Algoritmos probabilistas

Elvira Mayordomo

Universidad de Zaragoza

5 de octubre de 2022

Contenido de este tema

Contenido de este tema

Contenido de este tema

Contenido de este tema

Contenido de este tema

Contenido de este tema

Este tema está basado en:

- El capítulo 10 de G. Brassard, P. Bratley. Fundamentos de Algoritmia. Prentice Hall 1997.
- Transparencias sobre ese mismo capítulo de Javier Campos (con modificaciones de EM)
- Fuentes adicionales que os iré contando

Probabilidad

Definición

La **probabilidad (uniforme)** de un evento es el número de casos que producen ese evento dividido por el número de casos posibles.

Probabilidad

Definición

La **probabilidad (uniforme)** de un evento es el número de casos que producen ese evento dividido por el número de casos posibles.

A(n, m : *natural*)

1 $x :=$ uniforme_entero$(1, m)$

2 Resultado $(x > n)$

Probabilidad

Definición

La **probabilidad (uniforme)** de un evento es el número de casos que producen ese evento dividido por el número de casos posibles.

A(n, m : *natural*)

1 $x :=$ uniforme_entero$(1, m)$
2 Resultado $(x > n)$

$\Pr(A(n, m) = \mathit{True}) =$??

Probabilidad

Definición

La **probabilidad (uniforme)** de un evento es el número de casos que producen ese evento dividido por el número de casos posibles.

$\mathrm{A}(n, m : \mathit{natural})$

1 $x := \mathrm{uniforme\_entero}(1, m)$
2 Resultado $(x > n)$

$\Pr(A(n, m) = \mathit{True}) = ??$

Obviamente a veces consideramos unos casos más probables que otros y no es un escenario uniforme, entonces si Ω son los *distintos casos* entonces $\sum_{S \in \Omega} \Pr(S) = 1$

Algunas ecuaciones

1. $\overline{A} = \Omega - A$

Algunas ecuaciones

1. $\overline{A} = \Omega - A$
2. $\Pr(\overline{A}) = 1 - \Pr(A)$

Algunas ecuaciones

1. $\overline{A} = \Omega - A$
2. $\Pr(\overline{A}) = 1 - \Pr(A)$
3. Si $A_1 \cap A_2 = \emptyset$ entonces $\Pr(A_1 \cup A_2) = \Pr(A_1) + \Pr(A_2)$

Algunas ecuaciones

1. $\overline{A} = \Omega - A$
2. $\Pr(\overline{A}) = 1 - \Pr(A)$
3. Si $A_1 \cap A_2 = \emptyset$ entonces $\Pr(A_1 \cup A_2) = \Pr(A_1) + \Pr(A_2)$
4. Si A_1 y A_2 son independientes entonces $\Pr(A_1 \cap A_2) = \Pr(A_1) \cdot \Pr(A_2)$

Algunas ecuaciones

1. $\overline{A} = \Omega - A$
2. $\Pr(\overline{A}) = 1 - \Pr(A)$
3. Si $A_1 \cap A_2 = \emptyset$ entonces $\Pr(A_1 \cup A_2) = \Pr(A_1) + \Pr(A_2)$
4. Si A_1 y A_2 son independientes entonces $\Pr(A_1 \cap A_2) = \Pr(A_1) \cdot \Pr(A_2)$

¿Pero qué quiere decir independiente?

Ejemplos

A(n, m : *natural*)

1 $x :=$ uniforme_entero$(1, m)$
2 Resultado $(x > n)$

A(n, m : *natural*)

1 $x :=$ uniforme_entero$(1, m)$
2 $y :=$ uniforme_entero$(1, m)$
3 Resultado (x, y)

Variables aleatorias

Definición

Una **variable aleatoria** es una función que asigna un valor a cada $S \in \Omega$.

Variables aleatorias

Definición

Una **variable aleatoria** es una función que asigna un valor a cada $S \in \Omega$.

O sea, un valor a cada caso, usualmente un real

Variables aleatorias

Definición

Una **variable aleatoria** es una función que asigna un valor a cada $S \in \Omega$.

O sea, un valor a cada caso, usualmente un real

A(n, m : *natural*)

1 $u :=$ uniforme_entero$(1, m)$
2 **for** $i = 1$ **to** u
3 $a := a + 1$
4 Resultado a

Por ejemplo, $X_{n,m}$= tiempo del algoritmo anterior con entrada n, m

Esperanza de una variable aleatoria

Definición

La **esperanza** de una variable aleatoria $X : \Omega \to \mathbb{R}$ es $\mathrm{E}[X] = \sum_{S \in \Omega} X(S)\Pr(S)$.

Para el ejemplo anterior $\mathrm{E}[X_{n,m}] = \sum_{u \in \{1,\ldots,m\}} \cdots$

```
A(n, m : natural)
1   u := uniforme_entero(1, m)
2   for i = 1 to u
3         a := a + 1
4   Resultado a
```

Ecuaciones con variables aleatorias

1. $\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y]$

$\mathrm{A}(n, m : \mathit{natural})$

1 $u := \mathrm{uniforme\_entero}(1, m)$
2 **for** $i = 1$ **to** u
3 $\quad a := a + 1$
4 **for** $i = 1$ **to** u^2
5 $\quad a := a * a$
6 Resultado a

Ecuaciones con variables aleatorias

1. $\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y]$
2. $\Pr(|X| \geq a) \leq \mathrm{E}[|X|]/a$

```
A(n, m : natural)
1   u := uniforme_entero(1, m)
2   for i = 1 to u
3         a := a + 1
4   for i = 1 to u^2
5         a := a * a
6   Resultado a
```

Ecuaciones con variables aleatorias

1. $\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y]$
2. $\Pr(|X| \geq a) \leq \mathrm{E}[|X|]/a$
3. $\Pr\left(|X - \mathrm{E}[X]| \geq a\sqrt{\mathrm{E}[(X - \mathrm{E}(X))^2]}\right) \leq 1/a^2$

$\mathrm{A}(n, m : \mathit{natural})$

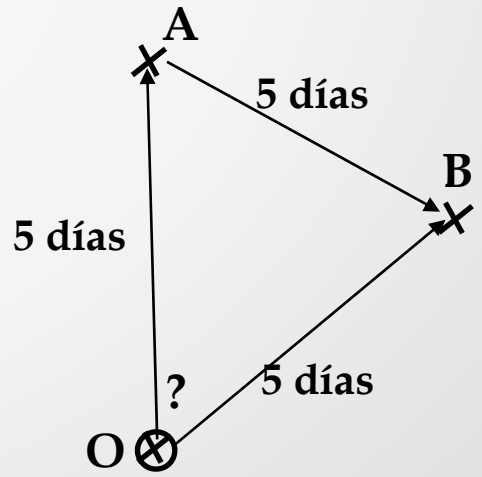
1 $u := \mathrm{uniforme\_entero}(1, m)$
2 **for** $i = 1$ **to** u
3 $\quad a := a + 1$
4 **for** $i = 1$ **to** u^2
5 $\quad a := a * a$
6 Resultado a

Algoritmos probabilistas

1. **Introducción**
   - Un poco de probabilidad y para qué la queremos
   - **Algoritmos probabilistas**
2. Clasificación de los algoritmos probabilistas
3. Algoritmos numéricos
4. Algoritmos de Monte Carlo
5. Algoritmos de Las Vegas

Introducción: un tesoro, un dragón, un elfo, un doblón y un computador

- En A ó B hay un **tesoro** de ***x*** lingotes de oro.
- Un **dragón** visita cada noche el tesoro llevándose ***y*** lingotes.
- Puedo saber en 4 días (en O) con mi **computador** dónde está.
- Un **elfo** me ofrece un trato:
  - solución ahora (pago lo que se llevaría el dragón en 3 noches).



Calculemos

- Si **me quedo** 4 días más en O hasta resolver el misterio, podré llegar al tesoro en **9 días**, y obtener ***x*-9*y*** lingotes.

- Si **acepto el trato** con el elfo, llego al tesoro en **5 días**, encuentro allí *x*-5*y* lingotes de los cuales debo pagar 3*y* al elfo, y obtengo ***x*-8*y*** lingotes.

Es **mejor aceptar el trato** pero…

¿Y si me la juego?

