Algoritmos y Estructuras de Datos II Clase 20

Carreras:

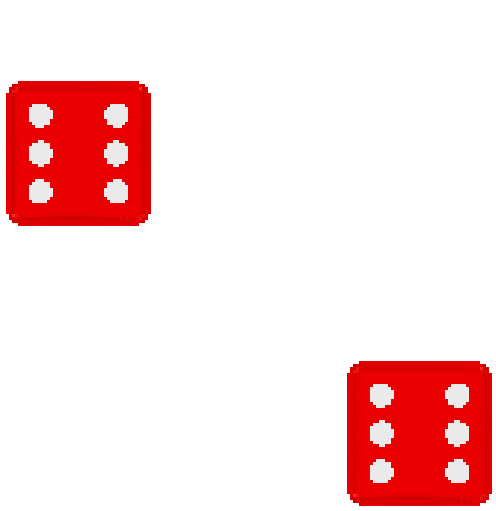
Licenciatura en Informática

Ingeniería en Informática

2024

Unidad III

Técnicas de diseño de algoritmos



Algoritmos Probabilistas(2)

Números primos

Algoritmo clásico:

• Criba de Eratóstenes (siglo II A.C.), obtiene todos los números primos menores que n.

El ***problema de la primalidad*** consiste en dado un numero entero n determinar si n es un numero primo.

Un algoritmo de ***prueba de primalidad*** es un algoritmo determinista que, dado un número de entrada n, verifica la hipótesis de un teorema cuya conclusión es que n es primo. Una prueba de primalidad es la verificación computacional de dicho teorema.

Teoremas y aportes a lo largo de varios siglos de Mersenne, Fermat, Euler, Legendre y Gauss.

Pruebas de primalidad

Solución deterministica:

- Desde los años siguientes a 1970 se ha logrado mejorar la eficiencia de los algoritmos clásicos para obtener mejores pruebas de primalidad.
- Lo más reciente: en el año 2004 el descubrimiento de un algoritmo determinista de tiempo polinomial que no se basa en ninguna conjetura no probada.
- Logro de tres académicos de la Universidad de Kanpur (Agrawal, Kayal y Saxena), que escribieron el algoritmo.
- Presentaron AKS, un algoritmo determinista de clase P para la determinación de la primalidad de un número.
- Además demostraron que este algoritmo puede ejecutarse en:

$$O((\log n)^{15/2})$$

*Publicación: "**PRIMES is in** P", By Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Annals of Mathematics, **160** (2004), no. 2, pp: 781–793*

Pruebas de primalidad

Cual es la eficiencia de los algoritmos determinísticos polinómicos para un número n muy grande?

- Las constantes escondidas de la complejidad de estos algoritmos hace que para números grandes resulte mucho más costoso de lo esperado y se usan soluciones con los llamados algoritmos probabilísticos.

Algoritmo Monte Carlo

Solución probabilística:

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

> Sea n un número primo,
> entonces (a^{n-1} mod n) = 1 para cualquier entero a tal que $1 \leq a \leq n-1$

Enunciado **contrarrecíproco** del teorema de Fermat:

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

Pruebas de primalidad

Por ejemplo, sea n = 7.

Como 7 es un número primo, entonces $(a^6 \bmod 7) = 1$,$1 \leq a \leq 6$

Entonces:

a=1 $a^{n-1} = 1^6 = 1 = 0 \times 7 + 1$

a=2 $a^{n-1} = 2^6 = 64 = 9 \times 7 + 1$

a=3 $a^{n-1} = 3^6 = 729 = 104 \times 7 + 1$

a=4 $a^{n-1} = 4^6 = 4096 = 585 \times 7 + 1$

a=5 $a^{n-1} = 5^6 = 15625 = 2232 \times 7 + 1$

a=6 $a^{n-1} = 6^6 = 46656 = 6665 \times 7 + 1$

Algoritmo Monte Carlo

- Para verificar que un número n es primo, habría que comprobar que ***todos los valores*** entre 1 y n-1 cumplen que a^{n-1} mod n = 1

- Se podría probar con un solo número elegido al azar entre: 1 y n-1

Una primera versión del algoritmo probabilista:

Que se puede concluir si retorna falso?

