

# Análisis de Algoritmos que permitan Generar Números Aleatorios grandes y que prueben la primalidad de un Número Grande

Becerra Sipiran, Cledy Elizabeth  
Oviedo Sivincha, Massiel  
Villanueva Borda, Harold Alejandro

## Resumen:

La cuestión de la determinación de si un número  $n$  dado es primo es conocida como el problema de la primalidad. Un test de primalidad (o chequeo de primalidad) es un algoritmo que, dado un número de entrada  $n$ , no consigue verificar la hipótesis de un teorema cuya conclusión es que  $n$  es compuesto.

Esto es, un test de primalidad sólo conjetura que “ante la falta de certificación sobre la hipótesis de que  $n$  es compuesto podemos tener cierta confianza en que se trata de un número primo”. Esta definición supone un grado menor de confianza que lo que se denomina prueba de primalidad (o test verdadero de primalidad), que ofrece una seguridad matemática al respecto.

Los números aleatorios son la base esencial de la simulación. Usualmente, toda la aleatoriedad involucrada en el modelo se obtiene a partir de un generador de números aleatorios que produce una sucesión de valores que supuestamente son realizaciones de una secuencia de variables aleatorias independientes e idénticamente distribuidas (i.i.d.)  $U(0, 1)$ . Posteriormente estos números aleatorios se transforman convenientemente para simular las diferentes distribuciones de probabilidad que se requieran en el modelo.

## 2. Algoritmos que prueben Primalidad

### 2.1. Lucas-Lehmer

#### Definición:

Es una prueba que sirve para determinar si un determinado número de Mersenne  $M_p$  es primo. Sea  $M_p = 2^p - 1$  el número de Mersenne a testear con  $p$  primo impar. Definase la sucesión  $\{S_i\}$  para todo  $i \geq 0$  según:

$$s_i = \begin{cases} 4, & \text{si } i = 0; \\ s_{i-1}^2 - 2 & \text{en caso contrario.} \end{cases}$$

Entonces,  $M_p$  es primo si y sólo si:

$$s_{p-2} \equiv 0 \pmod{M_p};$$

En otro caso,  $M_p$  es compuesto.

#### Algoritmo:

**Entrada:** Un número primo  $p$ .

**Salida:** COMPUESTO si  $M_p$  es compuesto y PRIMO si  $M_p$  es un primo.

1. Definase  $s_1 \leftarrow 4$
2. Definase  $M_p \leftarrow 2^p - 1$
3. Para  $j$  desde 2 hasta  $n - 1$  haga lo siguiente:
  1.  $s_j \leftarrow (s_{j-1}^2 - 2) \bmod M_p$
4. Si  $s_{j-1} = 0$  entonces:
  1. Retorne PRIMO
5. Si no, entonces
  1. Retorne COMPUESTO

#### Seguimiento numérico:

Si  $p=7$ , verificar primalidad de  $2^7 - 1$ .

Sabemos que  $2^7 - 1 = 127$

Entonces  $x = 127$ ,  $p = 7 - 1 = 6$ .

De ahí que la serie de Lucas-Lehmer sea:  
iteración 1: 4,

iteración 2:  $(4*4 - 2) \% 127 = 14$ ,  
 iteración 3:  $(14*14 - 2) \% 127 = 67$ ,  
 iteración 4:  $(67*67 - 2) \% 127 = 42$ ,  
 iteración 5:  $(42*42 - 2) \% 127 = 111$ ,  
 iteración 6:  $(111*111) \% 127 = 0$ .

La 6ta iteración es 0 entonces 127 es un primo.

### **Implementación:**

```
ZZ modulo(ZZ a, ZZ b)
{
    ZZ q = a/b;
    if (b < ZZ(0)) { q++; }
    return a - b * q;
}
ZZ expo_modular(ZZ a, ZZ n){
    ZZ result;
    result = ZZ(1);
    while( n != ZZ(0)) {
        if(modulo(n,ZZ(2))!=0)
            result = result*a;
        a = a*a;
        n >>= 1;
    }
    return result;
}
bool Lucas_Lehmer(ZZ p) {

    ZZ num = expo_modular(ZZ(2), p) - 1;

    ZZ nextval = modulo(ZZ(4),num);

    for (ZZ i = ZZ(1); i < p - 1; i++)
        nextval = modulo((nextval * nextval -
        2),num);

    return (nextval == ZZ(0));
}

int main() {
    ZZ p = ZZ(7);

    ZZ num = expo_modular(ZZ(2), p) - 1;

    if (Lucas_Lehmer(p))
        cout << num << " es primo.";
    else
        cout << num << " no es primo.";

    return 0;
}
```