Lanzo al aire para decidir a qué lugar voy primero (A ó B).

- Si **acierto** a ir en primer lugar al sitio adecuado, obtengo ***x*-5*y*** lingotes.

- Si **no acierto**, voy al otro sitio después y me conformo con ***x*-10*y*** lingotes.

El **beneficio esperado medio** es ***x*-7’5*y***.

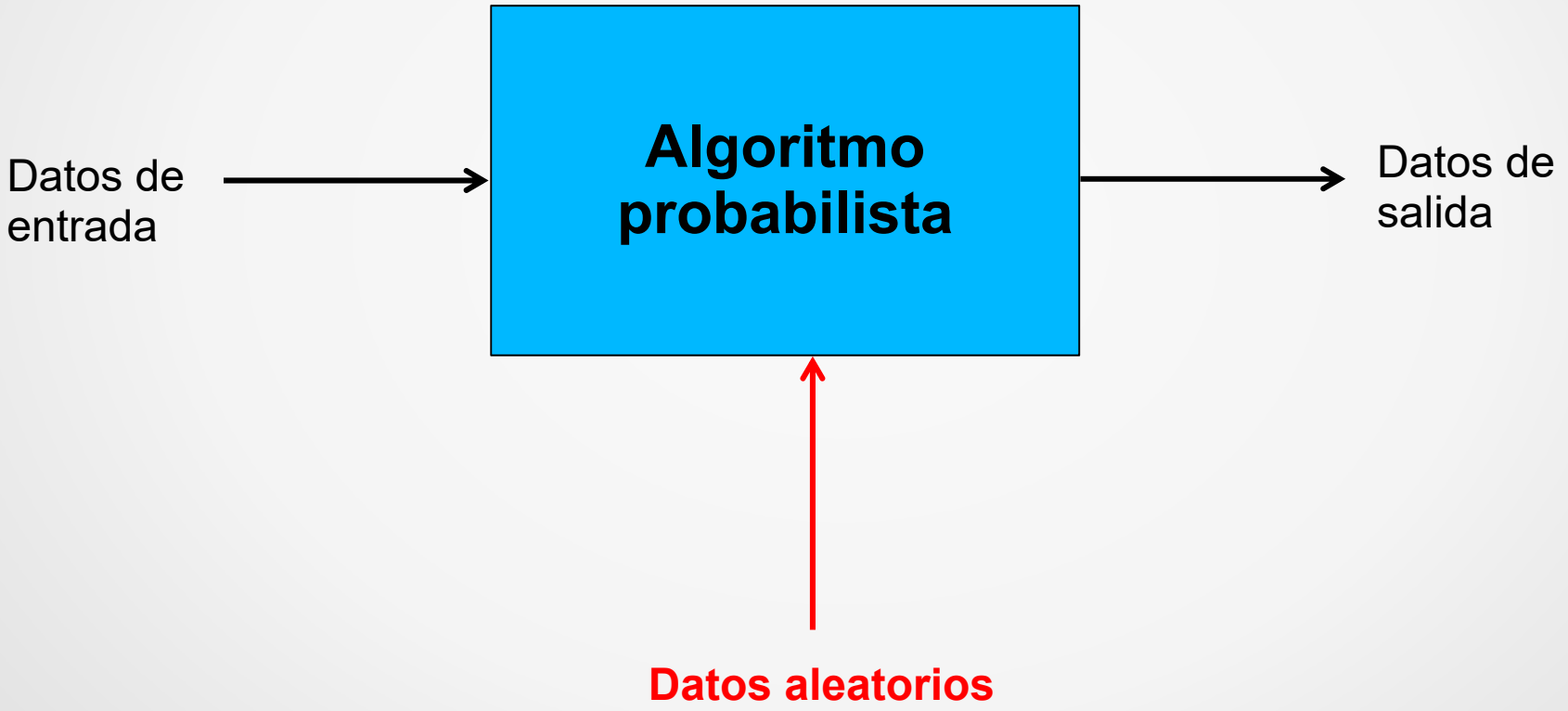
¿Entonces?

- 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.

¡Cuidado!

El mismo algoritmo puede comportarse de distinta forma aplicado a los mismos datos.

Flujo de datos



Azar e 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.

In some very real sense, computation is inherently randomized. It can be argued that the probability that a computer will be destroyed by a meteorite during any given microsecond of its operation is at least 2^{-100}.

Christos Papadimitriou

¡¡¡¡¿¿¿¿?????!!!!!

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

Más problemas

- Hay problemas para los que no se conoce **ningún algoritmo** (determinista ni probabilista) que dé la solución con certeza y en un tiempo razonable.

Ejemplos:

- La duración de la vida del programador,
- la vida del universo.

Por lo tanto, a veces...

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

Ejemplo: decidir si un nº de 1000 cifras es primo.

En realidad las cosas han cambiado un poco porque se conoce un algorimto con t. polinómico para primos.

Un ejemplo para el que no se conocen algoritmos eficientes pero sí probabilistas:

Polynomial Identity Testing:
Dado un polinomio q (caja negra, sin acceso a los coeficientes, acceso a los valores) ¿Es q siempre 0?

EMC

Resumen introducción

- Los algoritmos probabilistas utilizan información aleatoria cuando la necesitan …
- ¿Puedo estar seguro de que un algoritmo probabilista funciona?
- ¿Cómo garantizo el resultado?
- ¿Cómo garantizo las prestaciones (eficiencia)?
- ¿Puedo implementar un algoritmo probabilista? ¿Los generadores existentes son suficientes?

Algoritmos probabilistas

1. Introducción
   - Un poco de probabilidad y para qué la queremos
   - Algoritmos probabilistas
2. **Clasificación de los algoritmos probabilistas**
3. Algoritmos numéricos
4. Algoritmos de Monte Carlo
5. Algoritmos de Las Vegas

Tipos de algoritmos probabilistas

- Algoritmos que no garantizan la corrección de la solución:
  - Algoritmos Numéricos
  - Algoritmos de Monte Carlo
- Algoritmos que nunca dan una solución incorrecta:
  - Algoritmos de Las Vegas

Algoritmos Probabilitas. Numéricos

- Dan una solución aproximada.
- Dan un intervalo de confianza:

“ Con probabilidad del 90% la solución es 33 ± 3”.

- Más tiempo de ejecución => Mejor aproximación.

Ejemplo. 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 aproximada pero mejora aumentando el tiempo.
- Normalmente el error es O(1/sqrt(n)).
    - (100 veces más trabajo para una cifra más de precisión).

Algoritmos Probabilistas. Algoritmos de Monte Carlo

- Respuesta **exacta** con probabilidad alta.
- **Pueden** dar respuestas **incorrectas.**
- **No** se puede **saber** si la respuesta es correcta o incorrecta.
- Más tiempo de ejecución ⇒ Menor probabilidad de error.

Algoritmos Probabilistas. Algoritmos de Las Vegas

- Toman decisiones al azar.
- Si no encuentran una solución correcta lo admiten.
- Más ejecuciones  Mayor probabilidad de encontrar la solución.

“¿Cuándo descubrió América Cristobal Colón?”

Algoritmo numérico:

- 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%).

“¿Cuándo descubrió América Cristobal Colón?”

Algoritmo de Monte Carlo:

1492, 1492, 1492,

1491, 1492, 1492,

357 A.C., 1492, 1492,

1492

Aparentemente, un 20% de error.

Las respuestas incorrectas pueden ser **muy** incorrectas.

“¿Cuándo descubrió América Cristobal Colón?”

Algoritmo de Las Vegas:

1492, 1492, ¡Ops!,

1492, ¡Ops!, 1492,

1492, 1492, 1492,

1492

Aparentemente un 20% de error.

No hay **ninguna respuesta incorrecta**, pero el algoritmo **falla**.

Un poco de historia

Algoritmos de Monte Carlo

- 1946, laboratorio de Los Álamos (posteriormente en el Proyecto Manhattan).
- Secreto, nombre en clave “Monte Carlo”.
- Muy lentos al principio por la necesidad de los números aleatorios.
    - Se generaron los primeros PRNGs.