Que se puede concluir si retorna verdadero?

```
Función Fermat (n) : entero → bool

a←uniforme(1..n-1)
si (a^(n-1) mod n) = 1 entonces
        retorna verdadero
sino
        retorna falso
Fin
```

Pruebas de primalidad

- Cuando **Fermat(n)** devuelve falso es seguro de que **n** no es primo.
- Sin embargo, si **Fermat(n)** devuelve verdadero no se puede concluir nada.
- Para sacar alguna conclusión para la rama “true” se podría utilizar la versión **recíproca** del teorema:

~~Si n y a son enteros tales que $1 \leq a \leq n-1$ y $(a^{n-1} \bmod n) = 1$, entonces n es un número primo.~~

- Pero CUIDADO este resultado es falso.

Pruebas de primalidad

Casos triviales en que falla: $1^{n-1} \bmod n = 1$, para todo $n \geq 2$.

Más casos triviales en que falla:

$(n-1)^{n-1} \bmod n = 1$, para todo impar $n \geq 3$.

¿Falla el recíproco del teorema de Fermat en otros casos no triviales como que no sean: ($a \neq 1$ y $a \neq n-1$)?

SI FALLA.

El ejemplo más pequeño:

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

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.

Algoritmo Monte Carlo

- Una modificación del algoritmo “Fermat”:
  Elegir *a* entre 2 y *n*-2 (en lugar de entre 1 y *n*-1).

```
Función Fermat(n) :entero → bool
a←uniforme(2..n-2)
si a^(n-1) mod n = 1 entonces
        retorna verdadero
sino
        retorna falso
Fin
```

- El algoritmo falla para números no primos sólo cuando elige un *falso testigo de primalidad*.

Algoritmo Monte Carlo

La buena noticia:

- Hay “pocos” testigos falsos de primalidad.

- Si bien sólo 5 de los 332 números impares no primos menores que 1000 carecen de falsos testigos de primalidad: más de la mitad de ellos tienen sólo 2 falsos testigos de primalidad, menos del 16% tienen más de 15, en total, hay sólo 4490 falsos testigos de primalidad para todos los 332 números impares no primos menores que 1000 (de un total de 172878 candidatos existentes).

- La **probabilidad media de error** del algoritmo sobre los **números** impares no primos **menores que 1000** es menor que **0.033** y es **todavía menor** para **números mayores que 1000**.

Algoritmo Monte Carlo

La mala noticia:

- Hay números no primos que admiten muchos falsos testigos de primalidad.
- Recordar la característica fundamental de un algoritmo de Monte Carlo: “Con una alta probabilidad encuentra una solución correcta **sea cual sea la entrada**.”

- Por ejemplo, 561 admite 318 falsos testigos.
- Otro ejemplo peor: Fermat(65169305569368l) devuelve verdadero con probabilidad mayor que 0.999965 y sin embargo ese número no es primo.

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

Algoritmo Monte Carlo

Avances en la **Solución probabilística:**

Una extensión del método se debe a G.L. Miller y a M.O. Rabin que en 1976 publicaron un algoritmo Monte Carlo ¾ correcto para la comprobación de primalidad. (ver detalles en el libro Brassard & Bratley).

El tiempo total necesario para decidir acerca de la primalidad de n con una probabilidad de error acotada por ε está en $O(\log^3 n \log 1/\varepsilon)$.

Esto en la práctica significa que se puede usar para números de mil dígitos con una probabilidad de error menor que 10^{-100}

Publicación:

- *Miller, Gary L. "Riemann's Hypothesis and Tests for Primality", Journal of Computer and System Sciences **13** (3): 300–317, (1976)*
- *Rabin, Michael O. "Probabilistic algorithm for testing primality", Journal of Number Theory **12** (1): 128–138, (1980)*

Vector mayoritario

Dado un vector de **n** componentes T(1..n), se dice que T es un ***vector mayoritario*** si tiene un elemento mayoritario (un elemento que aparece repetido más de **n/2** veces).

