MATEMÁTICA DISCRETA

APUNTES DE CÁTEDRA
AGOSTO DE 2008

THOMAS N. HIBBARD
Índice general

Capítulo 1. **INTRODUCCIÓN**  
1. Algoritmos 3

Capítulo 2. **LOS NÚMEROS NATURALES**  
1. Axiomas 9  
2. División 11  
3. Primos 13  
4. Máximo común divisor 15  
5. El Teorema Fundamental de la Aritmética 16  
6. Aritmética Modular 18  
7. Grupos 22  
8. Cuerpos 23  
9. Más sobre primalidad y factorización 27  
10. Criptografía 28

Capítulo 3. **TEORÍA DE GRAFOS**  
1. Grafos 33  
2. Caminos y conectividad 33  
3. Los puentes de Königsgburg y ciclos de Euler 36  
4. Estructuras de datos para grafos 38  
5. Camino mínimo 40  
6. Árboles 42  
7. Redes de Transporte 44

Capítulo 4. **AUTÓMATAS**  
1. Autómatas finitos y lenguajes regulares 53  
2. Máquinas de Turing 58

Capítulo 5. **GRAMÁTICAS**  
1. Lenguajes libres de contexto 63  
2. Autómatas de Pila 67

Capítulo 6. **FUNCIONES GENERADORAS Y ECUACIONES DE DIFERENCIAS**  
1. Introducción 73  
2. Funciones generatoras racionales 75  
3. Ecuaciones de diferencia 76  
4. Una sola ecuación de diferencia 78  
5. Sobre la aritmética sobre series infinitas 84

Capítulo 7. **PROBABILIDAD DISCRETA**  
1. Introducción 87  
2. Autómatas estocásticos 92  
3. Cadenas de Markov 95

Capítulo 8. **LA INCOMPLETITUD DE LA ARITMÉTICA**  
1. Introducción 107  
2. Demostraciones 108  
3. El plan general 110  
4. Cálculo de predicados de primer orden 110  
5. El Teorema de Gödel 114
CAPÍTULO 1

INTRODUCCIÓN

Si bien la matemática discreta no es nueva en el siglo 20 – la teoría de grafos fue fundada por Euler, y Pitágoras fue un pionero en la teoría de números – no fue muy respetada antes: se consideraba más bien una diversión, comparada con el trabajo serio del matemático en la matemática continua. Euclides tuvo que disfrazar su teoría de números como geometría.

Con el advenimiento de la computación electrónica, algunos problemas discretos empezaban a verse como urgentes. Los matemáticos puros seguían buscando sus generalizaciones: calcular algo concreto estaba por debajo de su dignidad, pero la gente cuyo trabajo sí era el de calcular algo concreto con las nuevas computadoras, en forma que salga bien confiable y en tiempo aceptable, se enfrentaba a nuevos problemas. Ahora los problemas de esta clase se conocen como problemas del diseño de algoritmos. Y siendo la computadora digital un aparato de naturaleza discreta, que analizado cuidadosamente se ve que trabaja sólo con números enteros, la matemática involucrada era discreta.

Es por eso que cada libro de matemática discreta tiene un capítulo dedicado a la noción de algoritmo.

No basta con algunas observaciones generales sobre esa noción, sino que, para que el alumno capte su verdadera significancia, es preciso que trabaje con algoritmos concretos, y además que los ponga en marcha en una computadora. Hay que seleccionar unos problemas que condenan a esto, y allí cada profesor va a tener su propio gusto. Y así también con respecto a los temas: casi siempre se elige la teoría de números y la teoría de grafos. Aquí hemos elegido también la teoría de lenguajes, incluyendo automatas finitos y gramáticas, funciones generadoras y, finalmente, la formalización de la noción de algoritmo, necesaria para el teorema de incompletitud de Gödel.