Resumiendo

- Los **algoritmos numéricos** dan:
  - Solución aproximada
  - Intervalo de confianza
- Los de **Montecarlo**
  - pueden dar respuesta incorrecta
- Los de **Las Vegas**
  - Nunca mienten, a veces no contestan
- Todos ellos mejoran el error al aumentar el tiempo

Algoritmos probabilistas

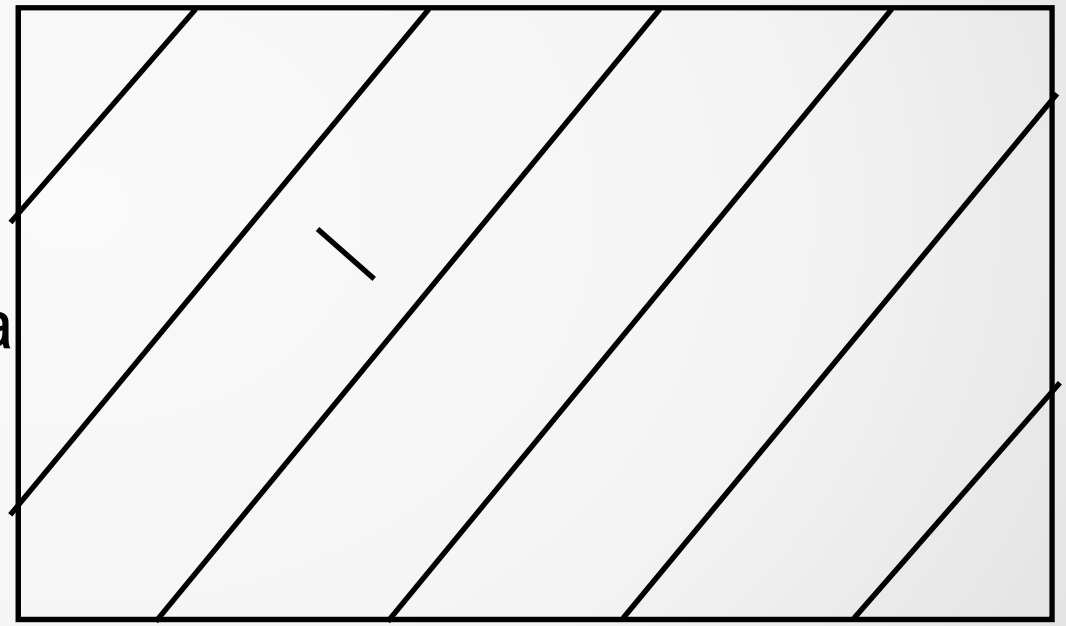
1. Introducción
2. Clasificación de los algoritmos probabilistas
3. **Algoritmos numéricos**
4. Algoritmos de Monte Carlo
5. Algoritmos de Las Vegas

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 $p=2\lambda/\omega\pi$.



La aguja de Buffon. Aplicación

- Si $\lambda=\omega/2$, entonces $p=1/\pi$.
- 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 π:

$$k \approx n/\pi \Rightarrow \pi \approx n/k$$

(Probablemente) el primer algoritmo probabilista.

¿Pero es útil?

- Cómo de rápida es la convergencia?

  (¿cuántas veces hay que tirar la aguja?)

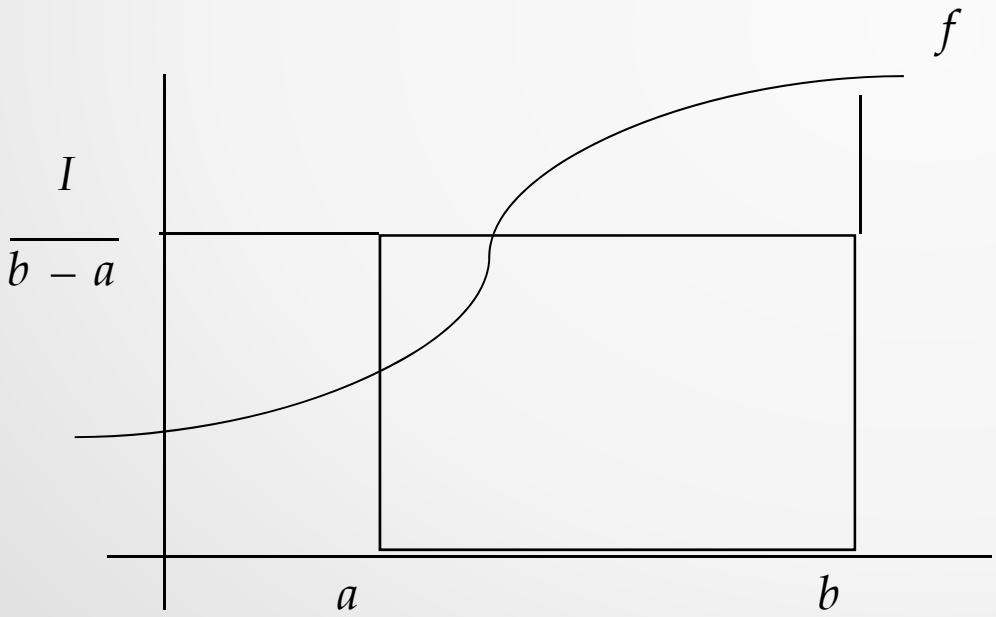
Es **muy lenta,** es decir el método no sirve [BB96]:

n=1.500.000 para obtener un valor de π±0’01 con probabilidad 0’9.

Algoritmos Numéricos. Integración probabilista.

Problema:

Calcular: $I = \int_a^b f(x)\, \mathrm{d}\, x, \quad$ donde $\quad f : \mathsf{R} \rightarrow \quad \mathsf{R}^+ \quad$ es continua y $\quad a \leq b$



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
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) // La media
fin
```

Número (seudo)aleatorio en [a,b)

Algoritmos Numéricos. Integración probabilista. ¿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.
- El error esperado es proporcional a la raiz de la varianza sqrt(Var(X)).
- 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. Versión determinista.

```
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
```

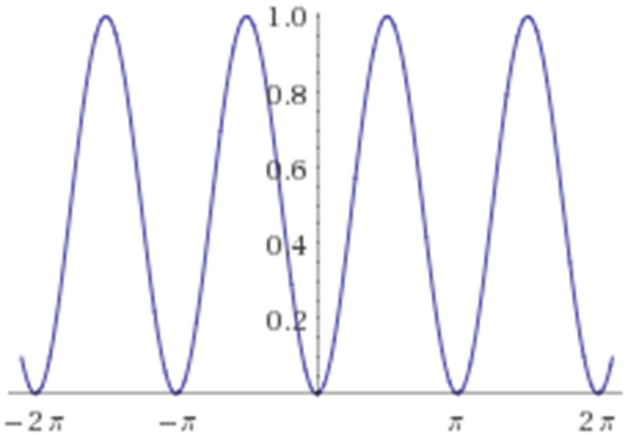
Puntos equidistantes

Algoritmos Numéricos ¿Entonces?

- 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).

Algoritmos Numéricos. ¿Entonces?

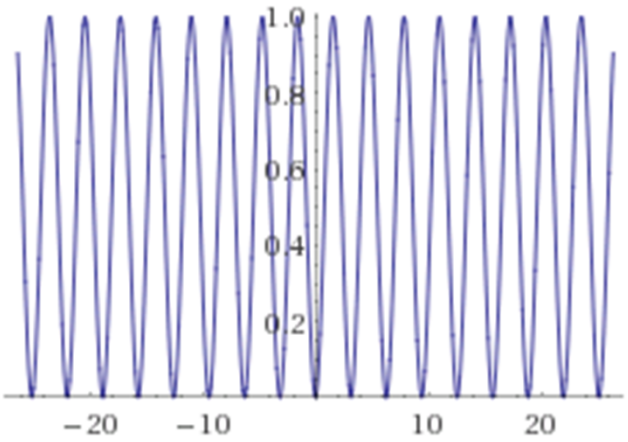
Plots:



(x from -6.283 to 6.2

Input:

$$\sin^2(100!\,\pi\,x)$$



(x from -20 to 20)

http://www.wolframalpha.com/input/?i=sin%C2%B2+%28100%21+pi*x%29

Algoritmos Numéricos. ¿Entonces?

- 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 ó mayor.
  - Existen técnicas híbridas (parcialmente sistemáticas y parcialmente probabilistas): *integración cuasi-probabilista*.

Complejidad de los Algoritmos Probabilistas Numéricos

- Para el ejemplo de integración, hemos dicho que el error esperado es inversamente proporcional a sqrt(n) (n era el número de puntos muestreados).
- **Medimos la complejidad en función del error** (más tiempo para obtener mayor precisión).
- En este ejemplo error = 1/sqrt(n), luego $n=1/error^2$

Complejidad de los Algoritmos Probabilistas Numéricos

