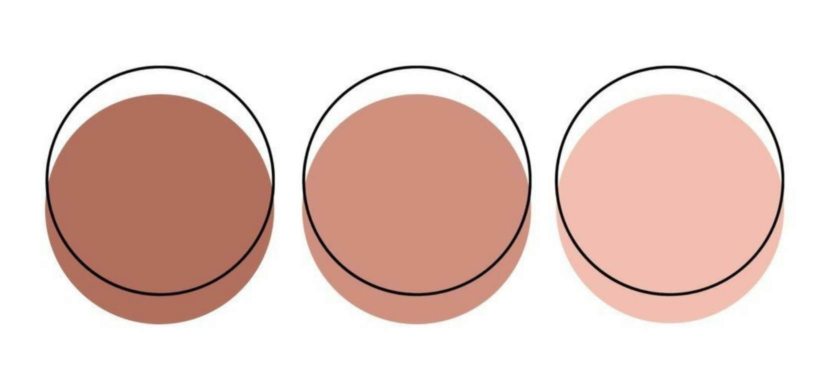
TRABAJO DE INVESTIGACIÓN

Álgebra abstracta



Integrantes	Aportes
❖ Dayevska Anabel Caceres Budiel	**Implementación de los algoritmos, conclusiones, resumen, análisis, introducción, contenido teórico, tablas de comparación, slides.**
❖ Sergio Leandro Ramos Villena	**Implementación de los algoritmos, seguimiento numérico, resumen, introducción, tablas de comparación, gráficas, slides.**
❖ Gabriel Alexander Valdivia medina	**Implementación de los algoritmos, conclusiones, pruebas del desempeño de los algoritmos, introducción, contenido teórico, slides.**
❖ Roberto Juan Cayro Cuadros	**Implementación de los algoritmos, resumen, contenido teórico, introducción, pruebas del desempeño de los algoritmos, slides.**

17/06/2021

CCOMP 3-1

RESUMEN

En este informe analizamos los diversos algoritmos para la generación de números aleatorios y primos, los cuales implementamos considerando distintos conceptos matemáticos mejorandolos en el proceso, siendo evaluados por su periodicidad, complejidad y semilla.

Llegamos así, que nuestro mejor algoritmo para la generación de aleatorios, es el middle square, y para la comprobación de primos Miller-Rabin, gracias a su relativa rapidez y sus resultados.

Contenido:

Descripciones de los algoritmos:

1. Generación de números aleatorios.
    1.1. Algoritmos GLC:

        Algoritmo para la generación de números aleatorios, mediante una escalera de operaciones modulares en secuencia, tomando en cuenta una semilla, a,b y un m.

    1.2. Algoritmo middle square:

        Algoritmo para la generación de números pseudoaleatorios. Al ponerlo en práctica no es un gran algoritmo ya que su periodicidad es usualmente corta. Sin embargo con las mejoras aplicadas, esa periodicidad ha mejorado y ha sido adecuada para generar números en un rango de Bits ingresados junto con una semilla.

2. Test de primalidad.

2.1. Test de Miller-Rabin:

Algoritmo probabilístico que tiene bases en el teorema de Fermat y que encuentra primos a través de diferentes intentos. Cada intento puede confirmar que el número en cuestión es un posible primo, o que es un compuesto. Si en una de las pruebas se determina que es compuesto, inmediatamente se declara al número como tal. Sin embargo, si todas las pruebas dan como resultado un posible primo, el número se declara así.

2.2. Divisibility Test:

Algoritmo simple de fuerza bruta que divide un número entre todos los menores a su raíz para determinar si es primo o no.

2.3. Criba de Eratóstenes:

Algoritmo que en esencia usa un vector que no tiene un límite de tamaño, funciona dependiendo de la capacidad del computador, este vector está compuesto por nodos, donde cada uno posee una key única para identificarlo y poder sacar su valor primo.

Se ingresa un número RSA donde hallamos un vector de nodos con números primos del rango de [(2**RSA)/2 , (2**RSA)-1]

INTRODUCCIÓN

En el presente informe se busca analizar el concepto de los algoritmos de Generación aleatoria y generación de primos, mediante algoritmos que prometen cumplir con los con diversos tipos de mejoras de cada uno, como trabajar con recursiones o bucles. Para solucionar el problema de una generación pseudo aleatoria, ya sea para aleatorios o primos.

Concepto de congruencia:

La congruencia es la inversa que puede utilizar la inversa de Euler. Donde se utilizan diferentes conceptos como “pi” de Euler que nos permite realizar comparaciones que nos permitan identificar si un número es congruente.

$$A \equiv B \pmod{3}$$

Teorema base de la divisibilidad:

Sean a,b que pertenecen a Z, con b >0 , entonces q,r pertenecen a Z, únicos tales que:

$$a = q*b+r;$$

Cada algoritmo funciona de una manera diferente al anterior, por ejemplo la congruencia lineal, utiliza una serie de operaciones sucesivas, muy similares a la recursividad, mientras que el middle square, escoge los números del centro elevarlos al cuadrado

Y a su vez en cada uno, se implementó cierto tipo de mejora.

CONTENIDO TEÓRICO:

➔ Algoritmo GLC

Descripción:

El algoritmo, tiene como base matemática la congruencia lineal, pero para su implementación se tuvo en cuenta, que para los valores a y b, se deben tomar números distintos, es decir no elegidos por el usuario, de modo que se aumenta la aleatoriedad del algoritmo, a su vez, se tuvieron que dejar valores predeterminados en la selección, para la generación de los números, ya que el algoritmo, regresa un ZZ pero este debe cumplir con un tamaño específico, correspondiente a los bits, entonces para cada variante de bits, se tiene planteadas algunas variables que aseguran dicho tamaño correspondiente de los bits, esto es una gran desventaja para este algoritmo, volviendo más predecible aun, tomando en cuenta que se tratan de operaciones modulares sucesivas y pueden ser seguidas con facilidad, por otra parte este algoritmo tiene una rapidez considerable gracias a esto, la aleatoriedad de este algoritmo proviene de la semilla que se le da, la cual viene de la posición x del mouse, aunque si el mouse permanece en el mismo lugar resulta el mismo número.

Para la implementación se tienen como valores predeterminados m,a, pero la semilla y el valor b, son determinados por la posicion del mouse

Congruencia:

$$a \equiv b \pmod{n}$$

Decimos que dos números son congruentes en un modulo m, cuando el resto de ambos, en modulo m, son iguales.