Ejemplo:

T= (8 3 5 3 3 3 6 3 1 2 3 1 3 3) n=14

El elemento x=3 se repite 8 veces, entonces 3 es un elemento mayoritario y el vector T es un vector mayoritario.

Algoritmo Monte Carlo

Una primer versión de un algoritmo probabilista.

```
función mayoritario1(T,n) : vector x entero>0 → bool
i←uniforme(1..n)
x ←T(i)
k← 0
para j =1 hasta n hacer
        si T(j) = x entonces k←k+1
retorna (k>n/2)
```

Este algoritmo 1/2-correcto.

¿Si el vector fuera mayoritario, existe alguna posibilidad de que el algoritmo retorne falso?

¿Si el vector no fuera mayoritario, existe alguna posibilidad de que el algoritmo retorne verdadero?

Algoritmo Monte Carlo

Una segunda versión del algoritmo probabilista invocando al algoritmo anterior:

función *mayoritario2*(T,n) : vector x entero>0 → bool
si ***mayoritario1*(T,n)** entonces
    retorna verdadero
sino
    retorna ***mayoritario1*(T,n)**

- La probabilidad de que mayoritario2(T,n) devuelva verdadero si el vector T es mayoritario es:

$$p + (1-p)\,p = 1 - (1-p)^2 > 3/4$$

- *Mayoritario2* es 3/4-correcto.
- *Mayoritario2* tiene tasa de error 1/4.

Algoritmo Monte Carlo

- Si se quiere resolver el problema con una tasa de error menor, inferior a un dado ε, se aplica varias veces el algoritmo:

```
función mayoritarioMC(T, n, ε ) :vector x entero>0 x real → bool
k← lg(1/ε)
para i = 1 hasta k hacer
        si mayoritario1(T,n) entonces
                retorna verdadero
retorna falso
```

Este algoritmo es ideal para ilustrar el método de Monte Carlo por su simplicidad, sin embargo existen algoritmos que resuelven este problema con costo lineal.

Algoritmos Las Vegas

- Toman decisiones al azar para encontrar una solución antes que un algoritmo determinista.
- **Nunca devuelven una solución errónea.**
- Si no encuentran la solución correcta lo admiten.
- **Pueden no terminar.**
- Es posible volver a intentarlo con los mismos datos hasta obtener la solución correcta.
- Cuanto mayor es el tiempo dedicado al cálculo, más fiable es la solución.

Algoritmos Las Vegas

Hay dos tipos de algoritmos de Las Vegas, atendiendo a la posibilidad de no encontrar una solución:

a) Los que **siempre** *encuentran una solución correcta*, aunque las decisiones al azar no sean afortunadas y la eficiencia disminuya.

b) Los que **a veces**, debido a decisiones desafortunadas, *no encuentran una solución.*

Algoritmos Las Vegas Tipo a

Algoritmos Sherwood

- *Devuelven siempre una respuesta, la cual es siempre exacta y correcta.*

- Se usan cuando un algoritmo determinista conocido es mucho más rápido en el caso medio que en el peor.

- Se incorpora un elemento aleatorio que permite reducir o eliminar la diferencia entre los casos buenos y malos.

Ejemplo: *quicksort*.

- Costo peor $\Omega(n^2)$ y costo promedio y mejor caso $O(n \log n)$.
- Costo promedio: se calcula bajo la hipótesis de **equiprobabilidad** de la entrada.

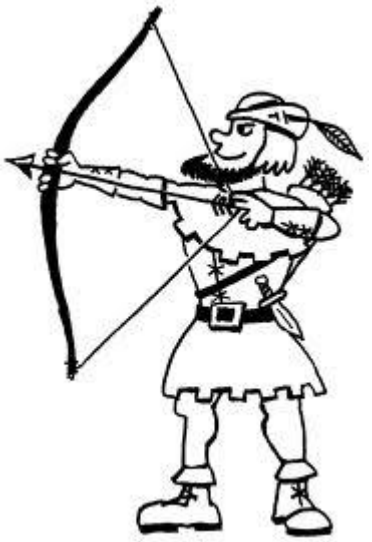
Algoritmos Sherwood

- En aplicaciones concretas, la equiprobabilidad puede no darse y pueden ocurrir entradas que producen una degradación del rendimiento en la práctica.