### **Pruebas:**

Procesador/Bits	AMD Ryzen 7 4800H 2.90 GHz	Intel(R) i5-6200U 2.30GHz	Intel(R) i3-3110M 2.40GHz
16	0.021 s	0.058 s	0.291 s
32	0.024 s	0.070 s	0.342 s
64	0.031 s	0.088 s	0.455 s
128	0.037 s	0.110 s	0.598 s
256	0.095 s	0.200 s	0.701 s
512	0.229 s	0.588 s	0.898 s
1024	0.577 s	0.679 s	0.978 s
2048	0.901 s	0.799 s	1.813 s

## **2.2.Solovay-Strassen**

### **Definición:**

Es una prueba probabilística para determinar si un número es compuesto o probablemente primo. El Solovay-Strassen fue la primera prueba popularizada por la llegada del público clave criptografía, en especial RSA criptosistema. Desarrollada de la prueba de Euler combinada con el símbolo de Jacobi. Recordar que los Símbolos jacobianos son equivalente a los símbolos de Legendre cuando es primo.

$$a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n}$$

Si n es un primo impar, entonces el algoritmo aleatorio de Solovay-Strassen produce el resultado primo, y si n es un compuesto impar produce COMPOSITE al menos para la mitad de posibles  $a \in \{1, 2, \dots, n-1\}$ . El algoritmo corre en tiempo polinomial, respecto al "tamaño de n".

Entonces se afirma que analiza si un número entero dado es primo, dando una respuesta segura en caso de que la respuesta sea negativa, mientras que si la respuesta es afirmativa lo hace con cierta probabilidad de error (tan baja como uno se proponga, según cómo aplique el test).

### **Algoritmo:**

**Input:** n, valor a probar primalidad  
k, parámetro de número de interacciones

**Output:** True si es primo y False si es compuesto

**Repetir** k veces:

elegir al azar en el rango  $[2, n - 1]$

$$x \leftarrow \left(\frac{a}{n}\right)$$

**if**  $x = 0$  or  $a^{(n-1)/2} \not\equiv x \pmod{n}$   
**then**  
    **return** False

**return** True

### Seguimiento Numérico:

Donde se desea saber si 13 es primo, “i” el número de interacciones y “a” los números aleatorios.

Como se cumple con las diez interacciones que “(jacobian+p)MOD p y Expo Binaria” son congruentes, se puede afirmar que 13 es un número primo.

i	a	jacobian	(jacobian+p)MODp	Expo_binaria
1	8	-1	(13-1)MOD13=12	8^6 MOD13=12
2	11	1	(13+1)MOD13=1	11^6 MOD13=1
3	10	-1	(13-1)MOD13=12	10^6 MOD13=12
4	8	-1	(13-1)MOD13=12	8^6 MOD13=12
5	6	-1	(13-1)MOD13=12	6^6 MOD13=12
6	8	-1	(13-1)MOD13=12	8^6 MOD13=12
7	11	-1	(13-1)MOD13=12	11^6 MOD13=12
8	1	1	(13+1)MOD13=1	1^6 MOD13=1
9	10	1	(13+1)MOD13=1	10^6 MOD13=1
10	2	-1	(13-1)MOD13=12	2^6 MOD13=12

### Implementación en C++:

```

ZZ MOD(ZZ a, ZZ n){
    ZZ q,r;
    q=a/n;
    r=a-(q*n);
    if(r<0){
        if(q<=0){--q;r=a-(q*n);}
        if(q>0){++q;r=a-(q*n);}
    }

    return r;
}

ZZ binary_expo_modular(ZZ a, ZZ n, ZZ m){
    ZZ result;
    result = ZZ(1);