Utilizando este concepto, se comienza con el proceso:

$$x_i \equiv ax_{i-1} + b(mod\ m)$$

Se realizara esta operación de forma sucesiva, comenzando con un x0 que vendría a ser nuestra semilla, y de manera recurrrente operar con el valor de x, en la posicion que se encuentre, siendo esta la nueva semilla

$$x_0 = 14, \qquad a = 5, b = 7, m = 4$$

$$x_1 \equiv (5)14 + 7(mod\ 4) \rightarrow 1$$

$$x_2 \equiv (5)1 + 7(mod\ 4) \rightarrow 0$$

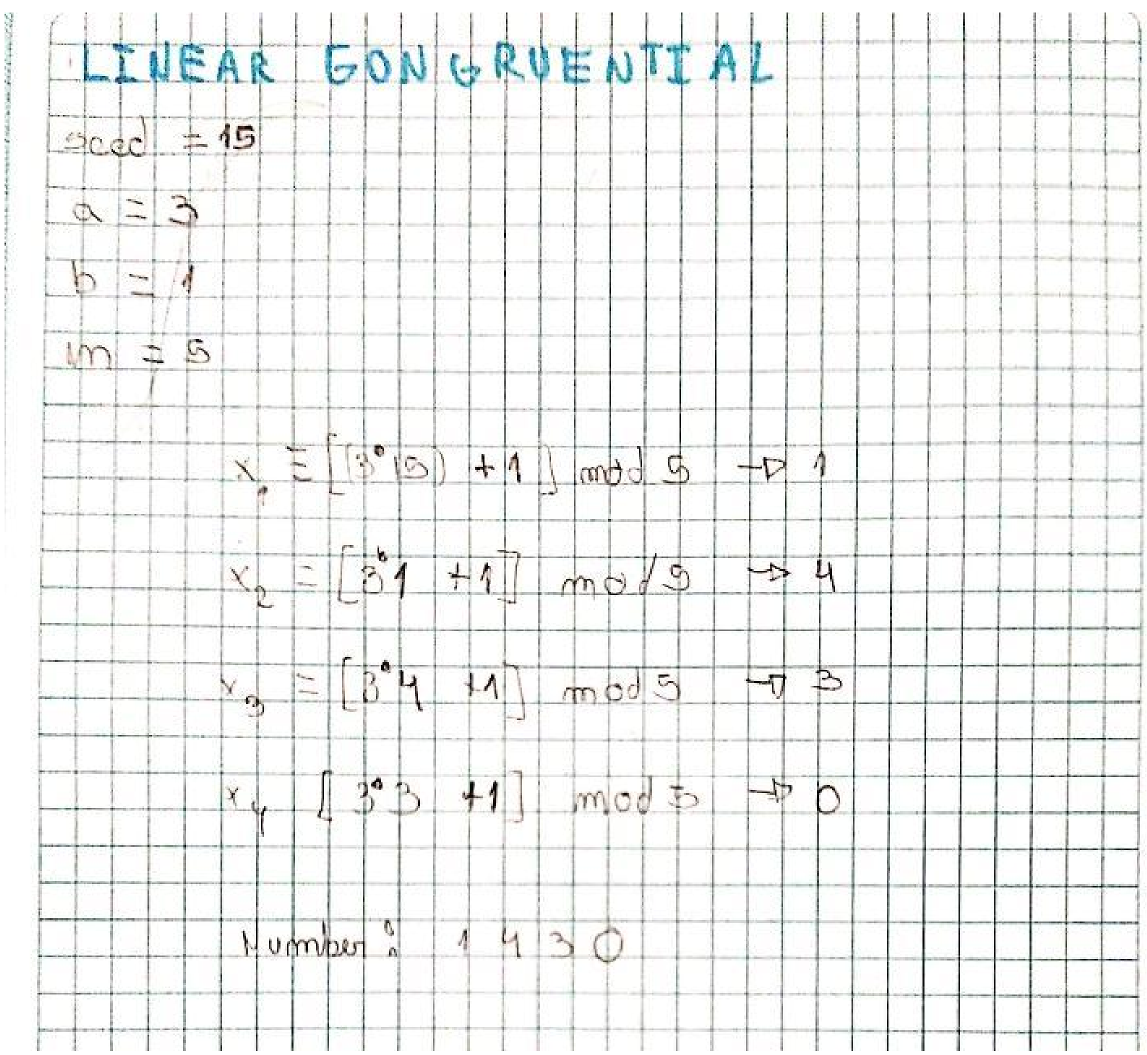
$$x_3 \equiv (5)0 + 7(mod\ 4)$$

Pseudo algoritmo:

```
ZZ a,b,m,seed, x0

for (int i=0; i< 3; i++){

x0 = ((a*seed)*b) mod m;

seed = x0;

return x0

}
```

Seguimiento numérico:

x	seed	a	b	m	resul
x1	15	3	1	5	1
x2	1	3	1	5	4
x3	4	3	1	5	5
x4	5	3	1	5	0
					1450



Tiempos de ejecución:

```
Process returned 0 (0x0)   execution time : 0.023 s
```

```
Process returned 0 (0x0)   execution time : 0.020 s
```

```
ocess returned 0 (0x0)   execution time : 0.018 s
```

➔ Algoritmo middle square

Descripción:

Para generar una secuencia de números pseudoaleatorios de “n” dígitos, se crea un valor inicial de “n” dígitos y se eleva al cuadrado, dándonos como resultado un número de “2n” dígitos. Solo que con las mejoras se le eleva el número al cubo dando como resultado un nuevo número de “3n” dígitos.

Entonces con los “n” dígitos del medio del resultado nos darían el siguiente número de la secuencia y se devolverá como resultado. Sin embargo, para mejorar el algoritmo se sacaron los primeros “n” dígitos, devolviendolos como resultado y continuando el ciclo.

Según la teoría matemática, para el generador de números de “n” dígitos, el periodo no puede ser superior a 8^n. Pero con las mejoras, el resultado “n” va a estar dentro del rango de bits ingresados, en este caso estará dentro del rango de:

$$intervalo\ mayor\ =\ 2^{bits} - 1$$

$$intervalo\ menor\ =\ 2^{bits}/2$$

Pseudo algoritmo:

Funcion MiddleSquareNumber:

Ingresa: Dos números que serán la semilla y el número de bits.

Salida: Número dentro del rango de bits ingresados.

