Tema numărul 1

- la disciplina Analiza Algoritmilor -

Marian Vlăduţ Jorăscu

Facultatea de Automatică si Calculatoare
Universitatea Politehnică din Bucureşti
Grupa 325 CD anul II
jorascuvlad@gmail.com

13 decembrie 2017

Abstract În cadrul acestei teme am ales 5 algoritmi care rezolvă problema alesă la etapa intermediară .Am implementat mai întai algoritmii şi am prezentat paşii efectivi pe care ii urmează , am calculat complexitaţile corespunzătoare şi în final am realizat un studiu comparativ pe baza rezultatelor obţinute de la testele pe care le-am efectuat pentru diferite date de intrare.

Cuprins

1 Prezentarea temei

1.1 Prezentarea problemei

Problema pe care am ales să o rezolv este: Determinarea numerelor prime mai mici ca N.Algoritmii cu ajutorul cărora am rezolvat această problemă sunt: algoritmul Miller-Rabin, algoritmul lui Fermat, Ciurul lui Eratostene, Ciurul lui Eratostene Optimizat şi Algoritmul Iterativ. Pentru primii doi am încercat să reduc numărul cazurilor când aceştia întorc TRUE pentru un număr compus, analizând ce efecte a avut această optimizare asupra timpului de rulare. Deoarece ştim că singurul număr prim par este 2, voi începe căutarea de la 3 , iterând doar prin numerele impare.Criteriile de evaluare pentru soluţiile luate în considerare sunt: timpul de rulare, spaţiul de memorie ocupat şi numarul de numere prime determinat. Ultimul criteriu este relevant doar pentru algoritmii probabilistici.Pentru diferite valori ale lui N voi rula fiecare program de 10 ori şi voi trece într-un tabel , cel mai bun rezultat (raportat la timpul de execuţie)pentru fiecare dintre algoritmii implementaţi.

1.2 Aplicatii preactice

Numerele prime au o mare importanţa în domenii precum:

- Criptografia

Mulţi algoritmi de criptare au în componenţa lor numerele prime, dintre acestea cel mai cunoscut este RSA. Numerele prime sunt esenţiale deoarece aceşti alogoritmi de criptare se bazează pe faptul că este uşor să înmulţim două numere prime mari şi să aflăm rezultatul , pe când este foarte costisitor sa facem operaţia inversă.Coplexitatea pentru înmulţirea a doua numere prime mari (reprezentate pe n biţi) este $O(n \log n \log n)$, iar pentru descompunerea numărului rezultat folosind un algoritm puternic $O(n^s)$ (cu s constant, mare). Dificultatea factorizării pentru RSA în funcţie de dimensiunea cheii se poate observa din figura 1.1 . Dacă avem un numar despre care ştim ca este un produs a doua numere prime ,găsirea lor nu poate fi realizată într-un timpul util cu resursele actuale de calcul pe care le avem la dispoziţie ca şi utilizatori. Astfel, determinarea primalităţii unui numar într-un mod cât mai eficient şi mai rapid stă la baza securităţii informaţiilor şi a unor algoritmi de criptare cât mai puternici

