Recursividad Esencial en la Resolución de Problemas

Sonia Rueda srueda@criba.edu.ar
Silvia Castro uscastro@criba.edu.ar
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
(8000) Bahía Blanca

Resumen

Este trabajo presenta el modo en que se introduce el concepto de Recursividad en la cátedra de Informática del Departamento de Ciencias de la Computación de la Universidad Nacional del Sur. Según nuestra experiencia, en las materias iniciales de programación la recursividad resulta ser uno de los temas más complejos. Esta complejidad no reside en el uso de la computadora, ni tampoco en el hecho de que las facilidades provistas por los lenguajes de programación para soportar recursividad resulten difíciles de ententer. La dificultad consiste en "plantear" soluciones recursivas. La recursividad resulta una forma diferente de pensar y razonar acerca de ciertos problemas. Es justamente desde este punto de vista que se encara la presentación del tema en la materia.

Introducción

El propósito de este artículo es ilustrar cómo se introduce el concepto de recursividad en el curso Informática de la carrera Licenciatura en Ciencias de la Computación de la Universidad Nacional del Sur. El artículo comienza describiendo las motivaciones que condujeron a la presentación de la Recursividad del modo que se hace desde hace seis años. Luego se detalla el problema con el que se presenta el tema, para continuar con una serie de ejercicios que permitirán al alumno introducirse gradualmente en el diseño de algoritmos recursivos. Posteriormente se muestra cómo se derivan en forma natural y original algoritmos en los que es apropiada la recursividad cruzada y el backtracking, luego de lo cual consideramos que los alumnos está en condiciones de *seguir* la ejecución de algoritmos. Recién entonces se comparan algoritmos recursivos e iterativos. Por último se muestran distintos ejemplos de mayor complejidad.

Cómo y por qué introducir así la Recursividad

En el año 1989 el plan de la Licenciatura en Ciencias de la Computación cambió, produciéndose una reestructuración de las materias "Informática" y "Estructuras de Información y Archivos". Estas dieron lugar a "Resolución de Problemas y Algoritmos", "Informática" y "Estructuras de Datos". Cada tema pudo ser tratado y ejercitado entonces con mayor profundidad, permitiendo que los alumnos alcanzaran un mejor nivel en la maduración de un tema, antes de pasar al siguiente.

Recursividad era uno de los temas que a los alumnos les resultaba más complejo y que parecía requerir más tiempo que el que se le había dedicado hasta el momento. En el plan anterior, poco después de presentar el tema, se utilizaba para resolver problemas que involucraban estructuras de datos complejas en sí mismas. Al cambiar el plan, se replanteó el momento en que se introduciría recursividad. "Resolución de Problemas y Algoritmos" e "Informática" se orientan a la resolución de problemas y su implementación en Pascal.

"Informática" se dicta en el segundo cuatrimestre del primer año, después del curso inicial de "Resolución de Problemas y Algoritmos". El objetivo fundamental de esta última es la resolución de problemas de simple complejidad mediante el uso de la computadora, usando el lenguaje de programación Pascal. La primera parte de la materia se dedica a la resolución de problemas en general, no necesariamente implementables en un lenguaje de programación y luego, al diseño de algoritmos usando un lenguaje de diseño propuesto ad hoc. Los alumnos que comienzan el curso de "Informática" deberían estar en condiciones de:

- Formular un algoritmo a partir del enunciado siguiendo la metodología top-down.
- Implementar una solución estructurada y modulada en Pascal.
- Distinguir convenientemente los conceptos de función y procedimiento y el pasaje de parámetros.
- Comprender el concepto de tipo de dato y consistencia de tipos, aplicado únicamente a datos simples.
- Seleccionar las estructuras de control adecuadas para el problema.