Cuadrado = semilla * semilla

Respuesta = semilla

intervalo_mayor = $2^{bits} - 1$

intervalo menor = $2^{bits}/2$

While(semilla < intervalo_mayor)

t = longitud de respuesta / 2

limite = 10^{t}

Cuadrado = respuesta / limite

Respuesta = Cuadrado * Cuadrado * Cuadrado

Respuesta = Respuesta mod intervalo_mayor

if(Respuesta < intervalo_menor)

Respuesta += Intervalo_menor

Retornar Respuesta

Seguimiento numérico:

Algoritmo Middle square

Primer pseudo aleatorio

Ingresa: Number = 125
bits = 8

Intervalo_mayor = 255
Intervalo_menor = 128
sqn = number * number
= 125 * 125
sqn = 15625
Next_number = 125

Entra al while

tam1 = 3
t = 3/2 => 1
limite = 10^1 => 10
sqn = 125/10 => 12
next_number = sqn * sqn * sqn
= 12 * 12 * 12
next_number = 1728

Sale del while 1728 < 255 NO

next_number = mod(next_number, intervalo_mayor)
=> 198

retorna $\boxed{198}$

Segundo pseudo aleatorio

Ingresa: Number = 198
bits = 8

Intervalo_mayor = 255
Intervalo_menor = 128
sqn = number * number
= 198 * 198
Sqn = 39204
Next_number = 198

Entra al while

tam1 = 3
t = 3/2 => 1
limite = 10^1 => 10 y sqn = 198/10 => 19
next_number = 19 * 19 * 19
= 6859

Sale del while

6859 < 255 No
next_number =

mod(next_number,
Intervalo_mayor)

=> 229

retorna $\boxed{229}$

Primer Pseudoaleatorio:

sqn	15625	12	12
next_number	125	1728	198
tam	-	3	-
t	-	1	-
límite	-	10	-

Segundo Pseudoaleatorio:

sqn	39204	19	19
next_number	198	6859	229
tam	-	3	-
t	-	1	-
límite	-	10	-

Tiempos de ejecución:

```
Process returned 0 (0x0)   execution time : 0.125 s
```

```
Process returned 0 (0x0)   execution time : 0.105 s
```

```
Process returned 0 (0x0)   execution time : 0.124 s
```

➔ Test de Miller Rabin

Descripción:

El test se basa en lo siguiente:

- Descomponer un número n impar en la forma (2^s *d) +1.
- Si cumple con alguna de estas condiciones, se considera posible primo:
  - $a^{d} \equiv 1 (mod\ n)$
  - $a^{2^{r} \cdot d} \equiv -1 (mod\ n)$, 0 <= r < s
- La primera condición se basa en el pequeño teorema de Fermat $a^{n-1} \equiv 1 (mod\ n)$, suponiendo que en la descomposición no hay ningún factor 2, **d** sería equivalente a n-1.
- La segunda condición se basa en que 1 solo puede tener de raíz 1 o -1.
- a será un número aleatorio en el rango 1 < a < n-1
- En base a ese número aleatorio, se empezará en la función en el modo de la primera condición, y poco a poco se incrementará **r** de acuerdo a la variable s que obtuvimos.
- En caso de que se cumpla alguna condición, se determina como posible primo.
- Para la siguiente prueba, se elige otro número **a** aleatorio.
- Si con otro número a se determina que **n** era compuesto, la primera a será declarada como *strong liar*.

Pseudo algoritmo:

Input #1: n > 3, que sea impar para ser evaluado.

Input #2: k, el número de rondas / pruebas.

Output: False si es compuesto, True si es primo.

Escribe n como $2^r \cdot d + 1$ with con d impar (factorizando n-1).

Loop: repite k veces:

a = random en el rango [2, n – 2]

$x \leftarrow a^d \bmod n$

if x = 1 or x = n – 1 :

continue WitnessLoop

repite r – 1 veces:

$x \leftarrow x^2 \bmod n$

if x = n – 1 :

continue WitnessLoop

return False

return True

Tiempos de ejecución: (1024 bits)

Si es primo:

```
Process returned 0 (0x0)   execution time : 1.442 s
Press any key to continue.
```

```
Process returned 0 (0x0)   execution time : 1.455 s
Press any key to continue.
```

```
Process returned 0 (0x0)   execution time : 1.443 s
Press any key to continue.
```

Si no es primo:

```
Process returned 0 (0x0)   execution time : 0.192 s
Press any key to continue.
```

```
Process returned 0 (0x0)   execution time : 0.174 s
Press any key to continue.
```

```
Process returned 0 (0x0)   execution time : 0.172 s
Press any key to continue.
```

➔ Divisibility test

Descripción:

Se basa en el concepto de que, para comprobar si un número **n** es primo o no, se tiene que comprobar si es divisible o no entre todos los primos menores a las raíz cuadrada de **n.**

En base a esto, comprueba la divisibilidad de un número entre todos los números menores a su raíz.

Pseudo algoritmo:

Input: número n.

Output: true, false.

while r < SqrRoot(n):

    if (r|n):

```
        return false;
    r = r + 1;
return true;
```

Tiempos de ejecución:

Si es primo: Demasiado alto.

Si no es primo:

```
Process returned 0 (0x0)   execution time : 0.048 s
Press any key to continue.
```

```
Process returned 0 (0x0)   execution time : 0.317 s
Press any key to continue.
```

```
Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

➔ Criba de eratóstenes

Descripción:

Algoritmo que usa nodos, uno unido al otro a través de punteros, de esta forma no tiene un límite de tamaño. Este vector construido por nodos consecutivos, funciona dependiendo de la arquitectura del computador donde se esté implementando el código.

Dicho vector contiene un identificador único “key” gracias a este se obtendrá el valor primo que guarda este key, además tiene un puntero que lo unirá al siguiente nodo en caso de haberlo, si no este puntero apuntará a nullptr.

Al tener claro cómo usamos esta especie de vector, el algoritmo de la criba de eratóstenes, recibe un int RSA en donde se hallará su valor mínimo en el rango y el valor máximo: [(2**RSA)/2 , (2**RSA)-1].

Luego se pasa a hacer un for donde recorre todo ese rango de números y a través del algoritmo probabilístico de Test de Miller-Rabin (algoritmo probabilístico) que tiene bases en el teorema de Fermat y que encuentra primos a través de diferentes intentos. Una vez que el algoritmo de Miller-Rabin nos indique que el número en efecto es primo, este número se agrega al final de la lista, con una key única generada por defecto en el constructor.

al terminar de correr el for nuestro puntero head apunta a un nodo con key 1 y el primer valor primo en el rango del RSA y así sucesivamente.

Podemos obtener el tamaño de este vector hecho por nodos con la función size de la clase y obtener el valor de una key en específico con la función de la clase "valor" toma como argumento la key, si la encuentra devuelve su valor primo en caso contrario devuelve 0.

Pseudo algoritmo:

entrada : RSA

salida: vector nodo

```
nodo *h= new nodo (ZZ(0), ZZ(0));

for ( ZZ min=2**RSA/2 ; min<= 2**RSA-1 ; min++){
    if (min es primo)  último puntero del nodo = new nodo(min);
}
```

Corrimiento a mano:

bucle	max	min	vector nodo
1	2**5 -1 = 31	2**5/2 = 16	0
2	31	17	0, 17
3	31	18	0, 17
4	31	19	0, 17, 19
5	31	20	0, 17, 19
6	31	21	0, 17, 19
7	31	22	0, 17, 19
8	31	23	0, 17, 19, 23
9	31	24	0, 17, 19, 23
10	31	25	0, 17, 19, 23
11	31	26	0, 17, 19, 23
12	31	27	0, 17, 19, 23
13	31	28	0, 17, 19, 23
14	31	29	0, 17, 19, 23, 29
15	31	30	0, 17, 19, 23, 29
16	31	31	0, 17, 19, 23, 29
17	31	32 (min<=max)	

Códigos- implementados en C++

- **Linear Congruential Generator**

