Test de primalidad.

Un paseo por las clases de complejidad.

Cruz Enrique Borges Hernández

7 de julio de 2005

Contenido

1 Introducción.

Contenido

1 Introducción.

2 Algoritmos.
- Algoritmos Exponenciales.
- Algoritmos Indeterministas.
- Algoritmos Esotéricos.
- Algoritmos Probabilísticos I.
- Algoritmos Probabilísticos II.
- Algoritmos Deterministas.

Contenido

1. Introducción.
2. Algoritmos.
   - Algoritmos Exponenciales.
   - Algoritmos Indeterministas.
   - Algoritmos Esotéricos.
   - Algoritmos Probabilísticos I.
   - Algoritmos Probabilísticos II.
   - Algoritmos Deterministas.
3. Resumen.

Contenido

1. Introducción.
2. Algoritmos.
   - Algoritmos Exponenciales.
   - Algoritmos Indeterministas.
   - Algoritmos Esotéricos.
   - Algoritmos Probabilísticos I.
   - Algoritmos Probabilísticos II.
   - Algoritmos Deterministas.
3. Resumen.
4. Implementación.
   - Motivación.
   - Implementación en MAPLE.
   - Implementación en C++
   - Experiencias.

Contenido

1. Introducción.
2. Algoritmos.
   - Algoritmos Exponenciales.
   - Algoritmos Indeterministas.
   - Algoritmos Esotéricos.
   - Algoritmos Probabilísticos I.
   - Algoritmos Probabilísticos II.
   - Algoritmos Deterministas.
3. Resumen.
4. Implementación.
   - Motivación.
   - Implementación en MAPLE.
   - Implementación en C++
   - Experiencias.
5. Conclusiones.

El problema PRIMES.

Definición (Número primo)

Sea $n \in \mathbb{N}$

n *es primo si y sólo si los únicos divisores que posee son* 1 *y* n

El problema **PRIMES**.

Definición (Número primo)

Sea $n \in \mathbb{N}$

n *es primo si y sólo si los únicos divisores que posee son* 1 *y* n

¿Por qué es importante?

- Porque sus propiedades matemáticas son bonitas.

El problema PRIMES.

Definición (Número primo)

Sea $n \in \mathbb{N}$

n es primo si y sólo si los únicos divisores que posee son 1 *y* n

¿Por qué es importante?

- Porque sus propiedades matemáticas son bonitas.
- Desafío matemático.

El problema PRIMES.

Definición (Número primo)

Sea $n \in \mathbb{N}$
n *es primo si y sólo si los únicos divisores que posee son* 1 *y* n

¿Por qué es importante?

- Porque sus propiedades matemáticas son bonitas.
- Desafío matemático.

  God may not play dice with the universe, but something strange is going on with the prime numbers. Paul Erdös. (1913 - 1996)

El problema **PRIMES**.

Definición (Número primo)

Sea $n \in \mathbb{N}$

n es primo si y sólo si los únicos divisores que posee son 1 *y* n

¿Por qué es importante?

- Porque sus propiedades matemáticas son bonitas.
- Desafío matemático.

  God may not play dice with the universe, but something strange is going on with the prime numbers. Paul Erdös. (1913 - 1996)

- Imprescindible en los protocolos criptográficos actuales.

Algoritmos Escolares.

Algoritmo (Criba de Eratóstenes)

Input $= n \in \mathbb{Z}$

1. *Escribir todos los números hasta* n.
2. *Tachar el* 1 *pues es una unidad.*
3. *Repetir hasta* n
   - *Tachar los múltiplos del siguiente número sin tachar, excepto el mismo número.*

Output $=$ *Los números tachados son compuestos y los no tachados son primos.*

Algoritmos Escolares.

Algoritmo (Criba de Eratóstenes)

Input $= n \in \mathbb{Z}$

1. *Escribir todos los números hasta* n.
2. *Tachar el* 1 *pues es una unidad.*
3. *Repetir hasta* n
   - *Tachar los múltiplos del siguiente número sin tachar, excepto el mismo número.*

Output $=$ *Los números tachados son compuestos y los no tachados son primos.*

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	

Algoritmos Escolares.

Algoritmo (Criba de Eratóstenes)

Input $= n \in \mathbb{Z}$

1. *Escribir todos los números hasta n.*
2. *Tachar el 1 pues es una unidad.*
3. *Repetir hasta n*
    - *Tachar los múltiplos del siguiente número sin tachar, excepto el mismo número.*

Output = Los números tachados son compuestos y los no tachados son primos.

~~1~~	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	

Algoritmos Escolares.

Algoritmo (Criba de Eratóstenes)

Input $= n \in \mathbb{Z}$

1. *Escribir todos los números hasta* n.
2. *Tachar el* 1 *pues es una unidad.*
3. *Repetir hasta* n
   - *Tachar los múltiplos del siguiente número sin tachar, excepto el mismo número.*

Output $=$ *Los números tachados son compuestos y los no tachados son primos.*

~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~
41	~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49	~~50~~
51	~~52~~	53	~~54~~	55	~~56~~	57	~~58~~	59	~~60~~
61	~~62~~	63	~~64~~	65	~~66~~	67	~~68~~	69	~~70~~
71	~~72~~	73	~~74~~	75	~~76~~	77	~~78~~	79	~~80~~
81	~~82~~	83	~~84~~	85	~~86~~	87	~~88~~	89	~~90~~
91	~~92~~	93	~~94~~	95	~~96~~	97	~~98~~	99	

Algoritmos Escolares.

Algoritmo (Criba de Eratóstenes)

Input $= n \in \mathbb{Z}$

1. *Escribir todos los números hasta n.*
2. *Tachar el* 1 *pues es una unidad.*
3. *Repetir hasta n*
   - *Tachar los múltiplos del siguiente número sin tachar, excepto el mismo número.*

Output $=$ *Los números tachados son compuestos y los no tachados son primos.*

~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	55	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	65	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	85	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	95	~~96~~	97	~~98~~	~~99~~	

Algoritmos Escolares.

Algoritmo (Criba de Eratóstenes)

Input $= n \in \mathbb{Z}$

1. *Escribir todos los números hasta n.*
2. *Tachar el 1 pues es una unidad.*
3. *Repetir hasta n*
   - *Tachar los múltiplos del siguiente número sin tachar, excepto el mismo número.*

Output = *Los números tachados son compuestos y los no tachados son primos.*

~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	

Algoritmos Escolares.