```

```

while( n != ZZ(0)) {
    if(MOD(n,ZZ(2))!=0){
        result = MOD(result*a,m);}

    a = MOD(a*a,m);
    n >>= 1;
}
return result;
}

ZZ calculateJacobian(ZZ a, ZZ n)
{
    if (a==ZZ(0))
        return ZZ(0);

    ZZ ans = ZZ(1);
    if (a < ZZ(0))
    {
        a = -a;
        if (MOD(n,ZZ(4)) == ZZ(3))
            ans = -ans;
    }

    if (a == ZZ(1))
        return ans;

    while (a!=ZZ(0))
    {
        if (a < ZZ(0))
        {
            a = -a;
            if (MOD(n,ZZ(4)) ==
ZZ(3))
                ans = -ans;
        }

        while (MOD(a,ZZ(2)) ==
ZZ(0))
        {
            a = a / ZZ(2);
            if (MOD(n,ZZ(8)) ==
ZZ(3) || MOD(n,ZZ(8)) == ZZ(5))
                ans = -ans;
        }

        swap(a, n);

        if (MOD(a,ZZ(4)) == ZZ(3)
&& MOD(n,ZZ(4)) == ZZ(3))
            ans = -ans;
        a = MOD(a,n);

        if (a > n / ZZ(2))
            a = a - n;
    }
}

```

```

    }
    if (n == ZZ(1))
        return ans;

    return ZZ(0);
}

bool SolovoyStrassen(ZZ p, int iterations)
{
    if (p < ZZ(2))
        return false;
    if (p != ZZ(2) && MOD(p,ZZ(2)) ==
ZZ(0))
        return false;

    for (int i = 0; i < iterations; i++)
    {
        int aux = 0;
        conv(aux, p);
        ZZ a = ZZ(rand() % (aux - 1)
+ 1);
        ZZ jacobian = MOD((p +
calculateJacobian(a, p)),p);
        ZZ mod =
binary_expo_modular(a, (p - ZZ(1)) / ZZ(2),
p);
        if (jacobian ==ZZ(0)|| mod !=
jacobian)
            return false;
    }
    return true;
}

```

### Pruebas:

Procesador/Bits	AMD Ryzen 7 4800H 2.90 GHz	Intel(R) i5-6200U 2.30GHz	Intel(R) i3-3110M 2.40GHz
16	0.025 s	0.023 s	0.322 s
32	0.028 s	0.069 s	0.359 s
64	0.028 s	0.073 s	0.468 s
128	0.029 s	0.088 s	0.633 s
256	0.050 s	0.105 s	0.647 s
512	0.164 s	0.249 s	0.793 s
1024	0.824 s	1.110 s	1.871 s
2048	1.118 s	1.268 s	1.977 s

## 2.3.Test de Miller- Robin

### Definición:

El test más implantado en la actualidad es el Miller-Rabin (también conocido como test fuerte del pseudoprimo). El test de Miller-Rabin (MR) está basado en el siguiente hecho:

Si tenemos un número primo  $n$  y  $n-1=2^s * r$  donde  $r$  es impar, se cumple que  $\forall a \in \mathbb{N}$   $\text{mcd}(a, n) = 1$ , entonces o bien  $a^r \equiv 1 \pmod{n}$  o bien  $\exists j \in [0, s-1] / a^{2^j * r} \equiv -1 \pmod{n}$ . [1]

Este test es el sistema que más se utiliza en la actualidad dada su rapidez, aunque se sacrifica la certitud de un test de primalidad, con pocas iteraciones que se realicen se alcanza una muy buena precisión. El algoritmo se basa en el siguiente lema ([2]pág. 14):

Sea  $p$  un número primo, y sea  $x \in \mathbb{Z}_p$  tal que:

$$x^2 \equiv 1 \pmod{p}$$

entonces,  $x \equiv 1$  o bien  $x \equiv p-1$  en módulo  $p$ .

Demostración. Como  $p$  es un número primo,  $\mathbb{Z}_p$  es un dominio de integridad, por tanto, tenemos

$$x^2 \equiv 1 \pmod{p}$$

$$x^2 - 1 \equiv 0 \pmod{p}$$

$$(x+1)(x-1) \equiv 0 \pmod{p}$$

Como  $\mathbb{Z}_p$  es un dominio de integridad, se tiene o bien  $x \equiv 1$  o bien  $x \equiv -1 \equiv p-1$

Teorema (Miller - Rabin). Sea  $p$  un número primo, escribimos  $p-1 = 2^s * m$  donde  $m$  es un número impar, entonces  $\forall a \in \mathbb{Z}_p$  se tiene

$$a^m \equiv 1$$

o, por el contrario, existe al menos un  $r$  entre  $0$  y  $s-1$  tal que

$$a^{2^r * m} \equiv 1$$

Demostración. Por el pequeño teorema de

Fermat tenemos que

$$a^{p-1} \equiv a^{2^r \cdot m} \equiv 1$$

luego por el lema anterior tenemos

$$a^{2^{s-1}m} \equiv -1 \text{ o bien } a^{2^{s-1}m} \equiv 1$$

iteramos sucesivamente hasta que encontramos  $-1$  o bien llegamos a

$$a^{2^0 \cdot m} \equiv a^m \equiv 1 \text{ o } a^m \equiv -1$$

En ambos casos, obtenemos el resultado.

