La Perplejidad como Recurso Didáctico

Jorge Aguirre* Nazareno Aguirre†

Resumen

En este trabajo se propone la utilización de la perplejidad y la sorpresa en la enseñanza de la computación. Se recalca la importancia de la utilización de los recursos computacionales en esta tarea, y se presentan algunos ejemplos que han sido utilizados con éxito.

1 Introducción

Las paradojas, enunciados que expresan contradicciones que producen perplejidad en un primer análisis, han desempeñado un importante papel motivador en la historia de la Ciencia.

Se conocen muchas paradojas formuladas en la antiguedad; una de éstas es la conocida paradoja de *Aquiles*: si al comenzar la carrera la tortuga está a cierta distancia delante de *Aquiles* y la velocidad de *Aquiles* duplica la suya, cuándo se encuentran?; si por más veces que *Aquiles* llegue a donde ella está, cuando esto suceda ella habrá avanzado la mitad del camino recorrido por *Aquiles* para llegar a su anterior posición, estando la tortuga de esta manera siempre delante de *Aquiles*. Esta paradoja sugiere el concepto de límite finito de una sucesión infinita y abre las puertas al concepto de Completitud y, por ende, al de número real.

Pero fue en este siglo cuando las paradojas jugaron su rol más destacado en la historia de la matemática.

La paradoja de *Russell* [6][1]- Sea $B = \{X | X \notin X\}$ entonces no puede cumplirse $B \in B$ ni $B \notin B$ - señaló una contradicción en la Teoría de conjuntos llevando a *Russell* a introducir el concepto de *tipo*, concepto que luego habría de tener gran relevancia en las Ciencias de la Computación.

La paradoja de *Berry* [6] - Sea n el menor número cuya definición castellana requiere no menos de mil palabras. El número n está bien definido, ya que como la cantidad de oraciones castellanas con menos de mil palabras es finita,

*Área de Computación. Facultad de Ciencias Exactas, Físico-Químicas y Naturales. Universidad Nacional de Río Cuarto. Departamento de Computación. Facultad de Ciencias Exactas, Físicas y Naturales. Universidad de Buenos Aires. e-mail: jaguirre@dc.uba.ar

†Área de Computación. Facultad de Ciencias Exactas, Físico-Químicas y Naturales. Universidad Nacional de Río Cuarto. e-mail: naguirre@dc.exa.unrc.edu.ar

[1]En su versión popular: en Sevilla hay un barbero que afeita a todos aquellos que no se afeitan a sí mismos, entonces quien afeita al barbero no puede ser él ni dejar de serlo.

es también finito el conjunto de números que ellas definen, por lo cual no es vacío el conjunto de números no definibles por ellas, existiendo entonces su mínimo n.

Sin embargo la frase anterior ha definido a n en sólo catorce palabras -. Esta paradoja condujo a nuestro compatriota Gregorio Chaitin, aún adolescente y luego de una obsesiva preocupación por alcanzar su profunda significación, a la redefinición del concepto de aleatoriedad en términos de complejidad algorítmica: Una cadena de *bits* es aleatoria si no existe ningún programa más corto que ella que la compute [2, 3, 4].

El uso de experiencias que producen perplejidad ha sido usado en la enseñanza de la Física, por Eduardo Flichman y Agustín Rela en cursos introductorios de la UBA, para despertar el interés del estudiante, incentivar la búsqueda de una explicación y destacar la importancia del razonamiento científico.

Los autores consideran que iguales resultados pueden lograrse en la enseñanza de la Computación. Si una presentación logra sorprender al alumno, es muy difícil que éste se olvide. El camino que siga para explicarla contará con una gran expectativa y tampoco será fácilmente olvidado. Además la presentación de ciertos casos que producen perplejidad puede contribuir a la creación del hábito de buscar rigurosamente la respuesta a problemas que parecen incomprensibles.

2 La Perplejidad como Herramienta Motivadora

Como ya fue señalado anteriormente, la perplejidad puede ser utilizada para lograr interés en los alumnos. Esto es principalmente útil en materias con gran contenido teórico, usualmente poco atractivas para los estudiantes.