Algoritmo (Divisiones Sucesivas)

$Input = n \in \mathbb{Z}$

1. $Mientras\ i < \sqrt{n}$
   - $Si\ n = 0 \mod i\ entonces$
     $Output = Compuesto$
2. $Output = Primo$

Algoritmos Escolares.

Algoritmo (Divisiones Sucesivas)

$Input = n \in \mathbb{Z}$

1. $Mientras\ i < \sqrt{n}$
   - $Si\ n = 0 \mod i\ entonces$
     $Output = Compuesto$
2. $Output = Primo$

- Hay que realizar muchas operaciones.

Algoritmos Escolares.

Algoritmo (Divisiones Sucesivas)

Input $= n \in \mathbb{Z}$

1. *Mientras* $i < \sqrt{n}$
   - *Si* $n = 0 \mod i$ *entonces* *Output* $=$ *Compuesto*
2. *Output* $=$ *Primo*

- Hay que realizar muchas operaciones.
- Algoritmos Exponenciales.

Algoritmos Escolares.

Algoritmo (Divisiones Sucesivas)

$Input = n \in \mathbb{Z}$

1. $Mientras\ i < \sqrt{n}$
   - $Si\ n = 0 \mod i\ entonces$
     $Output = Compuesto$
2. $Output = Primo$

- Hay que realizar muchas operaciones.
- Algoritmos Exponenciales.
- Se dice que pertenecen a la clase **E**.

Algoritmos Escolares.

Algoritmo (Divisiones Sucesivas)

$Input = n \in \mathbb{Z}$

1. $Mientras\ i < \sqrt{n}$
   - $Si\ n = 0 \mod i\ entonces$
     $Output = Compuesto$
2. $Output = Primo$

- Hay que realizar muchas operaciones.
- Algoritmos Exponenciales.
- Se dice que pertenecen a la clase **E**.
- Más simples y conocidos.

Algoritmos Tramposos.

Algoritmo (Test indeterminista de composición)

$Input = n \in \mathbb{Z}$

1. *Guess* d *divisor de* n.
2. *Si* $n = 0 \mod d$ *entonces*
   $Output = Compuesto$
   en otro caso $Output = Primo$

Algoritmos Tramposos.

Algoritmo (Test indeterminista de composición)

Input $= n \in \mathbb{Z}$

1. Guess d divisor de n.
2. Si $n = 0 \mod d$ entonces
   Output = Compuesto
   en otro caso Output = Primo

Algoritmo (Certificado de Pratt (simplificado))

Input $= n \in \mathbb{Z}$

1. Guess x testigo de n.
2. Si $o_n(x) = n - 1$ entonces
   Output = Primo.
   En otro caso Output = Compuesto.

Algoritmos Tramposos.

Algoritmo (Test indeterminista de composición)

Input $= n \in \mathbb{Z}$

1. *Guess* d *divisor de* n.
2. *Si* $n = 0 \mod d$ *entonces*
   Output = *Compuesto*
   en otro caso Output = *Primo*

Algoritmo (Certificado de Pratt (simplificado))

Input $= n \in \mathbb{Z}$

1. *Guess* x *testigo de* n.
2. *Si* $o_n(x) = n - 1$ *entonces*
   Output = *Primo*.
   En otro caso Output = *Compuesto*.

- Utilizan la trampa del Guess.

Algoritmos Tramposos.

Algoritmo (Test indeterminista de composición)

Input $= n \in \mathbb{Z}$

1. *Guess* d *divisor de* n.
2. *Si* $n = 0 \mod d$ *entonces*
   Output = *Compuesto*
   en otro caso Output = *Primo*

Algoritmo (Certificado de Pratt (simplificado))

Input $= n \in \mathbb{Z}$

1. *Guess* x *testigo de* n.
2. *Si* $o_n(x) = n - 1$ *entonces*
   Output = *Primo.*
   En otro caso Output = *Compuesto.*

- Utilizan la trampa del Guess.
- Algoritmos Indeterministas.

Algoritmos Tramposos.

Algoritmo (Test indeterminista de composición)

Input $= n \in \mathbb{Z}$

1. *Guess* d *divisor de* n.
2. *Si* $n = 0 \mod d$ *entonces*
   Output $=$ *Compuesto*
   en otro caso Output $=$ *Primo*

Algoritmo (Certificado de Pratt (simplificado))

Input $= n \in \mathbb{Z}$

1. *Guess* x *testigo de* n.
2. *Si* $o_n(x) = n - 1$ *entonces*
   Output $=$ *Primo.*
   En otro caso Output $=$ *Compuesto.*

- Utilizan la trampa del Guess.
- Algoritmos Indeterministas.
- Se dice que pertenecen a la clase **NP**.

Algoritmos Tramposos.

Algoritmo (Test indeterminista de composición)

Input $= n \in \mathbb{Z}$

1. *Guess* d *divisor de* n.
2. *Si* $n = 0 \mod d$ *entonces*
   Output $=$ *Compuesto*
   en otro caso Output $=$ *Primo*

Algoritmo (Certificado de Pratt (simplificado))

Input $= n \in \mathbb{Z}$

1. *Guess* x *testigo de* n.
2. *Si* $o_n(x) = n - 1$ *entonces*
   Output $=$ *Primo.*
   En otro caso Output $=$ *Compuesto.*

- Utilizan la trampa del Guess.
- Algoritmos Indeterministas.
- Se dice que pertenecen a la clase **NP**.
- No son implementables, hay que simular el Guess.

Otros Algoritmos.

Algoritmo (Test de Wilson)

Input $= n \in \mathbb{Z}$

1. *Si* $(n-1)! + 1 = 0$ *mod n entonces*
   Output $=$ *Primo*
   en otro caso Output $=$ *Compuesto*

Otros Algoritmos.

Algoritmo (Test de Wilson)

$Input = n \in \mathbb{Z}$

1. $Si\ (n-1)! + 1 = 0\ mod\ n\ entonces$
   $Output = Primo$
   $en\ otro\ caso\ Output = Compuesto$

- Complejidad desconocida.

Otros Algoritmos.

Algoritmo (Test de Wilson)

$Input = n \in \mathbb{Z}$

1. $Si\ (n-1)! + 1 = 0\ mod\ n\ entonces$
   $Output = Primo$
   $en\ otro\ caso\ Output = Compuesto$

- Complejidad desconocida.
- Nadie sabe como calcular $n!$ de forma eficiente.

Otros Algoritmos.

