



Algoritmos probabilistas

❖ Introducción	2
❖ Clasificación de los algoritmos probabilistas	9
❖ Algoritmos numéricos	13
– La aguja de Buffon	14
– Integración probabilista	16
❖ Algoritmos de Monte Carlo	21
– Verificación de un producto matricial	23
– Comprobación de primalidad	33
❖ Algoritmos de Las Vegas	50
– Ordenación probabilista	61
– Factorización de enteros	65

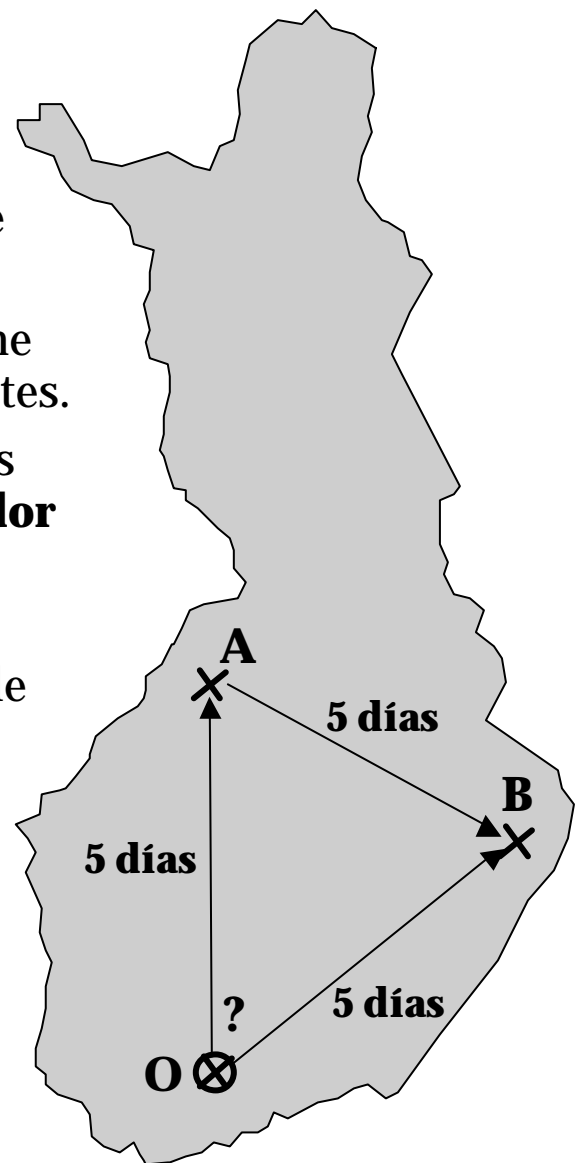


Algoritmos probabilistas: Introducción

❖ Una historia sobre un tesoro, un dragón, un computador, un elfo y un doblón.

- En A o B hay un **tesoro** de x lingotes de oro pero no sé si está en A o B.
- Un **dragón** visita cada noche el tesoro llevándose y lingotes.
- Sé que si permanezco 4 días más en O con mi **computador** resolveré el misterio.
- Un **elfo** me ofrece un trato: Me da la solución ahora si le pago el equivalente a la cantidad que se llevaría el dragón en 3 noches.

¿Qué debo hacer?





Algoritmos probabilistas: Introducción

- Si me quedo 4 días más en O hasta resolver el misterio, podré llegar al tesoro en 9 días, y obtener $x-9$ lingotes.
- Si acepto el trato con el elfo, llego al tesoro en 5 días, encuentro allí $x-5$ lingotes de los cuales debo pagar 3 al elfo, y obtengo $x-8$ lingotes.

→ Es mejor aceptar el trato pero...

... ¡hay una solución mejor!

¿Cuál?



Algoritmos probabilistas: Introducción

- ¡Usar el **doblón** que me queda en el bolsillo!
 - Lo lanzo al aire para decidir a qué lugar voy primero (A o B).
 - ♦ Si acierto a ir en primer lugar al sitio adecuado, obtengo $x-5y$ lingotes.
 - ♦ Si no acierto, voy al otro sitio después y me conformo con $x-10y$ lingotes.
- El beneficio esperado medio es $x-7'5y$.

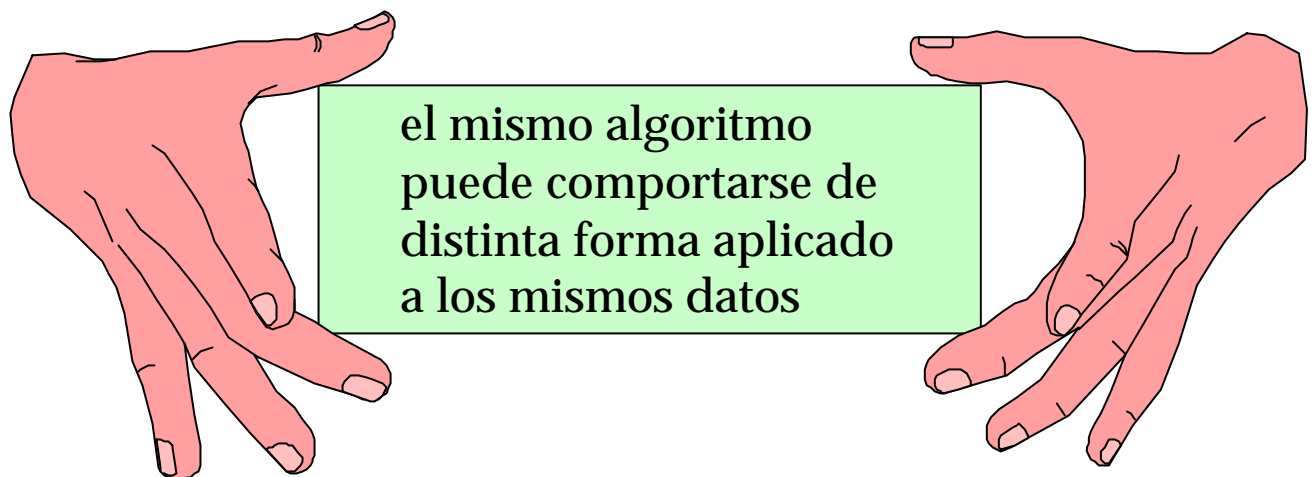


Algoritmos probabilistas: Introducción

❖ ¿Qué hemos aprendido?

- En algunos algoritmos en los que aparece una decisión, es preferible a veces elegir aleatoriamente antes que perder tiempo calculando qué alternativa es la mejor.
- Esto ocurre si el tiempo requerido para determinar la elección óptima es demasiado frente al promedio obtenido tomando la decisión al azar.

❖ Característica fundamental de un algoritmo probabilista:





Algoritmos probabilistas: Introducción

- ❖ Más diferencias entre los algoritmos deterministas y probabilistas:
 - A un algoritmo determinista **nunca** se le permite que no termine: hacer una división por 0, entrar en un bucle infinito, etc.
 - A un algoritmo probabilista se le puede permitir siempre que eso ocurra con una **probabilidad muy pequeña** para datos cualesquiera.
 - ♦ Si ocurre, se aborta el algoritmo y se repite su ejecución con los mismos datos.
 - Si existe más de una solución para unos datos dados, un algoritmo determinista siempre encuentra la **misma solución** (a no ser que se programe para encontrar varias o todas).
 - Un algoritmo probabilista puede encontrar **soluciones diferentes** ejecutándose varias veces con los mismos datos.