```
ZZ linear(ZZ seed, ZZ a, ZZ b, ZZ m){
ZZ x0;
string num;
x0 = mod((a*(seed)+b),m);
ostringstream numero;
for (int i=0; i < 3; i++){
  x0 = mod(((a*seed)+b),m);
  seed=x0;
  numero << x0;
   num += numero.str();
}

istringstream salida(num);
ZZ final_ ;
salida>>final_;

return final_;
}
```

- **Algoritmo Middle Square**

```
ZZ newTime()
{

   ZZ x;
   auto millisec_since_epoch =
duration_cast<milliseconds>(system_clock::now().time_since_epoch()).count();
   x = ZZ(millisec_since_epoch);

   x = string_a_int(int_a_string(x).substr(6,9));

   x = pot(x+5,ZZ(3));
   return x;
}

string int_a_string(ZZ conversion)

        {

          ostringstream convertido;

          convertido << conversion;

          return convertido.str();

        }

        ZZ pot(ZZ base, ZZ exponente) {

            if (exponente == 0) return ZZ(1);

            ZZ x = pot(base, exponente/2);

            if (exponente % 2 == 0) return x*x;

            return x*x*base;

        }
```

```
ZZ middleSquareNumber(ZZ number, ZZ bits) {
   ZZ intervalo_mayor = pot(ZZ(2),bits)-1;
   ZZ intervalo_menor = pot(ZZ(2),bits)/2;

   ZZ sqn = number * number, next_number = number;
   while(next_number < intervalo_mayor)
   {
     int tam1  = int_a_string(next_number).length();
     int t = (tam1 / 2);
     ZZ limite = pot(ZZ(10),conv<ZZ>(t));
      sqn =  next_number / limite;
      next_number = sqn * sqn * sqn;

   }
   next_number = residuo(next_number, intervalo_mayor);
   if(next_number < intervalo_menor)
   {
     next_number += intervalo_menor;
   }
   return next_number;
}
```

- **Test de Miller Rabin.**

```
ZZ mod(ZZ a, ZZ b);
ZZ modular_exponentation( ZZ a, ZZ e, ZZ n );
bool MillerRabinTest(ZZ n, ZZ k);
vector<ZZ>fact_2(ZZ n);
ZZ newTime();
string int_a_string(ZZ conversion);
ZZ string_a_int(string conversion);
ZZ pot(ZZ base, ZZ exponente);
ZZ middleSquareNumberRan(ZZ number, ZZ intervalo_mayor, ZZ
intervalo_menor);

int main(){
cout<<MillerRabinTest(conv<ZZ>("1721728409251164498040437802310245496140
8238376070901025989554340719541298014829793715475935432052112192931005
3887632724027983909765199447371871807072059039802809499044676718594095
4546832702086016052796084207550000670215656023954953991424780715867985
1636169066487438866147010438955236285099217993828446005O651"),ZZ(10))<<e
ndl;
}

bool MillerRabinTest(ZZ n, ZZ k){
  bool continueFor=false;

   //1. Write n as 2^r*d + 1; d odd; factoring out of n-1.
   ZZ r(0);
   ZZ d(1);
   vector<ZZ>facts = fact_2(ZZ(n-1));
   r = facts[0];
   d = facts[1];


   //2. WitnessLoop.
   ZZ a,x;
```

```
    for(int i = 0; i < k; i++){      //Repeat k times

        ZZ aux (newTime());
        a = middleSquareNumberRan(aux,n-2,ZZ(2));
        x = modular_exponentation(a,d,n);

        if(x == 1 || x == n - 1)
            continue;
        for(int i = 0; i < n-1; i++){//repeat n-1 times
            x = mod(x*x,n);
            if(x == n-1)
                continueFor=true;
                break;
        }
        if(continueFor==true)continue;
        return false;
    }
    return true;
}

vector<ZZ>fact_2(ZZ n){
    vector<ZZ>facts;
    ZZ a (0);
    while(mod(n,ZZ(2))==0){
        n = n/2;
        a++;
    }
    facts.push_back(a);
    facts.push_back(n);
    return facts;
}

ZZ mod(ZZ a, ZZ b){
            ZZ q= a/b;
            ZZ        r= a- (q*b);
            if(a<ZZ(0)){
                        ZZ ar=r;
```

```
            r= b+ar;
        }
        return r;
}

ZZ modular_exponentation( ZZ a, ZZ e, ZZ n ) {

    ZZ result (1);
    while( e != ZZ(0)) {
        if( mod(e,ZZ(2)) == ZZ(1))
            result  = mod(result*a,n);
        e >>= 1;
        a = mod(a*a,n);
    }
    return result;
}

ZZ newTime()
{

    ZZ x;
    auto millisec_since_epoch =
duration_cast<milliseconds>(system_clock::now().time_since_epoch()).count();
    x = ZZ(millisec_since_epoch);

    x = string_a_int(int_a_string(x).substr(6,9));

    x = pot(x+5,ZZ(3));
    return x;
}

string int_a_string(ZZ conversion){
  ostringstream convertido;
  convertido << conversion;
  return convertido.str();
}
```

```
ZZ string_a_int(string conversion){
  istringstream convertido(conversion);
  ZZ entero;
  convertido >> entero;
  return entero;
}

ZZ pot(ZZ base, ZZ exponente) {
    if (exponente == 0) return ZZ(1);
    ZZ x = pot(base, exponente/2);
    if (mod(exponente,ZZ(2))==0) return x*x;
    return x*x*base;
}

ZZ middleSquareNumberRan(ZZ number, ZZ intervalo_mayor, ZZ
intervalo_menor) {

   ZZ sqn = number * number, next_number = number;
   while(next_number < intervalo_mayor)
   {
     int tam1  = int_a_string(next_number).length();
     int t = (tam1 / 2);
     ZZ limite = pot(ZZ(10),conv<ZZ>(t));
      sqn =  next_number / limite;
      next_number = sqn * sqn * sqn;

   }
   next_number = mod(next_number, intervalo_mayor);
   if(next_number < intervalo_menor)
   {
     next_number += intervalo_menor;
   }
   return next_number;