Polinomio de Matijasevic

$$
\begin{aligned}
&(k+2)\{1-[wz+h+j-q]^2-[(gk+2g+k+1)(h+j)+h-z]^2-\\
&\quad-[2n+p+q+z-e]^2-[16(k+1)^3(k+2)(n+1)^2+1-f^2]^2-\\
&\quad-[e^3(e+2)(a+1)^2+1-o^2]^2-[(a^2-1)y^2+1-x^2]^2-\\
&\quad-[16r^2y^4(a^2-1)+1-u^2]^2-\\
&\quad-[((a+u^2(u^2-a))^2-1)(n+4dy)^2+1-(x+cu)^2]^2-\\
&\quad-[n+l+v-y]^2-[(a^2-1)l^2+1-m^2]^2-\\
&\quad-[ai+k+1-l-i]^2-\\
&\quad-[p+l(a-n-1)+b(2an+2a-n^2-2n-2)-m]^2-\\
&\quad-[q+y(a-p-1)+s(2ap+2a-p^2-2p-2)-x]^2-\\
&\quad-[z+pl(a-p)+t(2ap-p^2-1)-pm]^2\}
\end{aligned}
$$

Las imagenes positivas de valores enteros, del anterior polinomio, son exactamente los números primos positivos.

Otros Algoritmos.

Polinomio de Matijasevic

$$(k+2)\{1-[wz+h+j-q]^2-[(gk+2g+k+1)(h+j)+h-z]^2-$$
$$-[2n+p+q+z-e]^2-[16(k+1)^3(k+2)(n+1)^2+1-f^2]^2-$$
$$-[e^3(e+2)(a+1)^2+1-o^2]^2-[(a^2-1)y^2+1-x^2]^2-$$
$$-[16r^2y^4(a^2-1)+1-u^2]^2-$$
$$-[((a+u^2(u^2-a))^2-1)(n+4dy)^2+1-(x+cu)^2]^2-$$
$$-[n+l+v-y]^2-[(a^2-1)l^2+1-m^2]^2-$$
$$-[ai+k+1-l-i]^2-$$
$$-[p+l(a-n-1)+b(2an+2a-n^2-2n-2)-m]^2-$$
$$-[q+y(a-p-1)+s(2ap+2a-p^2-2p-2)-x]^2-$$
$$-[z+pl(a-p)+t(2ap-p^2-1)-pm]^2\}$$

- Soluciones muy difíciles de encontrar.

Las imagenes positivas de valores enteros, del anterior polinomio, son exactamente los números primos positivos.

Otros Algoritmos.

Polinomio de Matijasevic

$$
\begin{aligned}
&(k+2)\{1-[wz+h+j-q]^2-[(gk+2g+k+1)(h+j)+h-z]^2-\\
&-[2n+p+q+z-e]^2-[16(k+1)^3(k+2)(n+1)^2+1-f^2]^2-\\
&-[e^3(e+2)(a+1)^2+1-o^2]^2-[(a^2-1)y^2+1-x^2]^2-\\
&-[16r^2y^4(a^2-1)+1-u^2]^2-\\
&-[((a+u^2(u^2-a))^2-1)(n+4dy)^2+1-(x+cu)^2]^2-\\
&-[n+l+v-y]^2-[(a^2-1)l^2+1-m^2]^2-\\
&-[ai+k+1-l-i]^2-\\
&-[p+l(a-n-1)+b(2an+2a-n^2-2n-2)-m]^2-\\
&-[q+y(a-p-1)+s(2ap+2a-p^2-2p-2)-x]^2-\\
&-[z+pl(a-p)+t(2ap-p^2-1)-pm]^2\}
\end{aligned}
$$

- Soluciones muy difíciles de encontrar.
- Expresar secuencias de números recursivamente enumerables como soluciones de ecuaciones polinomiales.

Las imagenes positivas de valores enteros, del anterior polinomio, son exactamente los números primos positivos.

Tests de composición

Algoritmo (Test Solovay-Strassen)

Input $= n, k \in \mathbb{Z}$ *con* k *número repeticiones*

1. *Desde* $i = 1$ *hasta* k *haz*
   - $a =$ *número aleatorio* $\in \{2, \ldots, n-1\}$
   - $aux = \left(\frac{a}{n}\right)$
   - *Si* $aux = 0$ *o* $a^{\frac{p-1}{2}} - aux \neq 0 \mod n$ *entonces Output* $=$ *Compuesto.*
2. *Output* $=$ *Probable Primo.*

Tests de composición

Algoritmo (Test Solovay-Strassen)

Input $= n, k \in \mathbb{Z}$ *con k número repeticiones*

1. *Desde* $i = 1$ *hasta k haz*
   - $a =$ *número aleatorio* $\in \{2, \dots, n-1\}$
   - $aux = \left(\frac{a}{n}\right)$
   - *Si* $aux = 0$ *o* $a^{\frac{p-1}{2}} - aux \neq 0$ *mod n entonces Output* $=$ *Compuesto.*
2. *Output* $=$ *Probable Primo.*

Algoritmo (Test Miller-Rabin)

Input $= n, k \in \mathbb{Z}$ *con k número de repeticiones*

1. *Escribir* $n - 1 = 2^s d$
2. *Desde* $i = 1$ *hasta k haz*
   - $a =$ *número aleatorio* $\in \{1, \dots, n-1\}$
   - *Si* $a^d \neq 1$ *mod n entonces Output* $=$ *Compuesto.*
   - *Desde* $r = 0$ *hasta* $s - 1$ *haz*
     - *Si* $a^{2^r d} \neq -1$ *mod n entonces Output* $=$ *Compuesto.*
3. *Output* $=$ *Probable Primo.*

Tests de composición

Algoritmo (Test Solovay-Strassen)

Input $= n, k \in \mathbb{Z}$ *con* k *número repeticiones*

1. *Desde* $i = 1$ *hasta* k *haz*
   - $a =$ *número aleatorio* $\in \{2, \ldots, n-1\}$
   - $aux = \left(\frac{a}{n}\right)$
   - *Si* $aux = 0$ *o* $a^{\frac{p-1}{2}} - aux \neq 0 \mod n$ *entonces Output* $=$ *Compuesto.*
2. *Output* $=$ *Probable Primo.*

Algoritmo (Test Miller-Rabin)

Input $= n, k \in \mathbb{Z}$ *con* k *número de repeticiones*

1. *Escribir* $n - 1 = 2^s d$
2. *Desde* $i = 1$ *hasta* k *haz*
   - $a =$ *número aleatorio* $\in \{1, \ldots, n-1\}$
   - *Si* $a^d \neq 1 \mod n$ *entonces Output* $=$ *Compuesto.*
   - *Desde* $r = 0$ *hasta* $s - 1$ *haz*
     - *Si* $a^{2^r d} \neq -1 \mod n$ *entonces Output* $=$ *Compuesto.*
3. *Output* $=$ *Probable Primo.*

- Para toda entrada son polinomiales en tiempo.

