Identificarea Numerelor Prime: Solovay-Strassen si Fermat

Analiza Algoritmilor

Moldovan Gabriel-Nicolae

Universitatea Politehnica Bucuresti, Facultatea de Automatica si Calculatoare, Specializarea Calculatoare, Grupa 325CD

just.gaga97@icloud.com

1 Introducere

1.1 Tema aleasa

Pentru aceasta tema am ales subiectul “Identificarea Numerelor Prime”. Algoritmii pe care am ales sa ii studies sunt algoritmul Solovay-Strassen si algoritmul Fermat.

1.2 Exemple de utilizare in viata reala

Proiectarea cutiilor de viteze. Un exemplu de aplicatie real ce foloseste numerele prime este reprezentat de proiectarea cutiilor de viteze pentru automobile. Pentru a evita blocarea a doua rotite zimtate de dimensiuni diferite s-a observant ca daca numarul de dinti de pe fiecare rotita este un numar prim, atunci la un ciclu complet fiecare dinte de prima rotita se va intra in contact cu fiecare dinte de pe urmatoarea rotita. Pentru a ilustra un exemplu putem presupune ca avem doua rotite, una cu 3 dinti iar cealata cu 7:

1234567123456712345671234567123456712345671234567
1231231231231231231231231231231231231231231231231

Prima linie reprezinta dintii rotitei mari iar a doua linie reprezinta dintii rotitei mici. Se observa ca fiecare dinte de pe rotita mare intra in contact cu fiecare dinte de pe a doua rotita. Algoritmul Solovay-Strassen poate fi de folos in aceasta aplicatie.

Criptografie. Un alt exemplu foarte folosit se regaseste in criptografie. S-a descoperit ca folosind doua numere prime inmultite un cod devine mult mai greu de spart decat daca ar fi folosit alt procedeu deoarece un numar prim are doar 4 factori:

cifra unu, numarul insusi si cele doua numere folosite in produs. Ma voi lega de acest exemplu interpretand algoritmul Fermat.

2 Prezentarea si implementarea solutiilor

2.1 Algoritmul Solovay-Strassen

Algoritmul Solovay-Strassen de determinare a numerelor prime, este un test probabilist ce are ca scop sa determine daca un numar este compus sau probabil prim.

Inainte de prezentarea codului propriu zis va trebuii sa intelegem cateva concept si termini cheie pentru a fi capabili sa intelegem imprementarea acestuia.

Simbolul Legendre:

Acest simbol este definit pentru o pereche de numere intregi a si p astfel incat p sa fie numar prim. Este notat *(a/p)* si este calculate in felul urmator:

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & daca\ a\ \%\ p\ = 0 \\ 1, & daca\ exista\ un\ intreg\ k\ astfel\ incat\ k^2 = a(mod\ p) \\ -1, & in\ restul\ cazurilor \end{cases}$$

Eurler a dovedid faptul ca:

$$\left(\frac{a}{p}\right) = a^{\frac{(p-1)}{2}} \%\ p \quad \leftarrow \textit{Conditia (i)}$$

. Simbolul Jacobian:

Aces simbol este o generalizare a simbolului Legendre, unde p este inlocuit de catre n, unde n are valoarea:

$$n = p_1 k_1 * \ldots * p_n k_n$$

Apoi simbolul Jacobian este definit astfel:

$$\left(\frac{a}{n}\right) = (\left(\frac{a}{p1}\right)^{k1} * \left(\frac{a}{p2}\right)^{k2} * \ldots * \left(\frac{a}{pn}\right)^{kn}$$

Algoritmul.

Selectam un numar n pentru a verifica daca este prim sau nu, si un numar aleator, a, ce apartine intervalului [2, n - 1] si ii calculam Jacobianul (a/n), daca n este numar prim atunci Jacobianul va fi egal cu Legendre-ul si va satisfice *conditia (i)* conform teoriei lui Euler. Daca nu satisfice aceasta conditie atunci n este numar compus si programul se va opri.

