



# Análisis y diseño de algoritmos

---

## Sesión 13

# *Logro de la sesión*

*Al finalizar la sesión, el estudiante realiza un análisis y desarrollo de algoritmos probabilistas utilizando un lenguaje de programación*

# Agenda

---

- Características generales
- Clasificación de los algoritmos probabilistas
- Algoritmos probabilistas numéricos
- Algoritmos de Monte Carlo
- Algoritmos de Las Vegas

# Introducción

*Una historia sobre un tesoro, un dragón, un computador, un elfo y un intí.*

*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?**



# Introducción

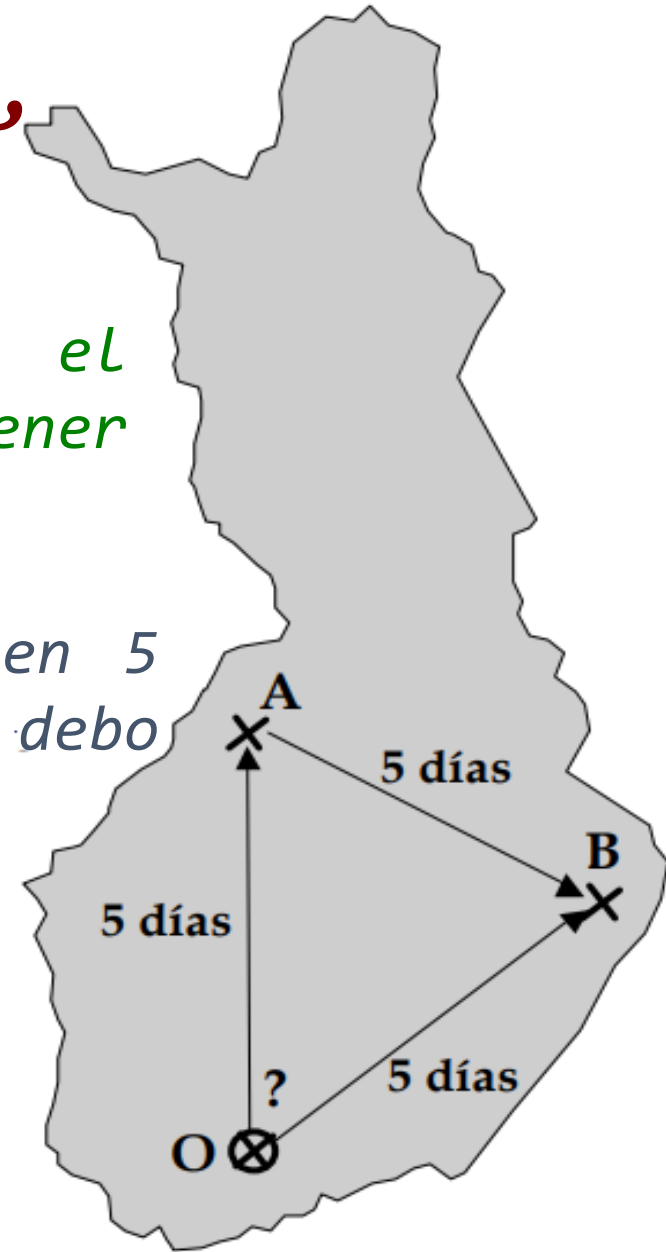
*Una historia sobre un tesoro, un dragón, un computador, un elfo y un intí.*

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

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

*Es mejor aceptar el trato pero...  
...¡hay una solución mejor!*

**¿Cuál?**



# Introducción

*Una historia sobre un tesoro, un dragón, un computador, un elfo y un inti.*

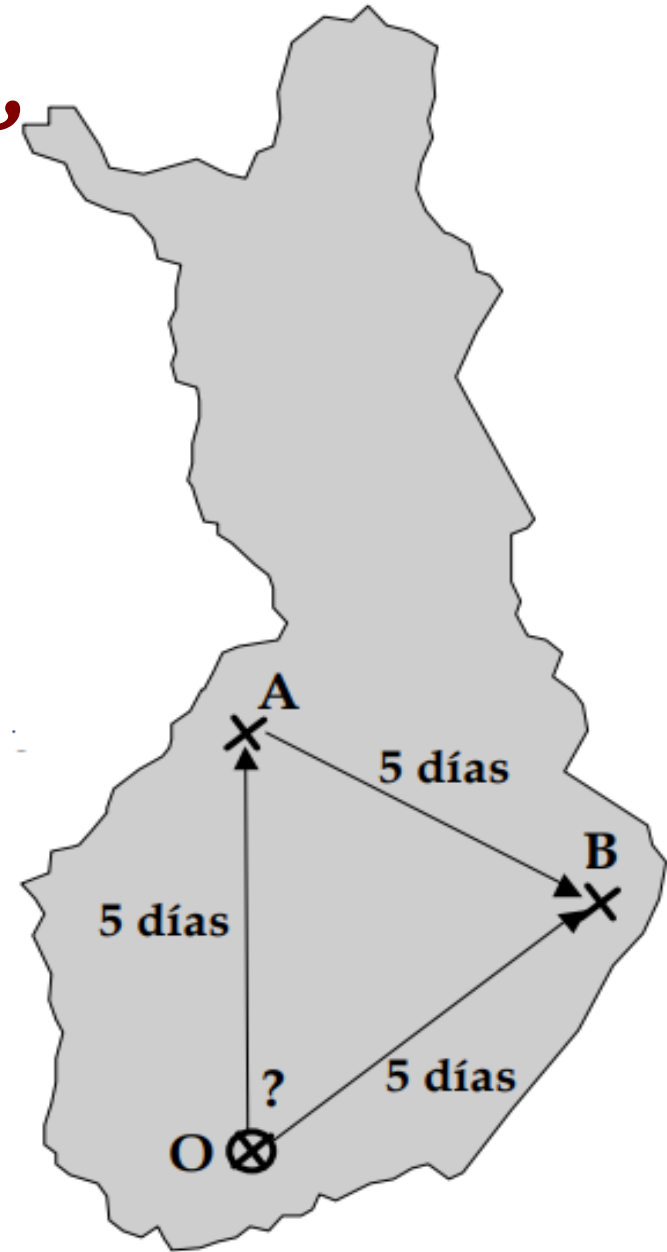
*¡Usar el inti que me queda en el bolsillo!*

*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  $x-7,5y$ .*



# Características generales

- En algunos algoritmos en los que aparece una **decisión**, es preferible a veces elegir aleatoriamente una de las posibles alternativas antes que perder tiempo calculando cuál de ellas es la mejor
- Esta situación se produce si el tiempo requerido para determinar la alternativa óptima es demasiado largo frente al promedio obtenido tomando la decisión al azar

## *Característica fundamental de Los algoritmos probabilistas:*

*Un algoritmo probabilista  $P$  puede comportarse de forma distinta cuando se aplica dos veces sobre los mismos datos de entrada*

# Características generales

## *Diferencias entre algoritmos deterministas y probabilistas:*

1. A un algoritmo **determinista** nunca se le permite que no termine.

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 y así tendremos una nueva oportunidad de éxito.

2. 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 las soluciones).

Un algoritmo **probabilista** puede encontrar soluciones diferentes ejecutándose varias veces con los mismos datos



# Características generales

## *Diferencias entre algoritmos deterministas y probabilistas:*

**3.** 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

**4.** 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

# Características generales

- 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 inadvertidos y que lleven a obtener una solución equivocada.

***Es decir, un algoritmo determinista tampoco garantiza siempre la certeza de la solución;***

*A veces la probabilidad de error en la respuesta obtenida por un algoritmo probabilista es menor que la de un fallo de hardware mientras se calcula la respuesta mediante un algoritmo determinista.*

# Características generales

*Por tanto, puede suceder que la respuesta incierta producida por un algoritmo probabilista sea:*

*Más rápida de calcular que la respuesta exacta de un algoritmo determinista.*

*Más fiable que la respuesta exacta de un algoritmo determinista*

En muchos casos es mejor un algoritmo probabilista rápido que dé la solución correcta con una cierta probabilidad de error.

*Ejemplo: decidir si un  $n^{\circ}$  de 1000 cifras es primo*

# Clasificación de los algoritmos probabilistas

## Algoritmos Probabilistas

Algoritmos que no garantizan la corrección de la solución

Algoritmos que nunca dan una solución incorrecta

### Algoritmos numéricos

Dan una solución aproximada

Cuanto más tiempo de ejecución se conceda a estos algoritmos, mejor sería la aproximación.

Ejemplo: “con probab. del 90% la respuesta es  $33 \pm 3$ ”

Son útiles si nos basta con una respuesta aproximada

### Algoritmos de Monte Carlo

Dan una respuesta exacta con una probabilidad muy grande. Pero también puede producir respuestas incorrectas.

No se puede saber si la respuesta producida es correcta o no

Se puede reducir la probabilidad de error si se alarga la ejecución

### Algoritmos de Las Vegas

Nunca dan una respuesta incorrecta

Verifican que la solución encontrada sea correcta. Si no es correcta lo indican.