```
función int_prob(f:función; epsilon: real
                 a,b:real) devuelve real
variables suma,x:real; i:entero; n:entero;
principio
  suma:=0.0;
  n:= 1/(epsilon*epsilon);
  para i:=1 hasta n hacer
    x:=uniforme(a,b);
    suma:=suma+f(x)
  fpara;
  devuelve (b-a)*(suma/n) // La media
fin
```

Complejidad de los Algoritmos Probabilistas Numéricos

- Tiempo para integración $O(1/epsilon^2)$
- En general en los algoritmos numéricos añadimos como parámetro de entrada el error épsilon y contabilizamos el tiempo necesario para obtener, con probabilidad alta, un resultado en

$$[x-epsilon, x+epsilon]$$

(siendo x el resultado exacto)

Resumen algoritmos numéricos

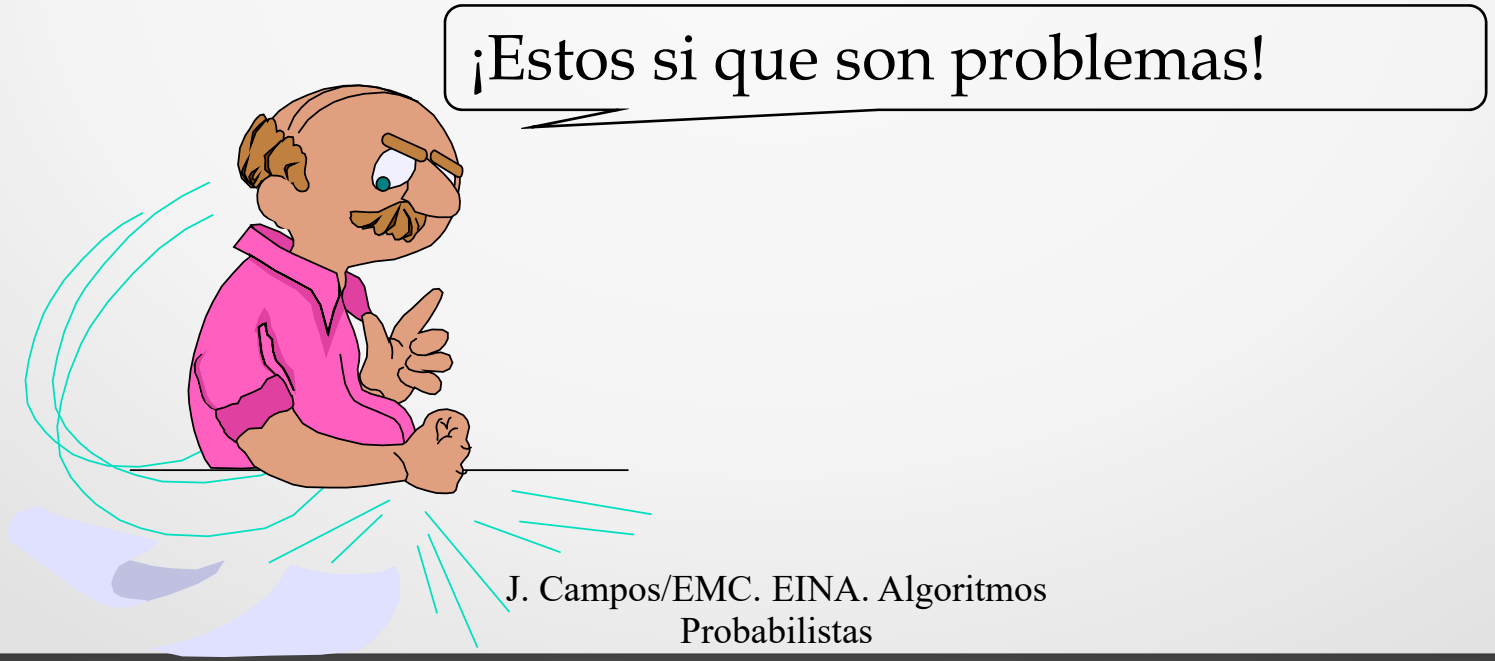
- Un algoritmo numérico probabilista puede ser más útil que uno determinista porque es **más difícil de engañar**
  - Ejemplo: Integración, si los puntos son fijos hay funciones que engañan, si elegimos los puntos al azar no hay función que lo consiga
- Se usa estadística (teorema central del límite) para razonar el error (O(1/sqrt(n) cuando se hace suma de n muestras)
- La **complejidad** se mide en función del error

Algoritmos probabilistas

1. Introducción
2. Clasificación de los algoritmos probabilistas
3. Algoritmos numéricos
4. **Algoritmos de Monte Carlo**
5. Algoritmos de Las Vegas

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).



Algoritmos de Monte Carlo. Introducción.

Un 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,

verificar si $C = AB$.

Solución trivial: hacer la multiplicación.

- Algoritmo directo: coste $\Theta(n^3)$.
- Algoritmo de Strassen (*Divide y vencerás*): $O(n^{2,81})$.
- Otros menos prácticos: $O(n^{2,373})$.

Algoritmos de Monte Carlo. Verificación de un producto matricial.

¿Podemos hacerlo más rápido?

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.

Algoritmos de Monte Carlo. Verificación de un producto matricial.

- Queremos hacer un algoritmo que dadas tres matrices A, B, C verifique que AB=C

```
función Freivalds(A,B,C:matriz) devuelve booleano
```

- Sea $D = AB - C$
- Queremos un algoritmo que distinga si D es 0.

Muy útil (demostrado a continuación)

- Si C = AB, $\Sigma_S (AB - C) = 0 \quad \forall S$

- Si C ≠ AB, $P (\Sigma_S (AB - C) \neq 0) \geq 1/2$

(Donde S es un conjunto cualquiera de filas y Σ_S es la suma de las filas S)

¿Por qué P (Σ_S (AB–C) ≠ 0) ≥ ½ ?

D = *AB* – *C*, supongamos que AB ≠ C, es decir, que D ≠ 0

- Sea *i* el índice de una fila no nula de *D*: $D_i \neq \vec{0}$
- Vamos a dividir en parejas los conjuntos $S \subseteq \{1,\ldots,n\}$.
  - Para cada S con $i \notin S$ cojo como pareja $S' = S \cup \{ i \}$
  - De cada pareja S, S' como mucho uno de los dos puede sumar 0 ya que $D_i \neq \vec{0}$ y

$$\sum_{S'}(D) = \sum_{S}(D) + D_i$$

¿Por qué P (Σ_S (AB–C) ≠ 0) ≥ ½ ?

D = *AB* – *C,* supongamos que AB ≠ C, es decir, que D ≠ 0

- Vamos a dividir en parejas los conjuntos $S \subseteq \{1,\ldots,n\}$.
  - De cada pareja S, S' como mucho uno de los dos puede sumar 0
  - Luego como mucho la mitad suman cero
  - **La mitad o más suman distinto de 0**

Algoritmo de Monte Carlo. Verificación de un producto matricial.

La mitad o más de los S suman distinto de 0, luego:

$$P\{\sum_{S}(D) \neq 0\} \geq 1/2$$

Muy útil

- Si C = AB, $\sum_S (D) = 0, \forall S$

- Si C ≠ AB, $P\{\sum_S (D) \neq 0\} \geq 1/2$

Algoritmo de Monte Carlo. Verificación de un producto matricial.

- **Idea:** Calcular $\sum_S(D)$ para un conjunto elegido al azar y comparar con 0.

- ¿Se puede calcular $\sum_S(D)$ de manera eficiente?

Algoritmo de Monte Carlo. Verificación de un producto matricial.

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

$$X_j = 1, j \in S$$

$$X_j = 0, j \notin S$$

Entonces:

$$\sum_S (D) = X.D$$

Algoritmo de Monte Carlo. Verificación de un producto matricial.

Se trata de decidir si: XAB = XC (X aleatorio)

El coste del cálculo de

$$XAB = (XA)\,B$$

$$XC$$

es

$$O(n^2)$$

Algoritmos de Monte Carlo. Verificación de un producto matricial.