### Algoritmo:

**Entrada:** Un número natural  $n > 1$ , el número  $k$  de veces que se ejecuta el test y nos determina la fiabilidad del test.

**Salida:** COMPUESTO si  $n$  es compuesto y POSIBLE PRIMO si  $n$  es un posible primo.

```
1. Definase  $r$  y  $s$  tal que  $r$  es impar y  $(n-1) = r \cdot 2^s$ 
2. Para  $j$  desde 1 hasta  $k$  haga lo siguiente:
  1.  $a \leftarrow$  Función Genera_numero_aleatorio_en_intervalo[2,  $n-2$ ]
  2.  $y \leftarrow a^r \bmod n$ 
  3. Si  $(y \neq 1) \wedge (y \neq n-1)$  entonces:
    1.  $j \leftarrow j-1$ 
    2. Mientras  $j \leq (s-1) \wedge y \neq (n-1)$  haga lo siguiente:
      1.  $y \leftarrow y^2 \bmod n$ 
      2. Si  $y = 1$  entonces:
        1. Retorne COMPUESTO
      3.  $j \leftarrow j+1$ 
    3. Si  $y \neq (n-1)$  entonces:
      1. Retorne COMPUESTO
3. Retorne POSIBLE PRIMO
```

### Seguimiento Numérico:

Supóngase que se desea determinar si un número  $n \in \mathbb{N}$  impar grande dado es un número primo o compuesto.

1. Elija de manera arbitraria e independiente  $k$  números  $a$  tal que  $2 \leq a \leq n-1$ .

2. Determine con el algoritmo de Euclides  $\text{mcd}(a, n)$ . Si  $\text{mcd}(a, n) \neq 1$  entonces  $n$  es compuesto.

3. Pruebe si bajo el  $a$  elegido se encuentra un testigo para la composición. Con el primer encuentro finaliza el algoritmo y  $n$  no es un número primo. Si no es posible encontrar tal número  $a$  entonces el algoritmo termina con fallido.

Observaciones sobre el Test de Miller – Rabin

Si el test de Miller - Rabin finaliza con fallido entonces  $n$  tiene la apariencia de un número primo, pero no necesariamente lo es. La probabilidad de que para un número

compuesto  $n$  el test de Miller - Rabin termine con fallido es menor que  $(1/4)^k$ .

Si se escoge  $k$  muy grande, entonces la probabilidad de que  $n$  sea un número primo es alta, sobre la base que el test de Miller - Rabin termine con fallido.

Un test de primalidad probabilístico menos efectivo que el de Miller - Rabin es el de Solovay - Strassen, el cual se fundamenta en el teorema de Euler

### Propiedades deseables[2]:

En general es deseable que los test de primalidad que se buscan cumplan una serie de propiedades que garanticen el buen comportamiento y la validez del algoritmo frente a cualquier situación.

- Un test de primalidad que ofrece una cierta probabilidad de que un número sea primo se denomina test aleatorizado, mientras que uno que afirma con determinación dicha propiedad se conoce como test determinista. Preferimos un test determinista sobre uno aleatorizado, como en este caso es el algoritmo de Miller-Rabin.

- La rapidez del algoritmo es vital, sobre todo nos interesa su comportamiento asintótico. Considerándose rápido un test si su orden de complejidad es polinómico respecto al número de cifras de entrada.

- Los algoritmos incondicionales son aquellos que funcionan sin tener que trabajar bajo ciertas suposiciones como en el test de Fermat, donde teníamos que suponer que no habíamos encontrado un número de Carmichael.

- Existen ciertos tests de primalidad que no pueden ser aplicados a cualquier candidato si no que solo funcionan utilizando cierto subconjunto de números. Buscaremos algoritmos generales, aunque en ciertas

ocasiones pueden resultar útiles dichos tests más específicos.

