Temă Analiza Algoritmilor

- Algoritmi de determinare a numerelor prime -

Cînjău Constantin-Iulian
grupa 323CD

Universitatea Politehnică București
Facultatea Automatică și Calculatoare
`constantin.cinjau@stud.acs.upb.ro`

Abstract. Aceasta tema consta in analiza a doi algoritmi de testare a numerelor prime: Algoritmul Fermat si Miller-Rabin, prin care vom evidentia diferentele de complexitate, durata de executie si rata de succes dintre acesti algoritmi, ce avantaje si dezavantaje prezinta fiecare in parte si de ce merita folositi pentru a calcula primalitatea unui numar atunci cand avem nevoie in aplicatii practice.

Keywords: Numar prim · Numar compus · Criptografie

1 Introducere

1.1 Utilizarea problemei in practica

Criptografia este unul dintre cele mai importante domenii ce are la baza utilizarea numerelor prime cu scopul asigurarii securitatii datelor si protectia acestora atunci cand noi le introducem pe anumite site-uri de pe internet.

De asemenea, tot sistemul de securitate online bancar se foloseste in mare masura de numerele prime si proprietatile pe care acestea le au, spre exemplu fara criptarea datelor nu am mai putea face plati online in siguranta(modalitate de plata tot mai raspandita in zilele noastre). Practic, fara criptografie, implicit, fara numere prime, nu ar mai putea exista sisteme de comunicatii sigure, deoarece odata cu lipsa acesteia se pierd multe lucruri esentiale precum:confidentialitatea si integritatea datelor dintr-un astfel de sistem.

Algoritmi de encriptare precum RSA creeaza chei private si publice folosindu-se de numere prime. Cu cat se poate calcula mai repede primalitate unui numar cu atat se pot construi niste chei cu un ordin de complexitate mai mare ce pot fi piratate mult mai greu.

1.2 Solutiile alese pentru rezolvarea problemei

In continuare am incercat sa prezint cum se aplica fiecare dintre cei doi algoritmi, explicand in mai multi pasi ideile fiecaruia.

Algoritmul Miller-Rabin

Algoritmul Miller-Rabin este folosit pentru a calcula daca un numar este probabil prim, spunem probabil pentru ca acesta nu ne poate confirma in toate cazurile daca un numar este prim sau nu.

Pasii algoritmului

- Primul pas consta in calcularea lui n-1 in urmatorul mod: n-1 = m * 2^k
- In al doilea pas verificam valoarea lui k:
  - Daca k <= 1:
    - Alegem un numar a din intervalul [2: n-2].
    - Calculam T = a^m mod n.
    - Daca T=1 sau T=-1, inseamna ca n este prim, altfel n este numar compus.
  - Daca k > 1
    - Calculam T = T^2 mod n.
    - Daca T=1 => n este compus.
    - Daca T=-1 => n este prim.
    - Altfel, numarul este compus.

O observatie pentru acest algoritm este ca functioneaza bine pentru numere foarte mari, dar, pot exista anumite exemple in care pentru un numar compus, testul pentru o anumita valoare a din intervalul ales ne poate intoarce ca numarul este prim, rezultat care ar fi eronat.

Algoritmul lui Fermat

Algoritmul lui Fermat este de asemenea un algoritm de testare a unui numar prim, acesta ne ofera un raspuns sigur, insa cu pretul duratei de executie mai mare, in cazul in care numarul nostru este prim, in cazul in care acesta nu este algoritmul ne va oferi un raspuns mai rapid.

Pasii algoritmului

- Fie x numarul asupra caruia dorim sa efectuam testul.
- Pentru fiecare a din intervalul $1 < a < x$ vom calcula a^{x-1} mod x.
- Daca rezultatul este 1 inseamna ca numarul poate fi prim si continuam algoritmul pentru urmatorul a.
- Daca rezultatul este diferit de 1 inseamna ca numarul nu este prim si ne oprim.

Ca si observatie pentru acest algoritm este ca functioneaza bine pentru numere mici(deoarece sunt putine valori a pentru care trebuie sa realizam testul), dar este foarte costisitor ca si timp pentru numere prime foarte mari.