Algoritmos probabilistas: Introducción

❖ Más diferencias:

- A un algoritmo determinista no se le permite que calcule una solución incorrecta para ningún dato.
- Un algoritmo probabilista puede equivocarse siempre que esto ocurra con una probabilidad pequeña para cada dato de entrada.
 - ♦ Repitiendo la ejecución un número suficiente de veces para el mismo dato, puede aumentarse tanto como se quiera el grado de confianza en obtener la solución correcta.
- El análisis de la eficiencia de un algoritmo determinista es, a veces, difícil.
- El análisis de los algoritmos probabilistas es, muy a menudo, muy difícil.



Algoritmos probabilistas: Introducción

❖ Un comentario sobre “el azar” y “la incertidumbre”:

- A un algoritmo probabilista se le puede permitir calcular una solución equivocada, con una probabilidad pequeña.
- Un algoritmo determinista que tarde mucho tiempo en obtener la solución puede sufrir errores provocados por fallos del hardware y obtener una solución equivocada.

→ Es decir, el algoritmo determinista tampoco garantiza siempre la certeza de la solución y además es más lento.

- Más aún:

Hay problemas para los que no se conoce ningún algoritmo (determinista ni probabilista) que dé la solución con certeza y en un tiempo razonable (por ejemplo, la duración de la vida del programador, o de la vida del universo...):

Es mejor un algoritmo probabilista rápido que dé la solución correcta con una cierta probabilidad de error.

Ejemplo: decidir si un n^o de 1000 cifras es primo.



Algoritmos probabilistas: Clasificación

Algoritmos probabilistas

Algoritmos que no garantizan la corrección de la solución

Algoritmos numéricos:

- dan una solución aproximada
- dan un intervalo de confianza (“con probab. del 90% la respuesta es 33 ± 3 ”)
- a mayor tiempo de ejecución, mejor es la aproximación

Algoritmos de Monte Carlo:

- dan la respuesta exacta con una alta probabilidad
- en algunas ocasiones dan una respuesta incorrecta
- no se puede saber si la respuesta es la correcta
- se reduce la probabilidad de error alargando la ejecución

Algoritmos que nunca dan una solución incorrecta

Algoritmos de Las Vegas:

- toman decisiones al azar
- si no encuentran la solución correcta lo admiten
- es posible volver a intentarlo con los mismos datos hasta obtener la solución correcta



Algoritmos probabilistas: Clasificación

❖ Ejemplo de comportamiento de los distintos tipos ante un mismo problema

“¿Cuándo descubrió América Cristobal Colón?”

– Algoritmo numérico ejecutado cinco veces:

- ◆ “Entre 1490 y 1500.”
- ◆ “Entre 1485 y 1495.”
- ◆ “Entre 1491 y 1501.”
- ◆ “Entre 1480 y 1490.”
- ◆ “Entre 1489 y 1499.”

Aparentemente, la probabilidad de dar un intervalo erróneo es del 20% (1 de cada 5).

Dando más tiempo a la ejecución se podría reducir esa probabilidad o reducir la anchura del intervalo (a menos de 11 años).



Algoritmos probabilistas: Clasificación

“¿Cuándo descubrió América Cristobal Colón?”

– Algoritmo de Monte Carlo ejecutado diez veces:

1492, 1492, 1492, 1491, 1492, 1492, 357 A.C., 1492,
1492, 1492.

De nuevo un 20% de error.

Ese porcentaje puede reducirse dando más tiempo para la ejecución.

Las respuestas incorrectas pueden ser próximas a la correcta o completamente desviadas.



Algoritmos probabilistas: Clasificación

“¿Cuándo descubrió América Cristobal Colón?”

– Algoritmo de Las Vegas ejecutado diez veces:

1492, 1492, ¡Perdón!, 1492, 1492, 1492, 1492, 1492,
¡Perdón!, 1492.

El algoritmo nunca da una respuesta incorrecta.

El algoritmo falla con una cierta probabilidad
(20% en este caso).



Algoritmos numéricos: Introducción

- ❖ Primeros en aparecer
 - SGM, clave “Monte Carlo”
- ❖ Un ejemplo ya conocido:
 - Simulación de un sistema de espera (cola)
 - ◆ Estimar el tiempo medio de espera en el sistema.
 - ◆ En muchos casos la solución exacta no es posible.
- ❖ La solución obtenida es siempre aproximada pero su precisión esperada mejora aumentando el tiempo de ejecución.
- ❖ Normalmente, el error es inversamente proporcional a la raíz cuadrada del esfuerzo invertido en el cálculo
 - Se necesita cien veces más de trabajo para obtener una cifra más de precisión.

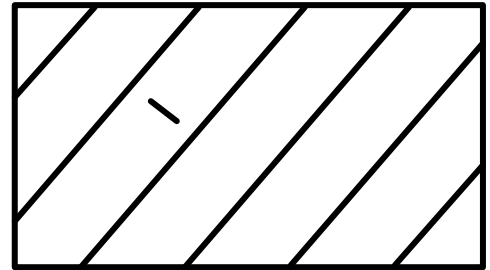


Algoritmos numéricos: *La aguja de Buffon*

G.L. Leclerc, Conde de Buffon:
“Essai d’arithmétique morale”, 1777.

❖ Teorema de Buffon:

Si se tira una aguja de longitud λ a un suelo hecho con tiras de madera de anchura ω ($\omega \geq \lambda$), la probabilidad de que la aguja toque más de una tira de madera es $p = 2\lambda/\omega$.



❖ Aplicación:

- Si $\lambda = \omega/2$, entonces $p = 1/2$.
- Si se tira la aguja un número de veces n suficientemente grande y se cuenta el número k de veces que la aguja toca más de una tira de madera, se puede estimar el valor de p :

$$k \approx n/p \Rightarrow p \approx n/k$$

Es (probablemente) el primer algoritmo probabilista de la historia.



Algoritmos numéricos: *La aguja de Buffon*

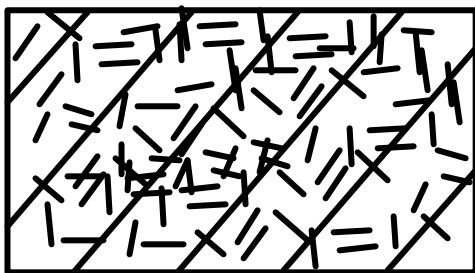
❖ Pregunta natural: ¿Es útil este método?

– ¿Cómo de rápida es la convergencia?

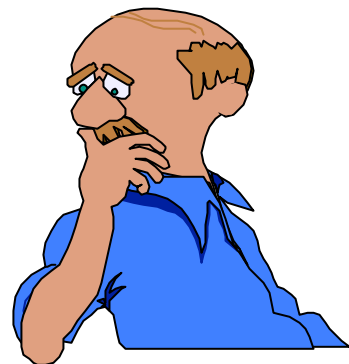
Es decir, ¿cuántas veces hay que tirar la aguja?