### **Implementación en C++:**

```
#include <bits/stdc++.h>
#include <NTL/ZZ.h>

using namespace NTL;
using namespace std;

//OPERACIONES
ZZ mod(ZZ a,ZZ b){
    ZZ r=a-(b*(a/b));
    if(r<0)r=b-r;
    return r;
}
bool even(ZZ a){
    if(mod(a,ZZ(2))==0) return 1;
    return 0;
}
ZZ ValAbs(ZZ a){
    if (a<0) return (a*-1);
    return a;
}
ZZ power(ZZ a, ZZ n, ZZ m){
    ZZ result;
    result = ZZ(1);
    while( n != ZZ(0)) {
        if(!even(n))
            result = mod(result*a,m);
        a = mod(a*a,m);
        n >>= 1;
    }
    return result;
}
bool miillerTest(ZZ d, ZZ n){

    ZZ a; a=2;
    ZZ x = power(a, d, n);
    if (x == 1 || x == n-1)return true;
    while (d != n-1){

        x = mod((x * x), n);
        d *= 2;
        if (x == 1)return false;
        if (x == n-1)return true;
    }
}
```

```
        return false;
    }
    //bool isPrime(ZZ n, ZZ k){}
    bool isPrime(ZZ n){
        ZZ k; k=0;
        if (n <= 1 or n == 4)return false;
        if (n <= 3)return true;

        ZZ d = n - 1;
        while (even(d)){
            //d /= 2;
            d >>= 1;
            k++;
        }

        for (int i = 0; i < k+1; i++)
            if (!miillerTest(d, n))
                return false;

        return true;
    }
}
```

### **Pruebas:**

Procesador/Bits	AMD Ryzen 7 4800H 2.90 GHz	Intel(R) i5-6200U 2.30GHz	Intel(R) i3-3110M 2.40GHz
16	0.019 s	0.021 s	0.202 s
32	0.025 s	0.022 s	0.216 s
64	0.027 s	0.029 s	0.288 s
128	0.029 s	0.067 s	0.480 s
256	0.034 s	0.069 s	0.606 s
512	0.043 s	0.078 s	0.619 s
1024	0.140 s	0.222 s	0.857 s
2048	0.405 s	0.615 s	1.510 s

## **3.Algoritmo para Generar Números Aleatorios Grandes**

### 3.1.RC4

#### **Definición:**

RC4 (esquemas de cifrado más utilizados del mundo , es un algoritmo simple) consiste en 2 algoritmos:

1-Algoritmo de programación de claves (KSA)

2-Algoritmo de generación Pseudoaleatoria (PRGA).

El KSA genera la permutación en el S-Box basándose en una clave de longitud variable ("key length", entre 40 y 2048 bits).

El Pseudo-Random Generation Algorithm (PRGA) tiene 2 contadores, el i y la j, en el cual ambos son inicializados en 0 para comenzar. Después de eso, cada bit de keystream(KSA) data es usado en el siguiente Pseudo-Code.

#### **Process Status API (PSAPI):**

##### **Definición:**

La interfaz de programación de aplicaciones de estado de procesos (PSAPI) es una biblioteca auxiliar que le facilita la obtención de información sobre procesos y controladores de dispositivos.

Estas funciones están disponibles en Psapi.dll.

##### **EnumProcesses function (psapi.h) [3]:**

Recupera el identificador de proceso para cada objeto de proceso en el sistema.

Sintaxis c++:

```
BOOL EnumProcesses(  
    DWORD *lpidProcess, // Un puntero a  
    una matriz que recibe la lista de  
    identificadores de proceso.  
    DWORD cb, //El tamaño de la matriz  
    pProcessIds, en bytes  
    LPDWORD lpcbNeeded //El número de  
    bytes devueltos en la matriz pProcessIds.  
);
```

Si la función tiene éxito, el valor devuelto es distinto de cero.

Si la función falla, el valor de retorno es cero. Para obtener información de error ampliada, llame a GetLastError

##### **Observaciones**

Es una buena idea usar una matriz grande, porque es difícil predecir cuántos procesos habrá en el momento de llamar a

EnumProcesses .

Para determinar cuántos procesos se enumeraron, divida el valor de lpcbNeeded por sizeof (DWORD). No se proporciona ninguna indicación cuando el búfer es demasiado pequeño para almacenar todos los identificadores de proceso. Por lo tanto, si lpcbNeeded es igual a cb , considere reintentar la llamada con una matriz más grande.

Para obtener identificadores de proceso para los procesos cuyos identificadores acaba de obtener, llame a la función OpenProcess .