Tests de composición

Algoritmo (Test Solovay-Strassen)

Input $= n, k \in \mathbb{Z}$ *con k número repeticiones*

1. *Desde* $i = 1$ *hasta k haz*
   - $a =$ *número aleatorio* $\in \{2, \ldots, n-1\}$
   - $aux = \left(\frac{a}{n}\right)$
   - *Si* $aux = 0$ *o* $a^{\frac{p-1}{2}} - aux \neq 0$ *mod n entonces Output* $=$ *Compuesto.*
2. *Output* $=$ *Probable Primo.*

Algoritmo (Test Miller-Rabin)

Input $= n, k \in \mathbb{Z}$ *con k número de repeticiones*

1. *Escribir* $n - 1 = 2^s d$
2. *Desde* $i = 1$ *hasta k haz*
   - $a =$ *número aleatorio* $\in \{1, \ldots, n-1\}$
   - *Si* $a^d \neq 1$ *mod n entonces Output* $=$ *Compuesto.*
   - *Desde* $r = 0$ *hasta* $s - 1$ *haz*
     - *Si* $a^{2^r d} \neq -1$ *mod n entonces Output* $=$ *Compuesto.*
3. *Output* $=$ *Probable Primo.*

- Para toda entrada son polinomiales en tiempo.
- Si el input es primo entonces siempre devuelven probable primo.

Tests de composición

Algoritmo (Test Solovay-Strassen)

Input $= n, k \in \mathbb{Z}$ *con* k *número repeticiones*

1. *Desde* $i = 1$ *hasta* k *haz*
   - $a =$ *número aleatorio* $\in \{2, \ldots, n-1\}$
   - $aux = \left(\frac{a}{n}\right)$
   - *Si* $aux = 0$ *o* $a^{\frac{p-1}{2}} - aux \neq 0$ *mod* n *entonces Output* $=$ *Compuesto.*
2. *Output* $=$ *Probable Primo.*

Algoritmo (Test Miller-Rabin)

Input $= n, k \in \mathbb{Z}$ *con* k *número de repeticiones*

1. *Escribir* $n - 1 = 2^s d$
2. *Desde* $i = 1$ *hasta* k *haz*
   - $a =$ *número aleatorio* $\in \{1, \ldots, n-1\}$
   - *Si* $a^d \neq 1$ *mod* n *entonces Output* $=$ *Compuesto.*
   - *Desde* $r = 0$ *hasta* $s - 1$ *haz*
     - *Si* $a^{2^r d} \neq -1$ *mod* n *entonces Output* $=$ *Compuesto.*
3. *Output* $=$ *Probable Primo.*

- Para toda entrada son polinomiales en tiempo.
- Si el input es primo entonces siempre devuelven probable primo.
- Si el input es compuesto la probabilidad de darlo por probable primo es menor que $1/2^k$.

Tests de composición

Algoritmo (Test Solovay-Strassen)

Input $= n, k \in \mathbb{Z}$ *con* k *número repeticiones*

1. *Desde* $i = 1$ *hasta* k *haz*
   - $a =$ *número aleatorio* $\in \{2, \ldots, n-1\}$
   - $aux = \left(\frac{a}{n}\right)$
   - *Si* $aux = 0$ *o* $a^{\frac{p-1}{2}} - aux \neq 0$ *mod* n *entonces Output* $=$ *Compuesto.*
2. *Output* $=$ *Probable Primo.*

Algoritmo (Test Miller-Rabin)

Input $= n, k \in \mathbb{Z}$ *con* k *número de repeticiones*

1. *Escribir* $n - 1 = 2^s d$
2. *Desde* $i = 1$ *hasta* k *haz*
   - $a =$ *número aleatorio* $\in \{1, \ldots, n-1\}$
   - *Si* $a^d \neq 1$ *mod* n *entonces Output* $=$ *Compuesto.*
   - *Desde* $r = 0$ *hasta* $s - 1$ *haz*
     - *Si* $a^{2^r d} \neq -1$ *mod* n *entonces Output* $=$ *Compuesto.*
3. *Output* $=$ *Probable Primo.*

- Para toda entrada son polinomiales en tiempo.
- Si el input es primo entonces siempre devuelven probable primo.
- Si el input es compuesto la probabilidad de darlo por probable primo es menor que $1/2^k$.
- Se dice que pertenecen a la clase **Co-RP**.

Test de primalidad.

- Test APRCL y ECPP.

Test de primalidad.

- Test APRCL y ECPP.
- Tremendamente complejos y largos de explicar.

Test de primalidad.

- Test APRCL y ECPP.
- Tremendamente complejos y largos de explicar.
- Usan matemáticas de alto nivel.

Test de primalidad.

- Test APRCL y ECPP.
- Tremendamente complejos y largos de explicar.
- Usan matemáticas de alto nivel.
  - Para toda entrada es polinomial en tiempo.
  - Si el input es compuesto entonces siempre devuelve probable compuesto.
  - Si el input es primo la probabilidad de darlo por probable compuesto es menor que $1/2^k$.

Test de primalidad.

- Test APRCL y ECPP.
- Tremendamente complejos y largos de explicar.
- Usan matemáticas de alto nivel.
  - Para toda entrada es polinomial en tiempo.
  - Si el input es compuesto entonces siempre devuelve probable compuesto.
  - Si el input es primo la probabilidad de darlo por probable compuesto es menor que $1/2^k$.
- Se dice que pertenecen a la clase **RP**.