De fapt, in algoritmul pe care il vom implementa in C nu vom calcula pentru fiecare 'a' din intervalul (1, x), tocmai pentru a mai reduce putin din timpul de executie, ci vom genera aleator cateva numere din acest interval pentru care vom realiza testul; cu cat generam mai multe numere cu atat vom avea o precizie mai mare a rezultatului.

1.3 Criteriile de evaluare:

Pentru a evalua acesti doi algoritmi voi incerca sa creez un set de teste cat mai variate, spre exemplu: teste cu putine numere dar cu valori mari; multe numere cu valori mici; multe numere cu valori mari(acestea ar trebui sa fie cele mai costisitoare din punct de vedere al timpului) si teste ce contin doar numere prime generate cu ajutorul algoritmului lui Eratostene.

De asemenea, voi incerca sa caut anumite cazuri particulare de numere pentru care algoritmii ne pot da un rezultat gresit, cum ar fi: Carmichael numbers(pseudoprime) pentru algoritmul lui Fermat. Aceste numere sunt de fapt niste numere compuse pentru care algoritmul lui Fermat ne va spune ca sunt prime pentru orice valoare 'a' din algoritmul prezentat anterior.

Pentru a evalua performantele celor doi algoritmi voi incerca sa gasesc o modalitate de a masura atat timpii in care cei doi algoritmi ne ofera raspunsul, facand astfel o comparatie intre cei doi, cat si corectitudinea rezultatului(intocmind niste fisiere de referinta pentru testele definite). Avand in calcul atat timpul de executie cat si valoare de adevar a raspunsului ne putem forma o parere despre avantajele si dezavantajele celor doi algoritmi.

2 Prezentarea solutiilor:

2.1 Descrierea modelului de lucru si analiza complexitatii:

Algoritmul Miller-Rabin

Intoarce 0 daca n este compus sau 1 daca n este probabil prim. k reprezinta un parametru de intrare ce determina nivelul acuratetii. Cu cat k este mai mare cu atat vom obtine un rezultat cu o exactitate mai mare.

int isPrime(int n, int k)

1. Verifica cazurile de baza pentru n < 3.
2. Daca n este par, intoarcem 0.
3. Gasim un numar impar d astfel incat n-1 poate fi scris ca si d * 2^r. Observam ca daca n este impar, inseamna ca n-1 este par si r trebuie sa fie mai mare ca 0.
4. Executa de k ori: if (MillerRabinTest(n, d) == 0) return 0
5. return 1

Functie apelata pentru fiecare din cele k iteratii.Aceasta intoarce 0 daca n este compus sau 1 daca n este probabil prim.

int MillerRabinTest(int n, int d)

1. Alege un numar random 'a' in intervalul [2, n-2].
2. Calculeaza x = pow(a, d) mod n.
3. Daca x == 1 sau x == n-1, intoarce 1.
4. Repeta cat timp d != n-1.
   (a) x = x^2 mod n.
   (b) Daca x == 1 intoarce 0.
   (c) Daca x == n-1 intoarce 1.
   (d) d = d * 2.

• Complexitatea algoritmului:
Operatiile de la subpunctele (1) si (2) din functia isPrime, se realizeaza in O(1). Cat despre operatia de la subpunctul (3), acesta se realizeaza in O(log(n-1)) = O(log(n)). Operatia de la subpunctul (4) reprezinta un loop ce aplica testul Miller-Rabin de k ori, deci O(k) pe care o vom inmulti cu complexitatea testului imediat ce o calculam.
Pentru a calcula complexitatea testului(celei de-a doua functii din pseudocod) trebuie sa tinem cont de operatia de modulare exponentiala de la subpunctul

(2) asociat acestei functii care se realizeaza in O(log(n)) si de loop-ul de la subpunctul (4) al acesteia, care ne da o complexitate de O(log(n)). Deci complexitatea finala a celei de-a doua functii este O(1 + 2 *(log(n)), unde 1 provine de la instructiunea (1) de generare a unui numar random. De asemenea aceasta complexitate poate fi aproximata cu O(log(n)). In acest moment avem si complexitatea functiei MillerRabinTest si putem calcula complexitatea loop-ului din prima functie:O(k*log(n)).
Pentru a obtine complexitatea finala a algoritmului, adunam toate complexitatile calculate pentru prima functie:O(1 + log(n) + k*log(n)), ce poate fi aproximata cu O(k*log(n)).