Es posible volver a ejecutar el algoritmo con los mismos datos hasta obtener la solución correcta

# Clasificación de los algoritmos probabilistas

*“¿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).*

# Clasificación de los algoritmos probabilistas

*“¿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.*

# Clasificación de los algoritmos probabilistas

*“¿Cuándo descubrió América Cristobal Colón?”*

*Algoritmo de Las Vegas ejecutado diez veces:*

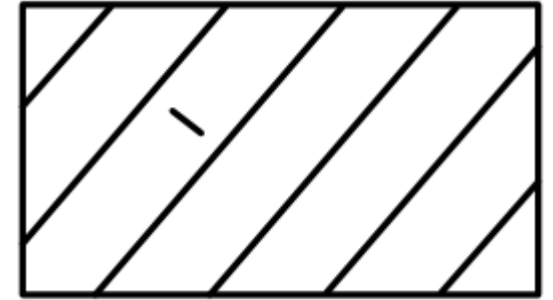
*1492, 1492, ¡Perdón!, 1492, 1492, 1492, 1492, 1492, ¡Perdón!, 1492.*

*El algoritmo nunca da una respuesta incorrecta.*

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

# Algoritmos numéricos: La aguja de Buffon

- Supongamos que se tira una aguja sobre el suelo
- Dicho suelo está hecho de planchas de madera de anchura constante.
- La separación entre las distintas planchas es  $\theta$ .
- La longitud de la aguja es la *mitad* de la anchura de las planchas
- En el siglo XVIII, el conde de Buffon demostró que la probabilidad de que la aguja caiga encima de una rendija es  $1/\pi$
- Así, este teorema sirve para predecir de forma aproximada cuántas agujas caerán entre dos planchas si dejamos caer  $n$  agujas:  $n/\pi$



**Más interesante: sirve para calcular el valor de  $\pi$**

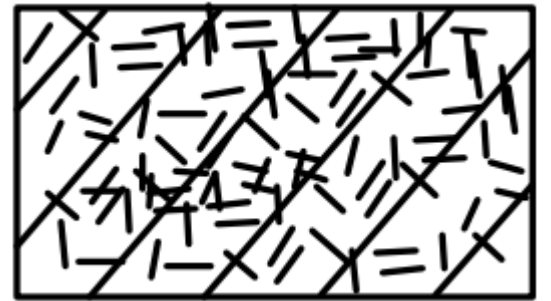
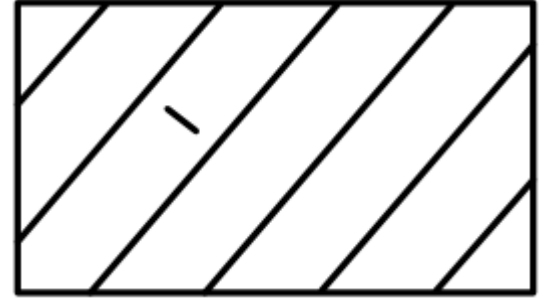
- Se dejan caer  $n$  agujas y se cuentan el número de agujas  $k$  que han caído entre dos planchas.
- Se puede utilizar  $n/k$  como aproximación de  $\pi$



# Algoritmos numéricos: La aguja de Buffon

## *Teorema de Buffon*

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



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  $\pi$ :

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

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

# Algoritmos numéricos: La aguja de Buffon

**Pregunta natural: ¿Es útil este método?**

*Cuanto más precisa se quiera la aproximación, más agujas de deben dejar caer*

*La convergencia es muy lenta: para obtener un valor de  $\pi \pm 0,01$  con probabilidad 0,9 se necesitan tirar 1.500.000 agujas*

*En la práctica, este mecanismo para calcular el valor aproximado de  $\pi$  no es útil, porque se pueden obtener aproximaciones de  $\pi$  mucho mejores utilizando algoritmos deterministas*

# ALGORITMO

## LA AGUJA DE BUFFÓN

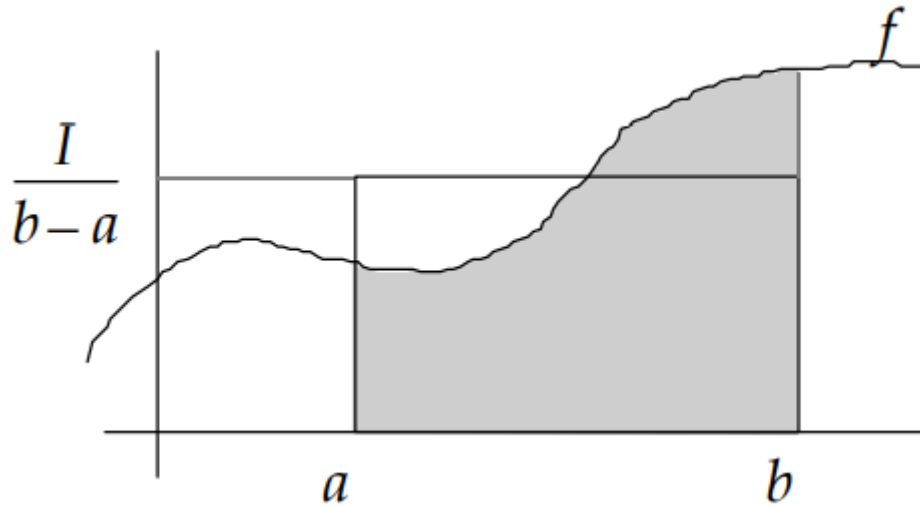
# SOLUCIÓN:



```
1 package Semana13;
2 // Tomar una aguja de 1cm y un papel con líneas paralelas a 2cm
3 // Lanzar la aguja sobre el papel, cada vez que la aguja corta una línea
4 // cuenta como un acierto
5 // pi (aprox) = intentos / aciertos
6 import java.util.Random;
7 import java.util.Scanner;
8 public class BuffonPiEstimation {
9     public static void main(String[] args) {
10         // TODO code application logic here
11         System.out.println("Aguja de Buffon");
12         System.out.println("Introducir número de lanzamientos: ");
13         Scanner in = new Scanner(System.in);
14         int tries = in.nextInt();
15         Random generator = new Random(42);
16         int hits = 0;
17         for (int i = 0; i < tries; i++) {
18             // Generar yLow la distancia de un extremo de la aguja a la línea
19             double yLow = generator.nextDouble() * 2;
20             // Generar un ángulo entre 0 y 180 grados
21             double angle = generator.nextDouble() * 180;
22             // Obtener la altura del otro extremo de la aguja
23             double yHigh = yLow + Math.sin(Math.toRadians(angle));
24             // Si el extremo está a mayor altura que la siguiente línea
25             // es un acierto
26             if (yHigh >= 2) {
27                 hits++;
28             }
29         }
30         double piEstimate = (float) tries / hits;
31         System.out.println();
32         System.out.println(piEstimate);
33     }
34 }
```

# Algoritmos numéricos: Integración numérica

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



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

*Sea el rectángulo de base  $b - a$  y de altura  $I/(b-a)$*

*El área de la superficie bajo la curva también es  $I$*

*Con lo anterior podemos concluir que el rectángulo y la curva tienen la misma altura media*

# Algoritmos numéricos: Integración numérica

*Se puede hacer una estimación de la altura media  $A_m$  de la curva por muestreo aleatorio*

*Una aproximación de la integral sería en este caso*  
$$I = A_m \times (b - a)$$

*Esto también se podría haber hecho con un algoritmo determinista, que calcule el valor de  $f(x)$  a intervalos regulares.*

*Sin embargo, esto no funciona en todos los casos.*

```
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 numérica

## *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
```

*En general, la versión determinista es más eficiente (menos iteraciones para obtener precisión similar).*

# Algoritmos numéricos: Integración numérica

- Pero, para todo algoritmo determinista de integración puede construirse una función que “lo vuelve loco” (no así para la versión probabilista).
  - Por ejemplo, para  $f(x) = \sin^2(100! \pi x)$  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 de algoritmos probabilistas: 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



# ALGORITMO

# INTEGRACIÓN NUMÉRICA

(VERSIÓN PROBABILÍSTICA)

# SOLUCIÓN:



**PYTHON**

## Integración Numérica

Para realizar este algoritmo, nos basaremos de la imagen mostrada para determinar su integral.

$$\int_{0.8}^3 \frac{1}{1 + \sinh(2x) \log^2(x)} dx = 0.67684$$

# SOLUCIÓN:

Podemos visualizar el programa completo en relación a la integración numérica.

Asimismo, para un mejor entendimiento, el mismo programa generará la gráfica de la determinada función.

## Codificación del programa

```
INTEGRACION NUMERICA - VERSION ALGORITMO PROBABILISTICO
"""
import numpy as np
import matplotlib.pyplot as plt
from numpy.random import uniform as unif

cant_num=100 #cantidad de numeros aleatorios que generamos

#Estas dos lineas servirá para realizar la grafica
c=np.linspace(0.0001,3.2)
f=1/(1+np.sinh(2*c)*(np.log(c))**2)

#Aqui genero numeros aleatorios especificamente en este intervalo
lim_inf=0.8
lim_sup=3
x=unif(lim_inf,lim_sup,cant_num)

#inicializamos la variable de la sumatoria
suma=0

for j in range(cant_num):
    suma= suma + 1/(1+np.sinh(2*x[j])*(np.log(x[j]))**2)

resultado=(lim_sup-lim_inf)*suma/cant_num

plt.xlabel('x')
plt.ylabel('y')
plt.plot(c,f)
plt.hist(x,density=True)

print('=====')
print('      INTEGRACIÓN NUMERICA      ')
print('=====')
print('El resultado de la integral es: ')
print(resultado)
```

# SOLUCIÓN:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.random import uniform as unif
```

- Agregamos la cantidad de números aleatorios para el programa.
- Las siguientes dos líneas servirán para la realización de la gráfica en el IDE Spyder.
- Colocamos el límite superior e inferior.
- Luego, generamos el arreglo de números aleatorios, teniendo en cuenta en base a la cantidad de números (100).
- Asimismo, para este caso que el número más pequeño puede ser 0.8 y el más grande el número 3.

## Explicación del código

- Importamos para poder utilizar la función seno hiperbólico y logarítmico.
- Importamos para realizar la gráfica.
- Generación de números aleatorios.