Din cauza ca este un test probabilist acuratetea sa este direct proportional cu numarul de iteratii. Deci vom rula testul pentru un anumit numar de iteratii in functie de ce acuratete dorim.

Nota: Nu ne intereseaza sa calculam Jacobianul unui numar par deoarece stim deja ca acestea nu sunt prime exceptie fiind doar numarul 2.

Pseudocod.

```
Algoritm pentru Jacobi:
Pasul 1    //omitem cazurile de baza
Pasul 2     if a>n then
Pasul 3         return ((a mod n)/n)
Pasul 4     else
Pasul 5         return (-1)^(^(^a^-^1^)^/^2^)^(^(^n^-
^1^)^/^2^)^(a/n)
Pasul 6     endif


Algorithm for Solovay-Strassen:
Pasul 1    Alegem aleator un intreg a, a < n
Pasul 2    if cel_mai_mic_divizor_comun(a, n) > 1 then
Pasul 3        return nrCompus
Pasul 4    end if
Pasul 5    Calculam a^(^n^-^1^)^/^2 folosind ridicarea la
patrat repetata
           si (a/n) folosind algoritmul Jacobian.
Pasul 6    if (a/n) != a^(^n^-^1^)^/^2 then
Pasul 7        return nrCompus
Pasul 8    else
Pasul 9        return nrPrim
Pasul 10   endif
```

Implementare.

Pentru simplitate am ales sa implementez acest algorit in Java.

```
/**
 ** Program Java pentru implementarea algoritmului de de-
terminare a numerelor prime, Solovay Strassen
 **/

import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Random;

/** Clasa SolovayStrassen **/
public class SolovayStrassen {
    /** Metoda pentru calculul Jacobi (a/n) **/
    public long Jacobi(long a, long b) {
        if (b <= 0 || b % 2 == 0)
            return 0;
        long j = 1L;
        if (a < 0) {
            a = -a;
            if (b % 4 == 3)
                j = -j;
        }
        while (a != 0) {
            while (a % 2 == 0) {
                a /= 2;
                if (b % 8 == 3 || b % 8 == 5)
                    j = -j;
            }

            long temp = a;
            a = b;
            b = temp;

            if (a % 4 == 3 && b % 4 == 3)
                j = -j;
            a %= b;
        }
        if (b == 1)
            return j;
```

```
        return 0;
    }

    /** Metoda ce verifica daca n este prim **/
    public boolean isPrime(long n, int iteration) {
        /** base case **/
        if (n == 0 || n == 1)
            return false;
        /** base case - 2 is prime **/
        if (n == 2)
            return true;
        /** an even number other than 2 is composite **/
        if (n % 2 == 0)
            return false;

        Random rand = new Random();
        for (int i = 0; i < iteration; i++) {
            long r = Math.abs(rand.nextLong());
            long a = r % (n - 1) + 1;
            long jacobian = (n + Jacobi(a, n)) % n;
            long mod = modPow(a, (n - 1) / 2, n);
            if (jacobian == 0 || mod != jacobian)
                return false;
        }
        return true;
    }

    /** Metoda ce calculeaza (a ^ b) % c **/
    public long modPow(long a, long b, long c) {
        long res = 1;
        for (int i = 0; i < b; i++) {
            res *= a;
            res %= c;
        }
        return res % c;
    }

    /**
     * Metoda Main
     *
     * @throws FileNotFoundException
     **/
    public static void main(String[] args) throws File-
NotFoundException {
        Scanner scan = new Scanner(System.in);
```

```
        System.out.println("SolovayStrassen Primality Al-
gorithm Test\n");
        SolovayStrassen ss = new SolovayStrassen();
        System.out.println("Enter number\n");
        long num = scan.nextLong();
        System.out.println("\nEnter number of itera-
tions");
        int k = scan.nextInt();

        boolean prime = ss.isPrime(num, k);
        if (prime)
            System.out.println("\n" + num + " is prime");
        else
            System.out.println("\n" + num + " is compo-
site");
    }
}
```