A partir de Windows 7 y Windows Server 2008 R2, Psapi.h establece los números de versión para las funciones de PSAPI. El número de versión de PSAPI afecta el nombre utilizado para llamar a la función y la biblioteca que debe cargar un programa.

Si PSAPI\_VERSION es 2 o mayor, esta función se define como K32EnumProcesses en Psapi.h y se exporta en Kernel32.lib y Kernel32.dll. Si PSAPI\_VERSION es 1, esta función se define como EnumProcesses en Psapi.h y se exporta en Psapi.lib y Psapi.dll como un contenedor que llama a K32EnumProcesses .

Los programas que deben ejecutarse en versiones anteriores de Windows, así como en Windows 7 y versiones posteriores, siempre deben llamar a esta función como EnumProcesses . Para asegurar la resolución correcta de los símbolos, agregue Psapi.lib a la

macro TARGETLIBS y compile el programa con `-DPSAPI_VERSION = 1`. Para utilizar la vinculación dinámica en tiempo de ejecución, cargue Psapi.dll.

### ***GetProcessMemoryInfo function (psapi.h) [3]:***

Recupera información sobre el uso de memoria del proceso especificado.

Sintaxis c++:

```
BOOL GetProcessMemoryInfo(  
    HANDLE Process,  
    PPROCESS_MEMORY_COUNTERS  
    ppsmemCounters,  
    DWORD cb  
);
```

Parámetros

Process

El identificador debe tener el derecho de acceso `PROCESS_QUERY_INFORMATION` o `PROCESS_QUERY_LIMITED_INFORMATION`.

**\*\* Windows Server 2003 y Windows XP: \*\***  
El identificador debe tener los derechos de acceso `PROCESS_QUERY_INFORMATION` y `PROCESS_VM_READ`.

ppsmemCounters

Un puntero a la estructura `PROCESS_MEMORY_COUNTERS` o `PROCESS_MEMORY_COUNTERS_EX` que recibe información sobre el uso de memoria del proceso.

cb

El tamaño de la estructura `ppsmemCounters`, en bytes.

Si la función tiene éxito, el valor devuelto es distinto de cero.

Si la función falla, el valor de retorno es cero. Para obtener información de error ampliada, llame a `GetLastError`.

Observaciones

A partir de Windows 7 y Windows Server 2008 R2, Psapi.h establece los números de versión para las funciones de PSAPI. El número de versión de PSAPI afecta el nombre utilizado para llamar a la función y la biblioteca que debe cargar un programa.

Si `PSAPI_VERSION` es 2 o mayor, esta función se define como `K32GetProcessMemoryInfo` en Psapi.h y se exporta en Kernel32.lib y Kernel32.dll. Si `PSAPI_VERSION` es 1, esta función se define como `GetProcessMemoryInfo` en Psapi.h y se exporta en Psapi.lib y Psapi.dll como un contenedor que llama a `K32GetProcessMemoryInfo`.

Los programas que deben ejecutarse en versiones anteriores de Windows, así como en Windows 7 y versiones posteriores, siempre deben llamar a esta función como `GetProcessMemoryInfo`. Para garantizar la resolución correcta de los símbolos, agregue Psapi.lib a la macro TARGETLIBS y compile el programa con `-DPSAPI_VERSION = 1`. Para utilizar la vinculación dinámica en tiempo de ejecución, cargue Psapi.dll.

### ***Estructura***

### ***PROCESS\_MEMORY\_COUNTERS (psapi.h) [3]:***

Sintaxis c++:

```
typedef struct  
_PROCESS_MEMORY_COUNTERS {  
    DWORD cb;  
    DWORD PageFaultCount;  
    SIZE_T PeakWorkingSetSize;  
    SIZE_T WorkingSetSize;  
    SIZE_T QuotaPeakPagedPoolUsage;
```



```

SIZE_T QuotaPagedPoolUsage;
SIZE_T QuotaPeakNonPagedPoolUsage;
SIZE_T QuotaNonPagedPoolUsage;
SIZE_T PagefileUsage;
SIZE_T PeakPagefileUsage;
} PROCESS_MEMORY_COUNTERS;

```

## Miembros

cb: El tamaño de la estructura, en bytes.

PageFaultCount: El número de errores de página.

PeakWorkingSetSize: El tamaño máximo del conjunto de trabajo, en bytes.

WorkingSetSize: El tamaño actual del conjunto de trabajo, en bytes.