AKS

Algoritmo (AKS)

Input $= n \in \mathbb{Z}$

1. *Si* $n = a^b$ *para algún* $a \in \mathbb{N}$ *y* $b > 1$ *entonces Output* $=$ *Compuesto.*
2. *Encuentra el menor* r *tal que* $o_r(n) > 4\log^2(n)$
3. *Si* $1 < (a, n) < n$ *para algún* $a \leq r$ *entonces Output* $=$ *Compuesto.*
4. *Si* $n \leq r$ *entonces Output* $=$ *Primo.*
5. *Desde* $a = 1$ *hasta* $\lfloor 2\sqrt{\varphi(r)}\log(n)\rfloor$ *comprueba*
   - *si* $(x + a)^n \neq x^n + a \mod (x^r - 1, n)$ *entonces Output* $=$ *Compuesto.*
6. *Output* $=$ *Primo.*

AKS

Algoritmo (AKS)

$Input = n \in \mathbb{Z}$

1. $Si\ n = a^b$ *para algún* $a \in \mathbb{N}$ *y* $b > 1$ *entonces Output = Compuesto.*
2. *Encuentra el menor* r *tal que* $o_r(n) > 4\log^2(n)$
3. *Si* $1 < (a, n) < n$ *para algún* $a \leq r$ *entonces Output = Compuesto.*
4. *Si* $n \leq r$ *entonces Output = Primo.*
5. *Desde* $a = 1$ *hasta* $\lfloor 2\sqrt{\varphi(r)}\log(n) \rfloor$ *comprueba*
   - *si* $(x+a)^n \neq x^n + a \mod (x^r - 1, n)$ *entonces Output = Compuesto.*
6. *Output = Primo.*

- Para toda entrada es polinomial en tiempo.

AKS

Algoritmo (AKS)

Input $= n \in \mathbb{Z}$

1. *Si* $n = a^b$ *para algún* $a \in \mathbb{N}$ *y* $b > 1$ *entonces Output = Compuesto.*
2. *Encuentra el menor* r *tal que* $o_r(n) > 4\log^2(n)$
3. *Si* $1 < (a, n) < n$ *para algún* $a \leq r$ *entonces Output = Compuesto.*
4. *Si* $n \leq r$ *entonces Output = Primo.*
5. *Desde* $a = 1$ *hasta* $\lfloor 2\sqrt{\varphi(r)}\log(n) \rfloor$ *comprueba*
   - *si* $(x+a)^n \neq x^n + a \bmod (x^r - 1, n)$ *entonces Output = Compuesto.*
6. *Output = Primo.*

- Para toda entrada es polinomial en tiempo.
- No falla NUNCA.

AKS

Algoritmo (AKS)

$Input = n \in \mathbb{Z}$

1. $Si\ n = a^b\ para\ algún\ a \in \mathbb{N}\ y\ b > 1\ entonces\ Output = Compuesto.$
2. $Encuentra\ el\ menor\ r\ tal\ que\ o_r(n) > 4\log^2(n)$
3. $Si\ 1 < (a, n) < n\ para\ algún\ a \leq r\ entonces\ Output = Compuesto.$
4. $Si\ n \leq r\ entonces\ Output = Primo.$
5. $Desde\ a = 1\ hasta\ \lfloor 2\sqrt{\varphi(r)}\log(n) \rfloor\ comprueba$
   - $si\ (x + a)^n \neq x^n + a\ mod\ (x^r - 1, n)\ entonces\ Output = Compuesto.$
6. $Output = Primo.$

- Para toda entrada es polinomial en tiempo.
- No falla NUNCA.
- Se dice que pertenecen a la clase **P**.

AKS

Algoritmo (AKS)

Input $= n \in \mathbb{Z}$

1. *Si* $n = a^b$ *para algún* $a \in \mathbb{N}$ *y* $b > 1$ *entonces Output* $=$ *Compuesto.*
2. *Encuentra el menor* r *tal que* $o_r(n) > 4\log^2(n)$
3. *Si* $1 < (a, n) < n$ *para algún* $a \leq r$ *entonces Output* $=$ *Compuesto.*
4. *Si* $n \leq r$ *entonces Output* $=$ *Primo.*
5. *Desde* $a = 1$ *hasta* $\lfloor 2\sqrt{\varphi(r)}\log(n)\rfloor$ *comprueba*
   - *si* $(x + a)^n \neq x^n + a \mod (x^r - 1, n)$ *entonces Output* $=$ *Compuesto.*
6. *Output* $=$ *Primo.*

- Para toda entrada es polinomial en tiempo.
- No falla NUNCA.
- Se dice que pertenecen a la clase **P**.
- En la forma actual es más lento que Miller-Rabin o incluso ECPP.

Conclusiones.

- El algoritmo AKS pone resuelve un problema abierto de hace más de 2.000 años.

Conclusiones.

- El algoritmo AKS pone resuelve un problema abierto de hace más de 2.000 años.
- Permitirá mejorar los sistemas criptográficos.

Conclusiones.

- El algoritmo AKS pone resuelve un problema abierto de hace más de 2.000 años.
- Permitirá mejorar los sistemas criptográficos.
- Actualmente se sigue investigando para mejorar el algoritmo.

Conclusiones.

- El algoritmo AKS pone resuelve un problema abierto de hace más de 2.000 años.
- Permitirá mejorar los sistemas criptográficos.
- Actualmente se sigue investigando para mejorar el algoritmo.
- Resultado elemental que abre todo una nueva vía de estudio.

Motivación.

Objetivos de la experimentación.

- Comprobar la velocidad del algoritmo (cotas asintóticas).

Motivación.

Objetivos de la experimentación.

- Comprobar la velocidad del algoritmo (cotas asintóticas).
- Comprobar la eficacia de los algoritmos probabilistas.

Motivación.

Objetivos de la experimentación.

- Comprobar la velocidad del algoritmo (cotas asintóticas).
- Comprobar la eficacia de los algoritmos probabilistas.
- Comprobar la siguiente conjetura.

Conjetura (Conjetura para la modificación del AKS)

Si r es un número primo que no divide a n y $(x-1)^n = x^n - 1 \mod x^r - 1, n$ entonces o n es primo o $n^2 = 1 \mod r$.

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Subrutinas internas.

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Subrutinas internas.
- Función exponencial dentro del bucle

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Subrutinas internas.
- Función exponencial dentro del bucle $\Longrightarrow$ Subrutinas internas.

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Subrutinas internas.
- Función exponencial dentro del bucle $\Longrightarrow$ Subrutinas internas.

Resultados.

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Subrutinas internas.
- Función exponencial dentro del bucle $\Longrightarrow$ Subrutinas internas.

Resultados.

- El algoritmo es extremadamente lento.

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Subrutinas internas.
- Función exponencial dentro del bucle $\Longrightarrow$ Subrutinas internas.

Resultados.

- El algoritmo es extremadamente lento.
- Suponemos que es debido a la implementación de la función exponencial dentro del bucle.