}
```

- **Divisibility test.**

```
bool divisibility_Test(ZZ n);
ZZ mod(ZZ a, ZZ b);

int main(){
   ZZ a = conv<ZZ>("137188");
   cout<<divisibility_Test(a);
}

bool divisibility_Test(ZZ n){
   ZZ r (2);
   while (r < SqrRoot(n)){
      if (mod(n,r)==0)
         return 0;
      if(r == 2){r += 1;}
      else{r += 2;}
   }
   return 1;
}

ZZ mod(ZZ a, ZZ b){
         ZZ q= a/b;
         ZZ      r= a- (q*b);
         if(a<ZZ(0)){
                  ZZ ar=r;
            r= b+ar;
         }
         return r;
}
```

- **Criba de eratóstenes**

```
class nodo{
public:
  ZZ key, val ;
  nodo *next ;
  nodo (ZZ k, ZZ v, nodo *n=nullptr ) : key (k), val(v), next (n) {}

  bool find_key (nodo* head, ZZ k, nodo *& pos){
    pos = nullptr;
    nodo *t= head->next;
    for ( ; t and t->key < k ;pos= t, t= t->next );
    if(t and t->key == k)  return 1;
    else return 0;
  }

  void add(nodo* head,ZZ v){
    nodo *pos = nullptr;
    nodo *t= head->next;
    for ( ; t ; pos= t, t= t->next );
    pos->next = new nodo( (pos->key)+1, v);
  }


  ZZ valor( nodo* head,ZZ k){ //devuelve el valor usando la key (indice del vector)
    nodo *pos = nullptr;
    nodo *t= head->next;
```

```
        for ( ; t and t->key < k  ; pos= t, t= t->next );
        if(t and t->key == k)  return t->val;
        else return ZZ(0);
    }

    ZZ size(nodo* head){
        nodo *pos = nullptr;
        nodo *t= head->next;
        for ( ; t ; pos= t, t= t->next );
        return pos->key;
    }


    void print_val(nodo* head){
        cout <<"head:" << head->next <<endl;
        for (nodo* temp = head->next; temp ; temp = temp->next){
        cout <<"key:" << temp->key <<endl;
        cout <<"val:" << temp->val <<endl;
        cout <<"next:" << temp->next <<endl;
        cout <<"...................... "<<endl;
       }
    }

    ~nodo(){
        cout << "delete"<<endl;
    }
};
```

```
int main(){
    int RSA=64;
        ZZ pot=power2_ZZ(RSA);
        ZZ max= pot-1;

