Tema: Algoritmos probabilistas

asignatura: Métodos Algorítmicos de Resolución de Problemas
profesores: Ricardo Peña y Narciso Martí

Curso 2016–2017

1. Introducción

⋆ Cuando un algoritmo debe realizar una decisión, a veces es preferible tomarla al azar que emplear tiempo en determinar la decisión óptima.

⋆ Un **algoritmo probabilista** (AP) toma decisiones de este tipo; por eso, si se ejecuta dos veces seguidas puede comportarse de modo distinto cada vez, bien empleando más o menos tiempo, o incluso dando respuestas distintas.

⋆ A veces se permite que un AP falle o dé una respuesta incorrecta en alguna ejecución, siempre que esto suceda con baja probabilidad. Si eso sucede, basta con iniciarlo de nuevo sobre el mismo ejemplar, o bien se ejecuta repetidas veces sobre el ejemplar para aumentar la confianza en la respuesta.

⋆ A un algoritmo determinista no se le permiten esas "libertades" porque, si falla o da una respuesta errónea sobre un ejemplar, lo hará todas las veces que se ejecute.

⋆ Muchos AP dan **respuestas probabilistas**, es decir, no necesariamente correctas. Sin embargo, la probabilidad del error puede reducirse tanto como se quiera (incluso por debajo de la de un error de hardware) empleando más tiempo.

⋆ Para ciertos problemas no se conocen algoritmos eficientes (por ejemplo, determinar si un número de 1000 cifras es primo); en cambio un AP puede dar una respuesta eficiente si se tolera una cierta probabilidad de error.

2. Clasificación de algoritmos probabilistas

⋆ Consideramos tres tipos de AP:

a) Numéricos: Dan respuesta numérica con un intervalo de error, por ejemplo, 59 ± 3. Si se les da más tiempo, pueden disminuir el **intervalo de confianza**.
Ejemplo: integral por muestreo.

b) Monte Carlo: Dan una respuesta exacta pero que puede ser errónea con una cierta probabilidad. Normalmente no es posible determinar si la respuesta dada es correcta o no, pero si se les deja más tiempo pueden disminuir la probabilidad de ser errónea.
Ejemplo: determinación de primalidad.

c) Las Vegas: Nunca dan una respuesta incorrecta pero pueden no dar ninguna. Internamente toman decisiones aleatorias, que pueden o no conducirles a la solución. Si fallan, basta con ejecutarlos de nuevo hasta que alguna vez triunfen.
Ejemplo: n reinas en un tablero $n \times n$ de forma que no se den jaque entre sí.

⋆ Ejemplo: ¿En qué año se descubrió América?

a) Numérico: $1500 \pm 300, 1600 \pm 150, 1490 \pm 20$.

b) Monte Carlo: 1492 (error 10^{-1}), 1492 (error 10^{-2}), 350 AC (error 10^{-3}), 1492 (error 10^{-4}).

c) Las Vegas: 1492, 1492, perdón, 1492, 1492, 1492.

3. Tiempo esperado

⋆ Para saber la eficiencia de un AP se introduce una noción nueva. El **tiempo esperado** en resolver un ejemplar concreto es el tiempo promedio empleado en resolver dicho ejemplar, tras ensayar ese ejemplar infinitas veces. Por ejemplo, el AP de Las Vegas para el problema de las n reinas tiene un tiempo esperado en el caso peor en $\Theta(n)$.

⋆ Nótense las diferencias con el tiempo promedio:

- media del tiempo empleado en cada ejemplar de un tamaño dado n,
- necesita una hipótesis sobre la frecuencia/probabilidad con la que aparece cada ejemplar en el contexto en que se ejecute el algoritmo.

4. Números aleatorios

⋆ Supondremos la existencia de un generador de números aleatorios que puede ser llamado cada vez con coste en $\Theta(1)$. Consideramos tres variantes:

- $x = \textbf{uniform}(a, b)$, con $a, b \in \mathbb{R}^+, a < b$, devuelve x tal que $a \leq x < b$. La distribución de x es uniforme en $[a, b)$ y cada x es independiente.
- $k = \textbf{uniform}(i..j)$, con $i, j \in \mathbb{Z}$ devuelve k tal que $i \leq k \leq j$, con las mismas características.
- $m = \textbf{uniform}(0..1)$ se utiliza como equivalente de lanzar una moneda.

⋆ En la práctica los generadores necesitan una **semilla** y generan una secuencia **pseudoaleatoria** indistinguible de una realmente aleatoria.

- Si se da la misma semilla, la secuencia se repite.
- Para no repetir, se suele tomar la semilla de un dispositivo hardware tal como el reloj.
- Suelen constar de dos funciones:
  $f : X \to X$ genera la secuencia $\{x_i = f^i(s) \mid i \geq 0\}$, donde s es la semilla;
  $g : X \to Y$ genera $\{g(x_i) \mid i \geq 0\}$, donde X es el dominio interno e Y es el dominio del usuario.