Implementación en MAPLE.

¿Por qué MAPLE?

- Familiarizado con el entorno.
- Software ampliamente probado y fiable.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Subrutinas internas.
- Función exponencial dentro del bucle $\Longrightarrow$ Subrutinas internas.

Resultados.

- El algoritmo es extremadamente lento.
- Suponemos que es debido a la implementación de la función exponencial dentro del bucle.
- Proponemos una modificación.

Implementación en MAPLE.

Problemas de la implementación.

- Función exponencial dentro del bucle

Implementación en MAPLE.

Problemas de la implementación.

- Función exponencial dentro del bucle $\Longrightarrow$ Programación propia.

Implementación en MAPLE.

Problemas de la implementación.

- Función exponencial dentro del bucle $\Longrightarrow$ Programación propia.

Resultados.

Mucho más lento aún.

Implementación en MAPLE.

Problemas de la implementación.

- Función exponencial dentro del bucle $\Longrightarrow$ Programación propia.

Resultados.

Mucho más lento aún.

Conclusiones.

- MAPLE realiza de forma óptima el bucle.
- El rendimiento global lo achacamos al propio MAPLE.

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Programación propia.

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Programación propia.
- Función exponencial dentro del bucle

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Programación propia.
- Función exponencial dentro del bucle $\Longrightarrow$ Subrutinas internas.

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Programación propia.
- Función exponencial dentro del bucle $\Longrightarrow$ Subrutinas internas.

Problema.

- Máximo grado que puede alcanzar un polinomio es un número *long* (números desde 0 hasta 4.294.967.296).

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Programación propia.
- Función exponencial dentro del bucle $\Longrightarrow$ Subrutinas internas.

Problema.

- Máximo grado que puede alcanzar un polinomio es un número *long* (números desde 0 hasta 4.294.967.296).
- Limita la experimentación.

Implementación sobre la librería NTL.

¿Por qué NTL?

- Documentación más clara.
- Amplio catálogo de subrutinas y tipos de datos ya implementados.

Problemas de la implementación.

- Algoritmo de factorización $\Longrightarrow$ Subrutinas internas.
- Cálculo del orden de un elemento $\Longrightarrow$ Programación propia.
- Función exponencial dentro del bucle $\Longrightarrow$ Subrutinas internas.

Problema.

- Máximo grado que puede alcanzar un polinomio es un número *long* (números desde 0 hasta 4.294.967.296).
- Limita la experimentación.
- Impide comparar con el algoritmo de Miller-Rabin en los números donde se podría observar las cotas asintóticas.

Benchmark.

Objetivos.

- Comparar la velocidad del AKS con Miller-Rabin.

Benchmark.

Objetivos.

- Comparar la velocidad del AKS con Miller-Rabin.
- Comparar la velocidad del algoritmo en distintas arquitecturas.

Benchmark.

Objetivos.

- Comparar la velocidad del AKS con Miller-Rabin.
- Comparar la velocidad del algoritmo en distintas arquitecturas.

Descripción.

El programa realiza un test de primalidad AKS y Miller-Rabin a los números entre 9.000 y 9.100 calculando el tiempo de ejecución y devolviéndolo en una tabla.

Benchmark.

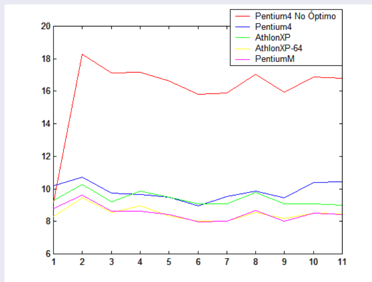
Objetivos.

- Comparar la velocidad del AKS con Miller-Rabin.
- Comparar la velocidad del algoritmo en distintas arquitecturas.

Descripción.

El programa realiza un test de primalidad AKS y Miller-Rabin a los números entre 9.000 y 9.100 calculando el tiempo de ejecución y devolviéndolo en una tabla.

Resultados.



Benchmark.

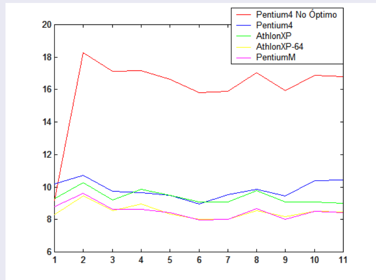
Objetivos.

- Comparar la velocidad del AKS con Miller-Rabin.
- Comparar la velocidad del algoritmo en distintas arquitecturas.

Descripción.

El programa realiza un test de primalidad AKS y Miller-Rabin a los números entre 9.000 y 9.100 calculando el tiempo de ejecución y devolviéndolo en una tabla.

Resultados.



MUCHO más rápido que MAPLE.

Computación GRID.

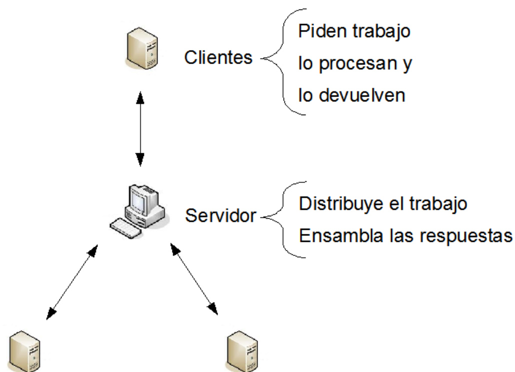
¿Qué es GRID?

Forma de distribuir grandes volumenes de trabajo entre muchos ordenadores. Sólo es útil cuando el trabajo es paralelizable.

Computación GRID.

¿Qué es GRID?

Forma de distribuir grandes volumenes de trabajo entre muchos ordenadores. Sólo es útil cuando el trabajo es paralelizable.



Computación GRID.

Objetivos.

- Comparar los tiempos de ejecución en un amplio rango de valores.

Computación GRID.

Objetivos.

- Comparar los tiempos de ejecución en un amplio rango de valores.
- Hallar la curva asintótica de complejidad del AKS.

Computación GRID.

Objetivos.

- Comparar los tiempos de ejecución en un amplio rango de valores.
- Hallar la curva asintótica de complejidad del AKS.
- Estimar $\pi(x)$. Esto es, verificar empíricamente el teorema de densidad de los números primos.

Computación GRID.

Objetivos.

- Comparar los tiempos de ejecución en un amplio rango de valores.
- Hallar la curva asintótica de complejidad del AKS.
- Estimar $\pi(x)$. Esto es, verificar empíricamente el teorema de densidad de los números primos.
- Encontrar errores en la ejecución de Miller-Rabin.