QuotaPeakPagedPoolUsage: El uso máximo del grupo paginado, en bytes.

QuotaPagedPoolUsage: El uso actual del grupo paginado, en bytes.

QuotaPeakNonPagedPoolUsage: El uso máximo del grupo no paginado, en bytes.

QuotaNonPagedPoolUsage: El uso actual del grupo no paginado, en bytes.

PagefileUsage: El valor de Commit Charge en bytes para este proceso. Commit Charge es la cantidad total de memoria que el administrador de memoria ha comprometido para un proceso en ejecución.

PeakPagefileUsage: El valor máximo en bytes del cargo de compromiso durante la vida útil de este proceso.

## Requisitos

Cliente mínimo admitido	Windows XP [aplicaciones de escritorio   Aplicaciones para UWP]
Servidor mínimo admitido	Windows Server 2003 [aplicaciones de escritorio   Aplicaciones para UWP]
Plataforma de destino	Ventanas
Encabezamiento	psapi.h
Biblioteca	Kernel32.lib en Windows 7 y Windows Server 2008 R2; Psapi.lib (si PSAPI_VERSION = 1) en Windows 7 y Windows Server 2008 R2; Psapi.lib en Windows Server 2008, Windows Vista, Windows Server 2003 y Windows XP
ETC	Kernel32.dll en Windows 7 y Windows Server 2008 R2; Psapi.dll (si PSAPI_VERSION = 1) en Windows 7 y Windows Server 2008 R2; Psapi.dll en Windows Server 2008, Windows Vista, Windows Server 2003 y Windows XP

## Implementación en C++:

```

vector<bool> RC4(vector<int> semilla){
    vector<int> Or;
    //permite que S solo pertenezca a este
scope
    vector<int> S;
    for(int i=0;i<256;i++) S.push_back(i);
    //permite que k solo exista en este scope,
optimiza memoria
    vector<int> K;
    for(int i=0,k=0;i<=51;i++)
        for(int j=0;j<5;j++,k++)
            K.push_back(semilla[j]);
    for(int i=0,f=0;i<256;i++){
        f= ModInteger(f + S[i] + K[i], 256);
        swap(S[i],S[f]);
    }
}
for(int i=0,f=0,k=0;k<8;k++){
    i= ModInteger(i + 1, 256);
    f= ModInteger(f + S.at(i), 256);
    swap(S.at(i),S.at(f));
    Or.push_back(S.at(ModInteger(S.at(i) +
S.at(f), 256)));//t
}
}
vector<bool> out;
for(int i=0;i<8;i++){
    bitset<8> aux(Or[i]);
    for(int j=0;j<8;j++)
        out.push_back(aux[j]);
}

```

```

    }
    return out;
}

```

### Pruebas:

Procesador/Bits	AMD Ryzen 7 4800H 2.90 GHz	Intel(R) i5-6200U 2.30GHz	Intel(R) i3-3110M 2.40GHz
16	0.022 s	0.026 s	0.131 s
32	0.025 s	0.029 s	0.164 s
64	0.027 s	0.031 s	0.165 s
128	0.030 s	0.052 s	0.178 s
256	0.031 s	0.065 s	0.179 s
512	0.032 s	0.068 s	0.181 s
1024	0.043 s	0.070 s	0.201 s

## 4. Análisis de Algoritmos

Los análisis de los algoritmos se realizaron en tres diferentes procesadores (podrá ver los diferentes resultados en las pruebas realizadas al final de cada descripción del algoritmo ) ,pero mismo sistema operativo:

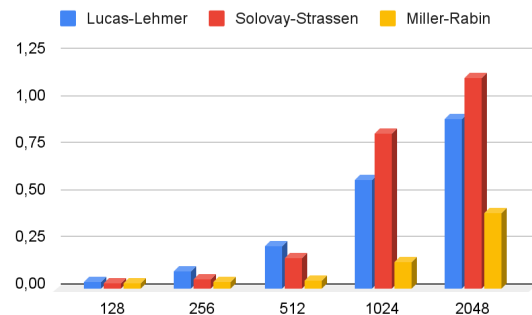
- AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz
- Intel(R) core(TM) i5-6200U CPU @ 2.30GHz 2.40GHz
- Intel(R) Core(TM) i3-3110M CPU @ 2.40GHz 2.40 GHz

Teniendo en cuenta que para el análisis tomamos el procesador más rápido de los tres; es decir , AMD Ryzen 7 4800H con Radeon Graphics 2.90 GHz.