En "Resolución de Problemas y Algoritmos" el énfasis está dado en el algoritmo y todos los problemas manejan datos simples. En "Informática" se amplía el espectro a problemas: encontrar una buena representación para los datos constituye en sí mismo

un problema. Dentro de este esquema "Resolución de Problemas y Algoritmos" hubiera sido quizás el marco ideal para presentar recursividad. Sin embargo, para implementar algoritmos recursivos es necesario haber madurado y ejercitado adecuadamente algunos temas tales como subprogramas y pasaje de parámetros. Se consideró entonces más adecuado que recursividad fuera presentado en "Informática", en la primera parte de la materia. Así, era posible dedicar más tiempo al tema y este podía ser madurado antes de combinarse con otros nuevos. Surgió entonces la pregunta: ¿esto es todo los que se necesita para que el tema resulte menos complejo?.

Según nuestra experiencia existen dos factores que inciden para que el tema tarde en ser comprendido:

- Cuando un alumno adquiere habilidad en el uso de un recurso tarda en aprender a usar otro alternativo. Concretamente, una vez que el alumno está acostumbrado a plantear y resolver problemas en forma iterativa, no es fácil seducirlo con un planteo recursivo. Le resulta algo mágico y no entiende muy bien para qué debe utilizar la recursividad, ya que con otras herramientas que le son familiares, podría haber alcanzado una buena solución.

- En la medida en que los alumnos toman más contacto con la computadora, es común que encaren la resolución pensando en cómo va a *funcionar* el programa cuando sea ejecutado.

El primer factor lo atacamos seleccionando ejemplos que ilustran el concepto de recursividad y no pueden ser resueltos iterativamente, salvo que se definan estructuras de datos más o menos complejas. Como a esta altura los alumnos sólo han trabajado con datos simples, no están en condiciones de plantear una solución iterativa. De esta manera se intenta lograr que no *piensen* una solución iterativa y la transformen luego en recursiva solo porque se les pide una solución recursiva. La intención es que usen la recursividad como estrategia de resolución.

El segundo punto fue un poco más difícil de contrarrestar. Es inevitable que los alumnos intenten saber qué está pasando *"adentro"* de la computadora. Consideramos que este hábito puede entorpecer la resolución misma. Si el alumno encara el problema pensando en su ejecución, probablemente no alcance una solución legible y elegante. Es natural que tienda a agregar variables indiscriminadamente, se maree intentando ligar parámetros actuales y formales y analizando cómo será el manejo de memoria.

Aunando los dos aspectos, nuestra propuesta apuntó y apunta a pasar del enunciado a un planteo recursivo que se le corresponda naturalmente. En el planteo recursivo debe quedar clara la condición de corte y la resolución del problema en términos de casos más simples del mismo problema. Recién cuando este planteo tome la forma de un algoritmo, estaremos en condiciones de encarar la implementación en Pascal y considerar todos aquellos aspectos que hacen a la ejecución.

El modo en que actualmente presentamos el concepto de recursividad es producto de una evolución progresiva a través de los años y de los cambios producidos en los planes de estudio de nuestra Licenciatura en Ciencias de la Computación. Implementamos nuestra propuesta en el año 1990 y a partir de entonces sólo se hemos hecho algunas modificaciones que no alteran los aspectos conceptuales, sino que constituyen pequeños ajustes.

Torres de Hanoi

Nos hemos cuestionado repetidamente qué problema resulta adecuado para presentar el concepto de recursividad. Algunos alumnos quedan inmediatamente atraídos por la recursividad, cualquiera sea el ejemplo con el que se encare el tema, e intentan aplicarla inmediatamente. Otros la consideran al principio un poco *mágica* y de ningún modo les resulta seductora.

Si presentamos recursividad a través de ejemplos sencillos y familiares tales como calcular una potencia o el factorial, la pregunta inmediata es: ¿para qué usar recursividad si el problema ya podía ser resuelto con las estructuras de control que conocíamos y nos resultaban familiares ? Optamos entonces por escoger un problema que, con las facilidades del lenguaje presentadas hasta el momento, no pueda ser resuelto en forma iterativa.