1. Algoritmos

No es que la matemática discreta no existía antes de la computadora, pero no existía el término. Euclides hacía matemática discreta, pero la tuvo que disfrazar como geometría para conseguir la atención de sus colegas. Leonardo Euler empezó la teoría de grafos en el siglo 18. John von Neumann empezó la teoría de los juegos al mismo tiempo que se estaban ideando las primeras computadoras electrónicas, cuando la división automática se hacía en segundos –en lugar de microsegundos– y estaba acompañada por una desconcertante serie de porrazos. Pero la matemática discreta no se consideraba parte de la corriente principal de la matemática, sino más bien una curiosa diversión.

Pero con la computadora, de golpe la inversión de una matriz de 50 por 50 no era trabajo de una semana sino de unos segundos, la simulación de una batalla se podía hacer en tiempo real, se podía modelar la economía de una nación por un año con bastante verosimilitud, y, en poco tiempo más, muchos problemas se abrieron a un estudio que antes parecía imposible.

Pero quizás la novedad principal que llegó con la computadora fue la realización concreta de algoritmos. Ya un algoritmo no era sólo una receta o una serie de instrucciones para hacer algo; residía dentro de la computadora en la forma de una serie de números, o si quiere, un solo número, porque la concatenación de una serie de números se puede leer como un solo número. Cambiando un solo dígito de ese número casi seguro iba a causar el fracaso del algoritmo. Se podía decir que los algoritmos habían pasado de la metamatemática a la matemática misma: habían llegado a ser objetos matemáticos.

Cabe mencionar que, en una de las muchas instancias en que la teoría se adelanta de la tecnología, en la década anterior, cuando Gödel necesitaba formalizar su requerimiento de la lógica que las demostraciones sean efectivamente computables, dió el primer intento de formalizar la noción de un algoritmo. Resultó después que no era completamente satisfactoria, aún que toda la lógica vista hasta este momento tienen demostraciones computables en ese sentido. Pero abrió por primera vez formalmente de qué era, formalmente, una función computable, contestado por Turing y Church con su famosa Tesis.

Es así que los algoritmos son fundamentales en la matemática discreta, y nuestra primera tarea es establecer una notación para describirlos.

Cada algoritmo está dirigido a calcular una función. Hay unos datos de entrada, los argumentos, y unos datos de salida (decimos “unos” pero pueden ser miles) que constituyen el valor de la función para los argumentos dados.
Aquí vamos a definir una función como una expresión, y vamos a ir definiendo el término expresión poco a poco.

Una expresión puede ser un número, y por ahora por número vamos a entender número natural. Por conveniencia incluimos 0 como número natural, dejando a un lado el debate de si 0 es natural en el sentido del lenguaje cotidiano. Cuando queremos enteros, ponemos un signo con un número natural, y cuando queremos un número real ponemos un par ordenado \((a, b)\) de enteros para significar \(a \times 10^b\) (o, a veces, \(a \times B^b\) donde \(B\), la base, es un número natural mayor que 1.)

Una expresión puede consistir en la invocación de una función ya conocida.

Ejemplo: \(5 + 7, 13 \times 8, 15 - 7\) (pero no \(7 - 15\) mientras trabajamos sólo con naturales), \(7^2\). Aquí las funciones son \(+, \times, -\) y exponenciación respectivamente. Funciones fundamentales como éstas, a menudo se expresan con un sintaxis especial, infija en el caso de \(+, \times y -\).

Una expresión puede dar un valor lógico, “verdadero” o “falso”. Por ejemplo \(5 < 7\) es verdadero y \(7 < 5\) es falso.

Una noción clave es la de la expresión condicional. Lo vamos a escribir en la forma

\[ \text{si} (\text{expresión 1}) \text{ expresión 2 } \text{sino} \text{ expresión 3} \]

donde expresión 1 es una expresión que da un valor lógico. El valor de tal expresión es el valor de la expresión 2 si el valor de la expresión 1 es verdadero, y es el de la expresión 3 si aquel es falso. Nótese que se evalúa la expresión 1 primero y después se evalúa la de las otras dos que corresponde. La otra opción, la de evaluar ambas de expresiones 2 y 3, y elegir el que corresponde al valor de la expresión 1, nos daría problema con una expresión como \(\text{si}(y = 0)\) 0 sino \(x/y\), que daría “indefinido” siempre cuando \(y = 0\) por que \(x/0\) es indefinido.