Es muy lenta, es decir el método no sirve [BB96]:

$n=1500000$ para obtener un valor de $p \pm 0'01$ con probabilidad 0'9.



$p \approx 3 \pm 0'5$, con probabilidad 0'85

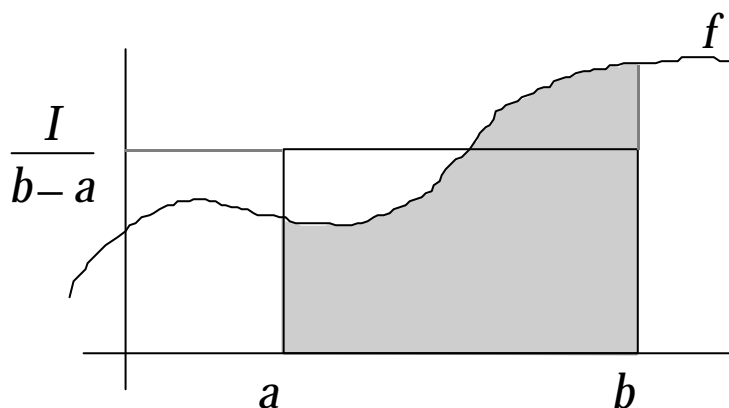


Algoritmos numéricos: Integración probabilista

❖ Problema:

Calcular:

$$I = \int_a^b f(x) \, dx, \text{ donde } f: \mathbb{R} \rightarrow \mathbb{R}^+ \text{ es continua y } a \leq b$$



$I/(b-a)$ es la altura media de f entre a y b .



Algoritmos numéricos: Integración probabilista

```
función int_prob(f:función; n:entero;  
                a,b:real) devuelve real  
{Algoritmo probabilista que estima la integral  
de f entre a y b generando n valores aleatorios  
 $x_i$  en  $[a,b)$ , haciendo la media de los  $f(x_i)$  y  
multiplicando el resultado por  $(b-a)$ .  
Se utiliza la función uniforme(u,v) que genera  
un número pseudo-aleatorio uniformemente  
distribuido en  $[u,v)$ .}  
variables suma,x:real; i:entero  
principio  
    suma:=0.0;  
    para i:=1 hasta n hacer  
        x:=uniforme(a,b);  
        suma:=suma+f(x)  
    fpara;  
    devuelve (b-a)*(suma/n)  
fin
```



Algoritmos numéricos: Integración probabilista

❖ Análisis de la convergencia:

- Puede verse [BB96] que la varianza del estimador calculado por la función anterior es inversamente proporcional al número n de muestras generadas y que la distribución del estimador es aproximadamente *normal*, cuando n es grande.
- Por tanto, el error esperado es inversamente proporcional a \sqrt{n} .
 - ♦ 100 veces más de trabajo para obtener una cifra más de precisión



Algoritmos numéricos: Integración probabilista

❖ La versión determinista:

- Es similar pero estima la altura media a partir de puntos equidistantes.

```
función int_det(f:función; n:entero;  
                a,b:real) devuelve real  
variables suma,x:real; i:entero  
principio  
    suma:=0.0; delta:=(b-a)/n; x:=a+delta/2;  
    para i:=1 hasta n hacer  
        suma:=suma+f(x);  
        x:=x+delta  
    fpara;  
    devuelve suma*delta  
fin
```



Algoritmos numéricos: *Integración probabilista*

- En general, la versión determinista es más eficiente (menos iteraciones para obtener precisión similar).
- Pero, para todo algoritmo determinista de integración puede construirse una función que “lo vuelve loco”
(no así para la versión probabilista).

Por ejemplo, para $f(x) = \sin^2(100!px)$ toda llamada a `int_det(f, n, 0, 1)` con $1 \leq n \leq 100$ devuelve 0, aunque el valor exacto es 0.5.

- Otra ventaja: cálculo de integrales múltiples.
 - ♦ Algoritmos deterministas: para mantener la precisión, el coste crece exponencialmente con la dimensión del espacio.
 - ♦ En la práctica, se usan algoritmos probabilistas para dimensión 4 o mayor.
 - ♦ Existen técnicas híbridas (parcialmente sistemáticas y parcialmente probabilistas): *integración cuasi-probabilista*.



Algoritmos de Monte Carlo: Introducción

- ❖ Hay problemas para los que no se conocen soluciones deterministas ni probabilistas que den siempre una solución correcta (ni siquiera una solución aproximada).



- ❖ Algoritmo de Monte Carlo:

- A veces da una solución incorrecta.
- Con una alta probabilidad encuentra una solución correcta **sea cual sea la entrada**.
(NOTA: Esto es mejor que decir que el algoritmo funciona bien la mayoría de las veces).

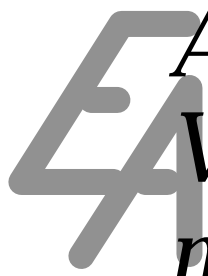


Algoritmos de Monte Carlo: Introducción

- ❖ Sea p un número real tal que $0 < p < 1$.
Un algoritmo de Monte Carlo es **p -correcto** si:

Devuelve una solución correcta con probabilidad mayor o igual que p , cualesquiera que sean los datos de entrada.

A veces, p dependerá del tamaño de la entrada, pero nunca de los datos de la entrada en sí.



Algoritmos de Monte Carlo: Verificación de un producto matricial

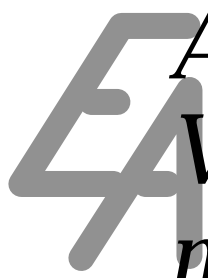
❖ Problema:

- Dadas tres matrices $n \times n$, A , B y C , se trata de verificar si $C = AB$.

❖ Solución trivial:

- Multiplicar A por B con:
 - ♦ El algoritmo directo: coste $\Theta(n^3)$.
 - ♦ El algoritmo de Strassen (*Divide y vencerás*, pág. 46): se puede llegar hasta $\Omega(n^{2,376})$.

❖ ¿Puede hacerse mejor?



Algoritmos de Monte Carlo: Verificación de un producto matricial

R. Freivalds: “Fast probabilistic algorithms”,
*Proceedings of the 8th Symposium on the Mathematical
Foundations of Computer Science*, Lecture Notes in
Computer Science, vol. 74, Springer-Verlag, 1979.

❖ Solución de Monte Carlo:

Suponer que $D = AB - C$

Sea i el índice de una fila no nula de D : $D_i \neq \vec{0}$.

– Sea $S \subseteq \{1, \dots, n\}$ cualquiera.

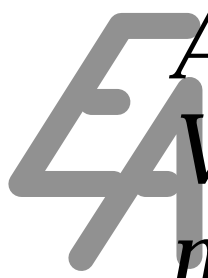
– Sea $\Sigma_S(D) = \sum_{i \in S} D_i$ ($\Sigma_\emptyset(D) = \vec{0}$)

– Sea
$$S' = \begin{cases} S \cup \{i\} & \text{si } i \notin S \\ S \setminus \{i\} & \text{si } i \in S \end{cases}$$

– Como D_i es no nulo, $\Sigma_S(D)$ y $\Sigma_{S'}(D)$ no pueden ser simultáneamente nulos.