2.2 Algoritmul Fermat

Algoritmul Fermat de determinare a numerelor prime, este o metoda probabilista si se bazeaza pe teorema lui Fermat de mai jos:

Daca n este numar prim, atunci pentru fiecare a din intervalul [1, n):

$$a^{n-1} \equiv 1 \ (mod\ n)$$

Sau

$$a^{n-1} \,\%\, n = 1$$

Exemplu:

Din moment ce 5 este un numar prim, $2^4 \equiv 1 \ (mod\ 5)$, $3^4 \equiv 1 \ (mod\ 5)$ si $4^4 \equiv 1 \ (mod\ 5)$

Din moment ce 7 este numar prim, $2^6 \equiv 1 \ (mod\ 7)$, $3^6 \equiv 1 \ (mod\ 7)$ si $4^6 \equiv 1 \ (mod\ 7)$, $5^6 \equiv 1 \ (mod\ 7)$ si $6^6 \equiv 1 \ (mod\ 7)$

Daca un numar este prim, atunci aceasta metoda va returna intotdeauna true. Daca numar primit este compus, atunci este posibil sa returneze true sau false, dar probabilitatea de a produce un rezultat eronat este mica si poate fi redusa prin efectuarea unui numar mai mare de iteratii.

Pseudocod:

```
// Cu cat este mai mare valoarea lui k, cu atat probabil-
itatea de a intoarce un rezultat eronat este mai mica.
// Pentru input-uri de numere prime, rezultatul va fi
intotdeauna corect
1)  for(i = 0 to k - 1):
      a) Se alege un intreg aleator din intervalul [2, n
- 2]
      b) Daca a^(n-1)  !≡ 1 (mod n) atunci intoarce false
2) Intoarce true, rezulta ca numarul este probabil prim.
```

Implementare:

```
/**
 ** Program java pentru implementarea algoritmului
 ** Fermat de determinare a numerelor prime
 **/

import java.util.Scanner;
import java.util.Random;

public class FermatPrimality {
    /** Metoda pentru verificarea primalitatii **/
    public boolean isPrime(long n, int iteration) {
        /** Cazuri de baza de baza **/
        if (n == 0 || n == 1)
            return false;
        /** Cazul pentru cifra 2 **/
        if (n == 2)
            return true;
        /** Orice numar par inafara de 2 este compus **/
        if (n % 2 == 0)
            return false;

        Random rand = new Random();
        for (int i = 0; i < iteration; i++) {
            long r = Math.abs(rand.nextLong());
            long a = r % (n - 1) + 1;
            if (modPow(a, n - 1, n) != 1)
```

```
                return false;
        }
        return true;
    }

    /** Metoda pentru calcul: (a ^ b) % c **/
    public long modPow(long a, long b, long c) {
        long res = 1;
        for (int i = 0; i < b; i++) {
            res *= a;
            res %= c;
        }
        return res % c;
    }

    /** Metoda Main **/
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Fermat Primality Algorithm
Test\n");
        /** Instantiem un obiect de tipul FermatPrimality
**/
        FermatPrimality fp = new FermatPrimality();
        /** Citeste numarul **/
        System.out.println("Enter number\n");
        long num = scan.nextLong();
        /** Citeste numarul de iteratii **/
        System.out.println("\nEnter number of itera-
tions");
        int k = scan.nextInt();
        /** Verifica daca numarul este prim **/
        boolean prime = fp.isPrime(num, k);
        if (prime)
            System.out.println("\n" + num + " is prime");
        else
            System.out.println("\n" + num + " is compo-
site");
    }
}
```

3 Complexitate si testare

3.1 Complexitatea si testarea algoritmului Solovay-Strassen

Fiind un algoritm probabilist, sansa ca programul sa intoarca un raspuns eronat este invers proportional cu numarul de iteratii folosit.