Con esto podemos definir funciones muy útiles, como por ejemplo la siguiente:

\[ \max(x, y) = \text{si} (x > y) x \text{sino} y \]

Note aquí que los nombres de las funciones que definimos pueden tener varias letras, en este caso “max”. Ahora teniendo este \(\max\) como conocido, podemos definir

\[ \max(x, y, z) = \max(x, \max(y, z)) \]

lo cual ilustra otra cosa: que podemos dar el mismo nombre a dos funciones distintas mientras tienen distinto número de argumentos. Y entonces podemos definir

\[ \max(x, y, z, w) = \max(x, \max(y, z, w)) \]

o si preferimos,

\[ \max(x, y, z, w) = \max(\max(x, y), \max(z, w)) \]

Cuando escribimos así la definición de una función, digamos \(f\), es permitido invocar a \(f\) en la misma expresión que usamos para definir \(f\). Por ejemplo, aunque vamos a suponer que todo el mundo sabe qué quiere decir \(x^y\) cuando \(x\) y \(y\) son naturales no ambos 0, podemos eliminar toda duda escribiendo

\[ x^n = \text{si} (n = 0) 1 \text{sino} x \times x^{n-1} \]

(esto definiría \(0^0 = 1\) pero incluimos en la definición que no vale cuando los dos argumentos son 0.) Pero este es mal algoritmo. Para \(2^{5000}\) por ejemplo hace 4999 multiplicaciones, pero se puede hacer como \((2^{10})^{10}\) con sólo 22.

**Ejercicio 1.** (1) Realice \(x^n\) en Maple como procedimiento pot(x,n), definiendo pot := proc(x,n) if n = 0 then 1 else x^t.. fi end;

**Ejercicio 2.** (1) Explique cómo calcular \(2^{5000}\) con menos de 50 multiplicaciones. Sugerencia: \(2^{5000} = (2^{1000})^5\), que multiplica 1005 veces en lugar de 5. Siga aplicando esta idea. Realice este cálculo como una expresión en Maple, empleando el procedimiento pot del ejercicico 1.

**Ejercicio 3.** (1) Explique por qué el mínimo número de multiplicaciones para engender \(x^n\) es igual al mínimo número de adiciones para engender \(n\), empezando con 1.

**Ejercicio 4.** (3) Explique cómo calcular \(x^n\) con no más que \(2(\lceil \log_2 n \rceil - 1)\) multiplicaciones. (No pedimos que lo explique en la notación algorítmica compacta porque hay que engendrar una sucesión y no hemos hablado todavía de ese tipo de datos – explique en buen castellano no más.)

**Ejercicio 5.** (3) Demuestre, examinando valores chicos de \(n\), que el algoritmo del ejercicio 4 es óptimo para algunos \(n\) pero no lo es para todos. (Alternativamente, si su algoritmo es óptimo para todo \(n\), mándelo a publicar inmediatamente).

Cuando la expresión que define a \(f\) invoca a \(f\) se dice que es una definición recursiva. Las definiciones recursivas son fundamentales; sin ellas no se puede definir nada de interés. Debe notarse que no hablamos de una sucesión de instrucciones, donde instrucción 10 puede ser “vuelva a la instrucción 5”, pero si estuviéramos hablando así, el volver a la instrucción 5 es una invocación recursiva de la función que se está evaluando cuando estamos en la instrucción 5.
También puede ocurrir que en la definición de $f$ invocamos una $g$ que invoca a $f$. También es una función recursiva. O puede ser que $f$ invoque a $g$ que invoca a $h$ que invoca a $f$, etcétera. Claro que no hay garantía de que tales definiciones siempre vayan a tener sentido, por ejemplo $f(x) = g(x) + 1$ y $g(x) = f(x) - 1$. ¿Qué idiota trataría de evaluar eso? ¡Ese idiota es la computadora!