– Si S se elige al azar (lanzando una moneda para cada j), la pertenencia de i a S es tan probable como la no pertenencia, luego:

$$P\{\Sigma_S(D) \neq \vec{0}\} \geq \frac{1}{2}$$



Algoritmos de Monte Carlo: Verificación de un producto matricial

- Por otra parte, si $C = AB$, $\Sigma_S(D) = \vec{0}$ siempre.
- Idea para decidir si $C = AB$ o no:
Calcular $\Sigma_S(D)$ para un conjunto elegido al azar S
y comparar el resultado con $\vec{0}$.
- ¿Cómo calcular $\Sigma_S(D)$ eficientemente?

Sea X el vector de n 0's y 1's tal que

$$X_j = \begin{cases} 1, & \text{si } j \in S \\ 0, & \text{si } j \notin S \end{cases}$$

Entonces:

$$\Sigma_S(D) = XD$$

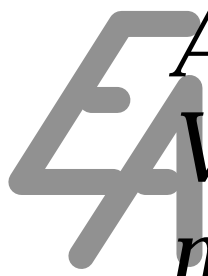
Es decir, se trata de decidir si $XAB = XC$ o no
para un vector binario X elegido al azar.

Algoritmos de Monte Carlo: Verificación de un producto matricial

- El coste del cálculo de $XAB = (XA)B$ y de XC es $\Theta(n^2)$.
- Algoritmo $1/2$ -correcto para decidir si $AB = C$:

```
tipo matriz=vector[1..n,1..n]de real

función Freivalds(A,B,C:matriz)
    devuelve booleano
variables X:vector[1..n]de 0..1
           j:entero
principio
    para j:=1 hasta n hacer
        X[j]:=uniforme_entero(0,1)
    fpara;
    si (X*A)*B=X*C
        entonces devuelve verdad
        sino devuelve falso
    fsi
fin
```



Algoritmos de Monte Carlo: Verificación de un producto matricial

❖ ¿Es útil un algoritmo $1/2$ -correcto para tomar una decisión?

❖ Es igual que decidir tirando una moneda al aire.

¡Y sin siquiera mirar los valores de las matrices!

❖ La clave:

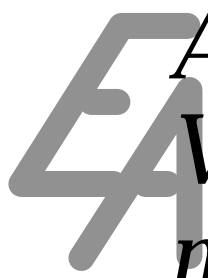
- Siempre que $\text{Freivalds}(A, B, C)$ devuelve falso, podemos estar seguros de que $AB \neq C$.
- Sólo cuando devuelve verdad, no sabemos la respuesta.

Algoritmos de Monte Carlo: Verificación de un producto matricial

❖ Idea: Repetir varias veces la prueba...

```
función repe_Freivalds(A,B,C:matriz;  
                        k:entero)  
    devuelve booleano  
variables i:entero; distinto:booleano  
principio  
    distinto:=verdad; i:=1;  
    mq i≤k and distinto hacer  
        si freivalds(A,B,C)  
            entonces i:=i+1  
            sino distinto:=falso  
        fsi  
    fmq;  
    devuelve distinto  
fin
```

- Si devuelve falso, es seguro que $AB \neq C$.
- ¿Y si devuelve verdad?
 - ¿Cuál es la probabilidad de error?



Algoritmos de Monte Carlo: Verificación de un producto matricial

- Si $C = AB$, cada llamada a `Freivalds` devuelve necesariamente el valor verdad, por tanto `repe_Freivalds` devuelve siempre verdad.

En este caso, la probabilidad de error es 0.

- Si $C \neq AB$, la probabilidad de que cada llamada devuelva (incorrectamente) el valor verdad es como mucho $1/2$.

Como cada llamada a `Freivalds` es independiente, la probabilidad de que k llamadas sucesivas den todas una respuesta incorrecta es como mucho $1/2^k$.

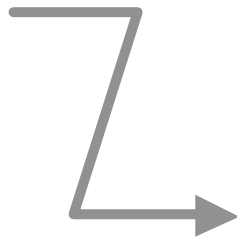
→ El algoritmo `repe_Freivalds` es $(1-2^{-k})$ -correcto.

Por ejemplo, si $k = 10$, es mejor que 0'999-correcto; si $k = 20$, la probabilidad de error es menor que uno entre un millón.

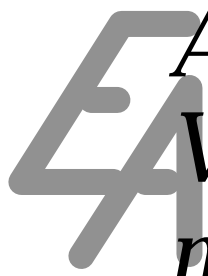
Algoritmos de Monte Carlo: *Verificación de un producto matricial*

- ❖ Situación típica en algoritmos de Monte Carlo para problemas de decisión:

Si está garantizado que
si se obtiene una de las
dos respuestas
(verdad o falso)
el algoritmo es
correcto



el decrecimiento de la
probabilidad de error es
espectacular repitiendo
la prueba varias veces.



Algoritmos de Monte Carlo: Verificación de un producto matricial

- ❖ Alternativa: diseñar el algoritmo con una cota superior de la probabilidad de error como parámetro.

```
función epsilon_Freivalds(A,B,C:matriz;  
                           epsilon:real))  
    devuelve booleano  
variable k:entero  
principio  
    k:= $\lceil \log(1/\text{epsilon}) \rceil$ ;  
    devuelve repe_Freivalds(A,B,C,k)  
fin
```

– Coste: $\Theta(n^2 \log 1/\text{epsilon})$.

Algoritmos de Monte Carlo: Verificación de un producto matricial

❖ Interés práctico:

- Se necesitan $3n^2$ multiplicaciones escalares para calcular XAB y XC , frente a las n^3 necesarias para calcular AB .
 - ♦ Si exigimos $\epsilon=10^{-6}$, y es cierto que $AB = C$, se requieren 20 ejecuciones de Freivalds, es decir, $60n^2$ multiplicaciones escalares, y eso sólo es mejor que n^3 si $n > 60$.
- Limitado a matrices de dimensión grande.



Algoritmos de Monte Carlo: Comprobación de primalidad

- ❖ Algoritmo de Monte Carlo más conocido: decidir si un número impar es primo o compuesto.
 - Ningún algoritmo determinista conocido puede responder en un tiempo “razonable” si el número tiene cientos de cifras.
 - La utilización de primos de cientos de cifras es fundamental en criptografía (ver *Divide y vencerás*, pág. 35 y siguientes).



Algoritmos de Monte Carlo: Comprobación de primalidad

❖ La historia comienza en 1640 con Pierre de Fermat...

– **Pequeño Teorema de Fermat.**

Sea n primo. Entonces,

$$a^{n-1} \bmod n = 1$$

para todo entero a tal que $1 \leq a \leq n-1$.

– Ejemplo: $n = 7$, $a = 5 \Rightarrow 5^6 \bmod 7 = 1$.

En efecto, $5^6 = 15625 = 2232 \times 7 + 1$.

– Enunciado **contrarrecíproco** del mismo teorema.

Si a y n son enteros tales que $1 \leq a \leq n-1$, y si $a^{n-1} \bmod n \neq 1$, entonces n no es primo.