⋆ Ejemplo: si p, q son primos tales que $p \equiv 3$ (mód 4) y $q \equiv 3$ (mód 4), sean $n = pq$ y $X = \{0, 1, \ldots, n-1\}$. Entonces $f(x) = x^2$ (mód n) tiene un periodo del orden de n y la secuencia $\{f^i(s)\}$ es indistinguible de una secuencia realmente aleatoria.

5. Algoritmos probabilistas numéricos

5.1. El experimento del Conde de Buffon

⋆ En el siglo XVIII Buffon demostró que si se arrojan de modo aleatorio n agujas al suelo (o n veces una aguja) tales que

- la longitud de cada aguja es l,
- el suelo está formado por tablas de anchura $2l$ y
- la separación entre tablas tiene anchura 0,

entonces la probabilidad de que una aguja caiga entre dos tablas es igual a $1/\pi$.

⋆ Si se arrojan n agujas, hay que esperar que caigan $m \leq n$ entre dos tablas, por lo que $m/n \approx 1/\pi$, o equivalentemente $\pi \approx n/m$. Es decir, el experimento puede servir para aproximar π.

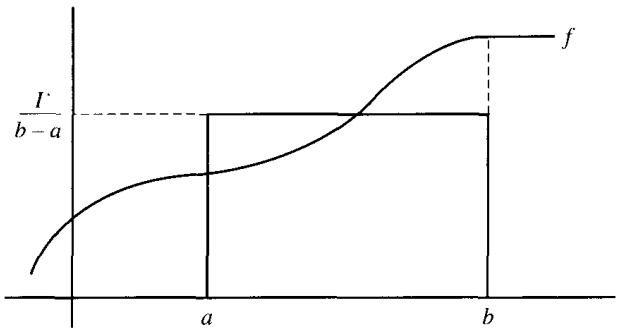
⋆ La distribución de la variable aleatoria $X = \Sigma_{i=1}^{n} X_i/n$, donde $X_i = 1(0)$ si la aguja i cae (no cae) entre dos tablas, es **normal** si n es suficientemente grande. Si escogemos un error ε, es posible demostrar que son necesarias $n > 1440/\varepsilon^2$ agujas para que la probabilidad de que $|m/n - 1/\pi| < \varepsilon$ sea mayor del 99 %.

En general, el número de agujas necesario crece con el inverso del cuadrado de la precisión requerida, de manera que un dígito decimal de precisión adicional necesita 100 veces más agujas.

5.2. Integración numérica

⋆ Es el algoritmo probabilista numérico más conocido. Se le conoce como "método de Monte Carlo" aunque nosotros usaremos el término "Monte Carlo" en un sentido completamente distinto.

⋆ Sean $f : \mathbb{R} \to \mathbb{R}^+$ continua y $a \leq b$. El área entre f y el eje x entre los puntos a y b viene dada por $I = \int_a^b f(x)dx$. Por tanto, la altura media de la curva f entre a y b es igual a $I/(b-a)$.



⋆ El AP de integración toma muestras aleatorias de f en el intervalo $[a, b)$ y hace la media.

fun integracionMC(f, n, a, b) **retorna** *int*
  $s = 0$
  para i **en** $1, n$ **hacer**
    $x =$ **uniform**(a, b)
    $s = s + f(x)$
  retorna $(b - a) * (s/n)$

⋆ Haciendo un análisis similar al del experimento de Buffon, se demuestra que el error esperado ε es inversamente proporcional a $\sqrt{n}$, es decir, el número de puntos crece cuadráticamente con el inverso del error: $\varepsilon \in O(1/\sqrt{n}) \Rightarrow n \in O(1/\varepsilon^2)$.

⋆ Un algoritmo determinista para este problema escogería puntos espaciados regularmente entre a y b y, en general, necesitaría menos iteraciones que el AP para el mismo grado de precisión.

⋆ Siempre es posible, sin embargo, encontrar funciones f que "engañarían" al algoritmo determinista; por ejemplo, la función seno(x) es periódica y podrían hacerse coincidir sus ceros con los puntos regularmente espaciados. En cambio, es muy improbable "engañar" al AP.

⋆ En la práctica el AP se utiliza para integrales dobles o triples en las que el algoritmo determinista necesita n^2 o n^3 puntos para ser preciso.

6. Algoritmos probabilistas de Monte Carlo

⋆ Ocasionalmente pueden dar una respuesta errónea, pero dan la respuesta correcta con una probabilidad tan alta como se desee.

⋆ Normalmente no es posible saber cuándo la respuesta es errónea, salvo utilizando un algoritmo determinista, que a veces no existe o tiene un coste inasumible.

⋆ El acierto/error es independiente del ejemplar considerado. Para un mismo ejemplar, la respuesta puede ser correcta o errónea en diferentes ejecuciones.