Examinemos esta función interesante:

$$\text{fibonacci}(n) = \begin{cases} \text{si} (n < 2) & n \\ \text{sino} & \text{fibonacci}(0, 1, n-2) \end{cases}$$

Los primeros términos de esta sucesión de Fibonacci son 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. Ud. puede continuar la sucesión, obteniendo el próximo término simplemente sumando los dos últimos que tiene.

**Ejercicio 6.** Realice la función fibonacci como procedimiento en Maple, escribiendo

$$\text{fibonacci} := \text{proc}(n) \text{ if } n < 2 \text{ then } n \text{ else } \text{fibonacci}(n-1) + \text{fibonacci}(n-2) \text{ end proc;}$$

Pruebe invocando varios fibonacci(n) con n<20. Ahora haga trace(fibonacci) e invoque fibonacci(8) (no más que 8 porque hay mucha salida). La palabra de inglés trace quiere decir rastrear. Cause el rastreo de toda la actividad de la función fibonacci cada vez que se invoca.

Pero note que esa idiota de computadora va a seguir esta definición a la letra, como hemos visto en el ejercicio 6, en lugar de simplemente agregar otro término de la sucesión sumando los dos últimos. Por ejemplo, para $f(4)$ va a calcular primero $f(3) = f(2) + f(1) = f(1) + f(0) + f(1)$, y luego $f(2) = f(1) + f(0)$ otra vez, y así hace $f(1) + f(0)$ + $f(1) + f(0)$, cinco invocaciones de $f$. Sea $t(n)$ el tiempo que demora la computadora en calcular fibonacci(n), y supongamos que $t(1) = t(2) = 1$ microsegundo. Entonces para $n > 1$, $t(n) \geq t(n-1) + t(n-2)$ y resulta que $t(n) \geq \text{fibonacci}(n)$. Entonces para calcular fibonacci(50) la computadora demoraría por lo menos 3 horas y media. Pero podríamos hacerlo a mano, aún demorando un promedio de 30 segundos en cada suma, en 25 minutos, y yo acabo de calcular con Maple fibonacci(50) = 12586269025 en una pequeña fracción de segundo.

Lo que pasa es que la definición, aunque funcionalmente correcta, es ineficiente. Es un mal algoritmo. Mejor es éste:

$$\text{fibonacci}(n) = \begin{cases} \text{si} (n < 2) & n \\ \text{sino} & \text{fibonacci}(0, 1, n-2) \end{cases}$$

Para fibonacci(x, y, n) := si(n = 0) x + y sino fibonacci(x, y, y + n, n-1)

**Ejercicio 7. (4) Demuestra que el último algoritmo es correcto demostrando lo siguiente:**

Si $f(n) = \text{si} (n = 0) a \text{ sino} \text{si} (n = 1) b \text{ sino} f(n - 1) + f(n - 2)$

y si $g(x, y, n) = \text{si} (n = 0) x + y \text{ sino} g(y, x + y, n-1)$

entonces $g(a, b, n) = f(n + 2)$.

Sugerencia: Demuestre por inducción en $n$ que $g(a, b, n) = g(f(k), f(k+1), n-k)$ para todo $k \leq n$.

**Ejercicio 8.** Realice este nuevo algoritmo para fibonacci en Maple. Haga trace(fibonacci) nuevamente (al redefinir “fibonacci” se cancela su trace) e invoque fibonacci(8). Haga untrace(fibonacci); luego invoque fibonacci(50).

El próximo ejemplo será el de calcular una raíz cuadrada. Algoritmo:

$$\text{raíz}(a) = \text{raíz}(a, 3)$$

$$\text{raíz}(a, x) = \text{si}(x = a/x) x \text{ sino } \text{raíz}(a, \frac{x + a}{2}).$$

Medio malo este algoritmo porque nunca para, excepto en casos especiales como $a = 9$. Pero pruebe viendo los sucesivos valores de $x$. Por ejemplo, para $a = 100000000$ en $166668.1667$, $83337.08332$, $41674.54139$, $20849.26843$, $10448.61588$, $5272.161168$, $2730.918352$, $1548.547770$, $1097.157047$, $1004.301796$, $1000.009213$. Claramente $x$ tiende al límite 1000, la raíz cuadrada de 1000000. Bueno, cuando se trata de la raíz cuadrada no podemos pretender más que llegar cerca. Entonces para tener un algoritmo que termina, terminamos cuando $x$ y $a/x$ están suficientemente cerca; es decir, elegimos un $\varepsilon$ y terminamos cuando $|x - a/x| < \varepsilon$.