Algoritmos de Monte Carlo: Comprobación de primalidad

❖ Una anécdota sobre Fermat y su teorema:

- El mismo formuló la hipótesis:

$$“F_n = 2^{2^n} + 1 \text{ es primo para todo } n.”$$

- Lo comprobó para: $F_0=3$, $F_1=5$, $F_2=17$, $F_3=257$, $F_4=65537$.
- Pero no pudo comprobar si $F_5=4294967297$ lo era.
- Tampoco pudo darse cuenta de que:

$$3^{F_5-1} \bmod F_5 = 3029026160 \neq 1 \Rightarrow F_5 \text{ no es primo} \\ (\text{por el contrarrecíproco de su propio teorema}) .$$

- Fue Euler, casi cien años después, quien factorizó ese número:

$$F_5 = 641 \times 6700417$$



Algoritmos de Monte Carlo: Comprobación de primalidad

❖ Utilización del pequeño teorema de Fermat para comprobar la primalidad:

- En el caso de F_5 , a Fermat le hubiera bastado con ver que

$$\exists a: 1 \leq a \leq F_5 - 1 \text{ t.q. } a^{F_5-1} \bmod F_5 \neq 1 \\ (a = 3)$$

- Esto nos da la siguiente idea:

```
función Fermat(n:entero) devuelve booleano
variable a:entero
principio
  a:=uniforme_entero(1,n-1);
  si  $a^{n-1} \bmod n = 1$ 
    entonces devuelve verdad
    sino devuelve falso
  fsi
fin
```



Algoritmos de Monte Carlo: Comprobación de primalidad

- El cálculo de $a^{n-1} \bmod n$ puede hacerse con el algoritmo de potenciación discreta que ya vimos (*Divide y vencerás*, pág. 33):

```
función potIter(a,n,z:entero) devuelve entero
{Devuelve  $a^n \bmod z$ .}
variable i,x,r:entero
principio
  i:=n; x:=a; r:=1;
  mq i>0 hacer
    si i es impar entonces r:=r*x mod z fsi;
    x:=x*x mod z;
    i:=i div 2
  fmq;
  devuelve r
fin
```



Algoritmos de Monte Carlo: Comprobación de primalidad

❖ Estudio del algoritmo basado en el pequeño teorema de Fermat:

- Si devuelve el valor falso, es seguro que el número no es primo (por el teorema de Fermat).
- Si devuelve el valor verdad:
¡No podemos concluir!
- Necesitaríamos el recíproco del teorema de Fermat:

“Si a y n son enteros tales que $1 \leq a \leq n-1$ y $a^{n-1} \bmod n = 1$, entonces n es primo.”

Pero este resultado es falso:

- ♦ Casos triviales en que falla: $1^{n-1} \bmod n = 1$, para todo $n \geq 2$.
- ♦ Más casos triviales en que falla: $(n-1)^{n-1} \bmod n = 1$, para todo impar $n \geq 3$.



Algoritmos de Monte Carlo: Comprobación de primalidad

- Pero, ¿falla el recíproco del teorema de Fermat en casos no triviales ($a \neq 1$ y $a \neq n-1$)?

SI.

El ejemplo más pequeño:
 $4^{14} \bmod 15 = 1$ y sin embargo 15 no es primo.

❖ Definición:

Falso testigo de primalidad.

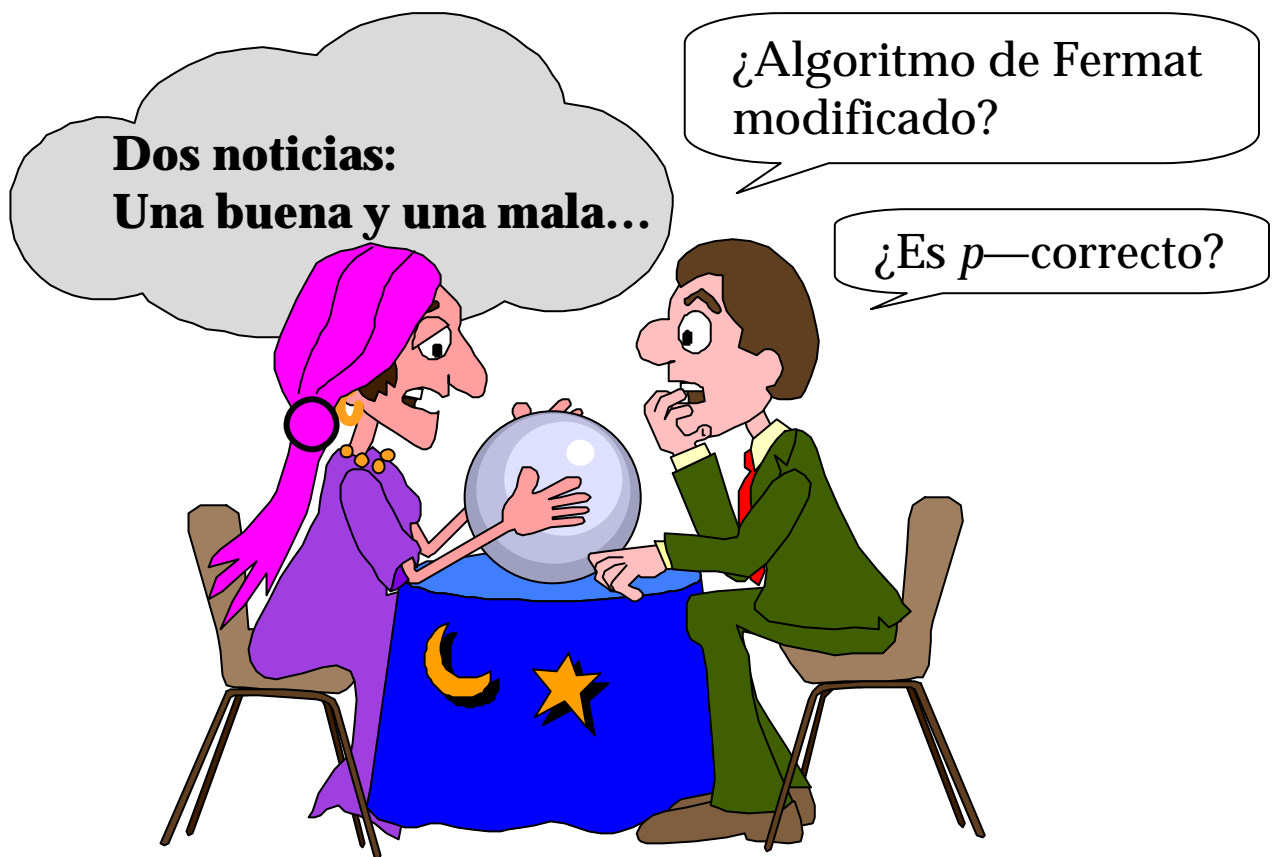
Dado un entero n que no sea primo, un entero a tal que $2 \leq a \leq n-2$ se llama falso testigo de primalidad de n si $a^{n-1} \bmod n = 1$.

- ♦ Ejemplo: 4 es un falso test. de prim. para 15.

❖ Modificación del algoritmo “Fermat”:

- Elegir a entre 2 y $n-2$ (en lugar de entre 1 y $n-1$).
- El algoritmo falla para números no primos sólo cuando elige un falso testigo de primalidad.