El juego de las Torres de Hanoi es probablemente un clásico de recursividad. El juego consiste en tres torres de igual tamaño colocadas sobre una plataforma. La primera contiene n discos apilados de mayor a menor. El objetivo del juego es mover los n discos de la torre 1 a la torre 3, usando la torre 2 como auxiliar, bajo la condición de que sólo puede moverse un disco a la vez y en ningún momento un disco de mayor tamaño puede quedar encima de uno más pequeño.

En la primera clase de la materia enunciamos el problema para que los alumnos jueguen *manualmente*. En la clase siguiente no les resulta difícil entender el planteo recursivo, aunque no haya sido capaz de esbozarlo por sí mismo. Planteamos la solución del problema en los siguientes términos:

> "El caso más simple se presenta cuando la cantidad de discos, n, es 1. La solución es directa, consiste en moverlo desde la torre 1 hacia la torre 3. Si n es 2 la solución nuevamente es sencilla, movemos el disco 1 a la torre 2, el disco 2 a la torre 3, y nuevamente el disco 1 de la torre 2 a la torre 3.

Es fundamental que cada paso de la explicación se complete con un dibujo en el pizarrón que grafique el movimiento.

> La solución para tres discos es algo más complicada y para cuatro resulta realmente difícil si no usamos un método sistemático. ¿Qué estrategia podríamos usar ? "

La solución en general no surge de los alumnos, pero sí pueden entenderla una vez que la presentamos en los siguientes términos:

> "Supongamos que nuestro problema consiste en mover 4 discos desde la torre fuente 1 a la torre destino 3 usando la torre 2 como auxiliar. El problema resulta complejo, pero si de alguna manera fuésemos capaces de mover los tres discos más pequeños desde la torre fuente a la torre auxiliar, podríamos mover el disco 4 a la torre destino y luego, mover los tres discos que están en la torre auxiliar a la torre destino."

El problema de mover cuatro discos se transforma en tres subproblemas:

1. Mover los 3 discos más pequeños a la torre auxiliar
2. Mover el disco 4 a la torre destino
3. Mover los 3 discos más pequeños a la torre destino

Los pasos 1 y 3 son muy similares y es claro que si resolviéramos uno, el otro quedaría resuelto también. El paso 2 es trivial.

La estrategia de resolución es en definitiva siempre la misma: **manejar la complejidad de un problema descomponiéndolo en subproblemas más simples**, sólo que en este caso dos de los subproblemas son instancias más simples del problema original.

> "Nuestro problema está resuelto siempre que seamos capaces de mover 3 discos de una torre a la otra y para hacerlo podemos usar la misma estrategia. El proceso no puede repetirse indefinidamente y es necesario encontrar una condición de corte..."

Durante todo este proceso graficamos los movimientos de los discos sobre las torres, para reflejar los distintos estados. Recién cuando la estrategia ha sido comprendida, refinamos el algoritmo, todavía muy informal, con datos constantes:

Mover 4 discos desde la torre 1 hacia la torre 3 usando la torre 2 como auxiliar
  Mover 3 discos desde la torre 1 hacia la torre 2
  Mover el disco 4 de 1 a 3
  Mover 3 discos desde la torre 2 hacia la torre 3

En cuanto intentamos ocuparnos de mover 3 discos notamos que esta versión no sólo es informal sino incompleta. Para mover 3 discos necesitamos una torre auxiliar. Una versión más refinada sería:

Mover 4 discos desde la torre 1 hacia la torre 3 usando 2 como auxiliar
  Mover 3 discos desde la torre 1 hacia la torre 2 usando 3 como auxiliar
  Mover el disco 4 de 1 a 3
  Mover 3 discos desde la torre 2 hacia la torre 3 usando 1 como auxiliar

Escribimos este algoritmos para 3 discos y luego inferimos la solución general donde los nombres de las torres se reemplazan por variables. Nos detenemos particularmente para analizar cómo *jugamos* con las variables de manera tal que la torre auxiliar del problema principal se transforma en la torre destino del problema para 3 discos y luego en la fuente. Probablemente este cambio de roles de las torres es lo que más cuesta entender y seguir.