$$\text{raíz}(a, \varepsilon) = \text{raíz}(a, 3, \varepsilon)$$

$$\text{raíz}(a, x, \varepsilon) = \text{si}(|x - \frac{a}{x}| < \varepsilon) x \text{ sino } \text{raíz}(a, \frac{x + a}{2}, \varepsilon).$$

Ahora cuando hacemos esto a mano no vamos a calcular $a/x$ dos veces, sino vamos a guardarlo para copiar la segunda vez que lo usamos. Pero si damos esto a algún idiota (por ejemplo, la computadora) para calcular,
capaz que no haga eso. Le tenemos que decir explícitamente que haga eso, escribiendo la subfunción $raíz(a, x, \epsilon)$ así:

$$raíz(a, x, \epsilon) = c := \frac{a}{x}; \text{sí} \ (|x - c| < \epsilon) \ x \ \text{sino} \ raíz\left(a, \frac{x + c}{2}, \epsilon\right)$$

“$c := \frac{a}{x}$” se llama asignación. Lo podemos considerar una abreviatura, “para el resto del cálculo de esta función $c$ significa el valor que tiene $a/x$ en este momento.” También se lo puede considerar una memoria, pero una local a la función; la asignación carece de validez cuando termine la función.

**Ejercicio 9.** Haga en Maple:

$a := 2; \ x := 3; \ (no \ se \ olvide \ del \ punto \ decimal, \ que \ indica \ que \ se \ quiere \ aritmética \ real – si no todos \ los \ valores \ se \ dan \ como \ racionales). \ Luego \ invoque \ repetidamente \ la \ siguiente \ línea \ (para \ invocar \ de \ nuevo \ haga ↑ hasta \ que \ el \ cursor \ llegue \ de \ nuevo \ a \ la \ línea \ y \ haga \ enter):$

$c := a/x; \ x := (x + c)/2;$

terminando cuando $|c - x| < 0.000001.$

**Ejercicio 10.** Haga en Maple la función $raíz(a, \epsilon)$, usando la segunda versión de la subfunción $raíz(a, x, \epsilon)$. Se la hace subfunción poniendo su definición ($raiz1 := \text{proc}(a, x, \epsilon)$ etc., llamándole raiz1 porque Maple, lamentablemente, no permite sobrecargar nombres) dentro de la definición de la función de $raíz(a, \epsilon)$. Invocar la función para probar. Luego haga trace(raiz) e invocarla de nuevo.

Las asignaciones casi abren la puerta para la programación imperativa. Más adelante vamos a abrir esa puerta completamente. Pero por ahora seguiremos con el punto de vista puramente funcional.

Para dos números naturales $x, y$, con $y \neq 0$, el cociente de $x$ dividido $y$ es el número de veces que se puede restar $y$ de $x$ y tener un saldo natural:

$$\text{cociente}(x, y) = \text{sí}(x < y) \ \text{sino} \ 1 + \text{cociente}(x - y, y).$$

Otro algoritmo correcto pero malo (desde el punto de vista de la eficiencia). Y aquí otro igualmente correcto y malo:

$$\text{resto}(x, y) = \text{sí}(x < y) \ x \ \text{sino} \ \text{resto}(x - y, y).$$

Vamos a ver mejores algoritmos para estas funciones, pero sirven para ilustrar un par de cosas más de la notación algorítmica. Notando que $\text{cociente}$ y $\text{resto}$ hacen la misma prueba, $x < y$, y hacen llamadas recursivas con los mismos argumentos, vemos que convendría combinar los dos algoritmos en uno, que llamaremos $cr$, $cr(x, y)$ nos va a entregar un par ordenado $(c, r)$ de números naturales tales que $x = cy + r$ con $r < y$, es decir que $c y r$ son el cociente y resto respectivamente de $x$ dividido $y$. Las funciones no están restringidas a entregar sólo números o valores lógicos, sino pueden entregar estructuras, desde pares ordenados a sucesiones, matrices, conjuntos, etc.