⋆ Sea $0 < p < 1$. Decimos que un AP de Monte Carlo es **p-correcto** si devuelve una respuesta correcta con probabilidad $\geq p$, independientemente del ejemplar considerado.

⋆ Usualmente repitiendo el algoritmo n veces, si la respuesta es consistentemente la misma, la probabilidad de que sea incorrecta será $(1-p)^n$, mucho menor que p. Esta estrategia se llama **amplificar la ventaja estocástica**.

6.1. Verificación del producto de matrices

⋆ Sean A, B, C matrices $n \times n$. Sospechamos que $C = AB$. ¿Cómo podemos verificarlo? Obviamente calculando AB y comparando con C. El algoritmo determinista que lo hace tiene un coste en $\Theta(n^3)$ (si se usa el algoritmo de divide y vencerás de Strassen, en $\Theta(n^{2,8})$). Vamos a intentar mejorarlo con un AP de Monte Carlo, si admitimos una pequeña probabilidad de error.

⋆ Supongamos que $C \neq AB$, lo que implica que $D = AB - C \neq 0$. Sea $S \subseteq \{1, 2, \ldots, n\}$ un conjunto de filas de D escogido aleatoriamente lanzando una moneda para cada fila entre 1 y n. Denotamos por $\Sigma_S(D)$ el vector resultante de sumar las filas de D indicadas por S.

⋆ Supongamos que $i \in \{1, 2, \ldots, n\}$ es una fila de D distinta de 0. La probabilidad de que $i \in S$ es $1/2$. Sea $S' \subseteq \{1, 2, \ldots, n\}$ igual que S salvo que $i \in S' \Leftrightarrow i \notin S$; claramente $\Sigma_S(D)$ y $\Sigma_{S'}(D)$ no son 0 a la vez. Por tanto, la probabilidad de que $\Sigma_S(D) \neq 0$ es al menos $1/2$ ("al menos" porque puede haber más filas distintas de 0 que no sean i). La idea del AP de Monte Carlo es escoger aleatoriamente S, calcular $\Sigma_S(D)$ y comprobar si es 0:

- si $\Sigma_S(D) \neq 0$, entonces $C \neq AB$ con seguridad;
- si $\Sigma_S(D) = 0$, entonces $C = AB$ con probabilidad de error $\leq 1/2$.

⋆ Repitiendo el experimento k veces, si todas las veces sale $\Sigma_S(D) = 0$, entonces $C = AB$ con probabilidad de error $\leq 1/2^k$ (porque si D fuera distinta de 0 el algoritmo se habría equivocado consistentemente k veces).

⋆ Lo único que resta es calcular $\Sigma_S(D)$ de forma eficiente. Sea $(x_1, \ldots, x_n)$, con $x_i \in \{0, 1\}$, el vector característico de S, donde los x_i se escogen con una moneda aleatoria. Entonces

- $\Sigma_S(D) = XD = XAB - XC$,
- $\Sigma_S(D) = 0 \Leftrightarrow XAB = XC$.

⋆ Obviamente XC puede calcularse con un coste en $\Theta(n^2)$ (producto de vector $1 \times n$ por matriz $n \times n$). XAB también si se hace en el orden $(XA)B$ por la misma razón dos veces.

```
fun zeroSigmaD(A, B, C, n) retorna bool
    para j en 1, n hacer
        x_j = uniform(0..1)
    si (XA)B = XC          {Θ(n²)}
        entonces retorna true
        si no retorna false
```

⋆ El algoritmo de Monte Carlo es entonces el siguiente:

```
fun prodMatMC(A, B, C, n, k) retorna bool
    para i en 1, k hacer          {Θ(kn²)}
        si ¬zeroSigmaD(A, B, C, n) entonces retorna false
    retorna true
```

⋆ Según el análisis precedente, el algoritmo prodMatMC es $(1 - 2^{-k})$-correcto. Si, por ejemplo, $k = 10$, el algoritmo es $99,9\,\%$-correcto. Se puede aumentar esta probabilidad aumentando k.

⋆ Si $\varepsilon = 1/2^k$ es la probabilidad de error, podemos sustituir el parámetro k por ε y calcular $k = \lceil \log_2 1/\varepsilon \rceil$. El coste entonces se expresaría como $\Theta(n^2 \log_2 1/\varepsilon)$ donde se aprecia claramente la dependencia del tamaño y de la probabilidad requerida de error.

⋆ El número exacto de productos escalares hechos por zeroSigmaD es $3n^2$. Si, por ejemplo, $\varepsilon = 10^{-6}$, tendríamos $k = 20$ y un coste $60n^2$, que solo compensa frente al algoritmo determinista de coste $\Theta(n^3)$ cuando $n > 60$.

