N-1);
    printf("\nValor de x es %d.\n", x);
}

int Recursivo(int v[], int n) {
    int aux;
    if(n==0)
        aux=v[n];
    else
        aux=Recursivo(v, n-1)+v[n];
    return aux;
}
```

3. ¿Qué ocurre al ejecutar el siguiente fragmento de código? Es decir, ¿qué hace el algoritmo `Recursivo`?

```
#define N 6
void Recursivo(int v[], int i, int n, float *p);

int main() {
    int v[N]={4, 5, 2, 7, 8, 1};
    float m;

    m=0;
    Recursivo(v, 0, N, &m);
    printf("\nEl valor de m es %4.2f.\n", m);
}
```

```
void Recursivo(int v[], int i, int n, float *p) {
   if (i<n){
        *p = *p + v[i];
        Recursivo(v, i+1, n, p);
    }
    else
        *p = *p / n;
}
```

4. ¿Qué ocurre cuando se ejecuta el algoritmo `Recursivo`? ¿Cómo varía el valor de la variable apuntada por `total`?

```
#define N 10
void Recursivo(float v[], int i, int n, float *total);

int main() {
    float v[N]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    float x;
    int i;

    x=0;
    Recursivo(v, 0, N, &x);
    printf("\nValor de x es %f.\n", x);
    for(i=0;i<N;i=i+1)
        printf("v[%d]=%f\n", i, v[i]);
}
void Recursivo(float v[], int i, int n, float *total){
    if(i<n){
        *total=*total+v[i];
        Recursivo(v, i+1,n, total);
        v[i]=v[i]/(*total);
    }
}
```

5. Indicar el *Dominio de Definición* del siguiente algoritmo:

```
int XXX (int x, int y){
  int aux;
  if (y==0)
    aux=x;
  else {
    y=y-1;
    aux=XXX(x-1,y-1)+x;
  }
  return aux;
}
```

6. ¿Qué operación realiza el algoritmo `noSe`?

```
int noSe(int b, int v[], int n);

int main() {
    int v[N];
    int x, p;
```

```
    ...
    p=noSe(x,v, N-1);
    ...
}

int noSe(int b, int v[], int n) {
    int nolose;
    if (n==-1)
        nolose=-1;
    else {
        if (b==v[n])
            nolose=n;
        else
            nolose=noSe(b,v,n-1);
    }
    return nolose;
}
```

7. Dado el algoritmo `Misterio`

```
int Misterio(int n1, int n2){
    int aux;

    if (n2==0)
        aux=0;
    else
        aux=Misterio(n1*10,n2-1)+n1;
    return aux;
}
```

   - Realizar una traza con n1=5 y n2=4.
   - ¿Qué hace el algoritmo?¿Seguro?. Hacer otra traza con n1=518 y n2=4.

8. Dado el algoritmo `Recursivo`

```
int Recursivo(int n1, int n2){
    int aux;

    if(n1==1)
        aux=n2;
    else {
        aux=Recursivo(n1/2, n2*2);
        if (n1%2 != 0)
            aux=aux+n2;
    }
    return aux;
}
```

   - Realizar una traza con n1=27 y n2=4.
   - Justificar qué hace el algoritmo.

9. Realizar una traza del Algoritmo Misterio, sabiendo que la matriz m, que se habrá definido de un tipo matriz de ENTERO de nxn componentes, tiene todas sus componentes inicialmente a 0, y que se realiza la llamada `Misterio(3, m, 2, 1, 1)`.

```
void Misterio(int n, matriz m, int i, int j, int x) {
  if (x <= n*n){
    Comprobar(n, &i, &j);
    if (m[i][j] == 0){
      m[i][j] = x;
      Misterio(n, m, i+1, j+1, x+1);
    }
    else
      Misterio(n, m, i+1, j-1, x);
  }
}

void Comprobar(int r, int *f, int *c){
  if(*f<0) *f=r-1;
  else {
    if (*f>(r-1)) *f=0;
  }
  if(*c<0) *c=r-1;
  else {
    if (*c>(r-1)) *c=0;
  }
}
```

10. Alguien ha propuesto el siguiente algoritmo para resolver el problema de la división entera:

```
void DivEntera(int dvdo, int dvsor, int *coc, int *resto){
    DivEntera(dvdo-dvsor,dvsor,coc,resto);
    *coc = *coc +1;
}
```

Es posible que no esté bien. Teniendo en cuenta que se supone que sobre `coc` se devuelve el cociente de la división entera y sobre `resto` el resto de la división entera, haz las modificaciones necesarias para que este algoritmo sea correcto.

11. Se quiere escribir un algoritmo recursivo que inicie los elementos de un vector de enteros a un valor dado, `k`. Para ello se ha hecho el siguiente razonamiento: *se asigna a un elemento cualquiera v[i] el valor k y, después, se hace lo mismo con los elementos v[i-1] y v[i+1], si es que existen*.

Se propone el siguiente algoritmo:

```
void iniciar(int i, int v[], int n, int k){
    if ((i < n) && (i > -1)){
        v[i]=k;
        iniciar(i+1, v, n, k);
        iniciar(i-1, v, n, k);
    }
}
```

- ¿Es correcto?¿Por qué? (nota: realizad una traza con n=5 e i=3).
- ¿Habría alguna forma simple de arreglarlo? (nota: NO VALE hacer una solución alternativa, se debe reformar para que sea correcto y admita cualquier valor inicial de i).

12. Dado el siguiente algoritmo:

```
def rimar(cad1, cad2, n):
  rima = 1
  if (n!=0):
     if (cad1[-1] == cad2[-1]):
        rima = rimar(cad1[:-1], cad2[:-1], n-1)
     else:
        rima = 0
  return rima
```

a) Haz la traza de las siguientes llamadas:

```
rimar('corazon', 'melon', 5)
rimar('oriente', 'poniente', 5)
rimar('amenaza', 'fantasmon', 3)
```

b) Indica la precondición y la postcondición de `rimar`. ¿Qué ocurre si se realiza la llamada `rimar('ser', 'coser', 5)`?. Modifica el código de la función `rimar` para que "eso" no ocurra.

13. Escribid un algoritmo recursivo que calcule la suma de los `n` primeros números naturales.

14. Escribid un algoritmo recursivo que calcule el número de dígitos de un número entero.

15. Escribid un algoritmo recursivo que calcule x^n, siendo `x` y `n` enteros, sabiendo que:

$$x^n = \begin{cases} 1 & si\ n = 0 \\ x * x^{n-1} & si\ n\ es\ impar \\ (x^{n/2})^2 & si\ n\ es\ par \end{cases} .$$

16. Escribid un algoritmo recursivo que acepte como parámetro un entero positivo y escriba por pantalla su representación en binario.

17. Escribid un algoritmo recursivo que, dada una cadena, la escriba invertida en la pantalla. Es decir, si la entrada es, por ejemplo, `"Hola, que tal!"`, en la pantalla se debe escribir `¡lat euq ,aloH"`.

18. Escribid dos algoritmos recursivos para imprimir los valores almacenados en una lista de tipo base real: el primero la debe imprimir desde el primer hasta el último elemento y el segundo desde el último hasta el primero.

19. Escribid una función recursiva que calcule el cuadrado de un número entero n, $n \geq 1$, mediante el siguiente método: el cuadrado de n es igual a la suma de los n primeros números impares. Por ejemplo, el cuadrado de 4 es 1+3+5+7=16.

20. Escribid una función recursiva que dada una cadena `cad` y dos caracteres `c1` y `c2`, indique si `c1` aparece inmediatamente antes de `c2` en `cad`, al menos una vez.

21. Escribir una función recursiva a la que se le pasará una cadena de caracteres; dicha cadena puede contener cualquier carácter. Nos interesa localizar los numéricos (entre `'0'` y `'9'`) ya que la función debe devolver la suma de los dígitos que haya en la cadena.

    Por ejemplo si la cadena fuera,

    `"Vivo en el numero 13, calle Melancolia"`

    tendría que devolver `1+3`, o sea, `4`.

22. Escribir una función recursiva a la que se le pasará una cadena de caracteres, que puede contener cualquier carácter. La función debe devolver una cadena idéntica a la original, pero en la que se hayan eliminado todos los caracteres que no sean alfanuméricos. Es decir, se eliminan blancos, signos de puntuación, etc.

    Por ejemplo si la cadena fuera, `''Vivo en el numero 13, calle Melancolia"` tendría que devolver `''Vivoenelnumero13calleMelancolia"`

23. Un número *diabólico* es un número que, cuando se pasa a base 2, tiene un número impar de `0`'s. Por ejemplo, el 0, el 2 (`10`), el 5 (`101`), el 6 (`110`), el 8 (`1000`), el 11 (`1011`), ...

    Hay que escribir una función recursiva que, dado un valor entero K, indique si K es o no es diabólico.

24. Escribir una función recursiva que, dado un número entero, indique si dicho número es o no es *apocalíptico*.

    Se dice que un número es apocalíptico si contiene entre sus dígitos la secuencia 666. Por ejemplo, el 16667234, el 6660, el 34666, el 66666666, el 10266663... El 1346786956, por ejemplo, NO lo es: los '6' no forman secuencia.

25. Escribir una función recursiva que, dado un número natural, devuelva la suma de sus dígitos pares. Por ejemplo, dado 16582234, debería devolver 22, ya que 6+8+2+2+4 = 22. Con 13553, debería devolver 0, ya que ningún dígito del número es par.

26. Escribir una función (o procedimiento) recursiva que calcule la suma de los divisores pares de un valor entero positivo. Por ejemplo, dado el 28, el resultado esperado es $2 + 4 + 14 = 20$.

27. Escribir una función recursiva que que dada una cadena devuelva `cierto` si todos sus caracteres están ordenados en orden creciente (según la tabla ASCII) y `falso` en caso contrario.

    Por ejemplo, la función devolverá `cierto` con `'LUZ'`, `'Racimos'`, `'ACEITe'`, `'ELOY'`, `'1358km'` y `'137AZ'` y devolverá `falso` con `'ACEITUNA'`, `'km1358'`, `'acuDIR'` y `'ABC248'`.