Una definición de $cr(x, y)$ para $x, y$ naturales y $y \neq 0$:

$$cr(x, y) =$$

. $\text{sí}(x < y) \ (0, x)$

. $\text{sino}$

. $(a, b) := cr(x - y, y)$

. $(a + 1, b)$

La cadena “$(a, b) := (c, r)$” (siendo $(c, r) = cr(x - y, y)$) significa las dos asignaciones $a := c$ y $b := r$.

**Ejercicio 11.** (2) Realice $cr(x, y)$ en Maple. Pruebelo con ejemplos con $y < 6000$ primero. Haga trace(cr) y pruebe con $y < 50$. Haga untrace(cr). Haga $cr(40000, 7)$. ¿Cuál es el número máximo $x$ tal que se puede hacer $cr(x, 7)$?

**Ejercicio 12.** (3) Realice en Maple el siguiente algoritmo:

$$cr2(a, b) = \text{sí}(a < b) \ (0, a)$$

. $\text{sino}$

. $(c, r) := cr2(a, 2b)$

. $\text{sí}(r < b) \ (2c, r) \ \text{sino} \ (2c + 1, r - b)$

¿Qué calcula $cr2$?

**Ejercicio 13.** (5) ¿Porqué? Sugerencia: Pruebe primero que siempre termina. Pruebe correctitud por inducción sobre el número de llamadas recursivas.

Volvemos al algoritmo $raíz$, convirtiéndolo en un algoritmo de la matemática discreta: calculamos la parte entera de $\sqrt{a}$, que escribimos $[\sqrt{a}]$, empleando sólo números enteros (ahora el argumento $a$, por supuesto, tiene que ser natural):

$$raíz2(a) := raíz2(a, 3)$$

**donde**

. $raíz2(a, x) =$

. $\text{sí}(|x - c| \leq 1) \ \text{min}(x, c) \ \text{sino} \ \text{raíz2}(a, \text{cociente}(x + c, 2))$
Por ejemplo, para el cálculo de $\sqrt{3}$ hasta 3 décimales, los pares ordenados que se encuentran después de la asignación $c := \left[ \frac{a}{2} \right]$ son $(3, 100000)$, $(500001, 5)$, $(250003, 11)$, $(125007, 23)$, $(62515, 47)$, $(31281, 95)$, $(15688, 191)$, $(7939, 377)$, $(4158, 721)$, $(2439, 1230)$, $(1834, 1635)$, $(1734, 1730)$, $(1732, 1732)$. Para el caso de raiz2(30000000, 3), son $(3, 1000000000)$, $(500000001, 5)$, $(250000003, 11)$, $(12500007, 23)$, $(6250015, 47)$, $(3125031, 95)$, $(1562563, 191)$, $(781377, 383)$, $(390880, 767)$, $(195823, 1531)$, $(98677, 3040)$, $(50858, 5898)$, $(28378, 10571)$, $(19474, 15405)$, $(17439, 17202)$, $(17320, 17321)$. Note que $1732^2 \leq 300000 < 1733^2$ y $17320^2 \leq 300000000 < 17321^2$.

**Ejercicio 14.** Haga en Maple:

- a := 2000000; x := 3; (preparándonos para $\sqrt{2}$ hasta 3 decimales). Repetir (ver ej. 9):
  - c := cociente(a, x); x := cociente(x + a, 2); hasta $|x - c| \leq 1$.

**Ejercicio 15.**

(a) (3) Demuestre que cuando raiz2 contesta, contesta correctamente (con $\lfloor \sqrt{a} \rfloor$).

(b) (5) Demuestre que raiz2 siempre termina. Sugerencia: Demuestre que en cada llamada recursiva $|x - c|$ se reduce en por lo menos 1.