Practic, algoritmul consta in verificarea egalitatii dintre Jacobian-ul si Legendre-ul numarului de verificat, de k ori (numarul de iteratii), de aceea, daca numarul de iteratii este mai mic decat numarul de verificat exista sansa ca programul sa omita anumite numere “a” pentru care Jacobian-ul si Legendre-ul ar fi diferite. Asadar, in acest caz programul va intoarce “true”, adica numarul este prim, desi exista posibilitatea ca verificand si restul Jacobian-elor si Lgendre-urilor sa observam ca numarul este de fapt compus.

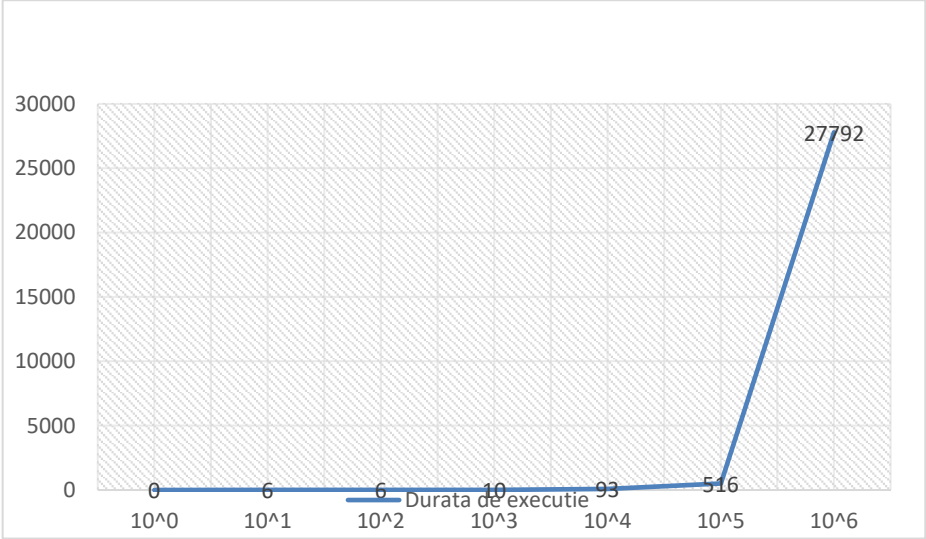
Complexitatea algoritmului pentru un singur numar este:

$O(k * \log^3 n)$, unde:
k reprezinta numarul de iteratii
n reprezinta numarul ce este supus verificarii

Analiza algoritmului:

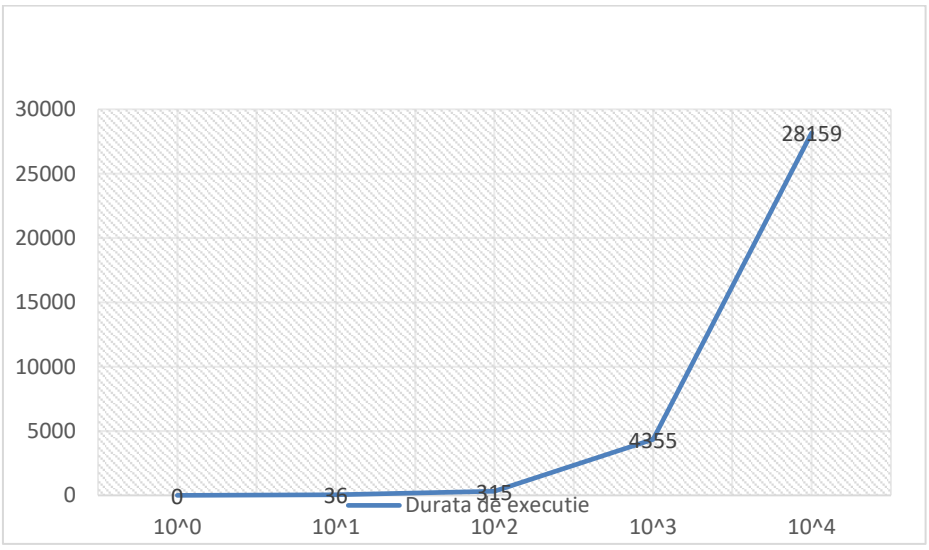
Voi realiza analiza algoritmului pe trei cazuri asupra unui vector cu 100 de elemente numere intregi generate aleator:

1) Cazul in care vom folosi un numar mic de iteratii fixate, 100, pentru a obtine un timp de rulare scurt. In acest caz numerele din vector vor avea initial valori cuprinse inre 0 si 10, urmand ca limita superioara sa creasca de 10 ori la fiecare test, pana ajungem la o limita superioara de 10^6.