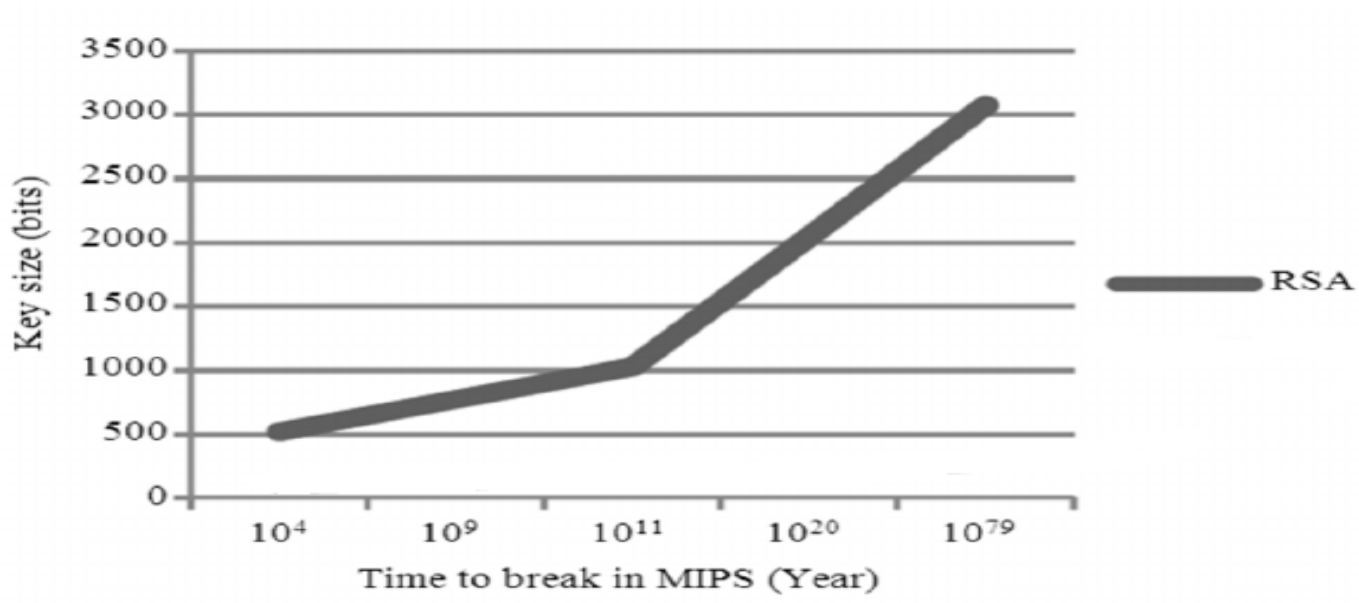

Figura 1.1:

- Generarea numerelor pseudoaleatoare
- Tabele de dispersie

2 Prezentarea şi implementarea soluţiilor

2.1 Algoritmul de ridicare repetată la pătrat

Algoritmul de ridicare repetată la pătrat şi reducerii modulo m nu este un algoritm de testare a primalităţii , dar este folosit de cei doi algoritmi probabilistici (Fermat şi Miller-Rabin) , de aceea este necesară şi o scurtă prezentare a sa. Această metodă este eficientă pentru valori mari ale lui m şi n , fiind des utilizată în multe protocoale criptografice care implică exponenţieri modulare.

Input : numerele naturale b, n, m
Output: $b^n \mod m$
$rezultat \leftarrow 1$
$A \leftarrow b$
Cat Timp $(n > 0)$ executa operatiile:
1) Daca n este impar atunci $rezultat \leftarrow A \cdot rezultat \mod m$
2)$A \leftarrow A^2 \mod m$
3)$n \leftarrow n/2$
returneaza rezultat

Algorithm 1: Algoritmul de ridicare repetată la pătrat

2.2 Algoritmul lui Fermat

Înainte de prezentarea efectivă a algoritmului este necesară introducerea teoremei cu acelaşi nume care stă la baza acestuia:

Mica Teoremă a lui Fermat : Dacă p este un număr prim si b este un număr întreg care nu este multiplu a lui p , atunci:

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

Fermat este un algoritm probabilistic de determinare a primalităţii unui număr şi se desfăşoară astfel: Pentru un număr natural n , a cărui primalitate o cercetăm , alegem $b > 1$ şi calculăm $b^{n-1} \mod n$.Dacă rezultatul nu este 1 , atunci numărul este compus şi b este martor Fermat al faptului că n este compus. Daca este egal cu 1 , atunci n este prim sau pseudoprim cu baza b.

Fie $b \geq 2$ un număr natural.Numărul natural n se numeşte pseudoprim cu baza b dacă verifică relatia : $b^n \equiv b \ (mod\ n)$, relaţie echivalentă cu $b^{n-1} \equiv 1 \ (mod\ n)$. Un număr natural compus n care verifică această relaţie pentru orice număr b relativ prim cu n , se numeşte număr Carmichael.

Este clar că testul va spune numai "Probabil-Prim" pentru numere prime. Puterea unui asemenea test rezidă însă în probabilitatea cu care testul se termină cu verdictul "Probabil-Prim" pentru numere compuse.