Es importante que este algoritmo se corresponda fielmente con el planteo. Aún así para la mayoría de los alumnos la cuestión es ahora, "lo entiendo, puedo ver que si ´dibujo´ cada estado llego a la solución, pero ¿cómo lo hace la computadora ? "

Nuestra intención es que en esta etapa no se preocupen por hacer una traza o ver cómo funciona efectivamente la recursividad. Es más, a esta altura transformamos el algoritmo en un programa para que puedan ir a la máquina y verificar que efectivamente funciona, pero ni siquiera nos detenemos a explicar detalladamente por qué elegimos pasaje de parámetros por valor en algunos casos y por referencia en otros.

Según nuestra experiencia, si desde el principio les explicamos detalladamente cómo es el manejo de memoria, no sólo les resulta difícil entender los algoritmos que les mostramos, sino que de allí en adelante les es todavía más difícil plantear ellos mismos soluciones recursivas, pues se pierden intentando paralelamente escribir el algoritmo y ver cómo va a funcionar. Nuestro objetivo es entonces encontrar *planteos recursivos* para cada uno de los problemas que proponemos, esto es:

- Identificar el Paso Recursivo, analizando cómo *reducir* el problema a uno o más casos más simples del mismo problema.
- Identificar el Caso Trivial o condición de corte.

Recién en una etapa posterior deberán ser capaces de transformar este planteo en un algoritmo más o menos formal. Cuando esta tarea les resulte natural empezaremos con las primeras trazas. Para lograr esta evolución presentamos una secuencia de problemas de complejidad creciente.

Diseño de algoritmos recursivos

En el primer grupo de problemas incluímos ejercicios numéricos muy sencillos y que podrían resolverse usando iteración. En esta etapa enfatizamos la importancia de NO plantear el problema en forma iterativa y luego transformar este planteo en una solución recursiva, ya que estas soluciones resultan por lo general poco naturales y difícil de leer.

Como a lo largo de toda la materia, intentamos que los alumnos entiendan que la legibilidad y estilo de programación son importantes aún para problemas sencillos. Nuestra intención es que adquieran buenos hábitos de diseño e implementación desde el principio, ya que aunque en un problema sencillo la falta de estas cualidades no resulte dramática, es difícil que el que comienza a programar en forma oscura, adquiera buenos hábitos cuando los programas crecen en tamaño y complejidad.

El primer problema es entonces:

"Hallar el dígito más significativo de un número natural N"

Este ejercicio es sumamente sencillo, tiene un único dato de entrada y uno de salida y la idea es que, además de hallar el planteo recursivo, los alumnos noten cómo a veces parte de la resolución del problema consiste en transformar las primitivas que tenemos en lo que en realidad necesitamos.

La condición de corte es trivial: "el dígito menos significativo de un número con un único dígito es justamente ese dígito". En caso contrario planteamos el problema en términos de "hallar el dígito más significativo del número que resulta de descartar el dígito menos significativo del número original". Es claro que la condición de corte va a ser alcanzada en algún momento.

No disponemos de una primitiva para "descartar un dígito" pero transformamos el problema de manera tal de poder usar la primitiva división entera. Esta transformación, natural para muchos, no lo es tanto para otros. Es por esto que aún en este problema sencillo comenzamos con un algoritmo informal que refleja fielmente el planteo y en

donde escribimos lo que necesitamos, en una versión más refinada vemos como las operaciones que necesitamos se transforman en las primitivas con las que realmente contamos.

"Recuperar el k-ésimo dígito menos significativo de un número natural N"

Este problema es ligeramente más complejo que el anterior ya que tiene dos datos de entrada y la condición de corte es un poco más elaborada, porque N puede tener menos de k dígitos. Comenzamos planteando entonces "el k-ésimo dígito menos significativo de un número es el k-ésimo-1 dígito menos significativo del número que resulta de descartar el dígito menos significativo en el número original" y nos ocupamos luego de las dos condiciones de corte posibles: "k es 0 y entonces el k-ésimo dígito es el dígito menos significativo o N es 0 y entonces no es posible descartar un dígito". Nuevamente la condición de corte va a ser alcanzada.