Utilizaremos como ejemplo en esta sección a la Complejidad de Algoritmos. Es usual en materias que abarquen este tema encontrar estudiantes que tengan la sensación de que, debido al poder computacional actual (aún la computadora hogareña más barata de hoy en día es mucho más poderosa que algunas de las mejores máquinas del mundo de hace 25 o 30 años), ya no es necesario "perder tiempo" o preocuparse por escribir programas eficientes. Los estudiantes pueden pensar que esta práctica era muy útil hace unos años, cuando el hardware era muy caro, pero, gracias a la tecnología actual, ya no tiene sentido estudiar la complejidad de los programas (al menos puede parecer una actividad inútil para programas que no manipulan cantidades importantes de datos).

Este pensamiento puede ser refutado fácilmente mediante algunos ejemplos sorprendentes. Consideremos el siguiente programa funcional:

```
selsort :: [Int] -> [Int]
selsort []     = []
selsort (x:xs) = (min (x:xs)):(selsort (qmin (x:xs))


min :: [Int] -> Int
min [x]    = x
```

```
min (x:xs) = x     , if x<=(min xs)
           = min xs, otherwise

qmin :: [Int] -> [Int]
qmin [x]    = []
qmin (x:xs) = xs         , if (min (x:xs))==x
            = x:(qmin xs), otherwise
```

Si bien este programa parece presentar algunas secciones ineficientes, parece ser bastante natural, y es probablemente la manera normal en la que un alumno resolvería el problema de codificar *Selection Sort* en un lenguaje funcional. Además, este programa no fue escrito para manejar estructuras de datos de gran envergadura, sino sólo una lista de, a lo sumo, algunas decenas de enteros. Lo sorprendente de este ejemplo es la ineficiencia con la cual resuelve el problema de ordenar una lista de enteros ordenada de mayor a menor. En este caso, para listas pequeñas, como por ejemplo, una lista de 30 elementos, el algoritmo "tarda" varios **minutos**, incluso en computadoras personales de gran poder (Celeron 300Mhz., por ejemplo). Algo aún más sorprendente con este caso es que el algoritmo presentado tardaría varios siglos en ordenar una lista de cien números enteros ordenados en forma decreciente (y téngase presente que son sólo 100 números enteros los que se desea ordenar...). Peor aún, ninguna mejora en hardware (actual o futura) hará funcionar a este algoritmo de manera más eficiente. Si en algún momento (en el futuro) se lograra ordenar los 100 elementos rápidamente, digamos en unos 2 segundos, bastaría con duplicar el tamaño de la lista para tener un problema tanto o más costoso computacionalmente que el anterior (y ahora son sólo 200 números enteros).

Otro ejemplo de este estilo, en algún sentido más sorprendente que el anterior, es el siguiente programa:

```
pot 0     = 1
pot (x+1) = (pot x) + (pot x)
```

que permite calcular, dado un valor x, el resultado de elevar 2 a la x. Si bien este programa no parece presentar ineficiencias importantes a simple vista, este programa, al igual que el anterior, es de complejidad *exponencial*, y por lo tanto tomaría un tiempo verdaderamente monstruoso en computar el resultado de valores relatimente pequeños (por ejemplo, calcular 2 a la 100).

Este ejemplo puede utilizarse para mostrar que con pequeñas modificaciones un algoritmo puede mejorarse **notoriamente**: Una simple factorización en la segunda ecuación transforma el programa anterior en:

```
pot 0     = 1
pot (x+1) = 2 * (pot x)
```

cuya complejidad es lineal y, por lo tanto, es mucho más eficiente que el original.

Citaremos, como último ejemplo en esta sección, el caso de la construcción de un programa de ajedrez que juegue "perfectamente". Hace ya varios años que se conoce un algoritmo que resuelve este problema (generación y exploración

del árbol de juego), pero la construcción del árbol de juego para el ajedrez es tan costosa que se estima tomaría varias veces la edad del universo terminar su construcción aún en la computadora actual mas poderosa (esto, sin hacer mención de la inimaginable cantidad de memoria necesaria para almacenar el árbol).

Ejemplos de este tipo sugieren al estudiante la fundamental importancia de la Complejidad algorítmica en nuestros días, y la necesidad de su consideración en la práctica cotidiana.

3 La Perplejidad para Resaltar la Importancia de la Teoría

En Ciencias de la Computación pueden utilizarse los recursos computacionales como una herramienta para la sorpresa y la perplejidad. Esto puede servir para fomentar en los alumnos el deseo de adquirir conocimientos teóricos, poco atractivos muchas veces debido a que generan una sensación de distancia respecto de la práctica.