Sa presupunem că la fiecare testare a unui număr Carmichael probabilitataea ca testul să întoarcă verdictul COMPUS este 1/2. atunci repetând testul de 100 de ori, probabilitatea sa obţiem COMPUS este foarte apropiată de 1, ceea ce înseamna ca daca am obţinut de

100 de ori "Probabil-Prim", atunci probabilitatea ca numaul sa fie totuşi Carmichael este extrem de mică.Însă nu putem fi siguri ca am eliminat in totalitate această posibilitate. Vom vedea ce efecte are asupra timpului de rulare aceasta marire a numărului de iteraţii. Numarul de iteraţii poate fi considerat un "parametru de securitate". Cu cat t se maresţe cu atât probabilitatea să întoarcă "TRUE" în cazul unui număr compus scade.Pentru implementarea propriu-zisă am folosit o soluţie existentă [1].

Input : Un număr $n > 2$ impar, un parametru de securitate t
Output: un răspuns referitor la primalitatea lui n
Pentru i de la 1 la t executa urmatoarele operaţii:
1)Alege un aleator întreg b cu $2 \leq b \leq n-2$
2)Calculează $r = b^{n-1}(\mod n)$ folosind algoritmul de la 2.1
3)Daca $r \neq 1$, atunci returneaza NUMĂR COMPUS şi se opreşte
Returnează NUMĂR PRIM

Algorithm 2: Algoritmul lui Fermat

2.3 Algoritmul Miller-Rabin

Punctul slab al metodelor de testare a primalităţii bazate pe mica teoremă a lui Fermat, îl reprezintă numerele Carmichael (numerele pseudoprime in raport cu o baza b), deci numerele care se comportă ca şi numerele prime , conform teoremei.Algoritmul Miller-Rabin se bazeaza pe o proprietate mai puternică numită pseudoprimalitate în sens tare.

Fie $n > 2$ număr natural impar cu $n - 1 = 2^s t$ unde s,t sunt numere naturale cu t impar.Spunem ca n trece testul Miller pentru baza b , un număr natural prim cu n , dacă este verificată una din următoarele condiţii:

$$b^t \equiv 1(\mod n) \tag{2}$$

sau

$$\exists 0 \leq j \leq s-1, b^{2^j t} \equiv -1(\mod n) \tag{3}$$

Spunem că un număr compus n este tare pseudoprim cu baza b unde $(b, n) = 1$ daca el trece testul Miller pentru baza b. Eficientă acestui algoritm in comparaţie cu Fermat reiese chiar din teorema enunţătă de Rabin : Daca n este un numar impar compus , acesta trece testul Miller-Rabin pentru cel mult $\frac{n-1}{4}$ baze b cu $1 \leq b \leq n-1$.Cu alte cuvinte probabilitatea ca acesta să treacă testul este mai mică decât $\frac{1}{4^k}$, unde k reprezintă numărul de iteraţii(numarul de baze pentru care testăm proprietatea). Deci, numarul cazurilor în care Algoritul Miller-Rabin întoarce "TRUE" pentru un număr compus este de 4 ori mai mic comparativ cu Algoritul lui Fermat.

Soluţia implementată alege baza în mod aleator, de aceea este nevoie de mai multe iteraţii (pentru eventualele numere tare pseudoprime) , dar pentru datele de testare pe care le-am utilizat ar fi fost nevoie de o singura iteraţie şi să aleg ca bază 2 ,3,5 sau 7 deoarece primul număr tare pseudoprim în raport cu una din aceste baze este 3215031751 (iar numarul maxim pe care îl testez în program este 10^8). Pentru implementarea propriu-zisă am folosit o soluţie existentă [2].Algoritmul se comportă astfel:

Input : Un număr $n > 2$ impar, un parametru de securitate r
Output: un răspuns referitor la primalitatea lui n
1. Determină s şi t astfel încât $n - 1 = 2^s t$ cu t impar
2. Pentru i de la 1 la r :
    2.1) Alege aleator un întreg $2 \leq b \leq n - 2$
    2.2) Calculează $y = b^t(\mod n)$ folosind algoritmul de la subsectiunea 2.1
    2.3) Dacă $y = 1$ sau $y = n - 1$ atunci
        2.3.1)Returnează NUMĂR PRIM
    2.4)Cât timp $t \neq n - 1$
        2.4.1) $y \leftarrow y^2 \mod n$
        2.4.2) $t \leftarrow t^2$
        2.4.3) Dacă $y = 1$ atunci:
            2.4.3.1) Returnează NUMĂR COMPUS
        2.4.4) Dacă $y = n - 1$ atunci:
            2.4.4.1) Returnează NUMĂR PRIM
    2.5) Returnează NUMĂR COMPUS