### 4.1. Tiempo de Ejecución

En la siguiente tabla y gráfico mostraré el tiempo de compilación en (s) dependiendo de los bits implementados.

	Lucas-Lehmer	Solovay-Strassen	Miller-Rabin
128	0,037	0,029	0,029
256	0,095	0,05	0,034
512	0,229	0,164	0,043
1024	0,577	0,824	0,140
2048	0,901	1,118	0,405



### 4.2. Evaluación de primalidad en números random

En las siguientes imágenes se muestran los 20 números de 16 bits aleatorios que generamos y el resultado del test de primalidad.

```

"C:\Users\Usuario\Documents\UCSP\III CICLO\ALGEBRA ABSTRACTA\SEMAM
37091 no es primo
42487 es primo
44255 no es primo
37111 no es primo
62541 no es primo
49205 no es primo
52293 no es primo
37269 no es primo
44743 no es primo
50645 no es primo
61091 es primo
59145 no es primo
55797 no es primo
43181 no es primo
59413 no es primo
62109 no es primo
45255 no es primo
39887 es primo
42969 no es primo
56167 es primo
Process returned 0 (0x0)   execution time : 0.054 s
Press any key to continue.


```





```
C:\Users\User\Documents\UCSP\III\8\Gebra i\FinalPaperPrimeRandom
44729 es primo
63017 no es primo
36215 no es primo
54709 es primo
41985 no es primo
54377 es primo
35347 no es primo
41035 no es primo
47715 no es primo
45047 no es primo
44599 no es primo
57623 no es primo
35873 no es primo
61011 no es primo
35659 no es primo
36981 no es primo
49103 es primo
48315 no es primo
52671 no es primo
47667 no es primo



Process returned 0 (0x0)   execution time : 0.044 s
Press any key to continue.
```

[illegible][illegible][illegible]

En la siguiente tabla y gráfico se muestra la ocupación de memoria que se realizaron en los algoritmos dependiendo de los bits implementados.

Name	Status	14% CPU	54% Memory	0% Disk	0% Network	5% GPU
Apps (8)						
>  Code:Blocks IDE		0.1%	21.3 MB	0 MB/s	0 Mbps	0%

		43%	65%
Nombre	Estado	CPU	Memoria
<b>Aplicaciones (5)</b>			
 Administrador de tareas		1.8%	24.4 MB
 Bloc de notas		0%	0.1 MB
 Brave Browser (34)		11.5%	1,930.0 MB
 Code:Blocks IDE		0.4%	103.2 MB

Nombre	Estado	25% CPU	76% Memoria	14% Disco	0% Red
>  Brave Browser (22)		12,5%	1.342,1 MB	0,1 MB/s	0,1 Mbps
>  main.cpp [Aleatorios1] - Code:...		3,0%	102,4 MB	0 MB/s	0 Mbps

La ventaja de este método respecto a los anteriores radica en que no existe ningún número  $n$  compuesto que supere el test para todos los posibles valores de  $a$  (tal como ocurría con los números de Carmichael en el test de Fermat), más aún, la proporción de valores de  $a$  que superan el test respecto a los

posibles valores que toma dicho parámetro siendo  $n$  compuesto es inferior a  $1/4$  con lo cual con  $k$  iteraciones superadas hay como mucho una probabilidad de  $(1/4)^k$  de obtener un falso positivo. Conforme aumenta el número de iteraciones la probabilidad tiende a 0 sin embargo, nunca podremos estar totalmente seguros de la primalidad de un número, siempre existirá cierto riesgo de habernos equivocado [2].

## 6. Referencias bibliográficas

- [1] Test de primalidad  
[https://es.wikipedia.org/wiki/Test\\_de\\_primalidad](https://es.wikipedia.org/wiki/Test_de_primalidad)
- [2] Test de primalidad para sistemas criptográficos  
<https://core.ac.uk/download/pdf/289981114.pdf>
- [3] Process Status API (PSAPI)  
<https://docs.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-enumprocesses>
- [4] Process Status API (PSAPI)  
<https://docs.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-getprocessmemoryinfo>
- [5] Process Status API (PSAPI)  
[https://docs.microsoft.com/en-us/windows/win32/api/psapi/ns-psapi-process\\_memory\\_counters](https://docs.microsoft.com/en-us/windows/win32/api/psapi/ns-psapi-process_memory_counters)