Algoritmul Fermat

Daca n este prim, intoarce 1, altfel, daca n este compus, intoarce 0 cu o probabilitate destul de mare. La fel ca si la Miller-Rabin, k mai mare inseamna o probabilitate mai mare de a obtine un rezultat corect.
int FermatTest(int n, int k)

1. Repeta urmatorii pasi de k ori:
   (a) Alege un numar 'a' random in intervalul [2, n-2].
   (b) Daca cmmdc(a, n) != 1, intoarce 0.
   (c) Daca a^{n-1} mod n != 1, intoarce 0.
2. Intoarce 1, deoarece n este probabil prim.

- Complexitatea algoritmului:O(k*log(n))

Aceasta depinde direct de numarul de iteratii pe care dorim sa le faca algoritmul, si anume: k.
Trebuie sa tinem cont de faptul ca pentru fiecare iteratie se realizeaza atat o operatie de modulare exponentiala(operatia de la subpunctul (c)), ce are o complexitate de O(log(n)), cat si o operatie de aflare a celui mai mare divizor comun dintre a si n(operatia de la subpunctul (b)), ce are o complexitate de O(log(max(a,n))), cum 'a' se afla in intervalul [2, n-2], inseamna ca max(a, n) = n, deci operatia de cmmdc are o complexitate de O(log(n)). Pentru a obtine complexitatea pentru fiecare iteratie adunam complexitatile cmmdc-ului si modularii exponentiale, mai exact O(log(n)) + O(log(n)) = O(2log(n)) care poate fi aproximata cu O(log(n)).De asemenea, am presupus ca operatia de la subpunctul (a) se realizeaza in O(1), deci ea nu va afecta complexitatea finala.
Deci avem de facut de k ori niste operatii de complexitate logaritmica, ceea ce inseamna ca avem o complexitate totala de O(k*log(n)).

Observatie: In situatia in care dorim o precizie foarte buna a algoritmilor, ceea ce implica in mod direct o valoare mai mare a lui k, eficienta in timp a acestora va scadea, deoarece complexitatea lor depinde direct de numarul de iteratii.

2.2 Prezentarea avantajelor si dezavantajelor ambilor algoritmi:

Avantaje Miller-Rabin

1. Principalul sau avantaj este ca functioneaza mai bine pe numere mari decat Fermat si de cele mai multe ori vom avea nevoie sa aplicam astfel de teste pe numere destul de mari(algoritmii de encriptare se folosesc de numere prime cu valori foarte mari).
2. Este mai rapid decat Fermat in majoritatea cazurilor.
3. In comparatie cu alte teste precum Euler sau Solovay-Strassen este mult mai puternic si probabilitatea ca acesta sa ne ofere un rezultat eronat este mai mica.
4. Un alt avantaj important al acestuia reprezinta faptul ca de cele mai multe ori ne ofera un rezultat corect pentru cazurile particulare ale algoritmului lui Fermat(numerele Carmichael).
5. Vom observa in sectiunea de evaluare a rezultatelor algoritmilor ca are nevoie de un numar mult mai mic de iteratii fata de Fermat pentru a ne oferi rezultate corecte.

Dezavantaje Miller-Rabin

1. Pot exista unele numere compuse pentru care testul ne ofera un rezultat eronat(depinde totusi de numarul de iteratii folosit).

Avantaje Fermat

1. La prima vedere poate fi un algoritm mai usor de inteles comparativ cu Miller-Rabin.
2. Este mai usor de implementat.
3. Timpi de executie decenti pentru numere mici.

Dezavantaje Fermat

1. Principalul dezavantaj este reprezentat de numerele Carmichael, pentru care in unele cazuri algoritmul intoarce un rezultat eronat(in functie de numarul de iteratii).
2. Este mai incet decat Miller-Rabin. In sectiunea de comparare a performantelor celor doi algoritmi vom putea vedea timpii pentru ambii si ne vom convinge de acest lucru.