Algorithm 3: Algoritmul Miller-Rabin

2.4 Algoritmul Iterativ

Algoritmul iterativ este cel mai simplu , dar şi ineficient mod de a testa proprietatea de prim pentru un număr n . Am ales să implementez şi acest algoritm pentru a evidenţia cât de importantă este o optimizare pentru un algoritm cu complexitat epolinomială atunci când avem date de intrare mari ,şi cât de vizibilă este diferenţa dintre n şi log(n). Pentru orice număr algoritmul iterativ testeaza toate numerele de la 2 la jumatatea acestuia. Dacă inputul ce trebuie testat nu se divide pentru niciunul din numerele de mai sus , atunci el este prim.

Input : Un număr $n > 2$ impar

Output: un răspuns referitor la primalitatea lui n

```
pentru i de la 1 la [n/2]
        daca n % i == 0
                returneaza numar compus

returneaza numar prim
```

Algorithm 4: Algoritmul Iterativ

2.5 Ciurul lui Eratostene

Acesta este cel mai vechi test de primalitate cunoscut, apărut în jurul anului 240 î.e.n El funcţionează corect pentru orice numere prime. Considerat un număr n, pentru a testa dacă este prim, întocmim o listă cu toate numerele naturale pornind de la 2 până la n . Din ea se înlătură toate numerele care sunt multiplii de numere prime $\leq n$. Cele care rămân în listă sunt toate numere prime.Algoritmul original parcurge toate numerele de la 2 la n , dar pantru ca singurul numar prim par este 2 , putem itera doar prin numerele impare începând cu 3 .Pentru testarea propriu-zisă am folosit o soluţie existentă [3]:

```
public int getTheNumber(int n) {
  int i, j, nr = 1;
  for (i = 3; i <= n; i+=2) {
    if (p[i] == 0) {
      nr++;
      for (j = i + i; j <= n; j += i) {
        p[j] = 1;
      }
    }
  }
  return nr;
}
```

Algorithm 5: Ciurul lui Eratostene

2.6 Ciurul lui Eratostene Optimizat

Prima optimizare ce se poate face pe Algoritmul original al Ciurului lui Eratostene este să nu mai luăm în calcul numerele pare deoarece ştim că singurul număr prim par este 2.Urmatoarea optimizare va fi marcarea multiplilor numărului prim i de la i*i nu de la 2 * i deoarece orice număr prim compus multiplu de i mai mic decat i*i are un factor prim mai mic decât i , şi acel factor a fost marcat la unul din paşii anteriori.De asemenea, nu

este necesar sa parcurgem numerele pana la n pentru a marca multiplii , ci până la $\sqrt{n}$(astfel la ultima etapă cea în care număram , nu vom mai lucra cu i ci cu 2*i+1). Ultima îmbunătăţire care este adusă este aceea de a folosi cât mai puţină memorie.Cum pentru fiecare număr este necesară doar o informaţie booleană,pe aceasta o putem ţine într-un bit , nu este necesar un char întreg.De asemenea,deşi îngreunează puţin citirea codului, sunt folosite foarte multe operaţi pe biţi, acolo unde este posibil întrucât sunt mai rapide decât operaţiile normale. Pentru testarea propriu-zisă am folosit o soluţie existentă [4]:

```
public int getTheNumber(int n) {
  int i, j, nr = 1;
  for (i = 1; ((i * i) << 1) + (i << 1) <= n; i += 1) {
    if ((p[i >> 3] & (1 << (i & 7))) == 0) {
      for (j = ((i * i) << 1) + (i << 1); (j << 1) + 1 <= n; j += (i << 1) + 1) {
        p[j >> 3] |= (1 << (j & 7));
      }
    }
  }
  for (i = 1; 2 * i + 1 <= n; ++i)
       if ((p[i >> 3] & (1 << (i & 7))) == 0)
           nr++;
  return nr;
}
```