Computación GRID.

Objetivos.

- Comparar los tiempos de ejecución en un amplio rango de valores.
- Hallar la curva asintótica de complejidad del AKS.
- Estimar $\pi(x)$. Esto es, verificar empíricamente el teorema de densidad de los números primos.
- Encontrar errores en la ejecución de Miller-Rabin.

Descripción.

Realizamos el test AKS y Miller-Rabin sobre el máximo conjunto de números que podemos alcanzar. Obtenemos los tiempos de ejecución de ambos algoritmos y el número de primos encontrados. Es una gran cantidad de cálculo a realizar. Debemos buscar alternativas al cálculo directo.

Computación GRID.

¿Cómo parelizamos el AKS?

Interna $\hookrightarrow$ Cada iteración del bucle por separado.

Computación GRID.

¿Cómo parelizamos el AKS?

Interna $\hookrightarrow$ Cada iteración del bucle por separado.

Desde $a = 1$ hasta $\lfloor 2\sqrt{\varphi(r)}\log(n) \rfloor$ comprueba:

- si $(x + a)^n \neq x^n + a \mod (x^r - 1, n)$ entonces Output $=$ Compuesto.

Computación GRID.

¿Cómo parelizamos el AKS?

Interna $\hookrightarrow$ Cada iteración del bucle por separado.

Desde $a = 1$ hasta $\lfloor 2\sqrt{\varphi(r)}\log(n)\rfloor$ comprueba:

- si $(x+a)^n \neq x^n + a \mod (x^r - 1, n)$ entonces Output $=$ Compuesto.

Externa $\hookrightarrow$ Cada ejecución completa del test por separado.

Computación GRID.

¿Cómo parelizamos el AKS?

Interna $\hookrightarrow$ Cada iteración del bucle por separado.

Desde $a = 1$ hasta $\lfloor 2\sqrt{\varphi(r)}\log(n)\rfloor$ comprueba:

- si $(x+a)^n \neq x^n + a \bmod (x^r - 1, n)$ entonces Output $=$ Compuesto.

Externa $\hookrightarrow$ Cada ejecución completa del test por separado.

Resultados.

- Fracaso.

Computación GRID.

¿Cómo parelizamos el AKS?

Interna $\hookrightarrow$ Cada iteración del bucle por separado.

Desde $a = 1$ hasta $\lfloor 2\sqrt{\varphi(r)}\log(n)\rfloor$ comprueba:

- si $(x+a)^n \neq x^n + a \mod (x^r - 1, n)$ entonces Output $=$ Compuesto.

Externa $\hookrightarrow$ Cada ejecución completa del test por separado.

Resultados.

- Fracaso.
- Estimación del tiempo de ejecución muy optimista.

Computación GRID.

¿Cómo parelizamos el AKS?

Interna $\hookrightarrow$ Cada iteración del bucle por separado.

Desde $a = 1$ hasta $\lfloor 2\sqrt{\varphi(r)}\log(n)\rfloor$ comprueba:

- si $(x+a)^n \neq x^n + a \bmod (x^r - 1, n)$ entonces Output $=$ Compuesto.

Externa $\hookrightarrow$ Cada ejecución completa del test por separado.

Resultados.

- Fracaso.
- Estimación del tiempo de ejecución muy optimista.
- El aula fue utilizada y apagaron varios ordenadores.

Test de estrés a Miller-Rabin.

Objetivos.

- Hacer fallar al algoritmo de Miller-Rabin.

Test de estrés a Miller-Rabin.

Objetivos.

- Hacer fallar al algoritmo de Miller-Rabin.

Descripción.

Realizamos 1.000.000 de iteraciones del algoritmo de Miller-Rabin (con una iteración) sobre un número de carmichael para ver si produce errores.

Test de estrés a Miller-Rabin.

Objetivos.

- Hacer fallar al algoritmo de Miller-Rabin.

Descripción.

Realizamos 1.000.000 de iteraciones del algoritmo de Miller-Rabin (con una iteración) sobre un número de carmichael para ver si produce errores.

Resultados.

- El algoritmo NO falló ni una sola vez.

Test de estrés a Miller-Rabin.

Objetivos.

- Hacer fallar al algoritmo de Miller-Rabin.

Descripción.

Realizamos 1.000.000 de iteraciones del algoritmo de Miller-Rabin (con una iteración) sobre un número de carmichael para ver si produce errores.

Resultados.

- El algoritmo NO falló ni una sola vez.
- Sospechamos que la implementación NO es el test de Miller-Rabin clásico.

Verificación de la conjetura.

Objetivos.

- Comprobar si la modificación del AKS, asumiendo la conjetura, es válida.

Conjetura

Si r es un número primo que no divide a n y $(x-1)^n = x^n - 1 \ mod \ (x^r - 1, n)$ entonces o n es primo o $n^2 = 1 \ mod \ r$.

Verificación de la conjetura.

Objetivos.

- Comprobar si la modificación del AKS, asumiendo la conjetura, es válida.
- Comprobar la velocidad del algoritmo modificado.

Conjetura

Si r es un número primo que no divide a n y $(x-1)^n = x^n - 1 \ mod \ (x^r - 1, n)$ entonces o n es primo o $n^2 = 1 \ mod \ r$.

Verificación de la conjetura.

Objetivos.

- Comprobar si la modificación del AKS, asumiendo la conjetura, es válida.
- Comprobar la velocidad del algoritmo modificado.

Descripción.

Realizamos una prueba similar al test de computación GRID, pero esta vez con el algoritmo modificado.

Conjetura

Si r es un número primo que no divide a n y $(x-1)^n = x^n - 1 \mod (x^r - 1, n)$ entonces o n es primo o $n^2 = 1 \mod r$.

Verificación de la conjetura.

Objetivos.

- Comprobar si la modificación del AKS, asumiendo la conjetura, es válida.
- Comprobar la velocidad del algoritmo modificado.

Conjetura

Si r es un número primo que no divide a n y $(x-1)^n = x^n - 1 \mod (x^r - 1, n)$ entonces o n es primo o $n^2 = 1 \mod r$.

Descripción.

Realizamos una prueba similar al test de computación GRID, pero esta vez con el algoritmo modificado.

Resultados.

- Sólo es un poco más lento que Miller-Rabin.

Verificación de la conjetura.

Objetivos.