3 Evaluarea algoritmilor

3.1 Descrierea testelor construite

In realizarea testelor am incercat sa acopar cat mai multe situatii pentru a putea compara corect cei doi algoritmi, pentru a putea vedea care dintre ei este mai bun pentru numere mici, numere mari, aflate in diferite intervale, daca unul dintre ei acopera cazurile particulare ale celuilalt sau daca sunt exemple in care ambele esueaza. Am incercat sa le compun astfel incat sa fie cat mai variate, complexe si sa acopere diferite dimensiuni, atat in ceea ce priveste numarul de elemente pe care le contin, cat si valorile acestora. Doua dintre teste au fost create cu ajutorul unui generator, mai exact, un program in care am implementat algoritmul lui Eratostene de generare a tuturor numerelor prime mai mici decat un numar dat. Restul de 16 teste sunt facute manual, cu ajutorul mai multor documentatii din interiorul carora am extras numere Carmichael cat si site-uri de unde am luat valori de numere prime foarte mari, pentru a putea testa comportamentul algoritmilor pe astfel de valori.

In continuare voi relua descrierea detaliata a testelor pe care am facut-o si in README-ul din arhiva cu implementarea algoritmilor.

- Testele 1, 2 si 3 reprezinta niste teste simple, cu numere mici, cat mai variate(atat numere prime cat si compuse).
- testele 4, 5, 6, 7 si 8 reprezinta teste de dimensiuni mici dar care contin cazuri particulare pentru algoritmul lui Fermat, si anume: numere Carmichael(numere care sunt de fapt compuse, dar pentru care algoritmul ne poate intoarce ca sunt prime) cu valori mici, amestecate cu alte numere compuse sau prime.
- Testele 9 si 10 sunt teste mai complexe, contin cate 8 numere cu valori de peste 10^5 care reprezinta de asemenea doar numere de tip Carmichael.
- Testele 11, 12 si 13 sunt niste teste care contin cate 10 valori mari(intre 10^6 si 10^7) printre are se afla atat numere prime cat si numere compuse.
- Testul 14 este un test care contine 20 de numere cu valori cuprinse intre 10^7 si 10^8 printre care se afla atat numere prime cat si numere de tip Carmichael.
- De asemenea, testul 15 contine 20 de numere cu valori de peste 10^8 de aceasta data, combinate cu numere de tip Carmichael;
- Testul 16 contine 10 valori prime foarte mari (de peste 10^9), l-am compus tocmai pentru a vedea cum se comporta algoritmii pe numere foarte mari(aproape de valoarea maxima cu semn reprezentabila pe 32 de biti).
- Testele 17 si 18 sunt niste teste generate cu ajutorul algoritmului lui Eratostene, pentru a vedea cum se comporta algoritmii cand primesc ca input un numar mare de valori prime si daca intorc rezultatele dorite.

3.2 Specificatiile sistemului de testare

Testele au fost rulate pe un laptop Lenovo Legion Y540, cu un procesor i5 9300HF si 8GB RAM DDR4, pe o masina virtuala Ubuntu si folosind Visual Studio Code ca IDE.

3.3 Ilustrarea rezultatelor

Pentru a putea evalua rezultatele algoritmilor am masurat atat timpii de rulare ai acestora pentru fiecare test in parte, cat si corectitudinea rezultatelor oferite, folosindu-ma de fisierele de referinta definite la etapa anterioara. De asemenea, testele au fost rulate pentru diferite valori ale numarului de iteratii.

Pentru a masura timpul de executie am folosit biblioteca time.c, iar pentru a verifica corectitudinea rezultatelor algoritmilor, am definit o functie care citeste valorile din fisierul de referinta pentru un anumit test si le compara cu cele obtinute in urma rularii algoritmilor. Am calculat un fel de rata de succes, care reprezinta de fapt raportul dintre numarul elementelor corecte(prime) din fisierul de output si totalul elementelor din fisierul ref. In cazul in care in fisierul de output aveam mai putine numere decat trebuie faceam raportul dintre cele obtinute si cele corecte(inmultit cu 100 pentru a obtine un procent), iar in cazul in care obtineam si alte numere in output(numere compuse pe care algoritmul ni le-a calculat ca fiind prime) am scazut din numarul elementelor corecte din output, numarul elementelor gresite din acesta si am impartit rezultatul la numarul elementelor pe care ar fi trebuit sa il obtinem, practic un numar gresit scris in fisier anula un numar corect scris in acelasi fisier. De asemenea, am definit atat un timp mediu, cat si o probabilitate medie, ambele raportate la numarul de teste.