- Los algoritmos de Sherwood pueden reducir o eliminar la diferencia de eficiencia para distintos datos de entrada:

  – Se logra un *costo uniforme del tiempo de ejecución* para todas las entradas de igual tamaño.
  – En promedio (tomado ejemplares de igual tamaño) no se mejora el costo.
  – Con alta probabilidad, ejemplares que requieren mucho tiempo (con algoritmo determinista) ahora se resuelven más rápido.
  – Otros ejemplares para los que el algoritmo determinista era muy eficiente, se resuelven ahora con más costo.

Algoritmos Sherwood



- Efecto *Robin Hood*:

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

- Filosofía: tomar una decisión aleatoria que disminuya el tiempo de ejecución en el peor de los casos, sin que perjudique la exactitud de la solución.

- La decisión aleatoria es clave y debe hacer que todas las instancias del problema se distancien del peor caso.

Algoritmos Sherwood

Ejemplo: Búsqueda binaria aleatoria

```
Function BusqBinaria(A, izq,der,x)
                        : arreglo(1..n) x indice x indice x item → indice

si izq<der entonces
        k ← uniforme(izq…der)
        si  x = A(k) entonces          // Eureka
                Retorna k
        sino si  x < A(k) entonces
                Retorna BusqBinaria(A, izq, k-1,x)
        sino    // x ≥ A(k)
                Retorna BusqBinaria(A,k+1,der,x)
sino
        si  A(der)=x entonces
                Retorna der
        sino
                Retorna -1
Fin
```

Algoritmos Sherwood

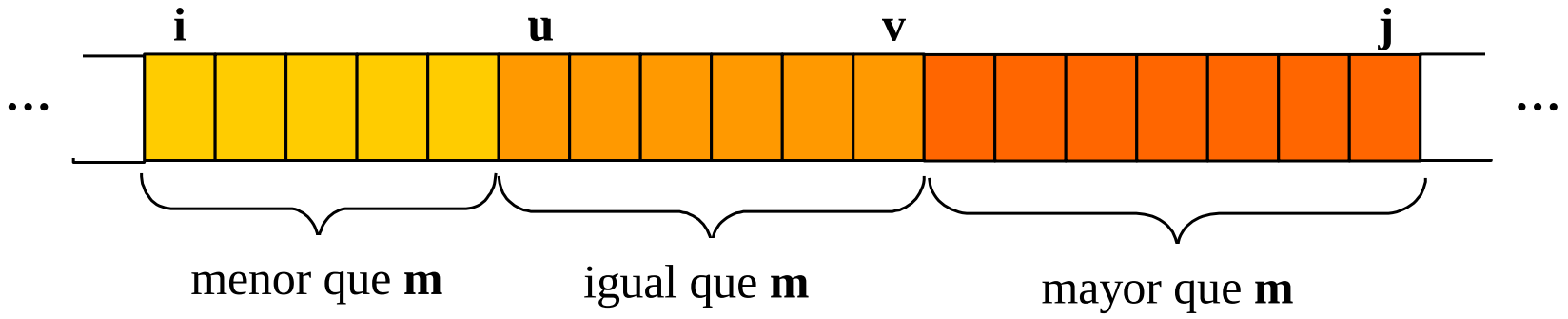
Ejemplo: Búsqueda del elemento k-ésimo menor en un arreglo

```
Funcion SeleccionarLV(T, k ) : arreglo(1..n) x indice → item
                                        // supone que 1 ≤ k ≤ n
i←1
j←n;
Mientras i < j hacer
        m ← T (uniforme(i..j))
        particionar(T, i, j, m, u, v);    //particiona el arreglo T segun valor de m
        si k<u Entonces
                j←u -1
        Sino Si k>v Entonces
                i←v+1
        Sino // k=m
                i←k
                j←k
Retorna T(i)
Fin
```

Algoritmos Sherwood

El algoritmo ***particionar*** recibe un arreglo T y separa sus elementos entre las posiciones i y j en tres partes, según el valor del m:

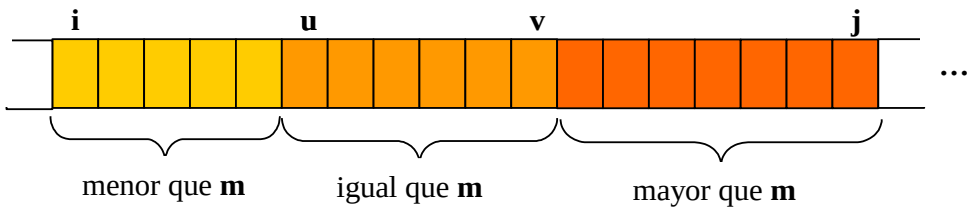
- Los elementos T(i,..,u-1) son menores que m
- Los elementos T(u,..,v) son iguales que m
- Los elementos T(v+1,..,j) son mayores que m
- Devuelve u y v



Algoritmos Sherwood

Algoritmo particionar(T,i, j, m, u, v)
: arreglo(1..n) x indice x indice x item → indice x indice

```
u ← v ← i
para k=i, j hacer
        x←T(k)
        si x < m entonces
                T(k) ← T(v)
                T(v) ← T(u)
                T(u) ← x
                u ← u + 1
                v  ← v + 1
        sino si x = m entonces
                T(k) ← T(v)
                T(v) ← x
                v ← v+1
v ← v – 1
Retorna u,v
Fin
```



Algoritmos Sherwood

- En el algoritmo del k-ésimo menor la esperanza matemática del tiempo de este algoritmo es *lineal*, independientemente del ejemplar.

- Siempre es posible que una ejecución emplee un tiempo cuadrático pero la probabilidad de que ocurra es menor cuanto mayor es n.

- Este Algoritmo de Sherwood es eficiente cualquiera que sea el ejemplar considerado.

Algoritmos Las Vegas Tipo b

- Algoritmos que, a veces, *no dan respuesta*.
- Son aceptables siempre y cuando fallen con probabilidad baja.
- Cuando fracasan *se vuelven a ejecutar* con la misma entrada buscando una posibilidad de éxito.
- 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 Las Vegas

Consideraciones sobre el costo:

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

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

algoritmo LV(x, s, éxito): entrada→solución x booleano
{éxito devuelve verdadero si LV encuentra la solución
y en ese caso devuelve la solución s encontrada}

función *obstinada*(*x*)
repetir
LV(*x,y,exito*)
hasta *éxito*
retorna *y*

Algoritmos Las Vegas

- $p(x)$: probabilidad de éxito para la entrada x , $(p(x) > 0)$
- $1/p(x)$: número de ejecuciones del lazo
- $v(x)$: esperanza matemática del tiempo esperado si éxito es verdadero
- $f(x)$: esperanza matemática del tiempo si éxito es falso.
- $t(x)$: esperanza matemática del tiempo requerido por *obstinada.*
- La primera llamada a *obstinada* tiene éxito al cabo de un tiempo: $v(x)$ con probabilidad: $p(x)$
- La primera llamada a *obstinada* 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 al tiempo del fracaso se le suma el tiempo del éxito.

Algoritmos Las Vegas

- Entonces el tiempo esperado $t(x)$ de *obstinada* está dado por la recurrencia:

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

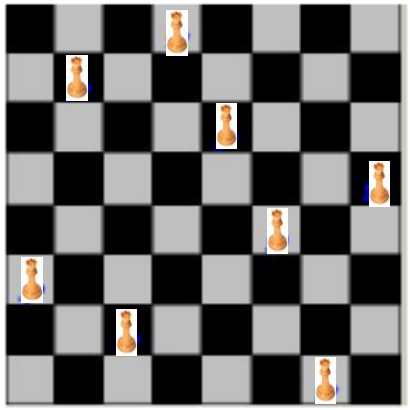
- Resolviendo la ecuación anterior se obtiene la esperanza matemática del tiempo requerido por *obstinada*:

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

- Esta ecuación es la clave para optimizar el rendimiento de este tipo de algoritmos.