- Comprobar si la modificación del AKS, asumiendo la conjetura, es válida.
- Comprobar la velocidad del algoritmo modificado.

Conjetura

Si r es un número primo que no divide a n y $(x-1)^n = x^n - 1 \ mod \ (x^r - 1, n)$ entonces o n es primo o $n^2 = 1 \ mod \ r$.

Descripción.

Realizamos una prueba similar al test de computación GRID, pero esta vez con el algoritmo modificado.

Resultados.

- Sólo es un poco más lento que Miller-Rabin.
- Sorprendentemente Miller-Rabin presenta errores. Repetimos el test de estrés en esos valores.

Verificación de la conjetura.

Objetivos.

- Comprobar si la modificación del AKS, asumiendo la conjetura, es válida.
- Comprobar la velocidad del algoritmo modificado.

Conjetura

Si r es un número primo que no divide a n y $(x-1)^n = x^n - 1 \; mod \; (x^r - 1, n)$ *entonces o n es primo o* $n^2 = 1 \; mod \; r$.

Descripción.

Realizamos una prueba similar al test de computación GRID, pero esta vez con el algoritmo modificado.

Resultados.

- Sólo es un poco más lento que Miller-Rabin.
- Sorprendentemente Miller-Rabin presenta errores. Repetimos el test de estrés en esos valores.
- Los errores siempre se producen en los mismos números. Semilla de aleatoriedad.

Verificación de la conjetura.

Objetivos.

- Comprobar si la modificación del AKS, asumiendo la conjetura, es válida.
- Comprobar la velocidad del algoritmo modificado.

Conjetura

Si r es un número primo que no divide a n y $(x-1)^n = x^n - 1 \ mod\ (x^r - 1, n)$ entonces o n es primo o $n^2 = 1 \ mod\ r$.

Descripción.

Realizamos una prueba similar al test de computación GRID, pero esta vez con el algoritmo modificado.

Resultados.

- Sólo es un poco más lento que Miller-Rabin.
- Sorprendentemente Miller-Rabin presenta errores. Repetimos el test de estrés en esos valores.
- Los errores siempre se producen en los mismos números. Semilla de aleatoriedad.
- Falla la conjetura. Posible error en la implementación.

Estimación del error en Miller-Rabin.

Objetivos.

- Observar el impacto del número de iteraciones en la probabilidad de error en el algoritmo de Miller-Rabin.

Estimación del error en Miller-Rabin.

Objetivos.

- Observar el impacto del número de iteraciones en la probabilidad de error en el algoritmo de Miller-Rabin.

Descripción.

El programa se ejecuta 20 veces, aumentando cada vez el número de iteraciones del algoritmo. En cada ejecución se realizan 100.000 test sobre uno de los números problemáticos.

Estimación del error en Miller-Rabin.

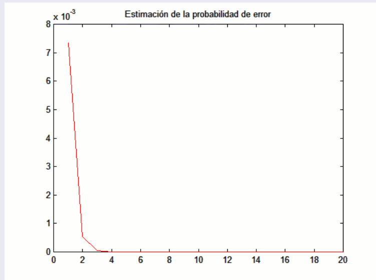
Objetivos.

- Observar el impacto del número de iteraciones en la probabilidad de error en el algoritmo de Miller-Rabin.

Descripción.

El programa se ejecuta 20 veces, aumentando cada vez el número de iteraciones del algoritmo. En cada ejecución se realizan 100.000 test sobre uno de los números problemáticos.

Resultados.

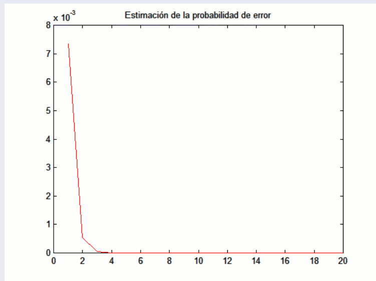


Estimación del error en Miller-Rabin.

Objetivos.

- Observar el impacto del número de iteraciones en la probabilidad de error en el algoritmo de Miller-Rabin.

Resultados.



Descripción.

El programa se ejecuta 20 veces, aumentando cada vez el número de iteraciones del algoritmo. En cada ejecución se realizan 100.000 test sobre uno de los números problemáticos.

- La probabilidad cae de forma exagerada.

Estimación del error en Miller-Rabin.

Objetivos.

- Observar el impacto del número de iteraciones en la probabilidad de error en el algoritmo de Miller-Rabin.

Descripción.

El programa se ejecuta 20 veces, aumentando cada vez el número de iteraciones del algoritmo. En cada ejecución se realizan 100.000 test sobre uno de los números problemáticos.

Resultados.

Estimación de la probabilidad de error

- La probabilidad cae de forma exagerada.
- Hay muy poco impacto en la velocidad de ejecución.

Conclusiones de la experimentación.

- Los algoritmos probabilistas son perfectamente válidos.

Conclusiones de la experimentación.

- Los algoritmos probabilistas son perfectamente válidos.
- El AKS estándar NO puede competir con Miller-Rabin.

Conclusiones de la experimentación.

- Los algoritmos probabilistas son perfectamente válidos.
- El AKS estándar NO puede competir con Miller-Rabin.
- De confirmarse la conjetura, el AKS modificado si podría competir con Miller-Rabin.

Trabajo futuro.

- Prueba de estrés al AKS modificado para averiguar en qué números falla.

Trabajo futuro.

- Prueba de estrés al AKS modificado para averiguar en qué números falla.
- Revisar la programación de la modificación y optimizarla.

Trabajo futuro.

- Prueba de estrés al AKS modificado para averiguar en qué números falla.
- Revisar la programación de la modificación y optimizarla.
- Revisar la implementación de Miller-Rabin (aleatoriedad y algoritmo).

Trabajo futuro.

- Prueba de estrés al AKS modificado para averiguar en qué números falla.
- Revisar la programación de la modificación y optimizarla.
- Revisar la implementación de Miller-Rabin (aleatoriedad y algoritmo).
- Modificar la librería NTL para poder realizar cálculos con exponentes de longitud arbitraria.

Trabajo futuro.

- Prueba de estrés al AKS modificado para averiguar en qué números falla.
- Revisar la programación de la modificación y optimizarla.
- Revisar la implementación de Miller-Rabin (aleatoriedad y algoritmo).
- Modificar la librería NTL para poder realizar cálculos con exponentes de longitud arbitraria.
- Estudio de los algoritmos de factorización y de los test de nulidad.