Pentru a putea avea toate aceste valori, atat timpi si rate de succes, cat si numarul de iteratii, concentrate intr-un singur loc, am definit o functie care creeaza pentru fiecare rulare un fisier de tip "performance" in care scriam toate aceste rezultate.

Ca si observatie, numarul de iteratii il citeam de la tastatura la fiecare rulare a unui test, pentru a fi mai usor sa creez fisiere de performanta pentru cat mai multe valori ale acestuia.

```
performances > miller-rabin_performance
Numarul de iteratii: 25000
Miller-Rabin time and correctness for test number 1 : 0.008874, 100.00
Miller-Rabin time and correctness for test number 2 : 0.028792, 100.00
Miller-Rabin time and correctness for test number 3 : 0.026653, 100.00
Miller-Rabin time and correctness for test number 4 : 0.005866, 100.00
Miller-Rabin time and correctness for test number 5 : 0.009534, 100.00
Miller-Rabin time and correctness for test number 6 : 0.004804, 100.00
Miller-Rabin time and correctness for test number 7 : 0.005569, 100.00
Miller-Rabin time and correctness for test number 8 : 0.000003, 100.00
Miller-Rabin time and correctness for test number 9 : 0.000004, 100.00
Miller-Rabin time and correctness for test number 10 : 0.000004, 100.00
Miller-Rabin time and correctness for test number 11 : 0.053051, 100.00
Miller-Rabin time and correctness for test number 12 : 0.021372, 100.00
Miller-Rabin time and correctness for test number 13 : 0.014167, 100.00
Miller-Rabin time and correctness for test number 14 : 0.072164, 100.00
Miller-Rabin time and correctness for test number 15 : 0.099725, 100.00
Miller-Rabin time and correctness for test number 16 : 0.109044, 100.00
Miller-Rabin time and correctness for test number 17 : 1.254281, 100.00
Miller-Rabin time and correctness for test number 18 : 1.512135, 100.00
Timpul mediu al algoritmului Miller-Rabin pentru un numar de 25000 iteratii este: 0.179225
Probabilitatea ca algoritmul Miller-Rabin sa ne dea un rezultat corect pentru 25000 iteratii este de: 100.00
```

Fig. 1. Exemplu de fisier.

Cu ajutorul acestor fisiere am creat niste tabele in care sunt reprezentati timpii celor doi algoritmi, pentru fiecare dintre teste, in functie de numarul de iteratii.

k = 10 iteratii		
Test	Miller-Rabin(time-ms)	Fermat(time-ms)
test1.in	0.007	0.008
test2.in	0.015	0.021
test3.in	0.013	0.023
test4.in	0.005	0.007
test5.in	0.007	0.011
test6.in	0.004	0.009
test7.in	0.005	0.008
test8.in	0.003	0.008
test9.in	0.004	0.018
test10.in	0.003	0.027
test11.in	0.024	0.035
test12.in	0.013	0.016
test13.in	0.010	0.013
test14.in	0.036	0.049
test15.in	0.046	0.066
test16.in	0.045	0.069
test17.in	0.531	0.787
test18.in	0.641	0.982

k = 100 iteratii		
Test	Miller-Rabin(time-ms)	Fermat(time-ms)
test1.in	0.038	0.056
test2.in	0.119	0.176
test3.in	0.114	0.165
test4.in	0.026	0.041
test5.in	0.048	0.062
test6.in	0.021	0.044
test7.in	0.024	0.052
test8.in	0.003	0.005
test9.in	0.003	0.040
test10.in	0.004	0.051
test11.in	0.214	0.327
test12.in	0.090	0.141
test13.in	0.061	0.099
test14.in	0.294	0.446
test15.in	0.409	0.615
test16.in	0.441	0.668
test17.in	5.048	7.586
test18.in	6.142	9.239

k = 1000 iteratii		
Test	Miller-Rabin(time-ms)	Fermat(time-ms)
test1.in	0.344	0.520
test2.in	1.189	1.733
test3.in	1.063	1.642
test4.in	0.233	0.360
test5.in	0.379	0.587
test6.in	0.191	0.296
test7.in	0.224	0.358
test8.in	0.003	0.005
test9.in	0.004	0.031
test10.in	0.004	0.047
test11.in	2.104	3.197
test12.in	0.852	1.315
test13.in	0.572	0.878
test14.in	2.879	4.380
test15.in	3.977	6.097
test16.in	4.349	6.645
test17.in	49.334	75.943
test18.in	60.356	92.010