```
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.

- En el caso C=AB Freivalds(A,B,C) acierta siempre
- En el caso C≠AB Freivalds(A,B,C) falla cuando elige X con XAB = XC
  - es decir, cuando elige S con Σ_S (D) =0
  - Pero sabemos que
    - Si C ≠ AB, $P(\Sigma_S(AB-C) \neq 0) \geq \frac{1}{2}$
  - Luego acierta con probabilidad ≥ ½
- Tenemos un algoritmo 1/2 - correcto

Algoritmos de Monte Carlo. Verificación de un producto matricial.

Pero …

- ¿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!

Algoritmos de Monte Carlo. Verificación de un producto matricial.

La clave:

- Si `Freivalds(A,B,C)` devuelve **falso**, podemos estar seguros de que $AB \neq C$.
- Sólo cuando devuelve **verdad**, **no sabemos** la respuesta.

¡Podemos repetir!

Algoritmos de Monte Carlo. Verificación de un producto matricial.

```
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
```

Algoritmos de Monte Carlo. Verificación de un producto matricial.

Si devuelve **falso**, es seguro que $AB \neq C$.

¿Y si devuelve **verdad**?

¿ Cuál es la **probabilidad** de error?

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**.

Algoritmos de Monte Carlo. Verificación de un producto matricial.

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:

$$1/2^k$$

Algoritmos de Monte Carlo. Verificación de un producto matricial.

Por lo tanto:

El algoritmo `repe_Freivalds` es

$(1-2^{-k})$–correcto.

***k* = 10**, es mejor que: **0’999–correcto**

***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.

```
función epsilon_Freivalds(A,B,C:matriz;
                          epsilon:real))
        devuelve booleano
variable k:entero
principio
  k:=⌈log(1/epsilon)⌉;
  devuelve repe_Freivalds(A,B,C,k)
fin
```

Diseñar el algoritmo con una cota superior de la probabilidad de error como parámetro.

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, (n^3 necesarias para calcular AB).
- Si epsilon=10^{-6}, y $AB = C$, 20 ejecuciones de `Freivalds:`
  - $60n^2$ multiplicaciones, mejor que n^3 si $n>60$

Limitado a matrices de **dimensión grande**

Algoritmos probabilistas

- Introducción
- Clasificación de los algoritmos probabilistas
- Algoritmos numéricos
- **Algoritmos de Monte Carlo**
  - Verificación de un producto matricial
  - **Comprobación de primalidad**
- Algoritmos de Las Vegas

Algoritmos de Monte Carlo. Comprobación de primalidad.

- Es el algoritmo de Monte Carlo más conocido: decidir si un número impar es primo o compuesto.
  - Ningún algoritmos determinista conocido puede responder en tiempo 'razonable' si el número es 'grande'
  - La utilización de números primos 'grandes' es fundamental en criptografía.

Muy importante

- Necesitamos algoritmos eficientes en función del tamaño de la entrada (el tamaño es lo que ocupa la entrada en memoria)
- En un algoritmo para primalidad la entrada es un número natural n. El tamaño de la entrada es d= log n que es el número de bits de la entrada
- Un algoritmo eficiente = **tiempo polinómico** tardará tiempo como $\mathbf{d^3 = (\log n)^3}$

Primos está en P (añadido EMC)

- En 2002 se publicó el AKS primality test (test de primalidad de Agrawal–Kayal–Saxena)
- Es el primer test de primalidad que es *general, tiempo polinómico, determinista e incondicional*
- Pero sigue siendo cierto que no se conocen algoritmos deterministas en tiempo razonable, ya que hoy en día la mejor cota conocida (mejora del AKS) es $O(d^6)$ (donde d=log n es el número de bits)

Clave pública y primalidad (añadido EMC)

- Los primos son fundamentales en criptografía
- Ejemplo: El protocolo de clave pública RSA permite codificar mensajes usando la *clave pública* y decodificar usando la *clave privada* (y firmar usando la *clave privada* y verificar usando la *clave pública*)
- La seguridad del sistema RSA está (en parte) basada en que no es posible *factorizar* números grandes en tiempo razonable
- Factorizar parece bastante más difícil que testear primalidad
    - Precisamente RSA usa el pequeño teorema de Fermat que veremos hoy

Algoritmos de Monte Carlo. Comprobación de primalidad.

- 1640, Pierre de Fermat.
- Pequeño Teorema de

Fermat: Sea n primo, a con (a mod n)>0.

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

Ejemplo: $n = 7$, $a = 5$

¿$5^6 \bmod 7 = 1$?

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

Algoritmos de Monte Carlo. Comprobación de primalidad.

Contrarrecíproco:

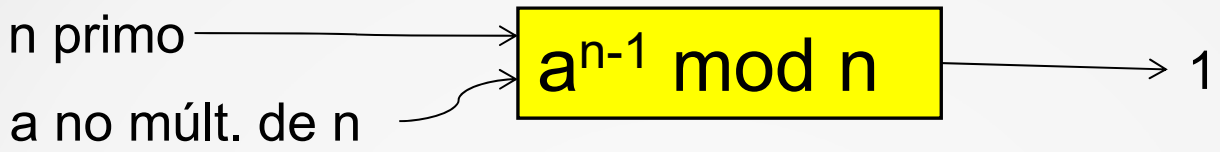
Si a y n son enteros tales que $1 \leq a \leq n-1$,

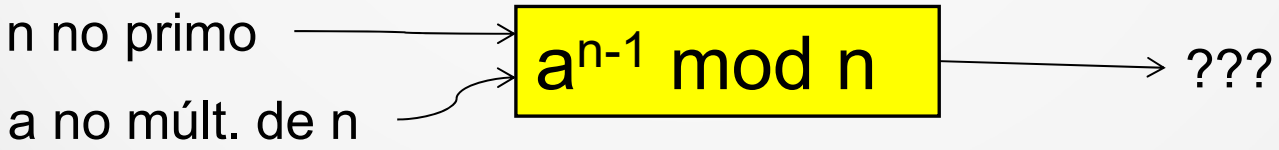
y si $a^{n-1} \bmod n \neq 1$,

entonces

n **no es primo**.

Buscando un test de primalidad





Algoritmos de Monte Carlo. Comprobación de primalidad.

Una **anécdota** sobre Fermat y su teorema:
Él mismo formuló la hipótesis:

$F_n = 2^{2^n} + 1$ Es primo para todo n.

Lo comprobó para:

- F_0=3,
- F_1=5,
- F_2=17,
- F_3=257,
- F_4=65537.

Algoritmos de Monte Carlo. Comprobación de primalidad.

Pero no pudo comprobar si F_5=4294967297 lo era.

Tampoco pudo darse cuenta de que:

$$3^{F_5-1} mod\ F_5 = 3029026160 \neq 1$$

No es primo (contrarrecíproco de su teorema).

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

$F_5 = 641 \times 670041$

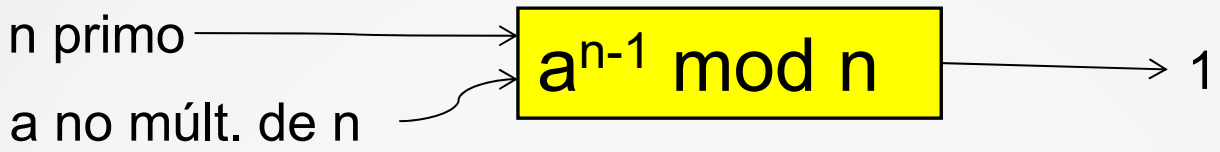
Algoritmos de Monte Carlo. Comprobación de primalidad.