Algorithm 6: Ciurul lui Eratostene Optimizat

3 Complexitate şi testare

3.1 Complexitate

• Algoritmul lui Fermat

În cadrul algoritmului propriu-zis complexitatea este dată de functia de ridicare repetată la pătrat care este $\Theta(\log n)$ multiplicată cu numărul de iteraţii pentru alegerea diferitelor baze k , deci $O(k \log n)$, acest $\log n$, provine din faptul că se calculeaza rezultatul lui b^n mod m, cât timp $n > 0$, iar la fiecare pas se execută operaţia $n \leftarrow n/2$. Cum problema noastră este determinarea numerelor prime mai mici ca un n dat , si verific primalitatea doar pentru numerele impare (n/2 numere), complexitatea worst case este $O(\frac{n}{2} k \log n) \rightarrow O(n \log n)$.

• Algoritmul Miller-Rabin

În cazul Miller-Rabin comparativ cu Fermat pe lângă complexitatea dată de funcţia de ridicare la pătrat mai avem şi complexitatea parţii din program în care se testează dacă numărul de testat are proprietatea de tare pseudoprim care este în cazul cel mai defavorabil $O(k \log n)$, unde k este o constantă (numărul de iteraţii), la care se adaugă si complexitatea pentru descompunerea lui n-1 în $n-1 = 2^s t$ (această descompunere se face o singură dată pentru fiecare număr ce trebuie testat) prin urmare :$O(k \log n) + O(k \log n) + O(\log n) = O((2k+1) \log n)$. Algoritmul este aplicat pe toate cele n/2 numere , prin urmare complexitatea finală este $O(\frac{n}{2}(2k+1) \log n) \rightarrow O(n \log n)$ .Deşi complexitatea worst case pentru Miller-Rabin este mai mare decat complexitatea lui Fermat(comparând factorii de amplificare ai complexităţii) , în practica se comportă mult mai bine deoarece acesta nu executa numărul total de iteraţii pentru $\frac{3}{4}$ din numerele Carmichael , ceea ce îmbunatăţeşte timpul de rulare al programului.Doresc să menţionez faptul că aceasta complexitate este doar pe implementarea algoritmului Miller-Rabin aşa cum a fost expusă şi în subcapitolul anterior.

• Algoritmul Iterativ

Complexitatea pentru a testarea primalităţii unui singur numar este un $O(n/4)$, acest $n/4$ provine din faptul că atunci cand testez primalitatea numărului n , pornesc cu un i de la 3 (incrementându-l cu 2 unităţi la fiecare pas), cât timp $i < n/2$. Iar acesta este aplicat pentru toate cele n/2 numere impare. Prin urmare complexitatea worst case este $O(\frac{n}{2}\frac{n}{4}) = O(\frac{n^2}{8}) \rightarrow O(n^2)$.

• Ciurul lui Eratostene

În cazul implementării noastre a ciurului lui Eratostene am evitat testarea tuturor celor n numere, iterând doar prin numerele impare. Pentru a calcula complexitatea mai uşor facem următoarea notaţie : $m = \frac{n}{2}$. Daca ne uităm in etapa de marcare(al doilea for) numerele sunt parcurse pe sărite , iar numărul de operaţii este : $\frac{m}{2} + \frac{m}{3} + \frac{m}{4} + ... + \frac{m}{n} = m(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n})$, iar această sumă din paranteză este $\log n$ ($\int_1^n \frac{1}{x}dx = ln(n)$). Deci bazându-ne pe observaţiile de mai sus complexitatea este $O(m \log n)$. O alta observaţie ce trebuie facută este aceea că noi marcam doar pornind de la valori prime(avem conditiţie de intrare), sărind peste multe valori . Astfel, suma noastră de fracţii devine $\frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{11} + ... + \frac{1}{p}$, unde p este cel mai mare număr prim mai mic decât n-ul nostru, această suma fiind egală cu $\ln n \ln n$ conform [5]. Deci complexitatea finală pentru implementarea noastră este $O(m \log \log n) = O(\frac{n}{2} \log \log n) \rightarrow O(n \log \log n)$