In urma acestor tabele putem observa mai multe aspecte, unul dintre ele este faptul ca timpul de executie creste pentru ambii algoritmi atunci cand numarul de iteratii creste si el, si un alt aspect ar fi diferenta de timpi dintre cei doi algoritmi, Miller-Rabin fiind mult mai rapid in toate cazurile. Pentru a putea observa mai bine aceste lucruri voi face atat un tabel cu timpii medii(media timpilor tuturor testelor) ai ambilor algoritmi in functie de numarul de iteratii, cat si o reprezentare grafica in Octave a acestor date.

Tabel timpi medii in functie de numarul de iteratii		
Nr. iteratii	Miller-Rabin(time-ms)	Fermat(time-ms)
1	0.012	0.015
5	0.042	0.061
10	0.078	0.117
25	0.183	0.283
50	0.366	0.554
100	0.723	1.104
250	1.788	2.733
500	3.572	5.467
1000	7.147	10.971
2500	17.858	27.352
5000	35.755	54.554
10000	71.480	109.115
25000	179.225	273.145

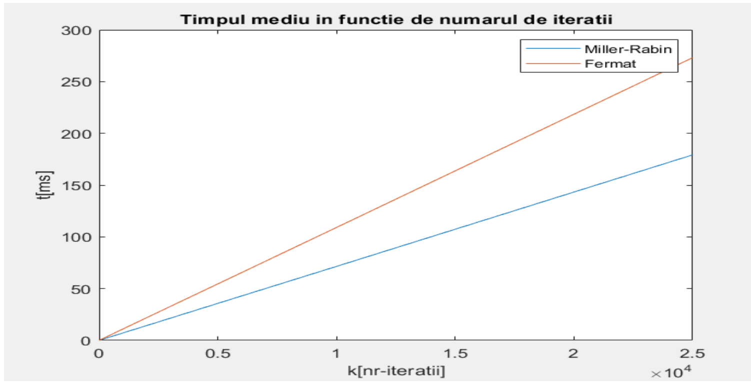

Fig. 2. Grafic timp mediu de executie in functie de numarul de iteratii.

• Observam din grafic faptul ca timpul mediu de executie al algoritmului Fermat asupra tuturor testelor definite creste mult mai mult decat cel al algoritmului Miller-Rabin atunci cand numarul de iteratii creste si el.

• Urmatorul grafic este definit pentru a ne putea face o idee despre cum creste precizia(rata de succes) a celor doi algoritmi in functie de numarul de iteratii.

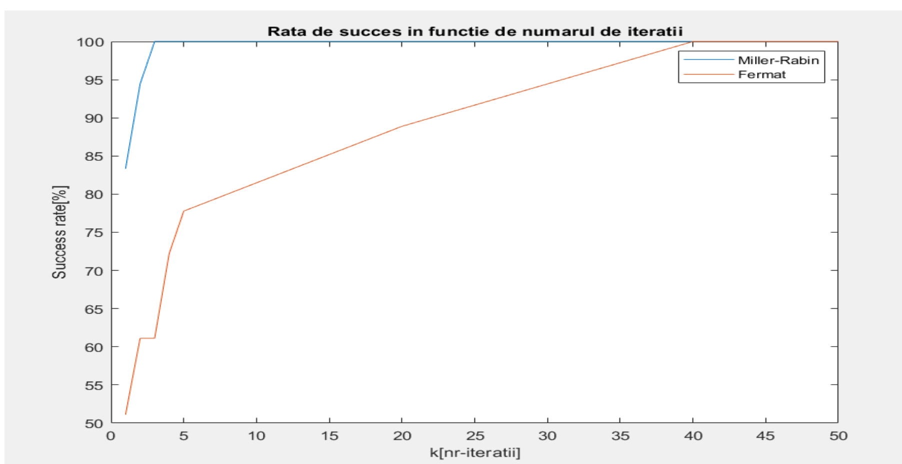

Fig. 3. Grafic rata de succes in functie de numarul de iteratii.