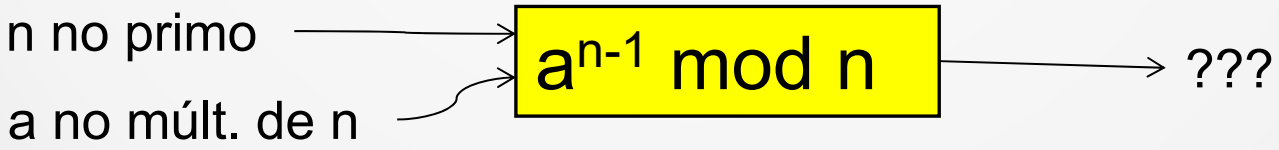
Utilización del pequeño teorema de Fermat para comprobar la primalidad:

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

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

Buscando un test de primalidad





Algoritmos de Monte Carlo. Comprobación de primalidad.

Recopilemos:

- Si existe a con $1 \leq a \leq n-1$ que cumple

  $a^{n-1} \bmod n \neq 1$,

  entonces n **no es primo**.

- Podemos plantear un algoritmo que elija a al azar …

Algoritmos de Monte Carlo. Comprobación de primalidad.

- Idea:

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

Algoritmos de Monte Carlo. Comprobación de primalidad.

¿Cómo calculamos $a^{n-1} \bmod n$?

Divide y vencerás

Al calcular multiplicaciones de números hasta n (por usar $\bmod n$) cada una de ellas tarda $O((\log n)^2)$

Recordemos que el tamaño de la entrada es $d = \log n$

En total $a^{n-1} \bmod n$ se calcula en tiempo $O(d^3)$

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!

Algoritmos de Monte Carlo. Comprobación de primalidad.

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."

Algoritmos de Monte Carlo. Comprobación de primalidad.

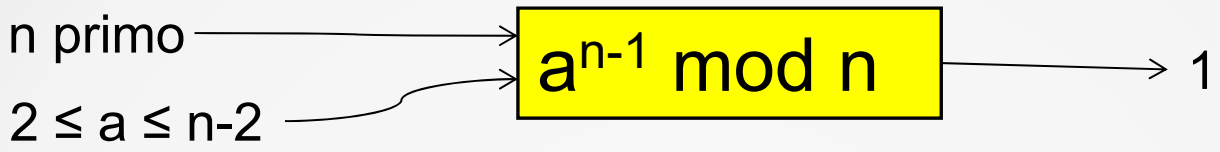
¡Falso!

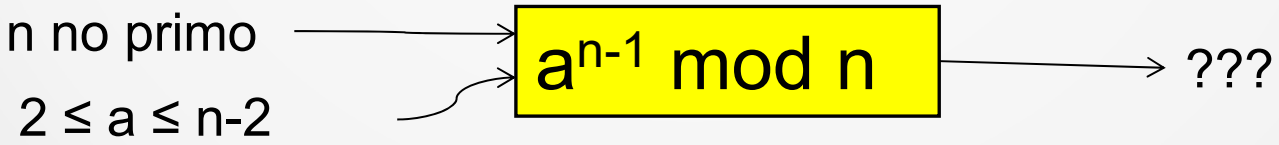
- Casos triviales en que falla:

$1^{n-1} mod\ n = 1$ para todo n≥2.

- Más casos triviales en que falla:
$(n\text{-}1)^{n\text{-}1}$ mod n = 1, para todo impar n≥3.

Buscando un test de primalidad





Algoritmos de Monte Carlo. Comprobación de primalidad.

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

SÍ

El ejemplo más pequeño:

$4^{14} \bmod 15 = 1$ y sin embargo 15 no es primo.

Algoritmos de Monte Carlo. Comprobación de primalidad.

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 testigo de primalidad para 15.

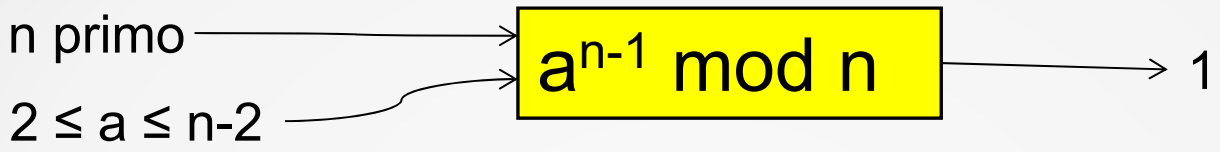
Algoritmos de Monte Carlo. Comprobación de primalidad.

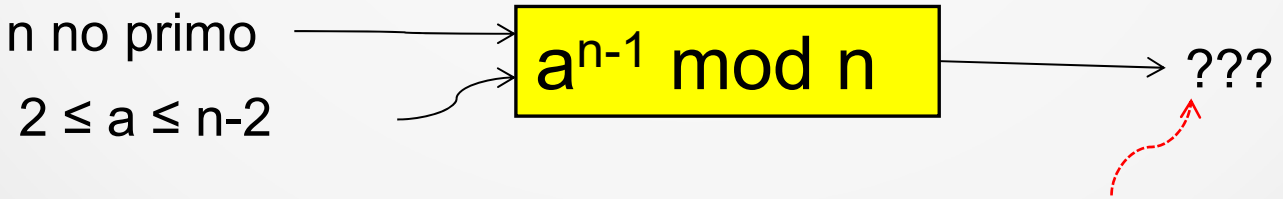
Modificación del algoritmo “Fermat”:

- 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**

“Carmichael numbers”

Buscando un test de primalidad





¿puedo al menos decir que es casi siempre ≠1?

Algoritmos de Monte Carlo. Comprobación de primalidad.

La **mala** noticia:
Hay números no primos que admiten muchos falsos testigos de primalidad.

Tan cerca del 100% como se quiera

"Un algoritmo probabilista debe encontrar la solución correcta con alta probabilidad **sea cual sea la entrada**."

Oooooops!!

Algoritmos de Monte Carlo. Comprobación de primalidad.

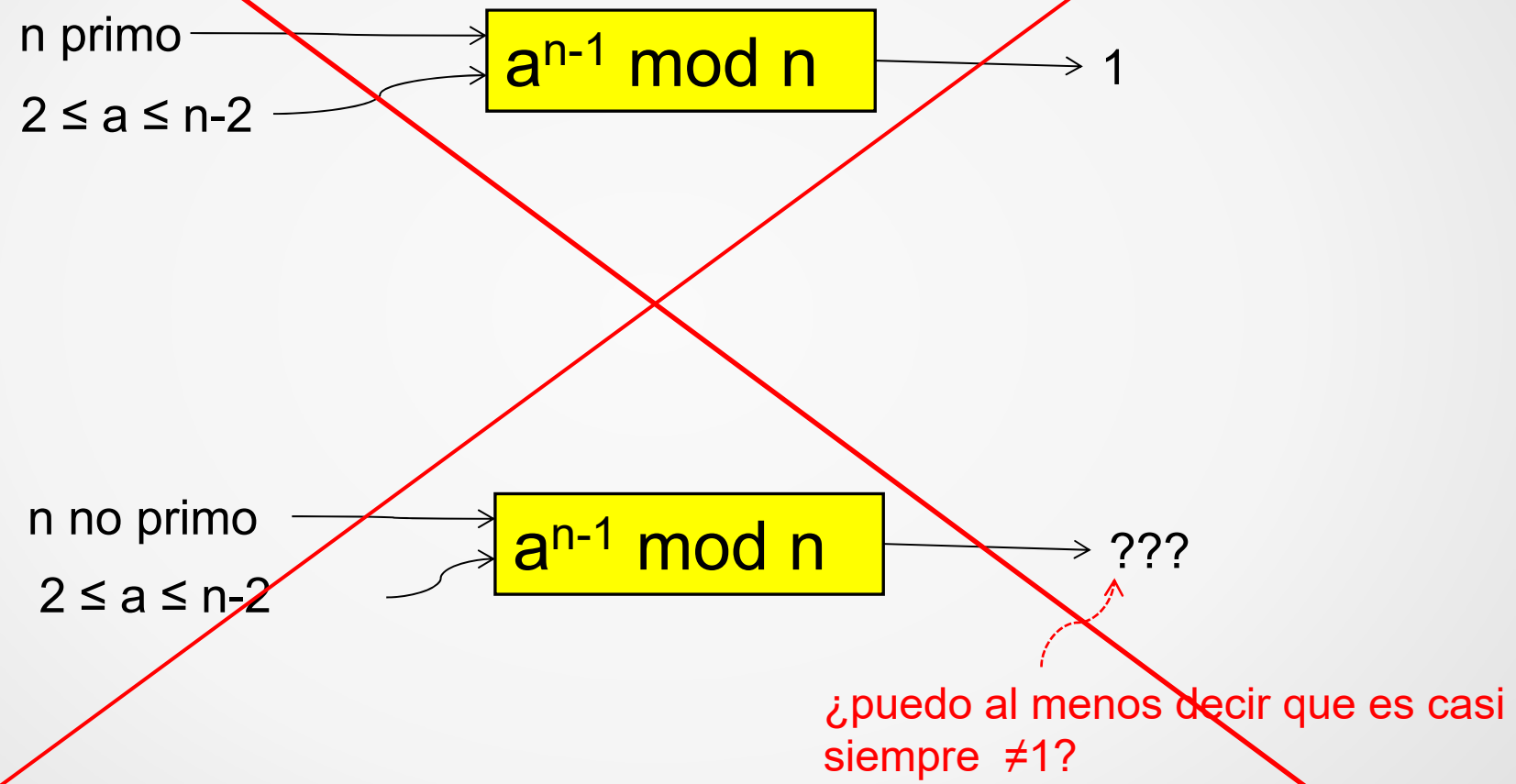
Por ejemplo, **561** admite **318** falsos testigos.