"Sumar los dígitos de un número natural N"

El problema puede como plantearse como "Sumar los dígitos de un número N con m dígitos, consiste en sumar los m-1 dígitos más significativos y a este valor sumarle el dígito menos significativo de N". Para este planteo proponemos dos algoritmos alternativos, uno de los cuales requiere una variable local. La dificultad de este problema respecto a los anteriores es que después de la llamada recursiva todavía quedan acciones pendientes.

"Recuperar la posición menos significativa en la que aparece un dígito x dentro de un número natural N"

Con este ejemplo mostramos cómo obtener una versión más formal del enunciado puede acercarnos a la solución. El enunciado se transforma en:

"Si $N = d_n\ d_{n-1} \ldots d_1\ d_0$ el problema existe en hallar un k tal que $d_k = x$ y no existe un j tal que $j > k$ y $d_j = x$"

Notamos además como en ocasiones el enunciado constituye en sí mismo un planteo o permite inferirlo fácilmente. Otra particularidad del problema es que el dígito x puede no estar en N.

A continuación proponemos una serie de problemas similares a los vistos hasta el momento - que van a ser retomados en la clase siguiente - y presentamos luego una serie de ejercicios que manipulan caracteres. Notemos que los alumnos todavía no manejan estructuras de datos, por lo tanto no están en condiciones de representar una cadena de caracteres en conjunto, pero sí pueden ir leyendo una cadena caracter a caracter. En los casos más simples la cadena no tiene más restricciones que el terminar con un punto. Esta serie de ejercicios se introduce con cuatro algoritmos que quedan planteados para que ellos mismos analicen:

```
procedure Leer
var ch : char ;
begin
  read(ch) ;
  write (ch) ;
  if ch <> "."
  then
    Leer ;
end
```

```
procedure Leer
var ch : char ;
begin
  read(ch) ;
  if ch <> "."
  then
    begin
      write (ch) ;
      Leer
    end
end
```

```
procedure Leer
var ch : char ;
begin
  read(ch) ;
  if ch <> "."
  then
    begin
      Leer ;
      write (ch)
    end
end
```

```
procedure Leer
var ch : char ;
begin
  read(ch) ;
  if ch <> "."
  then
    Leer ;
  write (ch) ;
end
```

Los problemas que proponemos y resolvemos a continuación son del tipo:

"Contar la cantidad de apariciones de una letra dada en una cadena de caracteres terminada con un punto, leída caracter a caracter"

En este ejercicio hacemos un planteo recursivo considerando como condición de corte la cadena vacía. Sobre esta base es posible presentar una gran variedad de ejercicios, probablemente imponiendo algunas restricciones sobre la cadena.

Nuevamente interesante mostrar cómo un enunciado formal puede transformarse para obtener una versión más precisa, en este caso usando diagramas sintácticos. Este recurso resulta atractivo porque una vez que se obtiene un diagrama que refleja las restricciones sobre la cadena, el planteo es inmediato y el algoritmo recursivo surge naturalmente. El riesgo de proponer esta alternativa como un método para encarar la resolución de este tipo de problemas, es que los alumnos se mecanizan demasiado.

Según hemos comprobado, una cantidad importante de alumnos son capaces de resolver problemas enunciados a través de diagramas sintácticos una vez que han adquirido cierta práctica. Sin embargo lo hacen mecánicamente, sin entender demasiado bien el algoritmo. Es bastante sencillo reconocer al alumno que se ha mecanizado ya que:

- Resuelve el problema siempre que venga enunciado a través de un diagrama sintáctico, pero le resulta muy difícil dibujar él mismo el diagrama si el problema está enunciado verbalmente.
- Le resulta muy difícil modificar su algoritmo si se modifica ligeramente el enunciado.
- Resuelven ejercicios más o menos complejos, siempre que vengan enunciados a través de diagramas sintácticos, y no son capaces de resolver otros más sencillos enunciados verbalmente.
- Cuando más adelante mostramos cómo es el manejo de memoria, les cuesta mucho hacer o hasta seguir una traza.