• Ciurul lui Eratostene Optimizat

Conform cu denumirea sa acest algoritm are anumite îmbunătaţiri comparativ cu varianta prezentată anterior, ele fiind atât asupra numarului de operaţii executate cât şi asupra memoriei ocupate, dupa cum au fost prezentat in secţiunea 2.6. Singura optimizare care are o implicaţie directă asupra complexităţii este cea în care nu mai marcăm multiplii numerelor prime pana la n , ci până la $\sqrt{n}$. Astfel Complexitatea pentru Ciurul lui Eratostene Optimizat este $O(\frac{n}{2} \log \log \sqrt{n}) \rightarrow O(n \log \log \sqrt{n})$.

3.2 Testare

Pentru implementarea algoritmilor prezentaţi anterior am utilizat un limbaj de nivel înalt şi anume Java. După cum am precizat şi în etapa intermediară a temei voi evidenţia comportamentul algoritmilor şi rezultatele pe care le-am obţinut în urma testării, cu ajutorul unor tabele , în locul graficelor.Datele de intrare sunt reprezentate de valori ale lui N . Vor exista 4 tabele corespunzătoare pentru fiecare valoare în parte (10^5,10^6, 10^7,10^8) .Am ales această metodă de reprezentare deoarece consider că in contextul problemei alese (Determinarea numerelor prime mai mici ca un N dat) sunt mai usor de observat performanţele în raport cu complexitatea(WorstCase) a fiecăruia , prezentând la final avantajele si dezavantajele lor în contextul problemei actuale dar şi în cazul altor probleme. Următoarele date au fost obţinute pe un procesor Intel Core i7 2.7GHz.Fiecare algoritm a fost testat de 10 ori(mai puţin algoritmul iterativ pentru N = 10^7 şi 10^8) , în mod programatic în fişierele sursă, pentru fiecare N în parte , şi au fost adaugate în tabele cele mai bune rezultate (de timp).

Înainte de vizualizarea rezultatelor testării doresc sa fac precizarea că am realizat calculul timpului de execuţie şi a memoriei utilizate cu ajutorul unor funcţii deja implementate

Algoritm N = 10^5	Complexitate Worst Case	Timp de rulare	Memorie Utilizata de program [KB]	Numere prime determiante
Iterativ	O(n * n)	0.324 s	0 KB	9592
Ciurul lui Eratostene	O(n log log n)	0,001 s	97 KB	9592
Ciurul lui Eratostene optimizat	O(n* log log(sqrt(n)))	0,001 s	0 KB	9592
Fermat, K = 5 iteratii	O(K n logn)	0.04 s	1966 KB	9595
Fermat K = 25 iteratii	O(K n logn)	0.099 s	1966 KB	9592
Fermat K = 50 iteratii	O(K n logn)	0.185 s	1966 KB	9592
Miller Rabin K = 5 iteratii	O(K n logn)	0.041 s	2621 KB	9592
Miller-Rabin K=25 iteratii	O(K n logn)	0.115 s	7209 KB	9592
Miller-Rabin K=50 iteratii	O(K n logn)	0.201 s	21627 KB	9592

Figura 3.1:

în Java. Rezultatele obţinute privind memoria sunt corecte la o analiză calitativă. Singura excepţie în care functia de determinare a memoriei utilizate nu are comportamentul dorit este la testarea algoritmului Miller-Rabin pentru $N = 10^7$ şi $N = 10^8$ când numărul iteraţiilor este 50.Cel mai probabil această neconcordanţa(deşi memoria ar trebui să crească ea scade) pe baza numaului mare de apeluri de funtii şi a modului efectiv în care JVM-ul eliberează memoria .