Otro ejemplo peor:

Fermat(651693055693681)

devuelve verdad con probabilidad mayor que 0’999965 y sin embargo ese número no es primo.

Buscando un test de primalidad



Algoritmos de Monte Carlo. Comprobación de primalidad.

- Idea:

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

Algoritmos de Monte Carlo. Comprobación de primalidad.

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.

G.L. Miller: "Riemann's hipothesis 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.

Algoritmos de Monte Carlo. Comprobación de primalidad.

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

"Sea n un entero impar mayor que 4 y primo.

Entonces para todo $2 \leq a \leq n-2$ se verifica el predicado

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

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

para s y t enteros tales que:

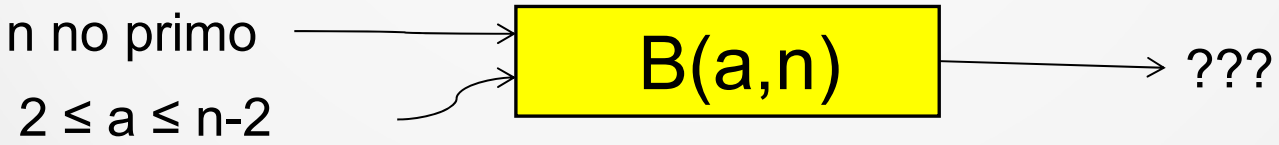
$n-1 = 2^s t$, con t impar."

Algoritmos de Monte Carlo. Comprobación de primalidad.

De nuevo, necesitaríamos el recíproco de ese teorema…

"Si n y a son enteros tales que $n>4$, $2 \leq a \leq n-2$, y se verifica $B(a,n)$,
entonces n es primo."

Buscando un test de primalidad



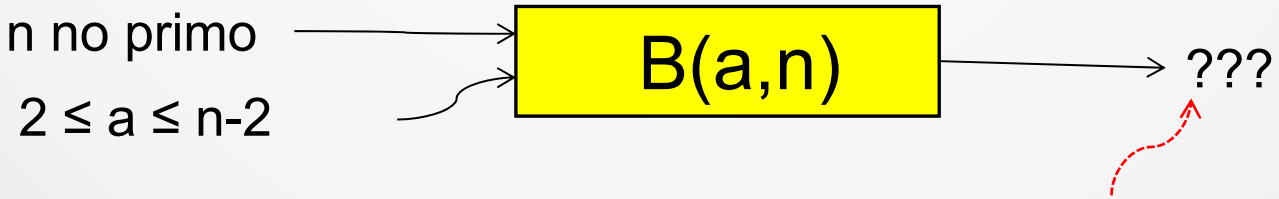
Algoritmos de Monte Carlo. Comprobación de primalidad.

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 $B(a,n)$ y ***n* no es primo**.

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

Buscando un test de primalidad



¿puedo al menos decir que es casi siempre False?

Algoritmos de Monte Carlo. Comprobación de primalidad.

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**.

Con falsos testigos de primalidad en sentido fuerte las cosas funcionan bastante mejor …

Algoritmos de Monte Carlo. Comprobación de primalidad.

Hacemos un algoritmo usando:

Si n es primo entonces para todo $2 \leq a \leq n-2$ se cumple $B(a,n)$

Al calcular multiplicaciones de números hasta n (por usar $\mathtt{mod}\ n$) cada una de ellas tarda $O((\log n)^2)$

Recordemos que el tamaño de la entrada es $d = \log n$

En total $B(a,n)$ se calcula en tiempo $O(d^3)$:

Algoritmos de Monte Carlo. Comprobación de primalidad.

```
función B(a,n:entero) devuelve booleano
{Pre: n es impar y 2≤a≤n-2}
{Post: B(a,n)=verdad ⇔ a,n verifican transparencia
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}

...
```

Algoritmos de Monte Carlo. Comprobación de primalidad.

```
….
  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
```

Algoritmos de Monte Carlo. Comprobación de primalidad.

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 sentido fuerte).

Algoritmos de Monte Carlo. Comprobación de primalidad.

- 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 $B(2,n) \wedge B(3,n) \wedge B(5,n) \wedge B(7,n) \wedge B(61,n)$ = 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**.

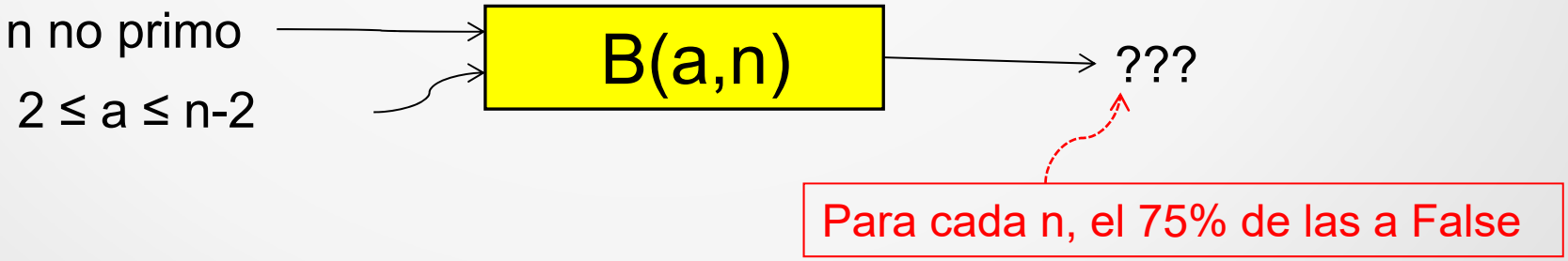
Teorema.
Sea *n* un entero impar mayor que 4.

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

Si *n* es compuesto, entonces

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

Miller-Rabin



Algoritmos de Monte Carlo. Comprobación de primalidad.

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.

```
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
```

Algoritmos de Monte Carlo. Comprobación de primalidad.

- Misma técnica que para verificar el producto de matrices (la respuesta "falso" siempre es correcta)
- 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: $O(d^3 \log {}^1/_{\varepsilon})$. Recordemos que $d = \log n$

(Es razonable para nºs de mil cifras con $\varepsilon<10^{-100}$.)

Resumiendo Monte Carlo

- Los tests de primalidad son los algoritmos de Montecarlo más conocidos (y utilizados)
- Un algoritmo de Montecarlo es **p-correcto** cuando para cualquier entrada el algoritmo da la respuesta correcta con probabilidad ≥ p
- Si un Montecarlo para un **decisional** es p-correcto y **da siempre la respuesta correcta en uno de los dos casos** entonces mediante k repeticiones lo convertimos en

  $1-(1-p)^k$ -correcto

Algoritmos probabilistas

1. Introducción
2. Clasificación de los algoritmos probabilistas
3. Algoritmos numéricos
4. Algoritmos de Monte Carlo
5. **Algoritmos de Las Vegas**
   - Problema de las 8 reinas
   - Ordenación probabilista
   - Factorización de enteros

Algoritmos de Las Vegas. Introducción.