Ejemplificaremos en esta sección con la temática *Lenguajes y Paradigmas de Programación*.

En asignaturas que abarcan este tema es común encontrarse con alumnos que consideran que para aprender un lenguaje basta con invertir muchas horas frente a la computadora, revisando manuales. Los alumnos suelen considerar que, para lograr el dominio de un lenguaje, es suficiente conocer una cantidad considerable de construcciones del mismo. Esto, obviamente, lleva a un falso "dominio" del lenguaje, debido a que el análisis de éste es muy superficial.

En este caso, ejemplos con resultados aparentemente incoherentes debido al desconocimiento de conceptos básicos que sustentan los lenguajes de programación y sus procesadores pueden utilizarse para sorprender al alumno, y así incentivarlo a estudiar las características del lenguaje más a fondo. Analicemos, por ejemplo, el siguente caso:

Ejemplo 3.1

El siguiente programa C calcula dos veces el mismo valor: la primera vez usando $t = c + a$ y $r = c + b$, y la segunda utilizando directamente $c + a$ y $c + b$ en lugar de t y r.

```
#include <stdio.h>

float a=11921., b=11800., c=1000000000, t, r;

main ()
{
 t=c+a;
 r=c+b;
 printf("%f   %f\n", (t-r)-120, ((c+a)-(c+b))-120);
}
```

Parece obvio que este programa debe imprimir dos valores iguales.

Sin embargo después de compilarlo con Turbo-C Versión 3, su ejecución muestra la siguiente salida:

```
8.0000000   1.0000000
```

Qué ha sucedido?

Todas las variables usadas son de tipo *float* por lo tanto serán representadas en notación flotante.

La notación flotante en base b con n dígitos de precisión representa a los números reales mediante pares $(mantisa, exponente)$, donde la mantisa es un racional que cumple $b - n \leq |\text{mantisa}| \leq b - 1$ o mantisa $= 0$. Esto equivale a decir que la mantisa es una fracción de la forma $0.m_1 m_2 ... m_n$, donde $m_1 \neq 0$ si $r \neq 0$. Cada real r se representa por un par (m, e) tal que

$$m \cdot be \cong r$$

Esta notación tiende a preservar el error relativo o porcentual, ya que la magnitud del error cometido varía proporcionalmente a la magnitud del número representado. Esta propiedad que la hace adecuada al calculo numérico, recibiendo también el nombre de notación científica desde mucho antes de la existencia de la computación. Sin embargo, la aritmética flotante no mantiene las propiedades de la aritmética real; por ejemplo, la suma flotante no es asociativa y además la resta de valores muy próximos puede aumentar significativamente el error relativo.

Esto explicaría que al realizar de formas distintas un mismo cálculo se obtengan valores distintos. Pero en el ejemplo se realizan aparentemente idénticas operaciones.

Analicemos ahora el problema. El compilador usado, como la mayoría de los actuales, usa una representación flotante que responde al estándar *IEEE* P 574; en ella, la base es 2 y la cantidad de dígitos -binarios- de precisión es 24. La precisión es similar a la que se obtendría si se usara base 10 y 8 dígitos de precisión. Si se siguen las operaciones realizadas por el programa, haciéndole imprimir valores intermedios se observa lo siguiente.

		Valores calculados por el programa	Valores Exactos
t	=	1000011**904.000000**	1000011921.000000
r	=	1000011**776.000000**	1000011800.000000
$t - r$	=	**128.000000**	121.000000
$(t - r) - 120$	=	**8.000000**	1.000000

Como la precisión del sistema es de entre 7 y 8 dígitos, el valor de los dígitos mostrados en negrita puede estar afectado por errores. Al restar t y r se produce un gran aumento del error relativo que afecta a todas la cifras del resultado; finalmente, al restar 128 y 120 se obtiene un valor que no tiene ninguna significación.

Esto sin embargo no explica que la segunda forma de calculo dé el valor correcto.

Analicemos lo que sucede.

El compilador realiza, en concordancia con la norma del *IEEE*, las operaciones en precisión extendida, usando una mantisa de 36 bits de precisión. El compilador utiliza para almacenar los valores temporales que genera también variables de precisión extendida. Entonces, al realizar los cálculos de la primera forma, los valores se computan con una precisión suficiente para obtener resultados exactos, pero al almacenarse en las variables t y r, que son de precisión simple, se obtienen los valores mostrados.