En la siguiente selección de ejercicios intentamos ilustrar cómo un mismo problema puede resolverse a través de algoritmos bien diferentes. Algunos problemas de este grupo son retomados poco más adelante ya que una de las alternativas consiste en plantearlos usando recursividad cruzada.

Recursividad cruzada

Hasta el momento hemos considerado problemas que podían resolverse a través de algoritmos que quedaban definidos en términos de sí mismos. Con frecuencia la solución de un problema puede quedar modelada a través de un algoritmo A que invoca a otro algoritmo B que a su vez invoca a A, hablamos entonces de recursividad cruzada o indirecta. Los ejercicios que proponemos a continuación pueden resolverse usando recursividad cruzada.

"Leer una cadena de caracteres terminando con un punto y mostrar la misma cadena, salvo las subcadenas encerradas entre paréntesis. Ninguna subcadena contiene subcadenas entre paréntesis y los paréntesis están bien balanceados".

En principio podemos notar que el conjunto de caracteres puede dividirse en cuatro subconjuntos:

- "."
- "("
- ")"
- otro caracter

dando lugar a estados diferentes:

- Terminar
- se interrumpe la impresión
- se reinicia la impresión
- se imprime o no según el estado de impresión

Escribimos el algoritmo de manera tal que cada alternativa quede claramente reflejada. Al implementar este algoritmo perdemos parte de la legibilidad debido a que los condicionales provistos por Pascal son limitados.

Si releemos el enunciado vemos que si no hay subcadenas anidadas y los paréntesis están bien balanceados, no todos los estados son posibles en cada momento. Para que estos hechos sean más evidentes deberíamos escribir un algoritmo en donde en cada paso solo se consideren las alternativas válidas.

Escribimos una segunda versión, mucho más natural, usando llamadas cruzadas. Dos algoritmos, Imprime y NoImprime, permiten reflejar los dos estados principales que se entrelazan alternativamente. Imprime se invoca recursivamente hasta que se lee un "." o un "(". En el primer caso el algoritmo termina, en el segundo invoca a NoImprime. NoImprime se invoca recursivamente hasta que se lee un ")" e invoca entonces recursivamente a Imprime.

A esta altura los alumnos ya deberían ser capaces de resolver problemas más o menos sencillos y consideramos que están en condiciones de entender cómo *funciona* la recursividad y qué características del manejo de memoria la hacen posible. Cuando son capaces de plantear e implementar soluciones recursivas y de entender los aspectos estáticos y dinámicos de la recursividad, podemos comenzar a comparar iteración y recursividad.

Recursividad en Ejecución

El seguimiento de soluciones recursivas en ejecución no debería ser difícil si los alumnos han entendido y madurado los conceptos asociados a la definición e invocación de funciones y procedimientos, en particular en lo que se refiere a pasaje de parámetros. En cualquier caso, en esta etapa retomamos los problemas más sencillos y simulamos su ejecución paso por paso, enfatizando todos las cuestiones que hacen al manejo de memoria.

Los algoritmos propuestos hasta el momento pueden agruparse en:

- Tienen una única llamada recursiva directa y es la última sentencia ejecutable
- Tienen un única llamada recursiva directa pero cuando el control vuelve todavía quedan instrucciones pendientes.
- Tienen más de una llamada recursiva directa.
- Dos o más algoritmos se invocan recursivamente en forma indirecta.

Es claro que la complejidad de esta categorización es creciente y por lo tanto el seguimiento se hace más difícil.

Backtracking

Hasta el momento la condición de corte quedaba siempre expresada por una sentencia condicional. La llamada recursiva puede producirse dentro de un bloque iterativo y esta facilidad permite modelar una estrategia de resolución muy poderosa llamada "backtracking". El backtracking consiste en avanzar en busca de la solución seleccionando un camino entre varios alternativos y luego retroceder y volver a avanzar siguiendo otro. Por lo general se retrocede cuando se comprueba que las alternativas seleccionadas no conducían a la solución, pero también es posible que la estrategia de backtracking se utilice justamente para recorrer todos los caminos posibles.