Algoritmos de Monte Carlo: Comprobación de primalidad





Algoritmos de Monte Carlo: Comprobación de primalidad

❖ La buena noticia:

- Hay “pocos” testigos falsos de primalidad.

Si bien sólo 5 de los 332 números impares no primos menores que 1000 carecen de falsos testigos de primalidad:

- ♦ más de la mitad de ellos tienen sólo 2 falsos testigos de primalidad,
- ♦ menos del 16% tienen más de 15,
- ♦ en total, hay sólo 4490 falsos testigos de primalidad para todos los 332 números impares no primos menores que 1000 (de un total de 172878 candidatos existentes)
- ♦ puede verse que la **probabilidad media de error** del algoritmo sobre los **números** impares no primos **menores que 1000** es menor que **0'033** y es **todavía menor** para **números mayores que 1000**.



Algoritmos de Monte Carlo: Comprobación de primalidad

❖ La mala noticia:

- Hay números no primos que admiten muchos falsos testigos de primalidad.

Recordar la característica fundamental de un algoritmo de Monte Carlo:

“Con una alta probabilidad encuentra una solución correcta **sea cual sea la entrada.**”

Por ejemplo, 561 admite 318 falsos testigos.

Otro ejemplo peor:

Fermat(651693055693681) devuelve verdad con probabilidad mayor que 0'999965 y sin embargo ese número no es primo.

- Puede demostrarse que el algoritmo de Fermat **no es p -correcto para ningún $p > 0$.**
 - ♦ Por tanto la probabilidad de error no puede disminuirse mediante repeticiones independientes del algoritmo.



Algoritmos de Monte Carlo: Comprobación de primalidad

❖ Una solución:

G.L. Miller: “Riemann’s hypothesis and tests for primality”,
Journal of Computer and System Sciences,
13(3), pp. 300-317, 1976.

M.O. Rabin: “Probabilistic algorithms”,
*Algorithms and Complexity: Recent Results and
New Directions*, J.F. Traub (ed.), Academic Press, 1976.

- Hay una **extensión del teorema de Fermat**:

Sea n un entero impar mayor que 4 y primo.

Entonces se verifica el predicado

$$B(n) = (a^t \bmod n = 1) \vee$$

$$\vee (\exists i \text{ entero}, 0 \leq i < s, \text{ t.q. } a^{2^i t} \bmod n = n-1)$$

para todo trío de enteros a , s y t tales que:

$2 \leq a \leq n-2$ y $n-1 = 2^s t$, con t impar.



Algoritmos de Monte Carlo: Comprobación de primalidad

- ❖ De nuevo, necesitaríamos el recíproco de ese teorema...

“Si n , s , t y a son enteros tales que $n > 4$, $n-1=2^st$, con n y t impares, $2 \leq a \leq n-2$, y se verifica $B(n)$, entonces n es primo.”

- ❖ Pero tampoco es cierto:

Existen números n y a con $n > 4$ e impar, $2 \leq a \leq n-2$, para los que se verifica $B(n)$ para algunos valores de s y t verificando $n-1=2^st$, con t impar, y **n no es primo.**

Por ejemplo: $n=289$, $a=158$, $s=5$, $t=9$.

- ❖ Si n y a son excepciones del recíproco del teorema se dice que n es un **pseudoprimo en el sentido fuerte** para la base a y que a es un **falso testigo de primalidad para n en el sentido fuerte.**



Algoritmos de Monte Carlo: Comprobación de primalidad

❖ Veamos primero como evaluar $B(n)$:

```
función B(a,n:entero) devuelve booleano
{Pre: n es impar y  $2 \leq a \leq n-2$ }
{Post:  $B(a,n)=\text{verdad} \Leftrightarrow a$  verifica  $B(n)$  para algún
valor de s y t tales que  $n-1=2^st$  con t impar}
variables s,t,x,i:entero; parar:booleano
principio
  s:=0; t:=n-1;
  repetir
    s:=s+1; t:=t div 2
  hastaQue t mod 2=1;
  x:= $a^t$  mod n; {se puede calcular con expdIter}
  si x=1 or x=n-1 entonces devuelve verdad
  sino
    i:=1; parar:=falso;
    mq i≤s-1 and not parar hacer
      x:=x*x mod n;
      si x=n-1
        entonces parar:=verdad
      sino i:=i+1
    fsi
  fmq;
  devuelve parar
fsi
fin
```



Algoritmos de Monte Carlo: Comprobación de primalidad

- ❖ Podemos basar el algoritmo probabilista de comprobación de primalidad en la función B :

```
función Miller_Rabin(n:entero)
    devuelve booleano
{Pre:  $n > 4$  e impar}
variable a:entero
principio
    a:=uniforme_entero(2,n-2);
    devuelve B(a,n)
fin
```

- Como con el algoritmo Fermat, **si la función devuelve falso, es seguro que el número no es primo** (por la extensión del teorema de Fermat).

- ¿Y si devuelve verdad?

El algoritmo puede fallar sólo para números pseudoprimos en el sentido fuerte (cuando elige como a un falso testigo de primalidad para n en el sentido fuerte).



Algoritmos de Monte Carlo: Comprobación de primalidad

❖ Por suerte (?), el número de falsos testigos de primalidad en el sentido fuerte es **mucho menor** que el de falsos testigos de primalidad.

- Considerando los impares no primos menores que 1000, la probabilidad media de elegir un falso testigo (en el sº fuerte) es menor que 0'01.
- Más del 72% de esos números no admiten ningún falso testigo (en el sº fuerte).
- Todos los impares no primos entre 5 y 10^{13} fallan como pseudoprimos (en el sº fuerte) para al menos una de las bases 2, 3, 5, 7 ó 61.

Es decir, para todo $n \leq 10^{13}$, n es primo si y sólo si $B(2,n) \wedge B(3,n) \wedge B(5,n) \wedge B(7,n) \wedge B(61,n) = \text{verdad}$ (éste es un algoritmo **determinista**, para $n \leq 10^{13}$).



Algoritmos de Monte Carlo: Comprobación de primalidad

❖ Y lo más importante:

- La proporción de falsos testigos de primalidad (en el sº fuerte) es pequeña **para todo impar no primo**.

❖ Teorema.

Sea n un entero impar mayor que 4.

- ♦ Si n es primo, entonces $B(n)$ =verdad para todo a tal que $2 \leq a \leq n-2$.
- ♦ Si n es compuesto, entonces

$$\left| \{a \mid 2 \leq a \leq n-2 \wedge B(n) = \text{verdad para } a\} \right| \leq (n-9)/4.$$

❖ Corolario.

La función `Miller_Rabin` siempre devuelve el valor verdad cuando n es primo.

Si n es un impar no primo, la función `Miller_Rabin` devuelve falso con una probabilidad mayor o igual que $3/4$.

Es decir, `Miller_Rabin` es un algoritmo $3/4$ -correcto para comprobar la primalidad.