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

- Son aceptables si fallan con probabilidad baja.
- Si fallan, se vuelven a ejecutar con la misma entrada.

Algoritmos de Las Vegas. Introducción.

- 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 $p(x)$ (>0 para todo 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 soluciór
 y en ese caso s devuelve la solución encontrada}
```

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
```

Algoritmos de Las Vegas. Introducción.

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:

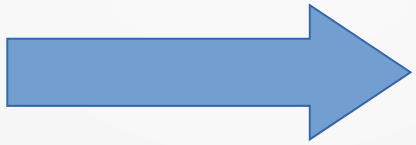
$$t(x) = p(x)v(x) + (1 - p(x))(f(x) + t(x))$$

$$t(x) = v(x) + f(x)(1 - p(x))/p(x)$$

Algoritmos de Las Vegas. Introducción.

$$t(x) = v(x) + f(x)(1 - p(x))/p(x)$$

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



Hay que optimizar esta función.

Algoritmos de Las Vegas. 8 reinas.

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

Algoritmo **determinista** (*Búsqueda con retroceso*)

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

Algoritmos de Las Vegas. 8 reinas.

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).

Algoritmos de Las Vegas. 8 reinas.

- 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. 8 reinas.

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.

Algoritmos de Las Vegas. 8 reinas.

Cuantas más reinas pongamos al azar:

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

Algoritmos de Las Vegas. 8 reinas.

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. 8 reinas.

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	
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	1 ms

Algoritmos de Las Vegas. 8 reinas.

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. 8 reinas.

Dimensiones mayores a 8:

Para 39 reinas en un tablero de dimensión 39.

Algoritmo determinista:

11402835415 nodos

41 **horas** en un computador

Algoritmos de Las Vegas. 8 reinas.

- Algoritmo Las Vegas, con 29 reinas al azar: (p=0,21)

$v \approx f \approx 100$ nodos $\Rightarrow$ $t \approx 500$ nodos

8,5 milisegundos (20×10^6 veces mejor)

- Algoritmo L.V. puro (39 reinas al azar): (p=0,0074)

150 milisegundos (10^6 veces mejor)

8 reinas, lo que hemos visto.

- Si colocamos k reinas de entre n al azar el tiempo esperado de respuesta es t_k la solución de:

$$t_k(x) = v_k(x) + f_k(x)\,(1-p_k(x)) \,/\, p_k(x)$$

$p_k(x)$ la probabilidad de respuesta si la entrada es x

$v_k(x)$ el tiempo esperado de ejecución si da respuesta

$f_k(x)$ el tiempo esperado si no da respuesta

- La idea sería elegir el k para el que el valor de t_k sea el menor
- En realidad puede haber ligeras divergencias porque no tarden lo mismo todas las instrucciones de alto nivel

Algoritmos de Las Vegas. Introducción.

Tipo *b*: Algoritmos de *Sherwood*

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

Ejemplo: Q*uickSort*.

Algoritmos de Las Vegas. Introducción.

- Coste peor $\Omega(n^2)$.
- 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:

Efecto *Robin Hood*:

"**Robar**" tiempo a los ejemplares "**ricos**" para dárselo a los "pobres".

Algoritmos de Las Vegas. Introducción.

- 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.

Algoritmos de Las Vegas. Ordenación probabilista

Ejemplo de algoritmo de Las Vegas “de tipo *b*” (algoritmo de Sherwood).

Recordar el método de ordenación de Hoare (*Divide y vencerás*)

Coste promedio: $O(n \log n)$

Coste peor: $\Omega(n^2)$

Algoritmos de Las Vegas. Ordenación probabilista

```
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
```

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) {Inicialmente T[i]=p}
{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

(En lugar de dos tramos, tres -teniendo en cuenta los que son iguales al pivote)

Algoritmos de Las Vegas. Ordenación probabilista

```
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) AND (i<=k<=m =>T[k]<p)
          AND  (r<=k<=d =>T[k]>p)}
      ordRápida(T,i,m);
      ordRápida(T,r,d)
  fsi
fin
```

Algoritmos de Las Vegas. Ordenación probabilista

Otro ejemplo del caso peor:

Secuencia ordenada en sentido contrario

(9, 8, 7, 6, 5, 4, 3, 2, 1)

Cualquier elección determinista de pivote tiene un caso peor cuadrático.

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,

Elegimos el **pivote al azar** en el intervalo `i..d`.

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

Algoritmos de Las Vegas. Ordenación probabilista

```
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) AND (i<=k<=m =>T[k]<p)
           AND (r<=k<=d=> T[k]>p)}
      ordRápida(T,i,m);
      ordRápida(T,r,d)
  fsi
fin
```

Algoritmos de Las Vegas. Ordenación probabilista

Fijaros en la sutil diferencia entre elegir como pivote el primer elemento o elegirlo al azar …

- Si eliges el primero claramente puedes equivocarte mucho en casos concretos (5, 4, 3, 2, 1)
- Si lo eliges al azar es imposible hacerlo mal siempre para un caso concreto (puedes equivocarte en una ejecución pero eso tiene probabilidad baja)

Empeoras los casos concretos en que elegir el primer elemento era lo mejor, pero mejoras los casos malos.

Resumen ordenación probabilista

- **Para cualquier entrada** el tiempo esperado es $O(n \log n)$
- Con la misma entrada distintas ejecuciones pueden tardar tiempo distinto, pero la mayoría estarán cerca del tiempo esperado

Algoritmos de Las Vegas. Factorización de enteros.

Ejemplo de algoritmo de Las Vegas “de tipo *a*”.

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.

Algoritmos de Las Vegas. Factorización de enteros.

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.

```
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
```

Algoritmos de Las Vegas. Factorización de enteros.

¡Solución ingenua!

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**.
- Partir un número *n* de 100 cifras: $\sqrt{(n)} = 7 \times 10^{49}$

Notas:

1) Número "duro" significa que es el producto de dos primos de tamaño parecido (justo los de RSA).
2) 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.

Nota: En 2009 se factorizó un número de 232 cifras usando cientos de máquinas durante 2 años

Algoritmos de Las Vegas. Factorización de enteros.

Existen varios algoritmos de Las Vegas para factorizar números grandes (véase [BrassardBratley96]).

Están basado en resultados avanzados de teoría de números

- Si $a^2 \bmod n = b^2 \bmod n$ $(a+b \neq n)$

  entonces mcd(a+b, n) es un divisor de n

Siguen teniendo costes altísimos

(factorizar un número de 100 cifras precisa del orden de 2×10^{15} operaciones).

Algoritmos de Las Vegas. Factorización de enteros.

- The RSA Challenge Numbers (oficialmente cerrado en 2007)
- http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-challenge-numbers.htm
- Se han factorizado 232 cifras en 2009 (en 2012, 212 cifras más difíciles)

RSA-640

Status: Factored

Decimal Digits: 193

3107418240490043721350750035888567930037346022842727545720161948823206440518081504556346829671723286782437916272838033415471073108501919548529007337724822783525742386454014691736602477652346609

Digit Sum: 806

RSA-704

Status: *Not Factored*

Decimal Digits: 212

7403756347956171282804679609742957314259318888923128908493623263897276503402826627689199641962511784399589433050212758537011896809828673317327310893090055250511687706329907239638078671008609696253793465056379635 9

Decimal Digit Sum: 1009

Algoritmos probabilistas

1. Introducción
2. Clasificación de los algoritmos probabilistas
3. Algoritmos numéricos
4. Algoritmos de Monte Carlo
5. Algoritmos de Las Vegas
   - Problema de las 8 reinas
   - Ordenación probabilista
   - Factorización de enteros

Conclusiones

- No es lo mismo una idea de cómo usar aleatoriedad que un algoritmo que funcione (es decir, con probabilidad de error baja en **todos los casos**)
- La dificultad suele estar en el análisis del error
- Para los **algoritmos numéricos** estimamos el tiempo en función de la precisión (aproximación)
- Los **algoritmos de Monte Carlo** pueden usar repetición para mejorar el error cuando una de las dos respuestas es siempre correcta (también en otros casos).
- Los **algoritmos de las Vegas** pueden usar repetición para aumentar la probabilidad de dar respuesta.