```
cant_num=100 #cantidad de numeros aleatorios que generamos

#Estas dos lineas servirán para realizar la grafica
c=np.linspace(0.0001,3.2)
f=1/(1+np.sinh(2*c)*(np.log(c))**2)

#Aqui genero numeros aleatorios especificamente en este intervalo
lim_inf=0.8
lim_sup=3
x=unif(lim_inf,lim_sup,cant_num)
```

# SOLUCIÓN:

## Explicación del código

```
#inicializamos la variable de la sumatoria
suma=0
for j in range(cant_num):
    suma= suma + 1/(1+np.sinh(2*x[j]))*(np.log(x[j]))**2)

resultado=(lim_sup-lim_inf)*suma/cant_num

plt.xlabel('x')
plt.ylabel('y')
plt.plot(c,f)
plt.hist(x,density=True)

print('=====')
print('      INTEGRACIÓN NUMERICA      ')
print('=====')
print('El resultado de la integral es: ')
print(resultado)
```

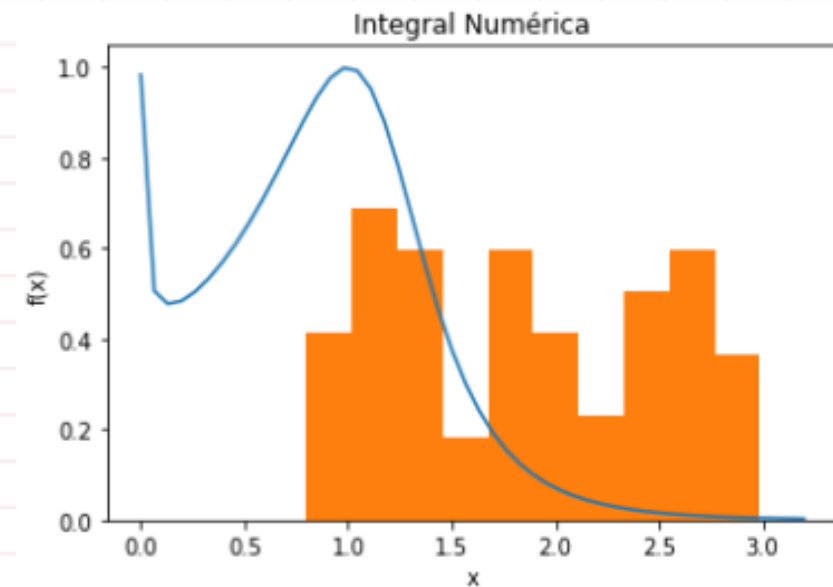
- Inicializamos la variable de la sumatoria (contador que mientras se vaya ingresando más términos, se va a ir haciendo más exacta esta aproximación.
- Aplicamos el ciclo for, la cual va a ir aumentando el contador j hasta la cantidad de números aleatorios que voy a generar.
- Evaluamos la función en base a los números aleatorios generados, para luego acumularse en “suma”.
- Luego, colocamos el resultado y realizamos la gráfica de la función para que se visualice.

# SOLUCIÓN:

## Ejecución del programa

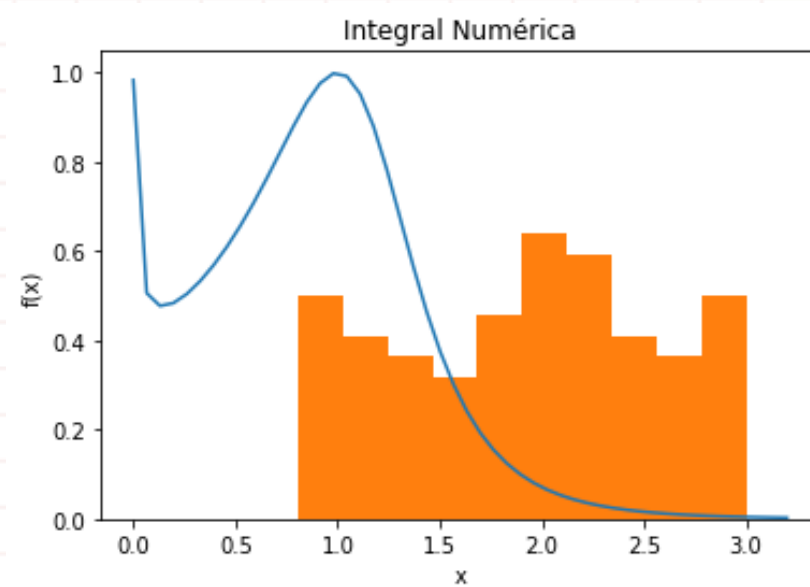
### PRUEBA 1:

```
=====
INTEGRACIÓN NUMERICA
=====
El resultado de la integral es:
0.7528332544226936
```



### PRUEBA 2:

```
=====
INTEGRACIÓN NUMERICA
=====
El resultado de la integral es:
0.6406091246758711
```



Realizamos dos pruebas para determinar la integral numérica de la función, obteniendo resultados cercanos al resultado original.

# ALGORITMO

# INTEGRACIÓN NUMÉRICA

(VERSIÓN DETERMINISTA)

# SOLUCIÓN:

Utilizamos como lenguaje de programación



**PYTHON**

## Integración Numérica

Para realizar este algoritmo, nos basaremos de la imagen mostrada para determinar su integral.

Para integrar la función en el intervalo  $[1,3]$  con 64 tramos.

$$f(x) = \sqrt{x} \sin(x)$$



# SOLUCIÓN:

## Codificación del programa

```
"""
INTEGRACION NUMERICA - VERSION ALGORITMO DETERMINISTA
"""
# Usando una función fx()
import numpy as np
import matplotlib.pyplot as plt

# INGRESO
fx = lambda x: np.sqrt(x)*np.sin(x)

# intervalo de integración
a = 1
b = 3
tramos = 4

# PROCEDIMIENTO
# Usando tramos equidistantes en intervalo
h = (b-a)/tramos
xi = a
suma = fx(xi)
for i in range(0,tramos-1,1):
    xi = xi + h
    suma = suma + 2*fx(xi)
suma = suma + fx(b)
area = h*(suma/2)

# SALIDA
print('=====')
print('      INTEGRACIÓN NUMERICA      ')
print('=====')
print('Número de Tramos: ', tramos)
print('El resultado de la integral es: ', area)
```

Este código permite obtener la integral de la función junto con su gráfica.

```
# GRAFICA
# Puntos de muestra
muestras = tramos + 1
xi = np.linspace(a,b,muestras)
fi = fx(xi)

# Línea suave
muestraslinea = tramos*10 + 1
xk = np.linspace(a,b,muestraslinea)
fk = fx(xk)

# Graficando
plt.plot(xk,fk, label = 'f(x)')
plt.plot(xi,fi, marker='o',
         color='orange', label = 'muestras')

plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Integración Numérica')
plt.legend()

# Trapecios
plt.fill_between(xi,0,fi, color='g')
for i in range(0,muestras,1):
    plt.axvline(xi[i], color='w')

plt.show()
```

# SOLUCIÓN:

## Explicación del código

```
import numpy as np
import matplotlib.pyplot as plt
```

Definimos la función.

El cálculo se realizará en un intervalo de  $[a,b]$  (en este caso es de 1 a 3, respectivamente) con una cantidad de tramos igual a 4.

```
# PROCEDIMIENTO
# Usando tramos equidistantes en intervalo
h = (b-a)/tramos
xi = a
suma = fx(xi)
for i in range(0,tramos-1,1):
    xi = xi + h
    suma = suma + 2*fx(xi)
suma = suma + fx(b)
area = h*(suma/2)
```

Importamos numpy para realizar los cálculos y matplotlib; para realizar la gráfica.

```
# INGRESO
fx = lambda x: np.sqrt(x)*np.sin(x)

# intervalo de integración
a = 1
b = 3
tramos = 4
```

En la sección del procedimiento, vamos a emplear los tramos equidistantes en el intervalo.

# SOLUCIÓN:

## Explicación del código

```
# SALIDA
print('=====')
print('      INTEGRACIÓN NUMERICA      ')
print('=====')
print('Número de Tramos: ', tramos)
print('El resultado de la integral es: ', area)
```

Para realizar la gráfica, debemos tener en cuenta los puntos de muestra. Para ello, definimos las muestras en base a los tramos en toda la función en un determinado intervalo. Además, realizamos la codificación para las líneas. Finalmente, graficamos la integral numérica de la función.

Imprimimos el número de tramos y la área calculada.

```
# GRAFICA
# Puntos de muestra
muestras = tramos + 1
xi = np.linspace(a,b,muestras)
fi = fx(xi)

# Línea suave
muestraslinea = tramos*10 + 1
xk = np.linspace(a,b,muestraslinea)
fk = fx(xk)

# Graficando
plt.plot(xk,fk, label = 'f(x)')
plt.plot(xi,fi, marker='o',
         color='orange', label = 'muestras')

plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Integración Numérica')
plt.legend()

# Trapecios
plt.fill_between(xi,0,fi, color='g')
for i in range(0,muestras,1):
    plt.axvline(xi[i], color='w')

plt.show()
```

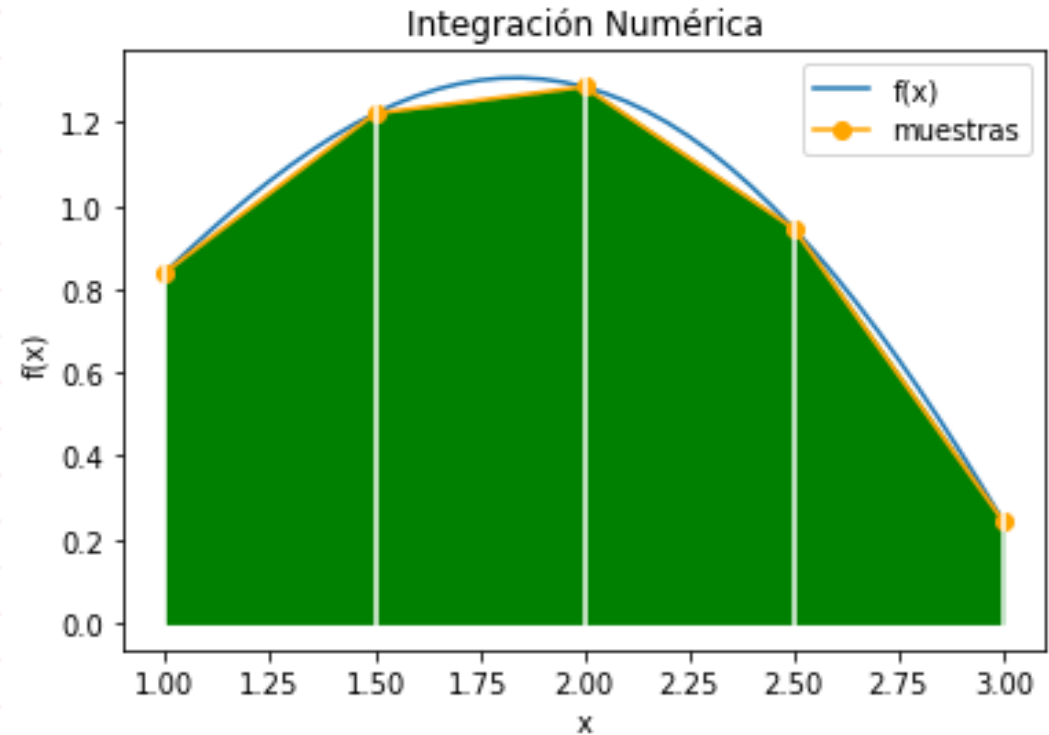
# SOLUCIÓN:

## Ejecución del programa

### PRUEBA 1:

```
=====
INTEGRACIÓN NUMERICA
=====
Número de Tramos: 4
El resultado de la integral es: 1.9984170862312691
```

Realizamos la prueba para determinar la integral numérica de la función en base a la versión determinista que tiene mejores resultados.



# Algoritmos de Montecarlo

- *Hay problemas para los que no se conocen soluciones deterministas ni probabilistas que den siempre una solución correcta (ni siquiera aproximada)*
- *Los algoritmos de Monte Carlo dan a veces una solución incorrecta, y con una probabilidad conocida una solución correcta, sea cual sea la entrada.*

*Para estos algoritmos es necesaria una definición de corrección probabilista:*

- *Un algoritmo de Monte Carlo es  $p$ -correcto, con  $0 < p < 1$ , si devuelve una solución correcta con probabilidad mayor o igual que  $p$ , cualesquiera que sean los datos de entrada.*

*A veces,  $p$  depende del tamaño de la entrada, pero no de los datos concretos del ejemplar.*

# 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  $C = AB$ .

La solución más sencilla (*trivial*) consiste en realizar la multiplicación de matrices y comparar el resultado con  $C$

El inconveniente de este enfoque es el coste: La multiplicación tradicional está en  $O(n^3)$ . Con Strassen, en  $O(n^{2,81})$ , y con el mejor método conocido hasta ahora para multiplicar matrices, en  $O(n^{2,37})$ .

Vamos a estudiar un algoritmo de Monte Carlo que resuelve este problema en  $O(n^2)$ , para una probabilidad de error dada  $\varepsilon$ .

# Algoritmos de Monte Carlo: Verificación de un producto matricial

- Supongamos que  $C \neq AB$ . Entonces,  $D = AB - C$ , donde  $D$  no es una matriz nula.
- También  $XD = X(AB - C) = XAB - XC$ , es decir, se trata de decidir si  $XAB = XC$  o no para un vector binario  $X$  elegido al azar.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{pmatrix} \quad C = \begin{pmatrix} 11 & 29 & 37 \\ 29 & 65 & 91 \\ 47 & 99 & 45 \end{pmatrix}$$

- ▶ si  $X = (1, 1, 0)$  entonces  $(XA)B = (40, 94, 128)$ ,  
 $XC = (40, 94, 128)$ , y la respuesta es **true**
- ▶ si  $X = (0, 1, 1)$  entonces  $(XA)B = (76, 166, 236)$ ,  
 $XC = (76, 164, 136)$  y la respuesta es **false**



# Algoritmos de Monte Carlo: Verificación de un producto matricial

- Supongamos que  $C \neq AB$ . Entonces,  $D = AB - C$ , donde  $D$  no es una matriz nula. Sea  $i$  la fila  $i$ -ésima de  $D$  que contiene al menos un elemento distinto de 0.  $D_i \neq \vec{0}$ .
- Sea  $S$  un subconjunto **cualquiera** de índices de filas:  $S \subseteq \{1, \dots, n\}$ .
- Sea  $\Sigma_S(D)$  la suma de las filas de  $D$  cuyos índices están en  $S$   
 $\Sigma_S(D) = \sum_{i \in S} D_i \quad (\Sigma_{\emptyset}(D) = \vec{0})$
- Sea  $S'$  un conjunto igual a  $S$ , excepto por el elemento  $i$ :
$$S' = \begin{cases} S \cup \{i\} & \text{si } i \notin S \\ S \setminus \{i\} & \text{si } i \in S \end{cases}$$
- Como  $D$  no es la matriz nula,  $\Sigma_S(D)$  y  $\Sigma_{S'}(D)$  no pueden ser nulos simultáneamente.



# Algoritmos de Monte Carlo: Verificación de un producto matricial

- *Podemos generar el conjunto  $S$  de forma aleatoria:*
  - ▶ Para cada  $j$  entre 1 y  $n$ , decidimos al azar si  $j$  se incluye en  $S$  o no.
  - ▶ La probabilidad de que  $i$  esté en  $S$  es de  $1/2$ .
  - ▶ Por tanto, la probabilidad de que  $\Sigma_S(D) \neq \mathbf{0}$  es como mínimo  $1/2$ .
- *Por otra parte,  $\Sigma_S(D)$  siempre es cero si  $C = AB$ .*
- *La solución está en calcular  $\Sigma_S(D)$  y compararlo con  $\mathbf{0}$ . Pero debemos evitar calcular  $D$ , pues el coste del producto  $AB$  es demasiado alto.*
- *¿Cómo calcular eficientemente  $\Sigma_S(D)$ ?*
  - *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:  $\Sigma_S(D) = XD$*

# Algoritmos de Monte Carlo: Verificación de un producto matricial

- Pero  $XD = X(AB - C) = XAB - XC$ , es decir, se trata de decidir si  $XAB = XC$  o no para un vector binario  $X$  elegido al azar.
- La agrupación apropiada de productos de matrices puede mejorar mucho la eficiencia del producto de matrices
- En concreto, el coste del cálculo de  $XAB = (XA)B$  y de  $XC$  es  $O(n^2)$ .
- Un algoritmo que calcule esto puede ser de la forma:

```
fun verifmat(A, B, C, n)
  desde j ← 1 hasta n hacer
    X[j] ← uniforme_entero(0..1)
  fin desde
  si (XA)B = XC entonces devolver cierto
  si no devolver falso
fin fun
```

# Algoritmos de Monte Carlo: Verificación de un producto matricial

- *Este algoritmo devuelve cierto como respuesta correcta con una probabilidad del 50 %.*
- *No parece demasiado interesante: tirar una moneda al aire para decidir si  $C = AB$  también tiene una probabilidad del 50 %... ¡Y sin siquiera mirar los valores de las matrices!*
- *La clave: Siempre que  $\text{verifmat}(A, B, C, n)$  devuelve falso, podemos estar seguros de que  $AB \neq C$ . Sólo cuando devuelve cierto, no sabemos la respuesta.*
- *Esto nos da una pista sobre cómo podemos tener más certeza sobre la respuesta afirmativa:*

# Algoritmos de Monte Carlo: Verificación de un producto matricial

- *Podemos aplicar `verifmat(A, B, C, n)` varias veces sobre el mismo caso, utilizando valores independientes para el vector  $X$  cada vez.*
- *Es suficiente que `verifmat(A, B, C, n)` devuelva falso una sola vez para decidir que  $AB \neq C$ .*
- *Podemos utilizar el siguiente algoritmo:*

```
fun repetir_verifmat(A, B, C, n, k)
  desde i ← 1 hasta k hacer
    si !verifmat(A, B, C, n) entonces devolver falso
  fin desde
  devolver cierto
fin fun
```

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

# Algoritmos de Monte Carlo: Verificación de un producto matricial

- *Vamos a analizar la probabilidad de error de este algoritmo.*
- *Si  $AB = C$ , toda llamada a `verifmat(A,B,C,n)` devuelve cierto, por lo que repetir `verifmat(A,B,C,n,k)` también devuelve cierto. En este caso, la probabilidad de error es  $0$ .*
- *Si  $AB \neq C$ , la probabilidad de que una llamada a `verifmat(A,B,C,n)` devuelva cierto incorrectamente es  $1/2$ .*
- *Dos llamadas independientes a `verifmat(A, B, C, n)` que devuelvan cierto de forma incorrecta tienen una probabilidad conjunta de  $1/4$ .*


# Algoritmos de Monte Carlo: Verificación de un producto matricial

- En el caso general, La probabilidad de que  $k$  llamadas consecutivas a `verifmat(A, B, C, n)` devuelvan cierto incorrectamente es  $2^{-k}$
- Dicho de otra forma, La probabilidad de que el algoritmo anterior devuelva un resultado correcto es  $1 - 2^{-k}$ : el algoritmo es  $(1 - 2^{-k})$ -correcto.
- Para  $k = 10$ , es 99,9 %-correcto
- Para  $k = 20$ , La probabilidad de error es menor a una entre un millón.
- La complejidad de este algoritmo está en  $\Theta(n^2k)$

# Algoritmos de Monte Carlo: Verificación de un producto matricial

- *Teniendo en cuenta que se realizan  $3n^2$  multiplicaciones escalares para calcular  $XAB$  y  $XC$ , si  $k = 20$  es necesario realizar  $60n^2$  multiplicaciones.*
  - *Por tanto, este método es más eficiente que el producto tradicional de matrices si la dimensión de las matrices  $n$  es mayor a 60.*
- ❖ 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



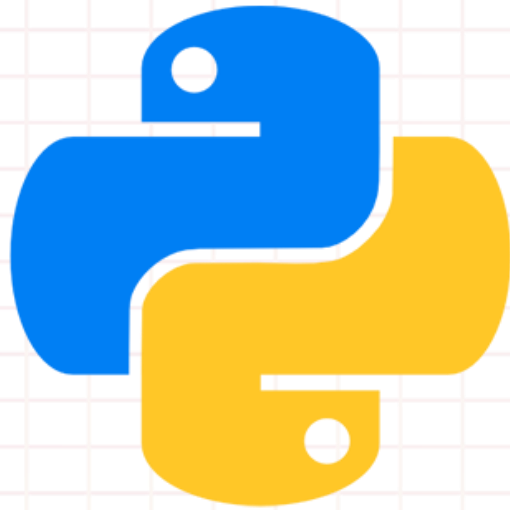
el decrecimiento de la probabilidad de error es espectacular repitiendo la prueba varias veces.

# ALGORITMO VERIFICACIÓN DE UN PRODUCTO MATRICIAL



# SOLUCIÓN:

Creamos las matrices a ser evaluadas por el programa y la condición a imprimir.



# PYTHON