Algoritm n = 10^6	Complexitate Worst Case	Timp de rulare	Memorie Utilizata de program [KB]	Numere prime determiante
Iterativ	O(n * n)	35.841 s	0 KB	78498
Ciurul lui Eratostene	O(n log log n)	0,004s	976 KB	78498
Ciurul lui Eratostene optimizat	O(n* log log(sqrt(n)))	0,003s	122 KB	78498
Fermat, K = 5 iteratii	O(K n logn)	0.351 s	27047 KB	78513
Fermat K = 25 iteratii	O(K n logn)	0.97 s	27047 KB	78499
Fermat K = 50 iteratii	O(K n logn)	1.821 s	27047 KB	78498
Miller Rabin K = 5 iteratii	O(K n logn)	0.373 s	10111 KB	78498
Miller-Rabin K=25 iteratii	O(n logn)	1.053 s	30358 KB	78498
Miller-Rabin K=50 iteratii	O(K n logn)	1.906 s	30732 KB	78498

Figura 3.2:

Algoritm n = 10^7	Complexitate Worst Case	Timp de rulare	Memorie Utilizata de program [KB]	Numere prime determiante
Iterativ	O(n * n)	O ora si 7 min	0 KB	664579
Ciurul lui Eratostene	O(n log log n)	0.109 s	9765 KB	664579
Ciurul lui Eratostene optimizat	O(n* log log(sqrt(n)))	0.046 s	1220 KB	664579
Fermat, K = 5 iteratii	O(K n logn)	5.136 s	13137 KB	664634
Fermat K = 25 iteratii	O(K n logn)	13.9 s	13751 KB	664595
Fermat K = 50 iteratii	O(K n logn)	25.09 s	14373 KB	664586
Miller Rabin K = 5 iteratii	O(K n logn)	4.987 s	9142 KB	664579
Miller-Rabin K=25 iteratii	O(K n logn)	13.633 s	19259 KB	664579
Miller-Rabin K=50 iteratii	O(K n logn)	19.874 s	14454 KB	664579

Figura 3.3:

Algoritm n = 10^8	Complexitate Worst Case	Timp de rulare	Memorie Utilizata de program [KB]	Numere prime determiante
Iterativ	O(n * n)	Peste 10 ore	0 KB	5761455
Ciurul lui Eratostene	O(n log log n)	1.75 s	97656 KB	5761455
Ciurul lui Eratostene optimizat	O(n* log log(sqrt(n)))	0.562 s	12207 KB	5761455
Fermat, K = 5 iteratii	O(K n logn)	44.82 s	22628 KB	5761587
Fermat K = 25 iteratii	O(K n logn)	105.008 s	26870 KB	5761496
Fermat K = 50 iteratii	O(K n logn)	179.498 s	22620 KB	5761477
Miller Rabin K = 5 iteratii	O(K n logn)	41.968 s	3336 KB	5761455
Miller-Rabin K=25 iteratii	O(K n logn)	108.753 s	23864 KB	5761455
Miller-Rabin K=50 iteratii	O(K n logn)	186.413 s	9641 KB	5761455

Figura 3.4:

3.3 Interpretarea Datelor

În ceea ce priveşte algoritmii probabilistici (Fermat si Miller-Rabin) , deşi factorul de amplificare pentru complexitatea worst case a lui Fermat este mai mic decât cel a lui Miller-Rabin(de doua ori mai mic) , pentru date de intreare ce depăşesc 10^6 ,cei doi au timpii de rulare foarte asemănători deoarece numărul numerelor Carmichael creşte iar cel de-al doilea depistează un procent mult mai mare (3/4 după cum am demonstrat în subsecţiunea de implementare). Astfel nu se vor mai executa toate iteraţiile , timpul de executare îmbunătaţindu-se(acest lucru se poate obersva si analizând tabelele).Prin urmare,atunci când vrem să verificăm dacă un număr este prim , pentru date mai mici de 10^6 putem folosi şi Fermat , altfel vom utilizat cel de-al doilea algoritm deoarece nu doar că are o precizie mult mai buna în detectarea numerelor pseudoprime(compuse) , dar sunt detectate şi mult mai rapid (pentru mai puţine baze/iteraţii).