• Putem observa faptul ca algoritmul Fermat are nevoie de un numar mult mai mare de iteratii pentru a ajunge la o precizie de 100%, mai exact 40 de iteratii, comparativ cu algoritmul Miller-Rabin care obtine aceasta precizie dupa doar

3 repetari. Dupa ce am studiat fisierele performance pentru fiecare numar de iteratii, am observat ca algoritmul Fermat intampina probleme chiar la testele ce contin numere Carmichael, necesitand un numar mare de iteratii pentru a ne oferi un rezultat corect in ceea ce priveste valorile continute de aceste teste. In ceea ce priveste algoritmul Miller-Rabin acesta ne ofera o precizie decenta inca de la a 2-a iteratie(94.11%), fapt ce ne demonstreaza inca o data ca acesta ar fi o alegere mai buna in cazul in care dorim sa implementam un algoritm de testare a numerelor prime pe o scara mai larga.

Observatie: Timpii descrisi mai devreme sunt unii orientativi, pot exista unele erori sau situatii in care algoritmii ne intorc rezultatele mai repede sau mai incet(depinde ce numere 'a' sunt generate random pentru fiecare dintre acestia). Putem obtine rezultate diferite la diferite rulari ale algoritmilor pentru acelasi numar de iteratii, dar diferenta dintre aceste rezultate si cele expuse mai sus este una neglijabila(0.001ms in unele cazuri). De asemenea, o alta observatie ar fi ca timpii de rulare nu sunt influentati de citirea, respectiv, scrierea in fisiere, ei reprezinta strict performantele algoritmilor de testare.

3.4 Evaluarea rezultatelor obtinute

• O concluzie pe care o putem formula asupra rezultatelor obtinute consta in faptul ca dupa fiecare evaluare, atat a timpilor pe fiecare test, timpilor medii, in mai multe situatii(pentru diferite valori ale numarului de iteratii), cat si a preciziei, am putut observa faptul ca algoritmul Miller-Rabin este mai eficient din toate punctele de vedere comparativ cu Fermat. Acesta ne ofera: timpi mai buni pentru fiecare test, timpi medii mai buni raportat la toate testele, o precizie mai buna pentru un numar mic de iteratii, si in cazul in care am avea nevoie de un astfel de algoritm ar fi o optiune foarte buna implementare sa, care desi poate parea putin mai greu de inteles decat Fermat, ne ofera mult mai multe avantaje comparativ cu acesta.

4 Concluzie

• Poate diferi de la situatie la situatie ce algoritm se pliaza mai bine cerintelor noastre, daca avem nevoie de un algoritm ce se poate implementa destul de repede, usor de inteles, si pe care il vom folosi pe numere mici si nu avem nevoie de o performanta foarte mare, algoritmul lui Fermat poate fi o optiune buna. In cazul in care avem nevoie de un algoritm care sa se comporte bine in toate situatiile, sa fie eficient pentru toate tipurile de numere(atat in ceea ce priveste timpii de executie cat si precizia), situatie mult mai probabila(deoarece majoritatea algoritmilor de encriptare, adica principalul domeniu de folosire a numerelor prime, se folosesc de numere cu valori foarte mari), algoritmul pe care trebuie sa il implementam este cu siguranta Miller-Rabin, acesta fiind mult mai robust decat Fermat.

References

1. https://crypto.stanford.edu/pbc/notes/numbertheory/millerrabin.html
2. https://www.baeldung.com/cs/fermat-primality-test
   De asemenea niste videoclipuri pe youtube pentru a vedea cum se aplica algoritmii matematic asupra unor exemple concrete:
3. https://www.youtube.com/watch?v=RcjxwCHRYfE
4. https://www.youtube.com/watch?v=qdylJqXCDGs
5. https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/
6. https://www.geeksforgeeks.org/primality-test-set-2-fermet-method/
7. https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/?ref=lbp
8. https://www.geeksforgeeks.org/sieve-of-eratosthenes/
9. https://stackoverflow.com/questions/26996736/fermat-primality-test-failure-in-c