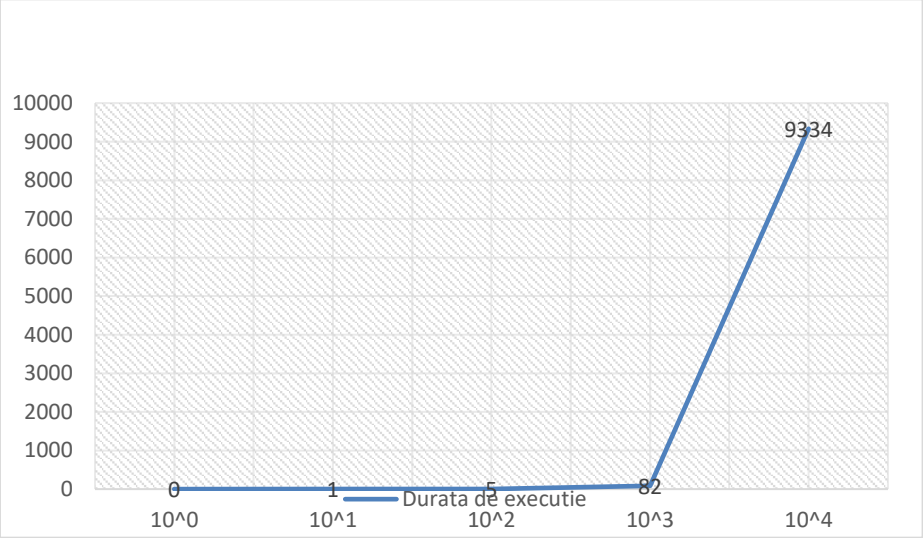
Din acest graphic putem observa ca pentru numere mai mici de 10^5 timpul de rulare este foarte mic. Din pacate, din cauza numarului mic de iteratii, acuratetea programului scade exponential odata cu cresterea numerelor.

2) In urmatorul caz vom fixa limita superioara a intervalului din care vor fi generate numerele la 10^6 si vom varia numarul de iteratii prin puterile lui 10 incepand de la 10^1 si pana la 10^4



Se observa ca, cu cat numarul de iteratii creste, creste si durata de executie a programului, iar acesta inca nu ajunge la o acuratete de 100%.

3) In continuare vom continua vom studia cazul in care dorim acuratete maxima si vom observa pentru ce fel de numeral acest program ruleaza intr-un timp de-cent.



Observam ca, in cazul ideal, atunci cand numerele studiate sunt mai mici de 10^3, algoritmul este foarte eficient si ruleaza intr-un timp foarte mic cu o acuratete maxima, deoarece numarul de iteratii ales este egal cu limita superioara a valorii pe care un numar din vectorul nostrum o poate lua. Dar odata cu cresterea numerelor timpul de executie la acuratete maxima creste rapid.

3.2 Complexitatea si testarea algoritmului Fermat

La fel ca si algoritmul Solovay-Strassen, algoritmul Fermat este unul de tip probabilist, asa ca nu va intoarce intotdeauna un rasuns corect. Programul are la baza teorema lui Fermat enuntata mai sus, verificand daca pentru un numar k, numarul de iteratii, cazuri, un numar respecta aceasta teorema. Programul poate intoarce un raspuns eronat deoarece se pot omite unul sau mai multe cazuri in care teorema lui Fermat nu va fi validate.

Acuratetea algoritmului creste direct proportional cu numarul de iteratii si cu timpul de rulare.