```
if __name__ == "__main__":  
    a = [[1,2,3], [4,5,6], [7,8,9]]  
    b = [[3,1,4], [1,5,9], [2,6,5]]  
    c = [[11,29,37], [29,65,91], [47,99,45]]  
  
    print("Con 3 repeticiones: ", repetir_verifmat(a, b, c, 3))
```

# SOLUCIÓN:

Definir verificar para analizar la verdad o falsedad

Para la verificación tenemos dos posibles respuestas. En el caso que retorna Falso, es seguro que  $AB \neq C$ .

En cambio para  $AB = C$ , la función utiliza `numpy` para devolver necesariamente el valor de verdad.

```
from random import randint
import numpy

def verificar(a, b, c, x):
    '''XAB == XC (se utiliza numpy)'''
    xc = numpy.dot(x, c) #devuelve XC
    xa = numpy.dot(x, a)
    xab = numpy.dot(xa, b)

    if (xab == xc).all(): #XAB == XC
        return True
    else:
        return False
```

# SOLUCIÓN:

Realizamos el llamado a `repetir_verifmat` el cual repetirá la verificación `k` veces, es suficiente que retorne 1 vez `False` para asegurar que es falso.

## Ejecución del programa

```
Con 3 repeticiones: False  
> |
```

## Implementación de funciones `verifmat` y `repetir_verifmat`

```
def verifmat(a, b, c):  
    '''AB == C'''  
    x = []  
    for i in range(len(a)):  
        x.append(randint(0,1))  
    #creo un vector X = [0-1, 0-1 , 0-1] de forma random del tamaño a  
  
    return verificar(a, b, c, x) #comparo xc == xab  
  
def repetir_verifmat(a, b, c, k):  
    '''Si solo 1 vez da false sera falso'''  
    for i in range(k+1):  
        if not verifmat(a, b, c):  
            return False  
    return True
```

Retorna la comprobación matricial con las repeticiones pedidas (3 en este caso).

# Algoritmos de Monte Carlo: Comprobación de primalidad

- *Un problema muy relevante que se puede resolver con el método de Monte Carlo es el de determinar si un número es primo.*
- *Es el algoritmo de Monte Carlo más conocido. Este problema es especialmente útil para las técnicas criptográficas, que se basan en la factorización de números muy grandes en factores primos.*
- *No se conoce ningún algoritmo determinista que resuelva este problema en un tiempo razonable para números muy grandes (cientos de dígitos decimales).*

# Algoritmos de Monte Carlo: Comprobación de primalidad

- *La verificación probabilista de primalidad de un número se basa en el teorema menor de Fermat(1640):*

Sea  $n$  un número primo. Entonces  $a^{n-1} \bmod n = 1$  para cualquier entero  $a$  tal que  $1 \leq a \leq n - 1$

- *Por ejemplo, sean  $n = 7$  y  $a = 5$ . Entonces,  $a^{n-1} = 5^6 = 15625 = 2232 * 7 + 1$*
- *Utilizaremos la versión contrapositiva de este teorema:*

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

# Algoritmos de Monte Carlo: Comprobación de primalidad

- Existe un algoritmo basado en la técnica Divide y Vencerás que permite realizar la exponenciación modular (resolver  $a^n \bmod z$ ) que está en  $\Theta(\log n)$ . Utilizando dos propiedades elementales de aritmética modular:

$$xy \bmod z = [(x \bmod z) \times (y \bmod z)] \bmod z$$
$$(x \bmod z)^y \bmod z = x^y \bmod z$$

```
fun expomod(a, n, z) // calcula  $a^n \bmod z$ 
   $i \leftarrow n$  ;  $r \leftarrow 1$  ;  $x \leftarrow a \bmod z$ 
  mientras  $i > 0$  hacer
    si  $i$  es impar entonces  $r \leftarrow rx \bmod z$ 
     $x \leftarrow x^2 \bmod z$ 
     $i \leftarrow \lfloor i/2 \rfloor$ 
  fin mientras
  devolver  $r$ 
fin fun
```

- Vamos a utilizar este algoritmo para dar una versión probabilista de la comprobación de primalidad

# Algoritmos de Monte Carlo: Comprobación de primalidad

- *Para verificar que un número  $n$  es primo, habría que comprobar que todos los valores  $a$  entre 1 y  $n-1$  cumplen que  $a^{n-1} \bmod n = 1$ .*
- *Podríamos probarlo con un solo número elegido al azar entre 1 y  $n - 1$*
- *La primera versión de nuestro algoritmo probabilista es:*

```
fun Fermat(n)
  a ← uniforme_entero(1..n-1)
  si expomod(a,n-1,n)=1 entonces devolver cierto
  devolver falso
fin fun
```

- *Como en el caso anterior, cuando  $\text{Fermat}(n)$  devuelva falso estaremos seguros de que  $n$  no es primo.*
- *Sin embargo, si  $\text{Fermat}(n)$  devuelve cierto no podemos concluir nada ...*

# Algoritmos de Monte Carlo: Comprobación de primalidad

- *Ejemplos:*
  - ▶ Por ejemplo,  $(1)^{n-1} \bmod n = 1$  para todo  $n \geq 2$ .
  - ▶ Por ejemplo,  $(n-1)^{n-1} \bmod n = 1$  para todo  $n \geq 3$ .
- *Dado un número  $n$  no primo, los valores de  $a$  tales que  $a^{n-1} \bmod n = 1$  se denominan testigos falsos de primalidad*
- *Modificación del algoritmo de Fermat: Elegir  $a$  entre 2 y  $n-2$*
- *Existen casos no triviales ( $a \neq 1, a \neq n-1$ ) en los que falla el algoritmo anterior. Ejemplo:  $4^{14} \bmod 15 = 1$  sin embargo 15 no es primo.*
- *Afortunadamente, no existen demasiados testigos falsos: La media de error de Fermat( $n$ ) para los impares menores a 1000 es inferior al 3,3 %.*



# Algoritmos de Monte Carlo: Comprobación de primalidad

- *Sin embargo, hay excepciones:*
  - 561 admite 318 testigos falsos.
  - Aún peor: `Fermat(651693055693681)` devuelve cierto con una probabilidad mayor al 99,9965 %, pero este número no es primo
- *Además, el algoritmo anterior no es  $p$ -correcto para ningún  $p > 0$ : no es posible reducir la probabilidad de error repitiendo la ejecución del algoritmo.*
- *Tenemos que buscar otra manera de enfrentarnos a este problema.*
- *Vamos a utilizar otro algoritmo basado en una extensión del teorema de Fermat.*

# Algoritmos de Monte Carlo: Comprobación de primalidad

- **Necesitamos la siguiente definición:**

- Sea  $n > 4$  un entero impar, y sean  $s$  y  $t$  enteros tales que  $n - 1 = 2^s t$ , con  $t$  impar. Definimos el conjunto  $B(n)$  de la forma siguiente:

$a \in B(n)$  si y sólo si  $2 \leq a \leq n - 2$  y además

- ▶  $a^t \bmod n = 1$ , o bien

- ▶ Existe un entero  $i$ ,  $0 \leq i < s$  tal que  $a^{2^i t} \bmod n = n - 1$

- **Por ejemplo, vamos a comprobar si  $a = 158$  está en  $B(289)$ .**

- Podemos comprobar que  $s = 5$  y  $t = 9$  cumplen las propiedades anteriores:  $t$  es impar y  $2^5 \cdot 9 = 289 - 1$

- $a^t \bmod n = 158^9 \bmod 289 = 131$ .

- No se cumple la primera parte de la definición. Vamos a ver si se cumple la segunda.

- Probamos diversos valores de  $i$ ,  $0 < i < 5$ :

$$a^{2^1 t} \bmod n = 158^{2 \cdot 9} \bmod 289 = 110$$

$$a^{2^2 t} \bmod n = 158^{4 \cdot 9} \bmod 289 = 251$$

$$a^{2^3 t} \bmod n = 158^{8 \cdot 9} \bmod 289 = 288 = 289 - 1$$

# Algoritmos de Monte Carlo: Comprobación de primalidad

## Teorema

Consideremos un número impar arbitrario  $n > 4$ .

- Si  $n$  es primo, entonces  $B(n) = \{a \mid 2 \leq a \leq n - 2\}$ .
- Si  $n$  no es primo, entonces  $|B(n)| \leq (n - 9)/4$ .

- ***A diferencia del teorema menor de Fermat,***
  - *Aunque pueden existir valores de  $a \in B(n)$  que cumplen las condiciones anteriores y  $n$  sea compuesto,*
  - *Se garantiza que la proporción de números entre 2 y  $n - 2$  que están en  $B(n)$  es pequeña para todo  $n$  compuesto.*
- *Calcular  $B(n)$  puede parecer complicado, pero se puede implementar fácilmente de forma eficiente*

# Algoritmos de Monte Carlo: Comprobación de primalidad

- *El algoritmo probabilista que permite determinar la primalidad según este teorema es el siguiente:*

*//algoritmo de Miller-Rabin*

*fun MillerRabin(n)*

*// n debe ser impar*

*// y mayor a 4*

*a ← uniforme\_entero(2..n-2)*

*devolver FermatB(a,n)*

*fin fun*

**fun** FermatB(a, n)

**s** ← 0 ; **t** ← n-1

**repetir**

**s** ← s+1 ; **t** ← ⌊t/2⌋

**hasta** t mod 2 = 1

**x** ← expomod(a, t, n)

**si** x=1 ∨ x=n-1 **entonces devolver** cierto

**desde** i ← 1 **hasta** s-1 **hacer**

**x** ← x<sup>2</sup> mod n (\*)

**si** x=n-1 **entonces devolver** cierto

**fin desde**

**devolver** falso

**fin fun**

- (\*) se puede demostrar que  $(x^a \bmod n)^b \bmod n = x^{a \cdot b} \bmod n$
- Este algoritmo es 3/4-correcto para la comprobación de primalidad.
- Además, garantiza que la respuesta falso es correcta. Por tanto, repitiendo el algoritmo se puede reducir rápidamente la probabilidad de error.

# Algoritmos de Monte Carlo: Comprobación de primalidad

- *El algoritmo que realiza sucesivas Llamadas al anterior es:*