6.2. Test de primalidad

⋆ Consiste en decidir si un número natural impar n con varios cientos de dígitos decimales es primo o no. Si es compuesto, no se espera que proporcione sus factores primos, ya que el problema de factorizar un número se considera hoy por hoy computacionalmente duro.

⋆ El interés práctico es enorme ya que numerosos protocolos criptográficos se basan en claves públicas o privadas que se construyen multiplicando dos números primos grandes. Precisamente las claves se consideran irrompibles debido a lo costoso de factorizar.

⋆ Hasta 1990 no se conocían algoritmos deterministas polinomiales para este problema. El ideado en esa fecha tiene complejidad en $\Theta(m^6)$ siendo m el número de cifras de n, es decir, complejidad en $\Theta(\log^6 n)$, y utiliza algunas conjeturas no demostradas (aunque se suponen ciertas) de teoría de números; se llama *test de la curva elíptica*. Hay otro posterior, **AKS**, de 2002, de complejidad en $\Theta(\log^{12} n)$ que no depende de ninguna conjetura. En general, estos algoritmos son difíciles de implementar y propensos a errores de programación.

⋆ El test de Gary L. Miller y Michael O. Rabin (Premio Turing 1976), de 1975, es un AP de Monte Carlo y tiene un coste en $O(\log^3 n \log 1/\varepsilon)$, siendo ε la probabilidad de error. Cuando el algoritmo dice "compuesto", la respuesta es correcta; si dice "primo", hay una probabilidad ε de error.

⋆ En el siglo XVII el matemático Pierre de Fermat enunció sin demostrarlo (fue demostrado en el siglo XVIII por Leonhard Euler) el siguiente teorema, conocido como el "pequeño teorema de Fermat":

$$\text{primo}(n) \Rightarrow \forall a : 1 \leq a \leq n-1 : a^{n-1} \text{ mód } n = 1.$$

⋆ Ejemplo: $n = 7$ es primo, por lo que $\forall a : 1 \leq a \leq 6 : a^6 \text{ mód } 7 = 1$.

a	a^6	a^6 mód 7
1	1	$0 \cdot 7 + 1$
2	64	$9 \cdot 7 + 1$
3	729	$104 \cdot 7 + 1$
4	4096	$585 \cdot 7 + 1$
5	15625	$2232 \cdot 7 + 1$
6	46656	$6665 \cdot 7 + 1$

⋆ Si consideramos el contrarrecíproco, tenemos

$$\exists a : 1 \leq a \leq n-1 : a^{n-1} \text{ mód } n \neq 1 \Rightarrow \neg\text{primo}(n).$$

⋆ La exponenciación módulo n se puede hacer eficientemente (con un número de productos módulo n en $\Theta(\log n)$). Esto sugiere el siguiente AP de Monte Carlo:

fun Fermat(n) **retorna** *bool*
  $a =$ **uniform**$(1..n-1)$
  si a^{n-1} mód $n \neq 1$
    entonces retorna false
    si no retorna true

⋆ La respuesta "false" es correcta, pero ¿qué podemos decir de la respuesta "true"? En general, no es cierto el recíproco del teorema de Fermat:

$$\forall a : 1 \leq a \leq n-1 : a^{n-1} \bmod n = 1 \not\Rightarrow \text{primo}(n).$$

Los números de Carmichael son números compuestos que cumplen la hipótesis anterior (añadiendo a coprimo con n); el más pequeño es $561 = 3 \cdot 11 \cdot 17$.

Para n compuesto puede haber algunos valores de a que pasen el test de la conclusión:

- $4^{14} \bmod 15 = 1$, pero 15 no es primo,
- $1^{n-1} \bmod n = 1$, para todo $n \geq 2$,
- $(n-1)^{n-1} \bmod n = 1$, para todo $n \geq 3$.

⋆ Si n es compuesto y $\exists a : 2 \leq a \leq n-2 : a^{n-1} \bmod n = 1$, decimos que a es un **falso testigo de primalidad** para n; por ejemplo, 4 es un falso testigo de primalidad para 15.

⋆ Si modificamos la función Fermat para escoger a entre 2 y $n-2$, fallará (devolverá true) cuando n es compuesto y a es un falso testigo de primalidad para n.

⋆ Los falsos testigos son escasos. Si tomamos los 332 números compuestos impares menores que 1000,

- 5 no tienen falsos testigos;
- $> 50\,\%$ tienen a lo sumo 2;
- $< 16\,\%$ tienen a lo sumo 15, pero el 561 tiene el $100\,\%$ de los coprimos con 561 (el $57\,\%$ de los $2 \leq a \leq n-2$).
- En total hay 4490 falsos testigos entre un conjunto de 172878 posibles candidatos $(2,6\,\%)$.

⋆ Sin embargo, hay algunos números que tienen una gran proporción de falsos testigos; por ejemplo, $n = 561$ tiene 318 falsos testigos (un $57\,\%$ de los candidatos). Eso hace el método inútil porque la probabilidad de error depende del ejemplar.