        nodo *h= new nodo (ZZ(0), ZZ(0));
        nodo *t= h;
        cout <<" rango = [" << pot/2<<", "<<pot-1<<" ]"<<endl;
        for ( ZZ min=pot/2 ;min<= max; min++){
            if ( MillerRabinTest(conv<ZZ>(min),ZZ(10))){
                t->next = new nodo( (t->key)+1, min);
                t= t->next;
            }
        }
   }
```

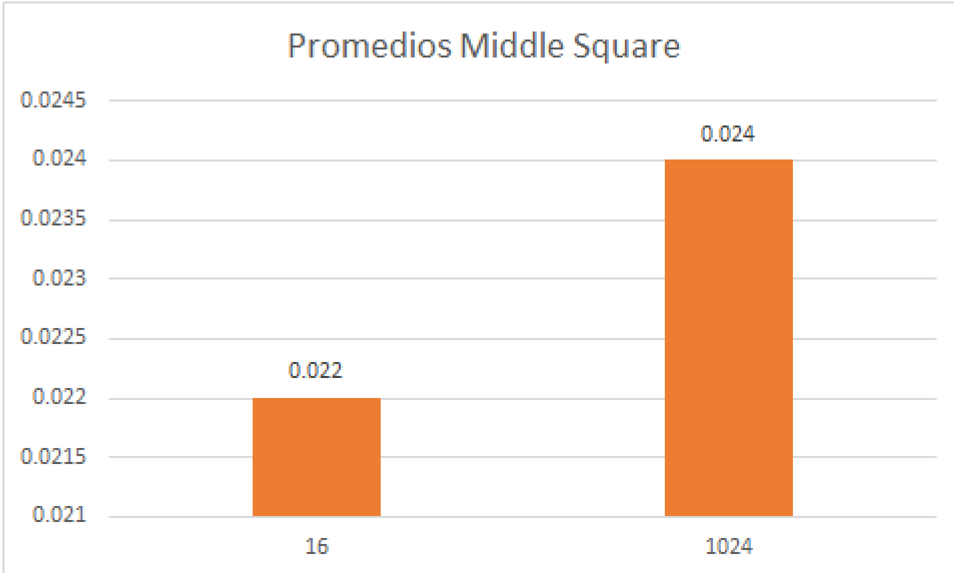
ANÁLISIS DE LOS ALGORITMOS:

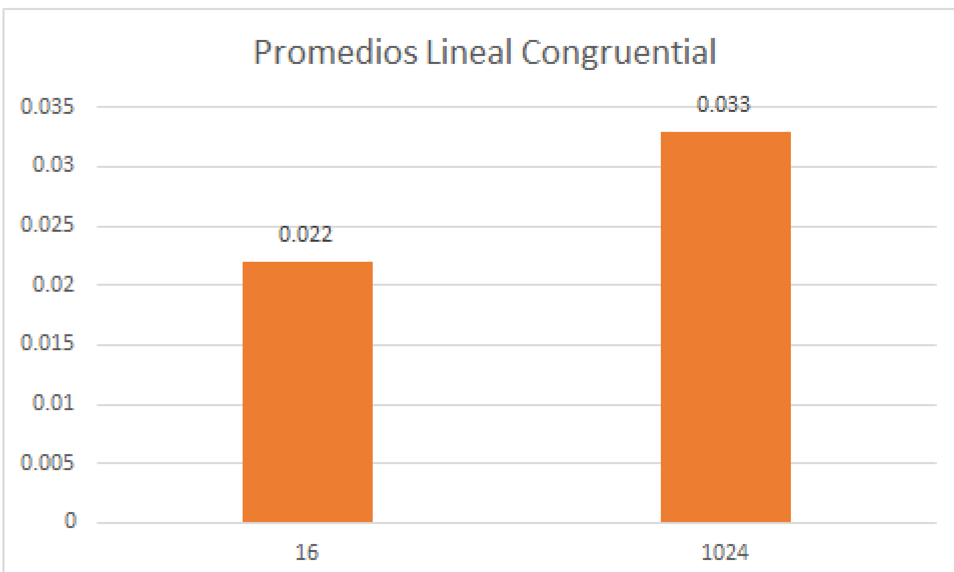
CARACTERISTICAS DEL PROCESADOR

HP Laptop 15-dw1xxx

- Procesador: Intel(R) Core(™) i7-10510U CPU @ 1.80GHz 2.30 GHz
- RAM instalada 12.0 GB (11.8 GB utilizable)
- Id. del dispositivo BFE252F1-EFB9-49C6-A3F2-7FE1602E1FF9
- Id del producto 00327-70000-00001-AA566
- Tipo de sistema Sistema operativo de 64 bits, procesador x64

	# bits	tiempo	memoria
Middle square	16	0,022	6,9 MB
Middle square	1024	0,024	6,8 MB
Linear Congruential	16	0,020	6,9 MB
Linear Congruential	1024	0,033	6,9 MB
Miller-Rabin	1024	1,443 / 0,179	6,9MB
Criba-Eratostenes	1024	Indefinido	Indefinida


Promedios Middle Square
0.0245
0.024
0.0235
0.023
0.0225
0.022
0.0215
0.021
0.024
0.022
16
1024


Promedios Lineal Congruential
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0.033
0.022
16
1024

CONCLUSIÓN

En síntesis, después de haber analizado los algoritmos de generación de aleatorios y los comprobadores de primalidad podemos decir que los algoritmos de mejor desempeño son la generación de middle square, y el test de miller Rabin, la evaluación tomada en cuenta fue respecto a la complejidad del algoritmo en cuestión y la periodicidad.

Durante la construcción del algoritmo se tomó en cuenta una semilla dada por el mismo usuario, y en el proceso de análisis, se optó por implementar otra semilla , esta vez por el lado del hardware, usando tanto la posición x del mouse (lineal congruencial), como el tiempo local de la computadora(middle square). Esto aumenta la aleatoriedad de los números en cuestión, ya que se toma un número x al azar dado en teoría por el usuario.

Si bien los tiempos entre los algoritmos de generación de aleatorios son similares, nosotros determinamos el mejor algoritmo de acuerdo a su complejidad y generación en gran medida, para el generador de congruencia lineal, debido a que se necesitan números predeterminados para la generación del número, no posee una periodicidad buena, ya que estos valores predeterminados comprometen la aleatoriedad, del algoritmo, tomando en cuenta además, que dicho algoritmo funciona de operaciones módulo sucesivas, por lo tanto puede llegar a ser predecible,

En cambio el middle square, es el algoritmo que más aleatoriedad posee, debido a que genera semillas en cuanto al tiempo local en milisegundos. Por lo tanto, la complejidad del algoritmo es medianamente alta, por los procedimientos y librerías que usa. En cambio, a pesar de que se cree un número pseudoaleatorio, las probabilidades de sacar un número primo son relativamente bajas. Por lo que el algoritmo deberá hacer más recursiones para sacar un número primo en poco tiempo.

Por otro lado, para la comprobación de primos, optamos por el test de miller-Rabin, debido a su velocidad, si bien se logró la implementación de la criba de Eratóstenes, esta tenía un gran tiempo de ejecución, llegando a ser mucho mayor a 30 minutos para 64 bits. Si bien el test de Miller-Rabin es un algoritmo probabilístico y no determinístico, si se aplican suficientes repeticiones en la prueba los resultados pueden ser bastante satisfactorios.

Implementación:

ZZ RANDBIGINTEGER (int nroBits){}

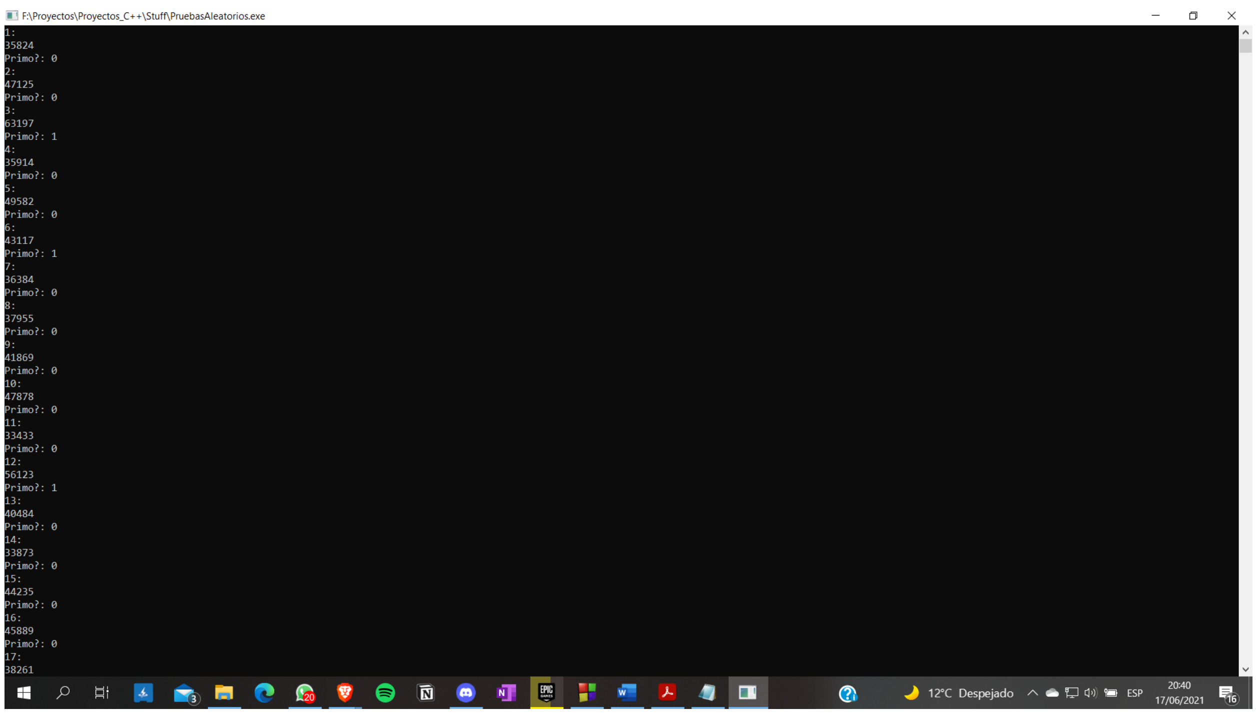
bool PRIMECHECK (ZZ aleatorio){}

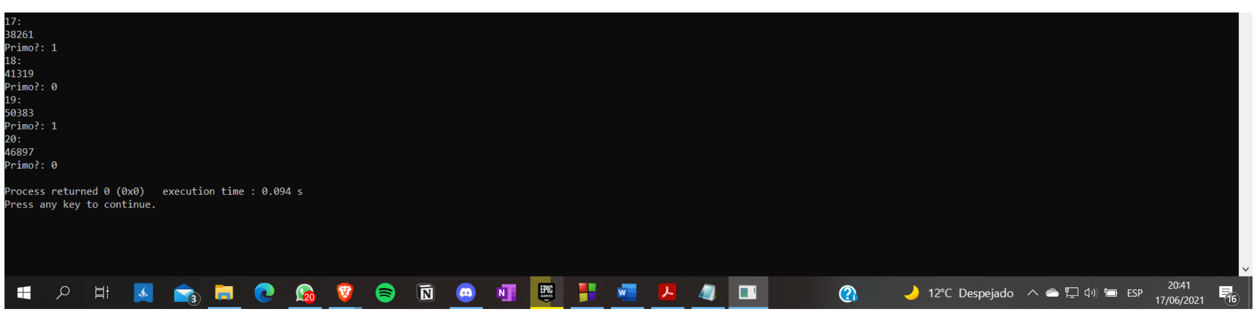
<u>Algoritmo Middle-square</u>

<u>Falso=0</u>

<u>Verdadero=1</u>

16 bits:



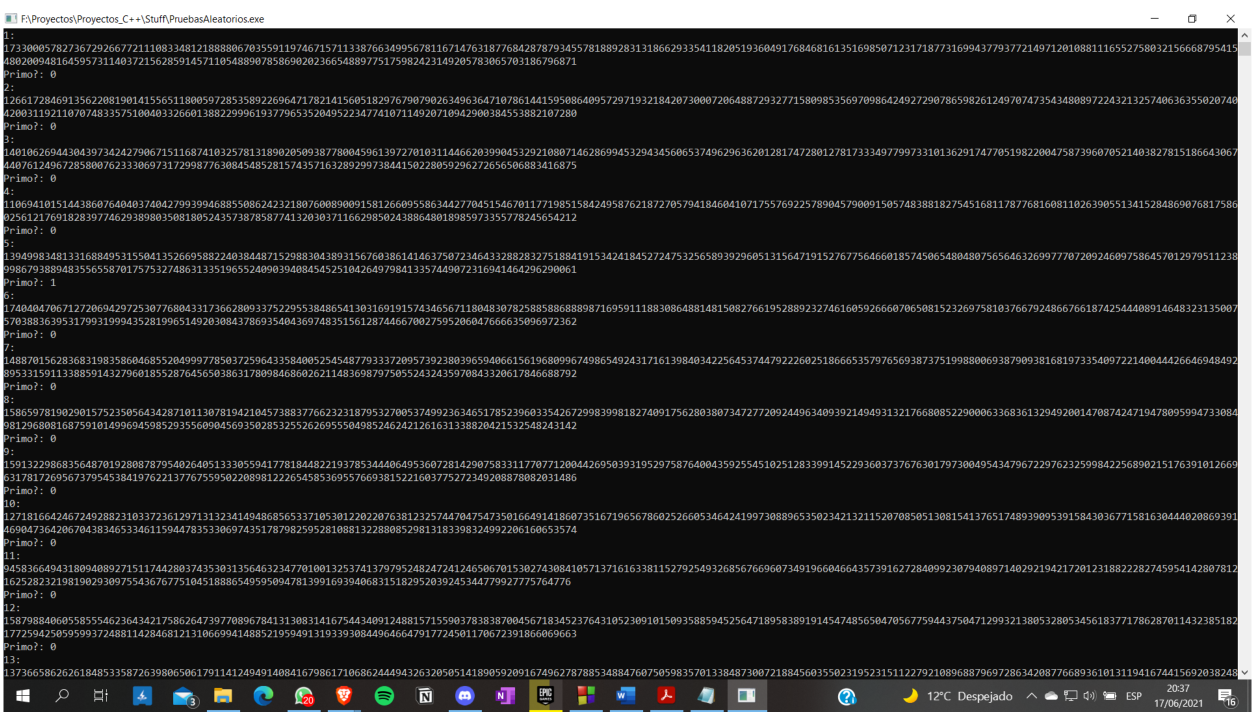


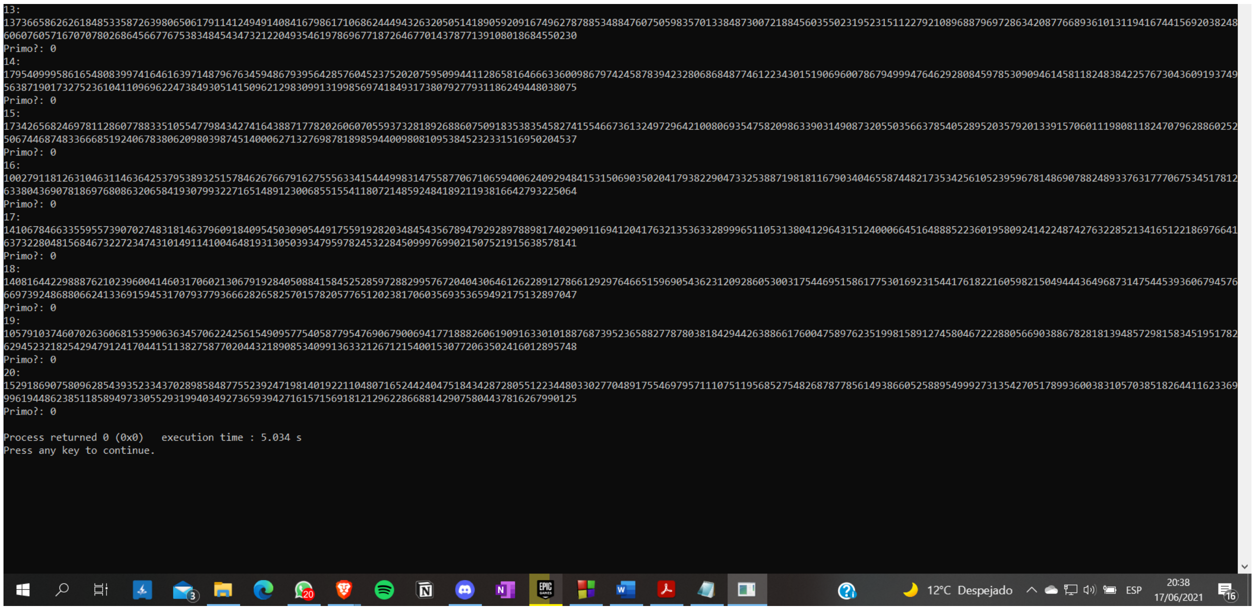
Transcripción:

1	**35824**	0