Algoritmos de Monte Carlo: Comprobación de primalidad

- Como la respuesta “falso” siempre es correcta, para reducir la probabilidad de error se puede aplicar la misma técnica que para verificar el producto de matrices:

```
función repe_Miller_Rabin(n,k:entero)
    devuelve booleano
{Pre: n>4 e impar}
variables i:entero; distinto:booleano
principio
    distinto:=verdad; i:=1;
    mq i≤k and distinto hacer
        si Miller_Rabin(n)
            entonces i:=i+1
            sino distinto:=falso
        fsi
    fmq;
    devuelve distinto
fin
```

- Es un algoritmo de Monte Carlo $(1-4^{-k})$ -correcto.
- Por ejemplo, si $k=10$ la probabilidad de error es menor que una millonésima.
- Coste con cota de probabilidad de error ε :
 $O(\log^3 n \log 1/\varepsilon)$.
(Es razonable para n^{os} de mil cifras con $\varepsilon < 10^{-100}$.)



Algoritmos de Las Vegas: Introducción

- ❖ Un algoritmo de Las Vegas **nunca** da una solución falsa.
 - Toma decisiones al azar para encontrar una solución antes que un algoritmo determinista.
 - Si no encuentra solución lo admite.
 - Hay dos tipos de algoritmos de Las Vegas, atendiendo a la posibilidad de no encontrar una solución:
 - a) Los que **siempre** encuentran una solución correcta, aunque las decisiones al azar no sean afortunadas y la eficiencia disminuya.
 - b) Los que **a veces**, debido a decisiones desafortunadas, no encuentran una solución.



Algoritmos de Las Vegas: Introducción

❖ Tipo *a*: Algoritmos de *Sherwood*

- Existe una solución determinista que es mucho más rápida en media que en el peor caso.

Ejemplo: *quicksort*.

Coste peor $\Omega(n^2)$ y coste promedio $O(n \log n)$.

- ♦ Coste promedio: se calcula bajo la hipótesis de **equiprobabilidad** de la entrada.
- ♦ En aplicaciones concretas, la equiprobabilidad es una falacia: entradas catastróficas pueden ser muy frecuentes.
- ♦ Degradación del rendimiento en la práctica.



Algoritmos de Las Vegas: Introducción

- Los algoritmos de Sherwood pueden reducir o eliminar la diferencia de eficiencia para distintos datos de entrada:
 - ♦ Uniformización del tiempo de ejecución para todas las entradas de igual tamaño.
 - ♦ En promedio (tomado sobre todos los ejemplares de igual tamaño) no se mejora el coste.
 - ♦ Con alta probabilidad, ejemplares que eran muy costosos (con algoritmo determinista) ahora se resuelven mucho más rápido.
 - ♦ Otros ejemplares para los que el algoritmo determinista era muy eficiente, se resuelven ahora con más coste.

Efecto *Robin Hood*:

“Robar” tiempo a los ejemplares “ricos” para dárselo a los “pobres”.



Algoritmos de Las Vegas: Introducción

❖ Tipo *b*: Algoritmos que, a veces, no dan respuesta.

- Son aceptables si fallan con probabilidad baja.
- Si fallan, se vuelven a ejecutar con la misma entrada.
- Resuelven problemas para los que no se conocen algoritmos deterministas eficientes (ejemplo: la factorización de enteros grandes).
- El tiempo de ejecución no está acotado pero sí es razonable con la probabilidad deseada para toda entrada.



Algoritmos de Las Vegas: Introducción

– Consideraciones sobre el coste:

- ♦ Sea LV un algoritmo de Las Vegas que puede fallar y sea $p(x)$ la probabilidad de éxito si la entrada es x .

```
algoritmo LV(ent x:tpx; sal s:tpsolución;  
             sal éxito:booleano)  
{éxito devuelve verdad si LV encuentra la solución  
 y en ese caso s devuelve la solución encontrada}
```

- ♦ Se exige que $p(x) > 0$ para todo x .
- ♦ Es mejor aún si $\exists \delta > 0: p(x) \geq \delta$ para todo x

(así, la probabilidad de éxito no tiende a 0 con el tamaño de la entrada).



Algoritmos de Las Vegas: Introducción

```
función repe_LV(x:tpx) devuelve tpsolución  
variables s:tpsolución; éxito:booleano  
principio  
  repetir  
    LV(x,s,éxito)  
  hastaQue éxito;  
  devuelve s  
fin
```

- ♦ El número de ejecuciones del bucle es $1/p(x)$.
- ♦ Sea $v(x)$ el tiempo esperado de ejecución de LV si éxito=verdad y $f(x)$ el tiempo esperado si éxito=falso.
- ♦ Entonces el tiempo esperado $t(x)$ de repe_LV es:

$$t(x) = p(x)v(x) + (1 - p(x))(f(x) + t(x))$$
$$\Rightarrow t(x) = v(x) + \frac{1 - p(x)}{p(x)} f(x)$$



Algoritmos de Las Vegas: Introducción

$$t(x) = v(x) + \frac{1-p(x)}{p(x)} f(x)$$

Notar que una disminución de $v(x)$ y $f(x)$ suele ser a costa de disminuir $p(x)$.

β

Hay que optimizar esta función.



Algoritmos de Las Vegas: Introducción

- Ejemplo sencillo: El problema de las 8 reinas en el tablero de ajedrez.

- ♦ Algoritmo determinista (*Búsqueda con retroceso*, pág. 16 y siguientes):

Nº de nodos visitados: 114
(de los 2057 nodos del árbol)

- ♦ Algoritmo de Las Vegas voraz: colocar cada reina aleatoriamente en uno de los escaques posibles de la siguiente fila.

El algoritmo puede terminar con éxito o fracaso (cuando no hay forma de colocar la siguiente reina).

Nº de nodos visitados si hay éxito: $v=9$

Nº esperado de nodos visitados si hay fracaso: $f=6'971$

Probabilidad de éxito: $p=0'1293$

(más de 1 vez de cada 8)

Nº esperado de nodos visitados repitiendo hasta obtener un éxito: $t=v+f(1-p)/p=55'93.$

¡Menos de la mitad!



Algoritmos de Las Vegas: Introducción

- Puede hacerse mejor combinando ambos:
 - ♦ Poner las primeras reinas al azar y dejarlas fijas y con el resto usar el algoritmo de búsqueda con retroceso.

Cuantas más reinas pongamos al azar:

- Menos tiempo se precisa para encontrar una solución o para fallar.
- Mayor es la probabilidad de fallo.

nº al azar	p	v	f	t
0	1,0000	114,00	–	114,00
1	1,0000	39,63	–	39,63
2	0,8750	22,53	39,67	28,20
3	0,4931	13,48	15,10	29,01
4	0,2618	10,31	8,79	35,10
5	0,1624	9,33	7,29	46,92
6	0,1357	9,05	6,98	53,50
7	0,1293	9,00	6,97	55,93
8	0,1293	9,00	6,97	55,93

Algoritmos de Las Vegas: Introducción

Mejor solución a mano: 3 reinas al azar (¡probadlo!)

nº al azar	p	v	f	t	REAL
0	1,0000	114,00	–	114,00	0,45 ms
1	1,0000	39,63	–	39,63	
2	0,8750	22,53	39,67	28,20	0,14 ms
3	0,4931	13,48	15,10	29,01	0,21 ms
4	0,2618	10,31	8,79	35,10	1 ms
5	0,1624	9,33	7,29	46,92	
6	0,1357	9,05	6,98	53,50	
7	0,1293	9,00	6,97	55,93	
8	0,1293	9,00	6,97	55,93	

Datos reales medidos en un computador:

¡Discrepancias!