⋆ G. L. Miller y M. O. Rabin hicieron el siguiente test modificado:

- Sea $n > 4$ impar tal que $n - 1 = 2^s \cdot t$ y t impar (nótese que $s > 0$ porque $n-1$ es par).
- Sea $B(n) = \{a \mid 2 \leq a \leq n-2 \wedge (a^t \bmod n = 1 \vee \exists i : 0 \leq i < s : a^{2^i t} \bmod n = n-1)\}$.
- Implementamos la función Btest(a, n) que devuelve "true" si y solo si $a \in B(n)$. Es fácil y eficiente; veremos que su coste está en $O(\log^3 n)$.
- Se usa el siguiente teorema:

$$\text{primo}(n) \Leftrightarrow \forall a : 2 \leq a \leq n-2 : a \in B(n).$$

```
fun Btest(a, n) retorna bool
    s = 0 ; t = n − 1
    repetir s = s + 1 ; t = t/2 hasta t mód 2 = 1
    x = a^t mód n
    si x = 1 ∨ x = n − 1
        entonces retorna true
        si no para i en 1, s − 1 hacer
                x = x^2 mód n
                si x = n − 1 entonces retorna true
            retorna false
```

Nótese que las dos primeras líneas solo dependen de n y no de a por lo que solo hacen falta una vez.

⋆ Si $n > 4$ es impar y compuesto y $a \in B(n)$, decimos que a es un **falso testigo fuerte de primalidad** para n. Todo falso testigo fuerte es un falso testigo en el sentido de Fermat, pero no a la inversa. El número de falsos testigos fuertes es mucho menor que el de falsos testigos (densidad menor que 1 %). Mejor aún, su número está acotado para todo n impar.

Teorema. Sea $n > 4$ un número impar arbitrario.

- $\text{primo}(n) \Rightarrow B(n) = \{a \mid 2 \leq a \leq n-2\}$.
- $\neg\text{primo}(n) \Rightarrow |B(n)| \leq \frac{1}{4}(n-9)$.

Nótese que la primera parte es de hecho una equivalencia porque en ese caso $|B(n)| = n - 3 > \frac{1}{4}(n-9)$.

⋆ Por tanto, si $\text{primo}(n)$, $\text{Btest}(a, n) = \text{true}$ (o sea, con probabilidad 1) para todo a con $2 \leq a \leq n-2$. Si, en cambio, $\neg\text{primo}(n)$, es decir, n es compuesto, $\text{Btest}(a, n) = \text{true}$ con probabilidad $< 1/4$ para a escogido con $2 \leq a \leq n-2$, y $\text{Btest}(a, n) = \text{false}$ con probabilidad $\geq 3/4$.

⋆ El siguiente AP de Monte Carlo es, por tanto, 3/4-correcto.

fun MillerRabin(n) **retorna** *bool* $\quad \{n > 4 \text{ impar}\}$
  $a =$ **uniform**$(2..n-2)$
  retorna Btest(a, n) $\quad \{\text{true con error} < 1/4\}$

⋆ Si lo repetimos k veces podemos reducir la probabilidad a 4^{-k}.

fun repetirMillerRabin(n, k) **retorna** *bool* $\quad \{n > 4 \text{ impar}\}$
  para i **en** $1, k$ **hacer**
    si $\neg$MillerRabin(n) **entonces retorna** false
  retorna true

⋆ Estudio del coste.

- $\log_2 n > \log_2(n-1) = \log_2(2^s t) = s + \log_2 t$, por lo que tanto s como $\log t$ están dominados por $\log n$.
- El cálculo a^t mód n dentro de $\text{Btest}(a, n)$ solo es necesario hacerlo una vez ya que, para cada i, $a^{2^i t}$ mód n se calcula como

$$(a^t \text{ mód } n)^{2^i} \text{ mód } n = (\ldots((a^t \text{ mód } n)^2 \text{ mód } n)^2 \text{ mód } n \ldots)^2 \text{ mód } n$$

  donde elevar al cuadrado se hace $0 \leq i < s$ veces.
- El número de cuadrados módulo n en $\text{Btest}(a, n)$ está en $O(\log n)$ y cada uno, considerándolo como una multiplicación de dos números de $\log n$ cifras, tiene coste en $O(\log^2 n)$.
- De la misma forma, el coste de calcular a^t mód n se basa en hacer $\log t$ (dominado por $\log n$) multiplicaciones, cada una de coste en $O(\log^2 n)$.
- Así pues, el coste total de $\text{Btest}(a, n)$, que es el mismo que el de MillerRabin(n), está en $O(\log^3 n)$.
- Si $2^{2k} \approx 1/\varepsilon$, entonces $k = \frac{1}{2}\lceil \log 1/\varepsilon \rceil$, por lo que el coste de repetirMillerRabin(n, ε) (que recibe el error máximo y calcula el k correspondiente) está en $O(\log^3 n \log 1/\varepsilon)$.

