DIMENZIÓK
Matematikai Közlemények
XII. kötet, 2024 85–90

doi:10.20312/dim.2024.09

A Fermat-féle prímteszt R-kódjának optimalizálása[1]

Tóth Zsolt
Soproni Egyetem, Faipari Mérnöki és Kreatívipari Kar, Alaptudományi Intézet
toth.zsolt@uni-sopron.hu, 0000-0003-0999-784X

ÖSSZEFOGLALÓ. A kutatás célja az R nyelven megvalósított Fermat-próba tesztelése és hatékonyabbá tétele volt. Az R-kódban a moduláris hatványozást manuálisan implementáltuk, míg a főleg összehasonlítási alapként szolgáló Python natív függvényét használtuk. A teljesítményteszteket két intervallumban végeztük el, és megvizsgáltuk a párhuzamos feldolgozás, valamint a R-be ágyazott C++ függvények hatását. Eredményeink rávilágítanak arra, hogy az R rugalmassága és egyes kódrészletek gépközelibb nyelveken való felgyorsítása jelentős előnyöket biztosíthatnak a nagy számításigényű feladatok megoldásában.

ABSTRACT. The aim of the research was to test and improve the efficiency of the Fermat primality test implemented in R. In the R code, modular exponentiation was manually implemented, while Python's native function, primarily used as a benchmark for comparison, was utilized. Performance tests were conducted over two intervals, and we examined the effects of parallel processing and C++ functions embedded in R. Our results highlight that the flexibility of R, combined with speeding up certain code segments using lower-level languages, can offer significant advantages in solving computationally intensive tasks.

1. Bevezetés

A prímtesztelés egy olyan matematikai eljárás, amelynek célja annak megállapítása, hogy egy adott természetes szám prím-e. Ez a probléma alapvető jelentőségű a számelméletben és számos gyakorlatias célú alkalmazásban, például a kriptográfiában. A prímtesztelés egyik fontos eljárása a Fermat-próba [2], amelynek több változata is létezik.

A dolgozat keretében megvizsgáltuk, hogy a Fermat-próba alapváltozatának a statisztikusok és adatelemzők által gyakran használt R programozási nyelvben [6] írt kódja milyen eljárásokkal tehető hatékonyabbá. Vizsgálataink nem voltak teljes körűek, de rávilágítottak arra, hogy az R környezetének rugalmassága és az adattudományban példátlan nyelvi gazdagsága a $gyorsR$ filozófiáját követve számos programozási területen kihasználható.

Az R (és elődje, az S) eredetileg általános célú és a magas szintű programozási nyelveken belül korántsem lassú nyelv, hiszen alapfüggvényeit a gyorsaság kedvéért jelentős részben Fortranban, C-ben és C++-ban írták, de használatát a nem feltétlenül programozói vénával megáldott tudományos kutatók, illetve a szakterületekhez kapcsolódó oktatás elvárásaihoz szabták [7]. Az adott tudományterület (kényelmi) igényeihez igazított csomagok viszont az R széles körű elterjedésének köszönhetően jelentős részben nem professzionális programozói megközelítés

[1] ENGLISH TITLE. Optimizing the Fermat primality test in R code.
KULCSSZAVAK. Fermat-féle prímteszt, R, Python, C++, benchmark.
KEYWORDS. Fermat primality test, R, Python, C++, benchmark.