Observăm de asemenea faptul că Fermat pentru un număr de iteratii mic(5) nu are o precizie de 100% în ceea ce priveşte rezultatul final obţinut , dar cu cât marim numărul iteraţiilor, numărul cazurilor în care întoarce TRUE pentru un număr compus se reduce. Iar pentru un numar de iteraţii destul de mare (50) am obţinut pentru N = 10^5 şi N = 10^6 datele de intrare numărul exact de numere prime(suntem siguri de acest rezultat prin

comparaţie cu rezultatele de la algoritmii determinişti).Pentru N mai mare de 10^6 , K = 50 de iteraţii nu este suficient pentru a determina exact numarul de numere prime (mai mici decat 10^7 şi 10^8).In particular am testat fermat pentru 100 de iteraţii , caz în care am obţinut rezultatul asteptat. Prin urmare , putem considera acest număr al iteraţiilor un factor de securitate, cu cât acesta creste , cu atât scade posibilitatea de a returna un rezultat greşit în testarea unui număr. Din păcate acest factor de securitate se reflectă şi în timpul de executie după cum reiese şi din tabele, timpul de rulare fiind proportional cu numărul iteraţiilor.

Cât despre ceilalţi 3 algoritmi determinişti se comporta la fel pentru toate datele de intrare. Dacă în zona de implementare nu erau foarte vizibile îmbunaţaţirile aduse Ciurului lui Eratostene, acestea se pot vizualiza foarte usor în zona de testare.Diferenţele exista atât pentru timpul de rulare , dar mai ales pentru memoria utilizată, acestea sunt din ce în ce mai evidente pe măsură ce mărim N-ul.De asemenea, din aceste tabele se poate observa mult mai uşor cât de mare este timpul de rulare atunci cand în complexitatea unui algoritm este n şi nu $\log n$ pentru date de intrare mari , fie acea complexitate $\log n$ înmulţită cu un K de dimensiuni mari.

4 Concluzii

După ce am analizat toţi algoritmii în secţiunile anterioare putem concluziona astfel: În cadrul problemei actuale (determinarea numerelor prime mai mici ca un N dat) cele mai bune performanţe dacă privim atât complexităţile , cât şi timpii de rulare , sunt obţinute de Ciurul lui Eratostene , aceast rezultat fiind oarecum de aşteptat , întrucât principala "întrebuinţare" a acestui algoritm este să rezolve problema pe care am ales-o pentru temă. Daca însă am privii comportamentul algoritmilor prezentaţi în contextul altei probleme cum ar fi "Testarea primalităţii unui numar X citit de la tastatură" , încercând sa soluţionam problema cu Ciurul lui Eratostene am obţine rezultate mult mai slabe comparativ cu Miller-Rabin sau Fermat(acesta este problema pentru care se potrivesc cel mai bine) .Cei doi vor rezolva problema în O(klog n), în schimb ciurul lui Eratostene va execută acelaşi număr de paşi pentru a vedea daca X este prim sau nu(în O(n log log n)) .

Dacă ar trebui sa alegem dintre cei doi algoritmi probabilistici avem doua opţiuni : Pentru numere mai mici de 10^6 putem folosim şi algoritmul lui Fermat,cu condiţia să avem un factor de securitate destul de mare , iar pentru valori mai mari folosim Miller-Rabin din două motive foarte importante : 1. frecventă de apariţie a numerelor compuse este din ce in ce mai mare 2. Acesta detecteaza un procent de $\frac{3}{4}$ din numerele pseudoprime, dupa o singură iteraţie (nedepinzând de baza aleasă în mod aleator), singurele numere care creeaza probleme acestui algoritm sunt cele cu proprietatea de tare pseudoprime în raport cu o bază b(doar $\frac{1}{4}$ din numerele pseudoprime au aceasta proprietate), putem rezolva acest dezavantaj alegând numâr de iteraţii suficient de mare.

Bibliografie

[1] Algoritmul lui Fermat: http://www.geeksforgeeks.org/primality-test-set-2-fermet-method/

[2] Algoritmul Miller-Rabin: http://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/

[3] Ciurul lui Eratostene: http://www.infoarena.ro/job_detail/153303?action=view-source

[4] Ciurul lui Eratostene Îmbunătăţit: http://www.infoarena.ro/ciurul-lui-eratostene

[5] Meissel-Mertens constant: http://mathworld.wolfram.com/MertensConstant.html

[6] Introducere în Criptografie: http://www.aut.upt.ro/ bgroza/Slides/-Carte_Intro_Cripto.pdf

[7] Introducere în Criptografie 2: http://math.ucv.ro/ dan/courses/carte_Alg.pdf