Algoritmos Las Vegas

- **Ejemplo:** problemas de las 8 reinas.



- Situar 8 reinas en un tablero de ajedrez de tal modo que ninguna de ellas amenace a ninguna de las demás.

- Una reina amenaza a los cuadrados de la misma fila, de la misma columna o de las mismas diagonales.

- **Solución determinística**: Prueba sistemática, imposible (4.426.165.368 casos)

- **Solución con vuelta atrás:** explorar los nodos del árbol implícito formado por los vectores k-prometedores. Se examinan 114 de los 2057 posibles en el árbol.

Problemas de las 8 reinas

¿Cómo se puede aplicar un algoritmo de Las Vegas a este problema ?

Solucion con Algoritmo Greedy Las Vegas:

Ideas:

- Ubicar de forma aleatoria las reinas en filas sucesivas de manera que no sean amenazada por las ya colocadas.

- Como el algoritmo es Greedy no puede cambiar las reinas ya ubicadas, si no se puede ubicar una reina mas, el algoritmo debe terminar sin solución.

- Si se consiguen colocar con éxito todas las reinas el algoritmo termina con *éxito*, si no lo consigue fracasa informando del *error*.

Problemas de las 8 reinas

El algoritmo ReinasLV (Solución, Exito) es un algoritmo Las Vegas que puede fallar.

Parametros:

- Solucion(1..8) Vector donde se van almacenando la columna en que esta ubicada cada reina.
- Exito: booleana, indica el resultado de la ejecucion
  - true: el vector Solucion tiene la ubicacion de las 8 reinas
  - false: hay que ejecutar de Nuevo.

Auxiliares

Libre(1..8) // un vector que contiene las posiciones libres y disponibles

col; diag45; diag135 // Conjuntos que contienen las columnas y las diagonales a 45 y a 135 grados que ya están ocupadas

```
Funcion ReinasLV (Solución, Exito)
Auxiliar :
        Vector: Libre(1..8) // contiene las posiciones disponibles
        Conjuntos: col← ∅ ; diag45← ∅; diag135← ∅ // columnas y diagonales ocupadas

PARA k=0,7 HACER          // Para situar la reina k+1
        n←0
        PARA j=1,8 HACER
            SI j ∉ col AND j-k ∉ col45 AND j+k ∉ col135 ENTONCES
                n←n+1                  // columna j disponible para ubicar la reina k+1
                Libre(n)←j
        SI n=0 ENTONCES                // no hay lugares disponibles
                Exito ← false ; Retorna
        SINO                           // se ubica la reina k+1
                j←Libre(uniforme(1..n))
                col← col U {j} ; diag45← diag45 U {j-k}; diag135← diag135 U {j+k}
                Solucion(k+1) ←j
Exito ← true
Retorna
Fin
```

Problemas de las 8 reinas

Análisis del algoritmo **ReinasLV** de Las Vegas

- El número medio de nodos que se explora en caso de éxito contando la raíz y la solución 8-prometedora es: s=9

- La probabilidad de éxito p calculada es: p=0.1293

- El numero medio de nodos que se explora con fracaso es: f= 6,971.

- La esperanza matemática del número de nodos explorados, si se repite el algoritmo hasta alcanzar exito, es:

$$v+f*(1-p)/p=55.94$$

Que resulta ser menos de la mitad de vuelta atras.

Problemas de las n reinas

- El problema de las 8 reinas se generaliza a n reinas.
- La ventaja de los algoritmos probabilisticos es mayor a medida que crece n.

Por ejemplo: n=39 reinas,

- Vuelta atrás examina 10^{10}=11.402.835.415 nodos
- Las Vegas tiene éxito con probabilidad del 21%, en el caso medio 500 nodos explorados antes de alcanzar el éxito.

Tiempo de computadora:

- Vuelta atrás: 41 horas de calculo
- Las Vegas: un intento de solución cada 8.5 milisegundos. La tasa de aciertos es de uno en 135 intentos. Se encuentra una solución en 150 milisegundos en caso medio.

Algoritmos Probabilistas

Trabajo Práctico no. 9