```
fun RepetirMillerRabin(n,k) // n debe ser impar y mayor a 4
  desde i ← 1 hasta k hacer
    si !MillerRabin(n) entonces devolver falso
  fin desde
  devolver cierto
fin fun
```

- *Este algoritmo devuelve siempre la respuesta correcta cuando n es primo.*
- *Cuando n no es primo, la probabilidad de que MillerRabin(n) devuelva la respuesta incorrecta es de 1/4.*
- *La probabilidad de que la respuesta sea incorrecta después de repetir la prueba k veces es  $4^{-k}$ : RepetirMillerRabin(n,k) es  $(1 - 4^{-k})$ -correcto.*
- *Por tanto, basta con tomar  $k = 10$  para que la probabilidad sea inferior a una entre un millón.*

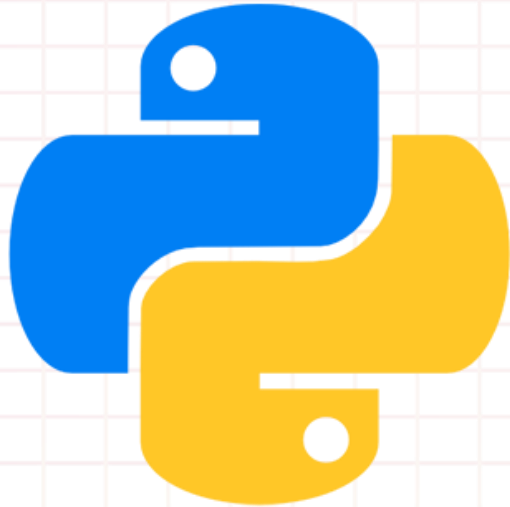
# Algoritmos de Monte Carlo: Comprobación de primalidad

- Sin embargo, existe la posibilidad de que el algoritmo anterior decida erróneamente que un número es primo.
- Pero cualquier entero mayor a 1 o es primo, o es compuesto, pero no “probablemente primo”
- Podríamos utilizar un método determinista más sofisticado que asegure si un número es primo o no. Este algoritmo sería bastante más costoso que `RepetirMillerRabin(n, 150)`.
- Pero  $4^{-150} \approx 10^{-100}$ . Es bastante menos probable que `RepetirMillerRabin(n, 150)` falle a que se produzca un fallo de hardware en el algoritmo determinista.

# ALGORITMO COMPROBACIÓN DE PRIMALIDAD

# SOLUCIÓN:

Inicializar la prueba de primalidad tomando un  $n=15$



**PYTHON**

```
if __name__ == "__main__":  
    n = 15  
  
    print(f"Probando primalidad de {n}: ", RepetirMillerRobin(n, 3))
```



# SOLUCIÓN:

## Creación de las Funciones MillerRobin y RepetirMillerRobin

La función MillerRobin siempre devuelve el valor de verdad cuando n es primo.

Si n es un impar no primo, la función MillerRobin devuelve falso.

```
def MillerRobin(n):  
    a = randint(2, n - 2)  
    return FermatB(a, n)  
  
def RepetirMillerRobin(n, k):  
    for i in range(k):  
        if not MillerRobin(n):  
            return False  
    return True
```

# SOLUCIÓN:

## Paso 3: Creación de la Función FermatB

Al aplicar esta función si retorna el valor de Falso, es seguro que el número no es primo (por el teorema de Fermat).

Por otro lado si devuelve el valor de verdad esto no se puede concluir.

```
def FermatB(a, n):  
    s = 0  
    t = n - 1  
    while t % 2 != 1:  
        s = s + 1  
        t = t // 2  
    x = expomod(a, t, n)  
    if x == 1 or x == n - 1:  
        return True  
    for i in range(1, s - 1):  
        x = (x * x) % n  
        if x == n - 1:  
            return True  
    return False
```

# SOLUCIÓN:

## Función expomod

Expomod retorna  $a^n \text{ MOD } z$   
Este cálculo lo realiza utilizando un bucle, para reducir su complejidad algorítmica.

## Ejecución del programa

```
Probando primalidad de 15: False
```

```
> |
```

Retorna la primalidad del número dado y lo realiza con el número de repeticiones dadas (3 en este caso).

```
def expomod(a, n, z):  
    i = n  
    r = 1  
    x = a % z  
    while i > 0:  
        if i % 2 != 0:  
            r = (r * x) % z  
        x = (x * x) % z  
        i = i // 2  
    return r
```

# Algoritmos de las Vegas

- *Un algoritmo de Las Vegas NUNCA da una solución falsa.*
- *Toman decisiones al azar (decisiones probabilistas) para encontrar una solución más rápido que un algoritmo determinista*
- *Si no son capaces de encontrar una solución, lo admiten*

*Clasificación (dependiendo de si encuentran o no la solución):*

- Los que siempre encuentran la solución correcta (Algoritmos de Sherwood). En este caso, es posible que sean poco eficientes si la decisión probabilista no es afortunada.
- Los que a veces, debido a decisiones poco afortunadas, llegan a un callejón sin salida y no son capaces de encontrar una solución.

# Algoritmos de las Vegas

- *Para el primer tipo: los que siempre encuentran la solución correcta.*
  - *Se aplican a problemas en los que la versión determinista es mucho más rápido en el caso promedio que en el caso peor (Ej. Quicksort)*
    - *Coste peor  $\Omega(n^2)$  y coste promedio  $O(n \log n)$*

# Algoritmos de las Vegas - Quicksort

- El algoritmo de ordenación Quicksort que vimos en la técnica Divide y Vencerás utiliza el primer elemento del vector ( o el central) como pivote para dividir el resto de la lista en dos sublistas: una con los elementos menores al pivote, y otra con los elementos mayores al pivote.
- El algoritmo determinista es muy eficiente en el caso medio: está en  $O(n \log n)$
- Sin embargo, si el vector ya está ordenado, está en  $O(n^2)$
- En este caso también podemos aplicar la técnica probabilista en la elección del pivote: en lugar de elegir siempre el primer elemento (o el central), lo que penaliza los vectores (casi) ordenados, se toma un elemento del vector al azar.

# Algoritmos de las Vegas - Quicksort

- *El algoritmo tiene el siguiente aspecto:*

```
proc quicksortLV(S[i..j])  
  si i < j entonces  
    p ← S[uniforme(i..j)]  
    particion(S[i..j], p, k, l) // particion in-place  
    quicksortLV(S[i..k])  
    quicksortLV(S[l..j])  
  fin si  
fin proc
```

```
213 // Esta función ayuda a calcular  
214 // numeros aleatorios entre low(inclusive)  
215 // y high(inclusive)  
216 static void random(int arr[], int low, int high) {  
217     Random rand = new Random();  
218     int pivot = rand.nextInt(high - low) + low;  
219     int temp1 = arr[pivot];  
220     arr[pivot] = arr[high];  
221     arr[high] = temp1;  
222 }  
223 /* Esta función toma un elemento como pivote, coloca el elemento pivote  
224 en su posición correcta en la matriz ordenada y coloca todos los elementos  
225 más pequeños (más pequeños que pivote) a la izquierda del pivote y todos  
226 los elementos mayores a la derecha del pivote */  
227 static int partition(int arr[], int low, int high) {  
228     // el pivote se selecciona aleatoriamente  
229     random(arr, low, high);  
230     int pivot = arr[high];  
231     int i = (low - 1); // índice de elemento más pequeño  
232     for (int j = low; j < high; j++) {  
233         // Si el elemento actual es más pequeño o igual  
234         // al pivote  
235         if (arr[j] < pivot) {  
236             i++;  
237             // intercambia arr[i] y arr[j]  
238             int temp = arr[i];  
239             arr[i] = arr[j];  
240             arr[j] = temp;  
241         }  
242     }  
243     // intercambia arr[i+1] y arr[high] (o pivote)  
244     int temp = arr[i + 1];  
245     arr[i + 1] = arr[high];  
246     arr[high] = temp;  
247     return i + 1;  
248 }
```

- *En este caso el tiempo esperado para el caso peor está en  $O(n \log n)$ , en lugar de tener un coste cuadrático.*