Al utilizarse la segunda forma todo el cálculo se realiza en precisión extendida, obteniéndose el resultado correcto.

Veamos ahora otro interesante ejemplo.

Ejemplo 3.2

Si a y b son variables enteras, el siguiente segmento de código realiza el intercambio de los valores de la variables a y b.

```
    { a=A y b=B }
swap:
      a:=  a + b   [1]
    { a=A+B y b=B }
      b:= a - b    [2]
    { a=A+B y b=A }
      a:= a - b
    { a=B y b=A }
```

Este procedimiento es por lo tanto equivalente al siguiente:

```
swap1:
      c:=a ; a:=b ; b:=c
```

Las implementaciones directas de ambos procedimientos en *Pascal* son

```
procedure swap(var a, b: integer)[3]
 begin
  a:= a + b
  b:= a - b
  a:= a - b
 end
```

```
procedure swap1(var a,b: integer)
 c:integer
 begin
  c:= a
  a:= b
  b:= a
 end
```

y obviamente con cualquier compilador las ejecuciones de los siguientes segmentos de procedimientos:

```
var a, b: integer
 begin
```

```
var a,b : integer
 begin
```

```
  a:= 1; b:= 2;
  swap(a,b)
  write(a,b)
 end
```

```
 a:= 1; b:= 2;
 swap1(a,b)
 write(a,b)
end
```

conducen a las salidas:

```
  1  2
```

```
2  1
```

Sin embargo, la ejecución de:

```
var a : integer
begin
 a:= 1;
 swap(a, a)   [4]
 write(a)
end
```

```
var a : integer
 begin
  a:= 1;
  swap1(a,a)
  write(a)
 end
```

conducen a las salidas:

```
  0  0
```

```
1   1
```

Lo sucedido se explica porque la invocación producida en [4] hace que en [3] ambos parámetros referencien la misma variable, creándose así una situación de sinonímia o *aliassing* entre las variables a y b, ya que el pasaje se realiza por referencia, que hace que el valor computado en [1] para a sea compartido por b, y por tanto que [2] ligue el valor 0 tanto a a como a b.

Otro manera útil de incentivar al alumno es mostrar al alumno como puede "extenderse" el poder de un lenguaje si uno conoce sus características a fondo (un ejemplo típico es el de creer que en *Pascal* las funciones no pueden retornar arreglos como resultados, cuando esto en realidad puede hacerse mediante un pequeño truco).

4 Conclusiones

Hemos mostrado varios ejemplos de la forma en la cual puede utilizarse la perplejidad en la enseñanza de la computación. Es necesario destacar que, siempre que sea posible, es recomendable que el alumno se enfrente con la computadora. El alumno debe "sentir" y padecer la ineficiencia de su programa, debe sentirse desconcertado frente a los resultados aparentemente erróneos generados por un programa aparentemente correcto; debe sentirse frustrado frente a los errores en tiempo de compilación que el compilador *Pascal* le reportará, al intentar retornar arreglos en la declaración de una función. Solo después de estas penurias el alumno se dará cuenta de la importancia del estudio de la teoría, y mas aún, de los beneficios que este estudio genera en la práctica. En una gran cantidad de casos, sólo de esta manera logra sorprenderse realmente al alumno. si uno se limita a realizar un comentario en una clase, el impacto logrado en el alumno es considerablemente menor.

Ejemplos sorprendentes han sido utilizados de forma excelente en algunos libros de texto de Computación. Un buen ejemplo de esto es [1].

Referencias

[1] S. Baase, *Computer Algorithms, Introduction to Design and Analysis*, Second Edition, Addison-Wesley Publishing Company, 1988.

[2] G. Chaitin, *On the Length of Programs for Computing Finite Binaries Sequences*, Journal of the ACM 13, pp 547-469, 1966.

[3] G. Chaitin, *The Limits of Mathematics*, Springer-Verlag, 1998.

[4] X. Caicedo, *La Paradoja de Berry revisitada, o la Indefinibilidad de Definibilidad y las Limitaciones de los Formalismos*, en Lecturas Matemáticas de la Sociedad Matemática de Bogotá, N. 1, 2 y 3, 1993.

[5] D. Knuth, *The Art of Computer Programming*, Addison Wesley, 1966.

[6] Mendelson, *Introduction to Mathematical Logic*, Van Nostrand, 1964.

[7] T. W. Pratt, *Programming Languages, Design and Implementation*, Prentice Hall, 1984.