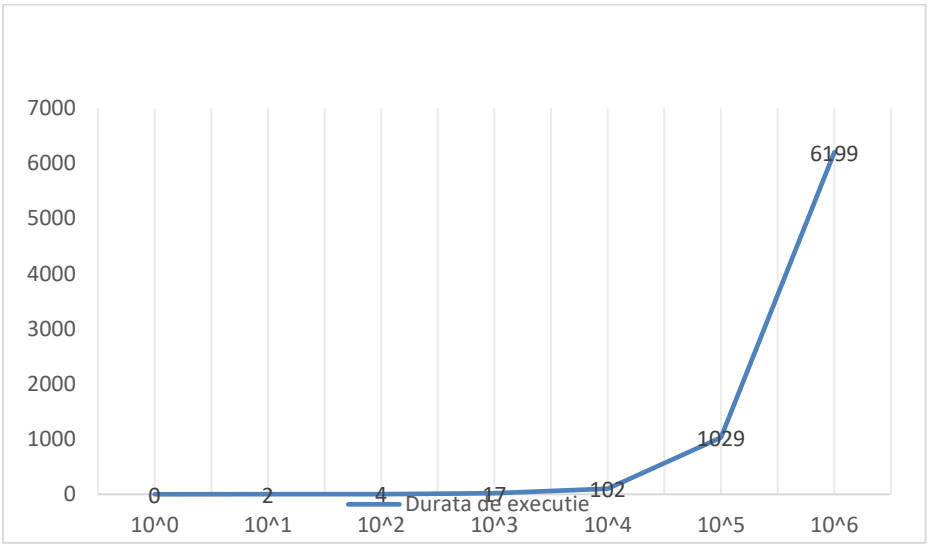
Complexitatea algoritmului pentru un singur numar este:

O(k * log n), unde:
k reprezinta numarul de iteratii
n reprezinta numarul ce este supus verificarii

Analiza algoritmului:

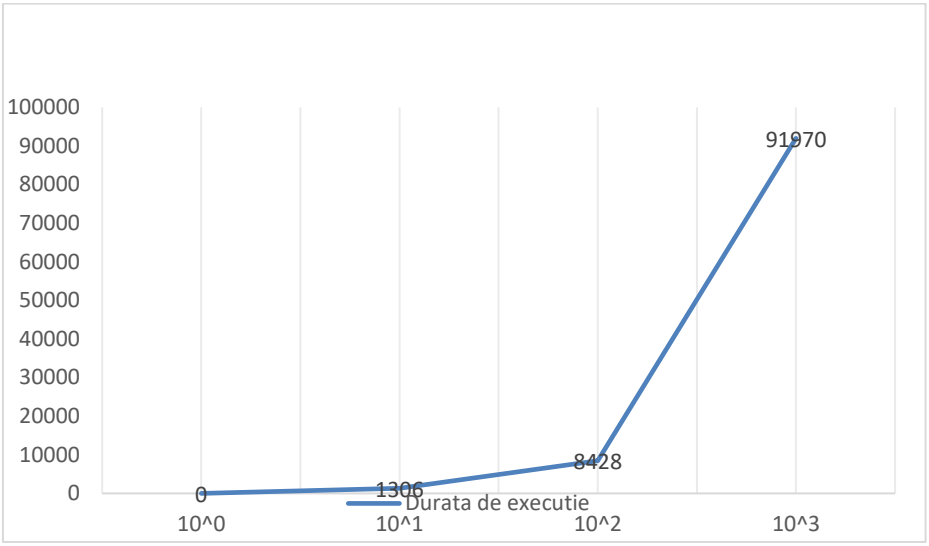
Analiza acestui algoritm va fi similara cu cea facuta precedent asupra algoritmului Solovay-Strassen. Asadar vom studia trei cazuri, asupra unui vector cu 100 de elemente numere intregi:

1) Cazul in care vom folosi un numar mic de iteratii fixate, 100, pentru a obtine un timp de rulare scurt. In acest caz numerele din vector vor avea initial valori cuprinse inre 0 si 10, urmand ca limita superioara sa creasca de 10 ori la fiecare test, pana ajungem la o limita superioara de 10^6.