```
249 /*La función principal que implementa QuickSort aleatorio  
250 arr [] -> Array a ordenar,  
251 low -> índice inicial,  
252 high -> índice final */  
253 public static void Qsortaleatorio(int arr[], int low, int high) {  
254     if (low < high) {  
255         /* pi es un índice de partición, arr [pi]  
256         ahora está en el lugar correcto */  
257         int pi = partition(arr, low, high);  
258         //Ordena elementos de forma recursiva  
259         //antes de la partición y después de la partición  
260         Qsortaleatorio(arr, low, pi - 1);  
261         Qsortaleatorio(arr, pi + 1, high);  
262     }  
263 }  
264 public static void QsortRandom(int a[]) {  
265     Qsortaleatorio(a, 0, a.length - 1);  
266 }  
267  
268 }
```

# Algoritmos de las Vegas

- *Los algoritmos de Las Vegas pueden reducir o eliminar las diferencias de eficiencia para distintos datos de entrada:*
  - *Con mucha probabilidad, los casos que requieran mucho tiempo con el método determinista se resuelven ahora mucho más deprisa*
  - *En los casos en los que el algoritmo determinista sea muy eficiente, se resuelven ahora con más coste*
  - *En el caso promedio, no se mejora el coste*
- Uniformización del tiempo de ejecución para todas las entradas de igual tamaño

Efecto *Robin Hood*:

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



# Algoritmos de las Vegas

- *Para el segundo tipo: algoritmos que a veces no dan respuesta.*
  - *Son aceptables si fallan con probabilidad baja.*
  - *Si fallan se vuelven a ejecutar con los mismos datos de entrada.*
  - *Se utilizan para resolver problemas para los que no se conocen algoritmos deterministas eficientes que los resuelvan.*
  - *Tiempo de ejecución no está acotado pero es razonable con una elevada probabilidad.*
  - *Se presentan en forma de procedimiento con una variable éxito que toma valor cierto si se obtiene solución y falso en otro caso.*

*LV(x, solución, éxito) donde x  
representa los datos de entrada*

- *Sea  $p(x)$  la probabilidad de éxito si la entrada es  $x$ .*

# Algoritmos de las Vegas

- Los algoritmos de Las Vegas exigen que  $p(x) > 0$  para todo  $x$

- Sea la siguiente función

```
fun repetir_LV(x)
  repetir
    LV(x, solucion, exito)
  hasta éxito
  devolver solución
fin fun
```

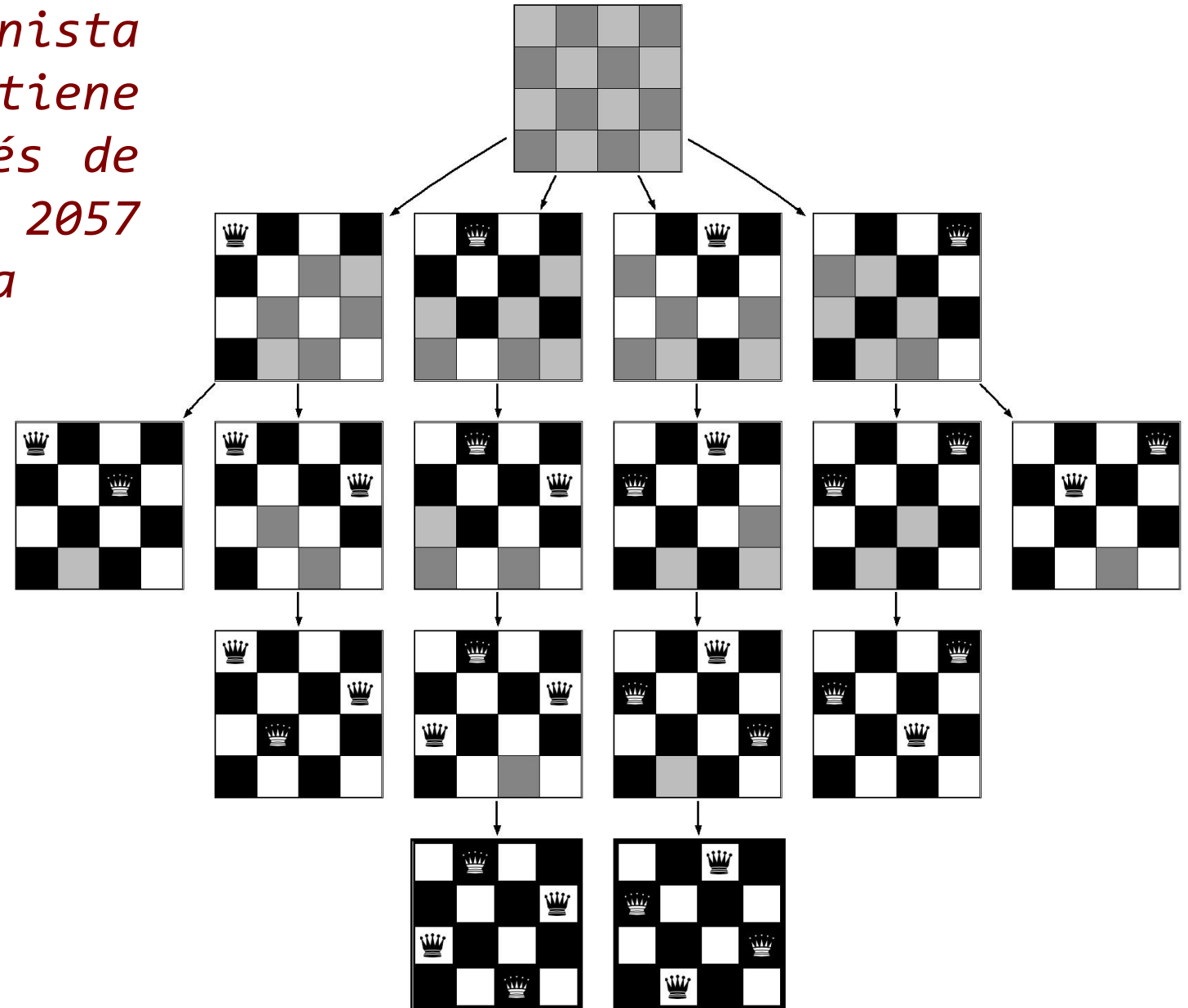
- El número de pasadas del bucle es  $1/p(x)$ 
  - Sea  $t(x)$  el tiempo esperado de `repetir_LV(x)`
  - La primera llamada a LV tiene éxito al cabo de un tiempo  $s(x)$  con probabilidad  $p(x)$
  - La primera llamada a LV fracasa al cabo de un tiempo  $f(x)$  con probabilidad  $1 - p(x)$ . El tiempo esperado total en este caso es  $f(x) + t(x)$ , porque después de que la llamada a LV fracase volvemos a estar en el punto de partida (y por tanto volvemos a necesitar un tiempo  $t(x)$ ).

# Algoritmos de las Vegas

- Así,  $t(x) = p(x)s(x) + (1 - p(x))(f(x) + t(x))$
- Resolviendo la ecuación anterior se obtiene:  $t(x) = s(x) + \frac{1-p(x)}{p(x)}f(x)$
- Esta ecuación es la clave para optimizar el rendimiento de este tipo de algoritmos. Notar que una disminución de  $s(x)$  y  $f(x)$  suele ser a costa de disminuir  $p(x)$ .

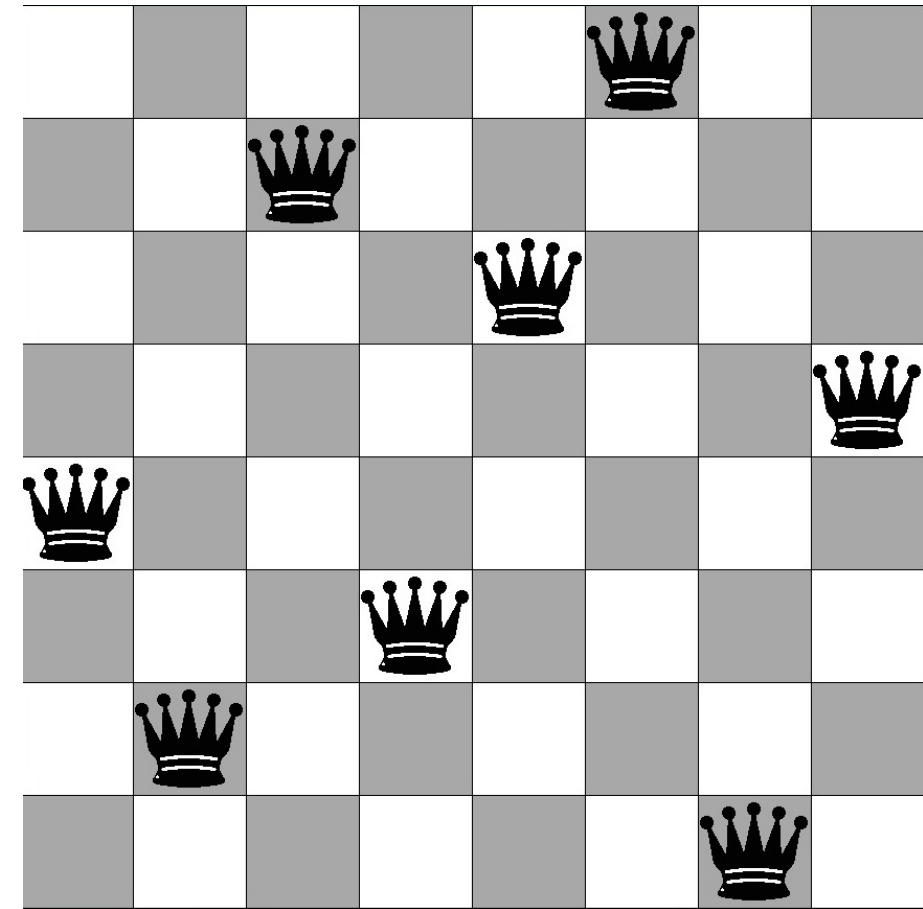
# Algoritmos de las Vegas - Problema de las 8 reinas

- El algoritmo determinista utilizando **Backtracking** obtiene la primera solución después de visitar 114 nodos de los 2057 nodos del árbol de búsqueda



# Algoritmos de las Vegas - Problema de las 8 reinas

- *Algoritmo de Las Vegas voraz: Colocar cada reina aleatoriamente en filas sucesivas ocupándose de no poner una nueva reina en una posición que sea amenazada por otra reina ya situada*
- *Como resultado:*
  - *El algoritmo acaba con éxito si es capaz de colocar todas las reinas en el tablero*
  - *Fracasa si no es capaz de colocar la siguiente reina*



# Algoritmos de las Vegas - Problema de las 8 reinas