⋆ ¿Cuántos primos hay?

Queremos generar un primo de 1000 dígitos. Un posible algoritmo iría probando números impares aleatorios en ese intervalo y usando el test de Miller-Rabin, por ejemplo con $k = 5$ (probabilidad de error $4^{-5} = 2^{-10}$).

fun randomprime **retorna** *int*
  repetir $n = 2 \cdot$ **uniform**$(10^{999}/2..10^{1000}/2 - 1) + 1$
    hasta repetirMillerRabin$(n, 5)$
  retorna n

¿Qué probabilidad hay de que n sea compuesto? Aparentemente $\frac{1}{4^5} = \frac{1}{2^{10}} \approx \frac{1}{10^3}$, pero no es cierto porque repetirMillerRabin se llama muchas veces sobre números compuestos antes de dar por azar con un primo. Muchas de esas veces el test falla y devuelve "falso" (es decir, compuesto con seguridad) pero las probabilidades de error se van acumulando.

La cuestión es saber cuántos compuestos hay que comprobar en promedio antes de dar con un primo.

El **teorema del número primo** asegura que, dado n, el número de primos menores que n es aproximadamente $n/\ln n$. Por ejemplo, obtenemos un $14\,\%$ si $n \leq 1000$, un $7,2\,\%$ si $n \leq 10^6$, y un $4,8\,\%$ si $n \leq 10^9$.

Particularizando en números impares de 1000 dígitos, obtenemos que hay un primo por cada 1000 números, es decir, randomprime ensaya 1000 compuestos antes de dar con un primo.

Para un n dado pueden ocurrir tres casos:

(1) n es primo, repetirMillerRabin$(n, 5)$ = true y randomprime devuelve n.

(2) n es compuesto, repetirMillerRabin$(n, 5)$ = false y randomprime vuelve a iterar.

(3) n es compuesto, repetirMillerRabin$(n, 5)$ = true y randomprime devuelve incorrectamente n.

(1) y (3) suceden ambos con probabilidad $1/10^3$ cada vez que se itera, mientras que (2) sucede 998 veces de cada 1000. Por tanto, el bucle terminará con igual probabilidad en (1) o (3). Cabe esperar entonces que, para números de 1000 cifras, este algoritmo devuelva tantos primos correctos como incorrectos.

6.3. Amplificación de la ventaja estocástica

⋆ Los AP de Monte Carlo vistos hasta ahora se denominan **sesgados** porque una de las dos respuestas es correcta con seguridad y la otra errónea con probabilidad p. Repitiendo el algoritmo, si se obtiene consistentemente k veces la respuesta probabilista, se reduce la probabilidad de error a p^k.

⋆ Hay otros algoritmos de Monte Carlo que son **no-sesgados** y p-correctos. Significa que ambas respuestas son correctas con probabilidad $\geq p$. Se puede obtener amplificación repitiendo el algoritmo siempre que $p > 1/2$. Si $p = 1/2$, es equivalente a tirar una moneda y no se obtiene ventaja al repetir.

⋆ Supongamos un AP de Monte Carlo MC 3/4-correcto, no-sesgado, y consideremos el siguiente algoritmo, que llama a MC tres veces y devuelve la respuesta mayoritaria:

fun MC3(x) **retorna** *algo*
  $uno = \text{MC}(x)$
  $dos = \text{MC}(x)$
  $tres = \text{MC}(x)$
  si $uno = dos \vee uno = tres$
    entonces retorna uno {incluye los 3 iguales}
    si no retorna dos $\{dos = tres\}$

⋆ Sea C la respuesta correcta y F la falsa. Veamos qué devuelve MC3 en función de los resultados obtenidos en las tres llamadas:

uno	*dos*	*tres*	MC3	prob. suceso
C	C	C	C	27/64
C	C	F	C	9/64
C	F	C	C	9/64
F	C	C	C	9/64
F	F	C	F	3/64
F	C	F	F	3/64
C	F	F	F	3/64
F	F	F	F	1/64

Así pues, MC3 devuelve F con una probabilidad 10/64, mientras que devuelve C con una probabilidad $54/64 = 84,4\,\%$.

⋆ En este caso, la repetición amplifica la ventaja de $3/4 = 0,75$ a $0,844$. Si MC se repitiera 5 veces, la ventaja aumentaría a $0,896$.

7. Algoritmos probabilistas de Las Vegas

⋆ Utilizan decisiones aleatorias para encontrar más rápidamente la solución. Nunca devuelven una respuesta errónea.

⋆ Dos tipos:

a) Siempre devuelven una respuesta, aunque las decisiones tomadas sean desafortunadas; en ese caso, simplemente tardan más.

b) Si las decisiones tomadas son desafortunadas, **pueden fallar**.