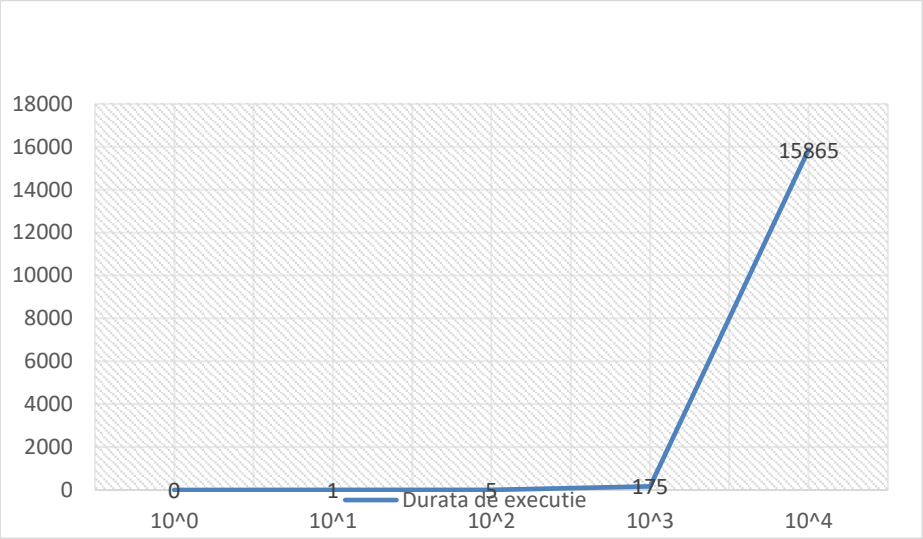
Se observa ca in acest caz timpul de rulare este mult mai scurt decat in cazul algoritmului anterior. Dar observam aceeasi problema, odata cu cresterea numerelor, acuratetea scade mult din cauza numarului mic de iteratii.

1) In urmatorul caz vom fixa limita superioara a intervalului din care vor fi generate numerele la 10^6 si vom varia numarul de iteratii prin puterile lui 10 incepand de la 10^1 si pana la 10^3



Pe acest caz algoritmul Solovay-Strassen se observa ca ruleaza mult mai rapid, numarul ridicat de iteratii nefiind pe placul algoritmului Fermat.

2) In continuare vom continua vom studia cazul in care dorim acuratete maxima si vom observa pentru ce fel de numere acest program ruleaza intr-un timp decent.



Din nou, odata cu trecerea pragului de numere mai mari decat 1000, cresterea timpului de rulare atunci cand se doreste acuratete maxima este semnificativa.

4 Concluzii

In urma testelor effectuate asupra celoi doi algoritmi, am observant ca Fermat este mai eficient decat Solovay-Strassen in cazul numerelor mai mici de 10^3, acesta fiind un prag dupa care ambii algorimti se plafoneaza oferind un timp de rulare cu o crestere foarte rapida. Deci, atunci cand avem de lucru cu mai multe elemente de ordin mic se prefera folosirea algoritmului Fermat iar atunci cand este necesar lucrul cu numere mari, comportamentul prezentat de Solovay-Strassen ne face sa preferam utilizarea acestui algoritm.

5 Bibligrafie

1) https://www.reddit.com/r/math/comments/nuob8/what_are_some_applications_of_prime_numbers_in/?st=ja051c0n&sh=3f029823 - cutia de viteze
2) https://www.youtube.com/user/EngineeringExplained
3) http://www.odec.ca/projects/2007/fras7j2/uses.htm
4) http://www.geeksforgeeks.org/primality-test-set-2-fermet-method/
5) http://www.geeksforgeeks.org/primality-test-set-4-solovay-strassen/
6) http://www.sanfoundry.com/java-program-solovay-strassen-primality-test-algorithm/
7) http://www.sanfoundry.com/java-program-implement-fermat-primality-test-algorithm/