```
proc reinasLV(solucion[1..8], exito)
  crear ok[1..8] //vector que contiene las posiciones disponibles
  columnas, diagonal1, diagonal2  $\leftarrow \emptyset$  //columnas y diagonales amenazadas
  k  $\leftarrow$  1, exito  $\leftarrow$  cierto
  mientras k  $\leq$  8  $\wedge$  exito hacer
    colPosibles  $\leftarrow$  0 //número de columnas no amenazadas por otras reinas
    desde j  $\leftarrow$  1 hasta 8 hacer
      si j  $\notin$  columnas  $\wedge$  j-(k-1)  $\notin$  diagonal1  $\wedge$  j+(k-1)  $\notin$  diagonal2 entonces
        colPosibles  $\leftarrow$  colPosibles + 1
        ok[colPosibles]  $\leftarrow$  j
      fin si
    fin desde
    si colPosibles = 0 entonces exito  $\leftarrow$  falso
    si no
      pos  $\leftarrow$  ok[uniforme(1..colPosibles)]
      columnas  $\leftarrow$  columnas  $\cup$  {pos}
      diagonal1  $\leftarrow$  diagonal1  $\cup$  {pos - (k - 1)} // Diagonal descendente
      diagonal2  $\leftarrow$  diagonal2  $\cup$  {pos + (k - 1)} // Diagonal ascendente
      k  $\leftarrow$  k+1
    fin si
  fin mientras
fin proc
```

# Algoritmos de las Vegas – Problema de las 8 reinas

- Podemos medir experimentalmente la complejidad de este algoritmo
- Utilizamos como medida de complejidad el número de nodos explorados
  - Número medio de nodos que explora en caso de éxito:  $s = 9$  (incluido el nodo inicial con 0 reinas colocadas)
  - Número esperado de nodos explorados en caso de fracaso:  $f \approx 6,971$
  - Probabilidad de éxito:  $p \approx 0,1293$
  - Número esperado de nodos visitados repitiendo hasta obtener éxito:

$$t(x) = s(x) + \frac{1-p(x)}{p(x)} f(x) \simeq 55,93$$

- Menos de la mitad del número de nodos explorados con backtracking

# Algoritmos de las Vegas - Problema de las 8 reinas

- *Se puede resolver este problema combinando las dos técnicas: backtracking y Las Vegas*
- *Se trata de poner las  $k$ -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*
- *El procedimiento quedaría así:*



# Algoritmos de las Vegas - Problema de las 8 reinas

```
proc reinasLV_b(solucion[1..8], reinasFijas, exito)
  crear ok[1..8] //vector que contiene las posiciones disponibles
  columnas, diagonal1, diagonal2  $\leftarrow \emptyset$  //columnas y diagonales amenazadas
  k  $\leftarrow$  1, exito  $\leftarrow$  cierto
  mientras k  $\leq$  reinasFijas and exito hacer
    colPosibles  $\leftarrow$  0 //número de columnas no amenazadas por otras reinas
    desde j  $\leftarrow$  1 hasta 8 hacer
      si j  $\notin$  columnas  $\wedge$  j-(k-1)  $\notin$  diagonal1  $\wedge$  j+(k-1)  $\notin$  diagonal2 entonces
        colPosibles  $\leftarrow$  colPosibles + 1
        ok[colPosibles]  $\leftarrow$  j
      fin si
    fin desde
    si colPosibles = 0 entonces exito  $\leftarrow$  falso
    si no
      pos  $\leftarrow$  ok[uniforme(0..colPosibles)]
      columnas  $\leftarrow$  columnas  $\cup$  pos
      diagonal1  $\leftarrow$  diagonal1  $\cup$  {pos - (k - 1)} ; diagonal2  $\leftarrow$  diagonal2  $\cup$  {pos + (k - 1)}
      k  $\leftarrow$  k+1
    fin si
  fin mientras
  si exito entonces backtracking(solucion, reinasFijas, exito)
fin proc
```

# Algoritmos de las Vegas - Problema de las 8 reinas

Reinas fijas	p	s	f	t	
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

Caso determinista puro

Caso probabilista puro

- *El valor óptimo es colocar 2 reinas al azar*
- *El caso ReinasFijas = 3 falla en la mitad de los casos, pero es muy rápido tanto si tiene éxito como si falla*

- *Para un tablero de 39 x 39*

Reinas fijas	p	t	Tiempo
0	1	11402835415	41 horas
29	0,21	≈ 500	8,5 ms
39	0,0074		150 ms

Determinista puro

Probabilista puro

# ALGORITMO

# PROBLEMA DE LAS N

# REINAS

(VERSIÓN LAS VEGAS)

# SOLUCIÓN:

Utilizamos como lenguaje de programación



# PYTHON

## Las Vegas

*El algoritmo Las Vegas ayudará a colocar las reinas en el tablero de forma aleatoria, de acuerdo, con las columnas disponibles. Si todas las columnas se “atacan”, se vuelve a empezar desde la primera reina.*

## Codificación del programa

```
In [10]: #PROBLEMA DE LAS N REINAS (VERSION LAS VEGAS)
import random
random.choice([1,2,3])
```

```
Out[10]: 1
```

```
In [11]: def disponibles(y,n):
          columnas_disponibles = []
          for x in range(n):
              if(columna[x] or diagonal_izquierda[x+y] or diagonal_derecha[x-y+n-1]):
                  #si la reina es atacada se debe volver a empezar
                  continue
              columnas_disponibles.append(x)
          return columnas_disponibles

          if([]):
              print("si")
          else:
              print("no")

          no
```

```
In [12]: n = 8 #Numero de reinas
          solucion = []

          columna = [False]*n
          diagonal_izquierda = [False]*(n*2)
          diagonal_derecha = [False]*(n*2)

          while(len(solucion)!=n and n>3):
              #for i in range(10):
              solucion = []
              columna = [False]*n
              diagonal_izquierda = [False]*(n*2)
              diagonal_derecha = [False]*(n*2)

              for y in range(n):
                  #colocar la reina en una columna aleatoria
                  columnas_d = disponibles(y,n)
                  #sólo si hay columnas no atacadas
                  if(columnas_d):
                      x = random.choice(columnas_d)
                  else:
                      break

                  columna[x] = diagonal_izquierda[x+y] = diagonal_derecha[x-y+n-1] = True
                  solucion.append((x,y))

          print("=====")
          print(" Problema de las N reinas ")
          print("=====")
          print("Ubicación de las reinas en el tablero: ")

          print(solucion)
```

# SOLUCIÓN:

## Explicación del código

```
In [8]: #PROBLEMA DE LAS N REINAS (VERSION LAS VEGAS)
import random
random.choice([1,2,3])
```

```
Out[8]: 3
```

Importamos la biblioteca para utilizar random.

Utilizamos random.choice para seleccionar la columna al azar de acuerdo a las columnas disponibles.

```
In [11]: def disponibles(y,n):
          columnas_disponibles = []
          for x in range(n):
              if(columna[x] or diagonal_izquierda[x+y] or diagonal_derecha[x-y+n-1]):
                  #si la reina es atacada se debe volver a empezar
                  continue
              columnas_disponibles.append(x)
          return columnas_disponibles

          if([]):
              print("si")
          else:
              print("no")
```

```
no
```

Esta sección del código determinará las columnas disponibles en la matriz.

# SOLUCIÓN:

Finalmente, al ejecutar programa se observa una de las posibles soluciones (ubicación de las reinas en el tablero) de la matriz.

Para (PRUEBA 1)

n = 8 reinas

```
=====
Problema de las N reinas
=====
Ubicación de las reinas en el tablero:
[(5, 0), (3, 1), (0, 2), (4, 3), (7, 4), (1, 5), (6, 6), (2, 7)]
```

```
In [8]: #PROBLEMA DE LAS N REINAS (VERSION LAS VEGAS)
import random
random.choice([1,2,3])

Out[8]: 3
```

Para (PRUEBA 2)

n = 8 reinas

```
=====
Problema de las N reinas
=====
Ubicación de las reinas en el tablero:
[(6, 0), (1, 1), (3, 2), (0, 3), (7, 4), (4, 5), (2, 6), (5, 7)]
```

```
In [11]: #PROBLEMA DE LAS N REINAS (VERSION LAS VEGAS)
import random
random.choice([1,2,3])

Out[11]: 2
```

Para (PRUEBA 3)

n = 8 reinas

```
=====
Problema de las N reinas
=====
Ubicación de las reinas en el tablero:
[(6, 0), (0, 1), (2, 2), (7, 3), (5, 4), (3, 5), (1, 6), (4, 7)]
```

```
In [14]: #PROBLEMA DE LAS N REINAS (VERSION LAS VEGAS)
import random
random.choice([1,2,3])

Out[14]: 1
```

## Ejecución del programa

```
In [12]: n = 8 #Numero de reinas
solucion = []

columna = [False]*n
diagonal_izquierda = [False]*(n*2)
diagonal_derecha = [False]*(n*2)

while(len(solucion)!=n and n>3):
    #for i in range(10):
        solucion = []
        columna = [False]*n
        diagonal_izquierda = [False]*(n*2)
        diagonal_derecha = [False]*(n*2)

        for y in range(n):
            #colocar la reina en una columna aleatoria
            columnas_d = disponibles(y,n)
            #sólo si hay columnas no atacadas
            if(columnas_d):
                x = random.choice(columnas_d)
            else:
                break

            columna[x] = diagonal_izquierda[x+y] = diagonal_derecha[x-y+n-1] = True
            solucion.append((x,y))

print("=====")
print(" Problema de las N reinas ")
print("=====")
print("Ubicación de las reinas en el tablero: ")

print(solucion)
```

**Tarea: Escribir el Código para  
los diferentes algoritmos  
probabilísticos vistos en clase**

# *¿Preguntas?*





***FIN***