mentén, R-ben készültek. Sebességük gyakran nem a legoptimálisabb, mint ahogy a nem optimalizált kódok bármely programozási nyelvben lehetnek lassúak. Ez jellemzően még közepes és nagy adatállományokra épülő kutatási és oktatási feladatok esetén sem jelent gondot, hiszen pl. a két ezredmásodperc helyett egy tizedmásodperc alatt lefutó kód az elemzések jellege miatt többnyire semmilyen problémát nem okoz. Extrém nagy adatállományok vagy számítási igények (pl. nagy prímek keresése, gépi tanulás) esetén viszont – ahol a kódok akár hetekig futnak – annál inkább. Tisztán programozói perspektívából természetesen vissza lehet térni a gyorsabb, gépközelibb nyelvekhez, azonban ez a legtöbb kutató és oktató számára frusztráló elvárás, hiszen az elemzési módszertan közvetlen megvalósítását teljesen „kiveszi a kezükből". Ráadásul nem reális, hogy a tudományos életben és az oktatásban felmerülő programozási feladatokat minden esetben alacsony szintű – vagy legalábbis sokkal gyorsabb – programozási nyelvekben professzionális tudásra szert tett programozók oldják meg, hiszen egyszerűen nincs ennyi humán kapacitás, illetve az ilyen típusú programozási feladatoknál az adott szakterületet is alaposan ismerni kell. Az oktatás – különösen a nem programozói kurzusokon tanuló, Z-generációhoz tartozó diákok oktatása – során pedig az amúgy is riasztó fejlesztőkörnyezet gépközelivé alakítása megfelelően (az elvárhatóhoz képest egészen kivételes módon) felkészült oktatók mellett is borítékolható pedagógiai csőd [3].

Követhetőbb módszernek tűnik, ha csak a problémás kódrészletekre koncentrálunk. R-ben optimalizáljuk, vagy esetleg más programozási nyelvet, professzionálisabb programozási módszertant követve felgyorsítjuk azokat. Az R mindkét megoldást „ab ovo" támogatja. Dolgozatunkban erre mutattunk néhány példát, majd az alkalmazásokat a programozási „benchmark" mérési és statisztikai eszközeivel hasonlítottuk össze.

2. Anyagok és módszer

Az elemzés során először felvázoltuk a Fermat-féle prímteszt alapvető jellemzőit és pszeudokódját. A módszertani jellemzők mellett a megvalósítás R-kódjának és az összehasonlítási alapként szolgáló Python-kód legfontosabb részeit és jellemzőit is közöltük, majd két eltérő méretű intervallumra teszteltük a kódok sebességét. Ezután párhuzamos feldolgozással és klaszterezéssel, ill. a moduláris hatványozást C++-ban implementálva az R-kódot felgyorsítottuk. A kódokat ebben az esetben is csak fő vonalakban közöltük, s itt is mértük azok sebességét. Végül a kapott eredményeket a megfelelő statisztikai próbákra építve összehasonlítottuk.

2.1. A vizsgált algoritmus

A Fermat-féle prímteszt a kis Fermat-tételen alapul. A tétel szerint, ha p prímszám és a egy olyan egész szám, amely nem osztható p-vel, akkor

$$a^{n-1} \equiv 1 \pmod{n}. \tag{1}$$

Ez azt jelenti, hogy ha p prímszám, akkor bármely a-ra, amely nem osztható p-vel, a^{p-1}-t p-vel elosztva 1 maradékot kapunk (Algoritmus 1.).

Léteznek olyan összetett n számok, amelyeknél (1) teljesül bizonyos a-kra. Ezeket az összetett számokat Fermat-pszeudoprímeknek nevezzük. A pszeudoprímek speciális osztályát alkotják a Carmichael-számok, amelyek különlegesek abból a szempontból, hogy bármely a-ra, amely relatív prím n-hez, (1) teljesül.

A Fermat-próba tehát egyszerű és gyors algoritmus a prímszámok ellenőrzésére, de nem minden esetben megbízható, mivel a pszeudoprímek és a Carmichael-számok hamis pozitív

Algoritmus 1. Fermat-féle prímteszt

Input: Egy egész szám p, és egy pozitív egész a, ahol $1 < a < p - 1$
Output: *Valószínűleg prím* vagy *Nem prím*
procedure FERMATPRÓBA(p, a)
    if $a^{p-1} \not\equiv 1 \pmod{p}$ **then**
        return *Nem prím*
    else
        return *Valószínűleg prím*
    end if
end procedure

eredményeket adhatnak, azaz nem prímekre is teljesíti az (1) tulajdonságot. Részben az anomáliák esélyének csökkentésére a Fermat-féle prímtesztnek több módosított változata létezik. Azonban a Fermat-próba „hatékonysága" megfelelő módszert követve magas [4]. A hamis pozitív eredmények kiszűrését szolgáló, összetettebb módszerek használatától most eltekintettünk.