2	**47125**	0
3	**63197**	1
4	**35914**	0
5	**49582**	0
6	**43117**	1
7	**36384**	0
8	**37955**	0
9	**41869**	0
10	**47878**	0
11	**33433**	0
12	**56123**	1
13	**40484**	0
14	**33873**	0

15	**44235**	0
16	**45889**	0
17	**38261**	1
18	**41319**	0
19	**50383**	1
20	**46897**	0

1024 bits:





Transcripción:

1:

1733000578273672926677211108334812188880670355911974671571133876634995678116
7147631877684287879345578188928313186629335411820519360491768468161351698507
1231718773169943779377214971201088111655275803215666879541548020094816459573
1140372156285914571105488907858690202366548897751759824231492057830657031867
96871

Primo?: 0

2:

1266172846913562208190141556511800597285358922696471782141560518297679079026
3496364710786144159508640957297193218420730007206488729327715809853569709864
2492729078659826124970747354348089722432132574063635502074042003119211070748
3357510040332660138822999619377965352049522347741071149207109429003845538821
07280

Primo?: 0

3:

1401062694430439734242790671511687410325781318902050938778004596139727010311
4466203990453292108071462869945329434560653749629636201281747280127817333497
7997331013629174770519822004758739607052140382781518664306744076124967285800
7623330697317299877630845485281574357163289299738441502280592962726565068834
16875

Primo?: 0

4:

1106941015144386076404037404279939946885508624232180760089009158126609558634
4277045154670117719851584249587621872705794184604107175576922578904579009150
5748388182754516811787768160811026390551341528486907681758602561217691828397
7462938980350818052435738785877413203037116629850243886480189859733557782456
54212

Primo?: 0

5:

1394998348133168849531550413526695882240384487152988304389315676038614146375
0723464332882832751884191534241845272475325658939296051315647191527677564660
1857450654804807565646326997770720924609758645701297951123899867938894835565
5870175753274863133519655240903940845452510426497984133574490723169414642962
90061

Primo?: 1

6:

1740404706712720694297253077680433173662809337522955384865413031691915743465
6711804830782588588688898716959111883086488148150827661952889232746160592666
0706508152326975810376679248667661874254440891464832313500757038836395317993
1999435281996514920308437869354043697483515612874466700275952060476666350969
72362

Primo?: 0

7:

1488701562836831983586046855204999778503725964335840052545487793337209573923
8039659406615619680996749865492431716139840342256453744792226025186665357976
5693873751998800693879093816819733540972214004442664694849289533159113388591
4327960185528764565038631780984686026211483698797505524324359708433206178466
88792

Primo?: 0

8:

1586597819029015752350564342871011307819421045738837766232318795327005374992
3634651785239603354267299839981827409175628038073472772092449634093921494931
3217668085229000633683613294920014708742471947809599473308498129680816875910
1499694598529355609045693502853255262695550498524624212616313388204215325482
43142

Primo?: 0

9:

1591322986835648701928087879540264051333055941778184482219378534440649536072
8142907583311770771200442695039319529758764004359255451025128339914522936037
3767630179730049543479672297623259984225689021517639101266963178172695673795
4538419762213776755950220898122265458536955766938152216037752723492088780820
31486

Primo?: 0

10:

1271816642467249288231033723612971313234149486856533710530122022076381232574
4704754735016649141860735167196567860252660534642419973088965350234213211520
7085051308154137651748939095391584303677158163044402086939146904736420670438
3465334611594478353306974351787982595281088132288085298131833983249922061606
53574

Primo?: 0

11:

9458366494318094089271511744280374353031356463234770100132537413797952482472
4124650670153027430841057137161633811527925493268567669607349196604664357391

6272840992307940897140292194217201231882228274595414280781216252823219819029