La mayoría de los problemas que pueden se resueltos usando backtracking requieren el uso de estructuras de datos y por lo tanto las presentamos más adelante. En particular en la última parte de la materia planteamos y resolvemos una colección de problemas clasificados. Una de las clases de problemas son Juegos y en este grupo muchos de los ejercicios puede resolverse usando backtracking. A esta altura solo estamos en condiciones de proponer algunos problemas simples. Por ejemplo:

"Dado un número entero positivo N con m dígitos mostrar todas las permutaciones de sus dígitos."

Planteamos la solución para este problema diciendo "Un conjunto importante de permutaciones puede obtenerse fijando el primer dígito del número en la posición más significativa y luego obteniendo todas las permutaciones de los m-1 dígitos menos significativos". Es claro que se trata de un planteo recursivo ya que queda expresado en términos de un caso más simple - un número con menos dígitos - del problema original. La condición de corte es trivial. Ahora bien, este razonamiento permite obtener

algunas de las permutaciones pero no todas. "Podemos ahora intercambiar el dígito más significativo con el que le sigue que quedará fijo en la primer posición, y obtener nuevamente todas las permutaciones de los m-1 dígitos restantes". A esta altura no es difícil inferir que la solución consiste en "Fijar cada uno de los dígitos del número en la posición más significativa y en cada caso permutar los m-1 dígitos restante de todas las maneras posibles". La solución de este problema es significativamente más compleja que la de los anteriores y requiere varios niveles de refinamiento hasta ser alcanzada.

Iteración vs. Recursividad

Para comparar iteración y recursividad usamos algunos problemas clásicos de recursividad, tales como el factorial o la sucesión de Fibonacci. Planteamos cada problema en forma iterativa y recursiva e implementamos ambas alternativas. La comparación se realiza en términos de legibilidad, naturalidad y eficiencia.

Ambas estructuras de control han sido maduradas lo suficiente como para notar la correspondencia entre cada planteo y su algoritmo asociado y además poder ser comparadas en términos de eficiencia y legibilidad.

Hemos notado que si desde el principio mostramos la versión recursiva e iterativa de cada problema, los alumnos tienden a saltear la etapa de identificar claramente el paso recursivo y la condición de corte, pasando directamente a escribir el algoritmo, obteniendo en general versiones oscuras, complicadas y muchas veces incorrectas.

Resolución de problemas más complejos

El último conjunto de ejercicios son considerablemente más complejos y en muchos casos los retomamos sobre el final de la materia, ya que no todos los alumnos han alcanzado en esta etapa el mismo dominio del tema. Enunciamos a continuación los ejercicios que proponemos actualmente.

Evaluar una expresión aritmética simple, leída de pantalla, consistente de operadores binarios +, -, * y / y y con números enteros no negativos de exactamente un dígito.

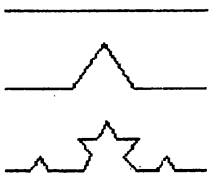
Este es uno de los problemas que formalizamos usando diagramas sintácticos y presenta varias características interesantes:

- La solución se corresponde directamente con el diagrama.
- Existe un único punto de lectura y contrariamente a lo que se podría pensar al comenzar a tratar el problema, Expresión no comienza leyendo la expresión.
- La recursividad cruzada provoca que sean necesarios algunos parámetros que no tuvimos en cuenta al comenzar a escribir el algoritmo.
- Con algunas variaciones da lugar a toda una serie de problemas similares.