⋆ Ejemplo de tipo a): quicksort.

```
proc quicksort(v, a, b)
    si a < b entonces
        particion(v, a, b, p)
        quicksort(v, a, p − 1)
        quicksort(v, p + 1, b)
```

- La elección del pivote en un subvector $v[a..b]$, con $a < b$, es simplemente $p = v[\textbf{uniform}(a..b)]$.
- Eso hace que el **tiempo esperado** para cualquier ejemplar de tamaño n sea $\Theta(n \log n)$.
- La elección aleatoria no previene que, ocasionalmente, el algoritmo se ejecute en su caso peor, con coste $\Theta(n^2)$. Lo que hace es **romper el vínculo** entre el caso peor y el ejemplar. El mismo ejemplar puede exhibir comportamientos benignos y patológicos en diferentes ejecuciones.
- El comportamiento promedio de quicksort no mejora por hacerlo probabilista. El efecto neto es que el AP "roba" eficiencia de los casos mejores, haciéndolos más lentos, y "se la da" a los ejemplares peores, haciéndolos más rápidos. Se suele referir a este comportamiento como "efecto Robin Hood".
- Los AP de Las Vegas de tipo a) son ventajosos en casos como este, en los que el coste peor es mucho peor que el promedio y este no difiere mucho del caso mejor.

⋆ Algoritmos de tipo b):

- Son capaces de reconocer el fallo.
- El fallo no está ligado al ejemplar. Basta ejecutarlo otra vez para que el fallo no se produzca (con gran probabilidad).

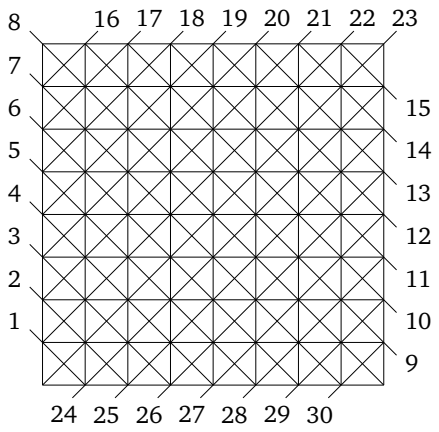
- La razón de usar un AP de Las Vegas en este caso es que en promedio encuentran la solución antes que el algoritmo determinista.
- El esquema general, si x es la entrada e y la salida, es el siguiente:
  fun repetirLV(x) **retorna** y
  repetir $LV(x, y, exito)$
  hasta $exito$
  retorna y
- Si $p(x)$ es la probabilidad de que $LV(x, y, exito)$ tenga éxito, el **número esperado** de iteraciones del bucle **repetir** es $1/p(x)$.
- Sea $e(x)$ el tiempo esperado en caso de éxito y $f(x)$ el tiempo esperado en caso de fallo. El tiempo esperado global es
$$t(x) = p(x)e(x) + (1 - p(x))(f(x) + t(x))$$
porque el tiempo de cada ejecución es independiente de las anteriores.
- Despejamos para obtener
$$t(x) = e(x) + \frac{1 - p(x)}{p(x)} f(x)$$
- La fracción $\frac{1-p(x)}{p(x)}$ representa el número de iteraciones esperado antes de tener éxito; por ejemplo, para $p(x) = 1/3$ hay que ejecutar 3 veces, 2 de ellas falla.

7.1. El problema de las 8 reinas

⋆ El algoritmo determinista de vuelta atrás explora efectivamente 2057 nodos (sin contar los podados) y encuentra la primera solución tras examinar 114, lo que no es demasiado malo. Empeora con más reinas.

⋆ El AP de Las Vegas coloca vorazmente las 8 reinas eligiendo aleatoriamente su posición dentro de la fila. Cada reina se coloca en alguna de las posiciones no amenazadas por las ya colocadas. Nunca se recoloca una reina ya colocada. Si la nueva reina no tiene ubicación posible, el algoritmo falla. Se representa la solución con un vector $sol[1..8]$ de posiciones 1..8, de forma que $sol[i]$ indica la columna que ocupa la reina en la fila i.

⋆ Para ver si una columna o diagonal están amenazadas, mantendremos dos conjuntos implementados como vectores de booleanos. $col[1..8]$ es un vector de booleanos de forma que $col[i] = \text{true}$ indica que la columna i está ocupada. El siguiente dibujo indica cómo podemos numerar todas las diagonales del tablero para $n = 8$. Las diagonales descendentes $\searrow$ se numeran de 1 a 15 y las diagonales ascendentes $\nearrow$ de 16 a 30 (las filas se numeran de 1 a 8 de arriba abajo y las columnas también de 1 a 8 de izquierda a derecha). Así tenemos un vector $diag[1..30]$ con el mismo significado que col.