En el caso “nº al azar = 8”, el 71% del tiempo se
gasta en **generar números pseudo-aleatorios**.

El valor óptimo es colocar 2 reinas al azar.



Algoritmos de Las Vegas: Introducción

- Para dimensiones mayores a 8:

Para 39 reinas en un tablero de dimensión 39.

Algoritmo determinista:

11402835415 nodos

41 horas en un computador

Algoritmo Las Vegas, con 29 reinas al azar:

$p=0,21$

$v \approx f \approx 100$ nodos

$\Rightarrow t \approx 500$ nodos (20×10^6 veces mejor)

8,5 milisegundos

Algoritmo L.V. puro (39 reinas al azar):

$p=0,0074$

150 milisegundos (10^6 veces mejor)



Algoritmos de Las Vegas: Ordenación probabilista

❖ Ejemplo de algoritmo de Las Vegas “de tipo a” (algoritmo de Sherwood).

- Recordar el método de ordenación de Hoare (*Divide y vencerás*, pág. 14)):

```
algoritmo ordRápida(e/s T: vect[1..n] de dato;  
                  ent i, d: 1..n)  
{Ordenación de las componentes i..d de T.}  
variable p: dato; m: 1..n  
principio  
  si d-i es pequeño  
    entonces ordInserción(T, i, d)  
  sino  
    p := T[i]; {p se llama 'pivote'}  
    divide(T, i, d, p, m);  
    {i ≤ k < m ⇒ T[k] ≤ T[m] = p ∧ m < k ≤ d ⇒ T[k] > T[m]}  
    ordRápida(T, i, m-1);  
    ordRápida(T, m+1, d)  
  fsi  
fin
```

- Coste promedio: $O(n \log n)$
- Coste peor: $\Omega(n^2)$



Algoritmos de Las Vegas: Ordenación probabilista

```
algoritmo divide(e/s T:vect[1..n]de dato;  
                ent i,d:1..n; ent p:dato;  
                sal m:1..n)  
{Permuta los elementos i..d de T de forma que:  
  i ≤ m ≤ d,  
  ∀k t.q. i ≤ k < m: T[k] ≤ p,  
  T[m] = p,  
  ∀k t.q. m < k ≤ d: T[k] > p}  
variables k:1..n  
principio  
  k := i; m := d + 1;  
  repetir k := k + 1 hasta que (T[k] > p) or (k ≥ d);  
  repetir m := m - 1 hasta que (T[m] ≤ p);  
  mq k < m hace  
    intercambiar(T[k], T[m]);  
    repetir k := k + 1 hasta que T[k] > p;  
    repetir m := m - 1 hasta que T[m] ≤ p  
  fmq;  
  intercambiar(T[i], T[m])  
fin
```



Algoritmos de Las Vegas: Ordenación probabilista

- Un ejemplo del caso peor:
Si todos los elementos son iguales, el algoritmo anterior no se percata.
- Mejora evidente:

```
algoritmo ordRápida(e/s T:vect[1..n]de dato;  
                  ent i,d:1..n)  
{Ordenación de las componentes i..d de T.}  
variable m:1..n  
principio  
  si d-i es pequeño  
    entonces ordInserción(T,i,d)  
  sino  
    p:=T[i]; {pivote}  
    divideBis(T,i,d,p,m,r);  
    {m+1≤k≤r-1⇒T[k]=p ∧ i≤k≤m⇒T[k]<p ∧  
      ∧ m≤k≤d⇒T[k]>p}  
    ordRápida(T,i,m);  
    ordRápida(T,r,d)  
  
  fsi  
fin
```



Algoritmos de Las Vegas: Ordenación probabilista

- Versión probabilista:

En lugar de elegir el pivote p como el primer elemento del vector, lo ideal sería elegir la mediana, pero esto sería muy costoso, luego elegimos el pivote al azar en el intervalo $i..d$.

```
algoritmo ordRápidaLV(e/s T: vect[1..n] de dato;  
                      ent i, d: 1..n)  
{Ordenación de las componentes i..d de T.}  
variable m: 1..n  
principio  
  si d-i es pequeño  
    entonces ordInserción(T, i, d)  
  sino  
    p := T[uniforme_entero(i, d)]; {pivote}  
    divideBis(T, i, d, p, m, r);  
    {m+1 ≤ k ≤ r-1 ⇒ T[k] = p ∧ i ≤ k ≤ m ⇒ T[k] < p ∧  
      ∧ m ≤ k ≤ d ⇒ T[k] > p}  
    ordRápida(T, i, m);  
    ordRápida(T, r, d)  
  fsi  
fin
```

- Tiempo **esperado** en el peor caso: $O(n \log n)$



Algoritmos de Las Vegas: Factorización de enteros

- ❖ Ejemplo de algoritmo de Las Vegas “de tipo b ”.
- ❖ Problema: descomponer un número en sus factores primos.
- ❖ Problema más sencillo: **partición**
 - Dado un entero $n > 1$, encontrar un divisor no trivial de n , suponiendo que n no es primo.
- ❖ Factorización =
 - = test de primalidad + partición
 - Para factorizar n , hemos terminado si n es primo, si no, encontramos un divisor m de n y recursivamente factorizamos m y n/m .



Algoritmos de Las Vegas: Factorización de enteros

- ❖ Solución ingenua para el problema de la partición:

```
función partición(n:entero) devuelve entero
variables m:entero; éxito:booleano
principio
  m:=2; éxito:=falso;
  mq m ≤ ⌊sqrt(n)⌋ and not éxito hacer
    si m divide a n
      entonces éxito:=verdad
      sino m:=m+1
    fsi
  fmq;
  si éxito
    entonces devuelve m
    sino devuelve n
  fsi
fin
```

– Coste en el peor caso: $\Omega(\sqrt{n})$



Algoritmos de Las Vegas: Factorización de enteros

❖ El coste de la solución ingenua es demasiado alto:

- Partir un número “duro” de unas 40 cifras:

Si cada ejecución del bucle tarda 1 nanosegundo, el algoritmo puede tardar **miles de años**.

Número “duro” significa que es el producto de dos primos de tamaño parecido.

- Partir un número n de 100 cifras:

$$\sqrt{n} \approx 7 \times 10^{49}$$

(Nota: 10^{30} picosegundos es el doble de la edad estimada del Universo.)



Algoritmos de Las Vegas: Factorización de enteros

❖ Recordar el sistema RSA de criptografía

- En 1994 se factorizó un número duro de 129 cifras tras 8 meses de trabajo de más de 600 computadores de todo el mundo.

Se utilizó un algoritmo de Las Vegas.

- Existen varios algoritmos de Las Vegas para factorizar números grandes (véase [BB96]).
 - ◆ Están basado en resultados avanzados de teoría de números.
 - ◆ Siguen teniendo costes altísimos (factorizar un número de 100 cifras precisa del orden de 2×10^{15} operaciones).