Copo de Nieve. Dibujar el Copo de Nieve suponiendo provistas las siguientes primitivas gráficas:

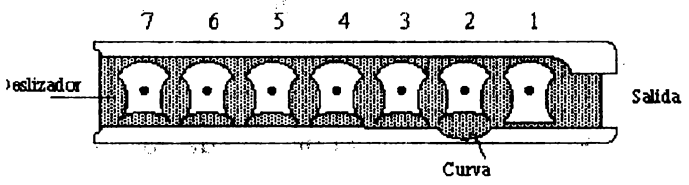
RotarIzq(ang)
RotarDer(ang)
DibujarLinea(long)

Para generar el Copo de Nieve, se debe partir de un triángulo equilátero. Cada lado de este triángulo se reemplazará por cuatro segmentos de línea, cada uno de los cuales es de 1/3 del segmento original, de acuerdo al siguiente esquema:



Este problema, si bien plantea un desafío en cuanto al gráfico, es ilustrativo ver como de acuerdo a la profundidad de la recursividad, obtenemos un determinado nivel de detalle. El poder tener *una representación gráfica* de la recursividad da una idea visual de qué es lo que está ocurriendo.

Spin Out. Este juego consiste en 7 piezas que están sujetas a una tabla deslizante mediante un clavito, alrededor del cual pueden girar sobre sí mismas. Estas pueden estar verticales, como en la figura, en cuyo caso se dirá que están bloqueadas, o podrán estar horizontales, y entonces se dirá que están desbloqueadas.



La meta de este juego consiste en sacar la barra deslizante (en gris) totalmente hacia la derecha, de modo que salga de la caja que la contiene. Para ello, las piezas deben ponerse previamente en posición desbloqueada.

La forma de la caja que contiene a la tabla deslizante y a las piezas, impone determinadas restricciones en cuanto a la forma en que se podrán rotar las piezas. En este problema, que es realmente complejo, se pide a los alumnos que *manualmente* se familiaricen con el juego. Recién entonces estarán en condiciones de comenzar a esbozar un algoritmo. Cabe destacar que las tareas que deben realizarse para llegar a la solución son dos tareas distintas pero entrelazadas entre sí.

Conclusiones

La selección de los ejercicios demandó un esfuerzo considerable. Intentamos obtener una secuencia de problemas ilustrativos, atractivos y de complejidad creciente, de

manera tal que los alumnos no solo sean capaces de entender la solución ya formulada, sino de resolver problemas de complejidad equivalente por sí mismos.

Es importante destacar que el tema Recursividad no queda agotado en este punto. La materia continua presentando los constructores provistos por Pascal para definir y manipular estructuras de datos. Se retoma entonces el concepto de tipo de dato y se describen las facilidades de Pascal para definir Tipos de Datos Estructurados. Se enfatiza la importancia de la abstracción y se enumeran las limitaciones de Pascal para soportar Abstracción de Datos. Extenderemos además nuestro lenguaje de diseño para resolver problemas en donde los datos constituyen en sí mismos un problema. La tercer y última parte de la materia la dedicamos a presentar una serie de problemas clasificados de acuerdo al dominio de aplicación, que conjugan los conceptos, técnicas y recursos del lenguaje presentados anteriormente.

Por último queremos remarcar la importancia de que en la práctica de la materia se mantenga el espíritu con que el tema es presentado en teoría. Por supuesto no es razonable pretender que toda la cátedra trabaje con total uniformidad y hasta es bueno que los alumnos noten que cada persona puede tener un estilo diferente para encarar la resolución de problemas, sin embargo tiene que haber una coherencia en el mensaje y una estrecha comunicación entre los profesores, asistentes y ayudantes de cátedra de manera tal que cada tema y cada ejercicio forme parte de un todo integrado.

Bibliografía

[1] Dromey, *How to Solve It By Computer*, Prentice Hall International, C.A.R Hoare Series Editor, 1982.

[2] Grogono, Peter, *Programming in Pascal.*

[3] Pruhs, Kirk *The SPIN-OUT Puzzle*, ACM SIGCSE Bulletin, Vol. 25, No. 3, Septiembre 1993.

[4] Rueda, Sonia y Castro, Silvia *Apuntes de Cátedra*, Cátedra de Informática, Dpto. de Cs. de la Computación, Universidad Nacional del Sur, Bahía Blanca.