En el caso general de un tablero de tamaño $n \times n$, las diagonales descendentes $\searrow$ se numeran de 1 a $2n-1$, y las diagonales ascendentes $\nearrow$ de $2n$ a $4n-2$. La reina en la posición $\langle i, j \rangle$ ocupa la diagonal descendente $j - i + n$ y la diagonal ascendente $i + j + 2n - 2$. Con esta información basta consultar tres posiciones en los dos vectores de columnas y diagonales ocupadas para ver si es posible colocar cada reina sin amenazar a las ya colocadas.

⋆ El bucle principal $k = 0..7$ supone que $sol[1..k]$ es **prometedor**, es decir, que las k reinas colocadas no se amenazan entre sí. En la iteración $k+1$ trata de colocar la reina $k+1$.

- investiga el número nb de columnas no amenazadas;
- si $nb = 0$, todas están amenazadas, por lo que falla;
- en caso contrario, elige una al azar entre las no amenazadas en el vector $ok[1..nb]$.

```
proc reinasLV(out sol[1..8], out exito)
    var ok[1..8], col[1..8], diag[1..30]
    col, diag = ∅
    para k en 0, 7 hacer
        {calcula columnas no amenazadas}
        nb = 0
        para j en 1, 8 hacer
            si j ∉ col ∧ j − k + 7 ∉ diag ∧ k + j + 15 ∉ diag
                entonces nb = nb + 1 ; ok[nb] = j
        si nb = 0 entonces retorna exito = false
        j = ok[uniform(1..nb)]
        col = col ∪ {j} ; diag = diag ∪ {j − k + 7} ∪ {k + j + 15}
        sol[k + 1] = j
    retorna exito = true
```

⋆ Análisis del tiempo esperado.

- Sea e el número de nodos explorados en caso de éxito. Claramente $e = 9$ contando la raíz del árbol $k = 0$ como nodo.
- Sea f el número de nodos explorados en caso de fallo. Cálculos experimentales dan $f = 6,971$.
- Igualmente dan $p(x) = 0,1293$ para la probabilidad de éxito (es decir, una vez de cada 8).
- Aplicando la fórmula: $t(x) = e(x) + \frac{1-p(x)}{p(x)} f(x) \approx 55,93$ nodos explorados esperados.
- Es menos de la mitad de los 114 de la solución determinista.

⋆ Se puede mejorar la solución empleando el AP de Las Vegas para las primeras reinas y la vuelta atrás para las restantes, sin descolocar las primeras. Estudios experimentales indican que la combinación 3 (Las Vegas) + 5 (Vuelta atrás) es superior a cualquier otra para $n = 8$. Para $n > 8$ las ventajas del AP de Las Vegas aumentan con n; para $n = 39$ la combinación 29 + 10 da lugar a 500 nodos esperados, mientras que el algoritmo determinista encuentra la primera solución tras explorar 10^{10} nodos.

7.2. La selección del k-ésimo menor elemento de un vector

⋆ Supongamos un algoritmo particion2$(v, a, b, p, \textbf{out}\ i, \textbf{out}\ j)$ al que se le da el pivote p y debe devolver el vector reorganizado de tal forma que se divide en tres sectores:

- los elementos en $v[a..i-1]$ son $< p$,
- los elementos en $v[i..j]$ son $= p$,
- los elementos en $v[j+a..b]$ son $> p$.

Claramente puede hacerse con un coste en $\Theta(b - a + 1)$.

⋆ Un algoritmo recursivo determinista para seleccionar el k-ésimo menor elemento de un vector $v[1..n]$ es el siguiente:

```
{ llamada inicial: seleccion(v, 1, n, k, kesimo) }
proc seleccion(v, a, b, k, out kesimo)        {a ≤ k ≤ b}
    si a = b entonces kesimo = v[a]
    si no
        particion2(v, a, b, v[a], i, j)
        casos
            k < i → seleccion(v, a, i − 1, k, kesimo)
            i ≤ k ≤ j → kesimo = v[k]
            k > j → seleccion(v, j + 1, b, k, kesimo)
```

⋆ Puede probarse que el coste en el caso promedio está en $\Theta(n)$, pero en el caso peor está en $\Theta(n^2)$ (pues responde a la recurrencia $t(n) = t(n-1) + c_2 n$ para $n > 1$ y $t(1) = c_1$).

⋆ El AP de Las Vegas (tipo a) puede conseguir un **coste esperado** en $\Theta(n)$ sustituyendo $v[a]$ en la llamada a particion2 por $v[\mathbf{uniform}(a..b)]$. Este coste es independiente del ejemplar considerado.

Lecturas complementarias

Estas notas están basadas en el capítulo 10 de [1], donde hay más explicaciones y ejemplos con los que el lector puede completar lo resumido aquí.

Referencias

[1] Gilles Brassard & Paul Bratley. *Fundamentos de algoritmia*. Prentice Hall, 1997.