3097554367677510451888654959509478139916939406831518295203924534477992777576
4776

Primo?: 0

12:

1587988406055855546236434217586264739770896784131308314167544340912488157155
9037838387004567183452376431052309101509358859452564718958389191454748565047
0567759443750471299321380532805345618377178628701143238518217725942505959937
2488114284681213106699414885219594913193393084496466479177245011706723918660
69663

Primo?: 0

13:

1373665862626184853358726398065061791141249491408416798617106862444943263205
0514189059209167496278788534884760750598357013384873007218845603550231952315
1122792108968879697286342087766893610131194167441569203824860607605716707078
0268645667767538348454347321220493546197869677187264677014378771391080186845
50230

Primo?: 0

14:

1795409995861654808399741646163971487967634594867939564285760452375202075950
9944112865816466633600986797424587839423280686848774612234301519069600786794
9994764629280845978530909461458118248384225767304360919374956387190173275236
1041109696224738493051415096212983099131998569741849317380792779311862494480
38075

Primo?: 0

15:

1734265682469781128607788335105547798434274164388717782026060705593732818926
8860750918353835458274155466736132497296421008069354758209863390314908732055
0356637854052895203579201339157060111980811824707962886025250674468748336668
5192406783806209803987451400062713276987818985944009808109538452323315169502

04537

Primo?: 0

16:

1002791181263104631146364253795389325157846267667916275556334154449983147558
7706710659400624092948415315069035020417938229047332538871981811679034046558
7448217353425610523959678148690788248933763177706753451781263380436907818697
6808632065841930799322716514891230068551554118072148592484189211938166427932
25064

Primo?: 0

17:

1410678466335595573907027483181463796091840954503090544917559192820348454356
7894792928978898174029091169412041763213536332899965110531380412964315124000
6645164888522360195809241422487427632285213416512218697664163732280481568467
3227234743101491141004648193130503934795978245322845099976990215075219156385
78141

Primo?: 0

18:

1408164422988876210239600414603170602130679192840508841584525285972882995767
2040430646126222891278661292976466515969054362312092860530031754469515861775 3
0169231544176182216059821504944436496873147544539360679457666973924868806624
1336915945317079377936662826582570157820577651202381706035693536594921751328
97047

Primo?: 0

19:

1057910374607026360681535906363457062242561549095775405877954769067900694177
1888260619091633010188768739523658827787803818429442638866176004758976235199
8158912745804672228805669038867828181394857298158345195178262945232182542947
9124170441511382758770204432189085340991363321267121540015307720635024160128
95748

Primo?: 0

20:

152918690758096285439352334370289858487755239247198140192211048071652442404751843428728055122344803302770489175546979571110751195685275482687877856149386605258895499927313542705178993600383105703851826441162336999619448623851185894973305529319940349273659394271615715691812129622866881429075804437816267990125

Primo?: 0

Link Github:

https://github.com/GalexVM/ProbabilisticAndPrimality

REFERENCIAS:

Miller, Gary L. (1976), "Riemann's Hypothesis and Tests for Primality", Journal of Computer and System Sciences, 13 (3): 300–317, doi:10.1145/800116.803773,S2CID 10690396

Rabin, Michael O. (1980), "Probabilistic algorithm for testing primality", Journal of Number Theory, 12 (1): 128–138, doi:10.1016/0022-314X(80)90084-0

Meng Xiannong. “Linear Congruential Method.” *bucknell*, 2002, https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node40.html.

Shu Tezuka. “Linear Congruential Generator.” *Springer link* https://link.springer.com/chapter/10.1007/978-1-4615-2317-8_3.

Meenakshi rana. “Linear Congruential Generator Method | Random Numbers.” *Youtube* https://www.youtube.com/watch?v=LUusa5Mhx_g.