2.2. A kódok kialakításának szempontjai

Mindkét nyelv esetén definiáltunk egy-egy függvényt, amelynek futási idejét a megfelelő számú elemből álló objektum segítségével teszteltük. Emellett törekedtünk az alkalmazott módszert a programozás alapelemeire (szekvencia, feltételes elágazás, iteráció) korlátozni, de esetenként az alapcsomagok (gyors) függvényeit is használtuk. A kódolás során speciális könyvtárakat tehát csak „végszükségből" hívtunk meg, a lehető legelemibb programozási módszertant követtük. A futtatás során mindent megtettünk a futtatókörnyezet eltéréseinek és anomáliáinak kiküszöböléséért [10]. Az R-ben, a Fermat-féle prímteszt során kénytelenek voltunk a moduláris hatványozást definiálni, míg a Pythonban erre nem volt szükség a rendelkezésre álló `pow` függvény miatt, amelyet C-ben írtak, s meglehetősen gyors függvénynek számít [1]. Viszont a Python használata során külön meg kellett hívnunk a `random` és a `time` modulokat.

A prímszámkeresést az $\{x \in \mathbb{Z} \mid 1 \leq x \leq N\}$ intervallumokban végezzük el, ahol $N = 100$ és $N = 10000$. Bár variabilitásra és anomáliákra nem számítunk, az $1-\beta = 0{,}8$ statisztikai erőt általában biztosító $m = 30$ kísérletet (futtatást) $m = 50$-re növeltük. A kiinduló Fermat-próba során eltértünk az általánosan használt és ajánlott $k = 10$ értéktől, $k = 30$-ra módosítottuk. (A k a véletlenszerűen választott a alapok számát jelöli.) Az összehasonlíthatóság kedvéért $N = 100$ esetében is, ahol erre nem lett volna szükség [5]. Nagyobb intervallumok esetén folytatott vizsgálatok során ez kiküszöbölheti a hamis pozitív eredményeket. Az előtesztelés során a Fermat-próbánál előfordult, hogy $k = 10$ és $N = 10000$ mellett nem minden esetben kaptunk 1229 prímet, $k = 30$ esetén ez nem fordult elő.

A kísérletek során – valószínűleg a legtöbb matematika- és statisztikatanár számára is elérhető konfigurációkhoz hasonló – Intel(R) Core(TM) i3, 3 GHz-es processzort, 8 GB RAM-ot és 64 bites Windows 10 operációs rendszert használtunk. A programkódokat R 4.3.2-ben, az `Rcpp` csomag R-verzióval kompatibilis C++ nyelvi változatában és Python 3.12.0-ban írtuk. A programkódoknak csak a legfontosabb részleteit és jellemzőit adtuk meg, azonban az összes kód elérhetőségét biztosítottuk [8]. A prímteszt megvalósítás szempontjából központi jelentőségű függvényét R-ben és Pythonban hasonló módon valósítottuk meg, a korábban jelzett jellemzőkkel és eltérésekkel (1. táblázat). Az R-kódban a futási időt a `Sys.time()` függvénnyel mértük, amely minden teszt futásának kezdetén és végén rögzítette az időt.

A Python-kód hasonló módon működött, de a `time.perf_counter()` függvényt hasz-

R

```r
fermat_teszt <- function(n, k) {
 if (n <= 1) {
  return("Összetett")
 }
 if (n == 2) {
  return("Prím")
 }
 for (i in 1:k) {
  a <- sample(2:(n - 2), 1)
  # Moduláris hatványozás
  result <- 1
  base <- a %% n
  exp <- n - 1
  while (exp > 0) {
   if (exp %% 2 == 1) {
    result <- (result * base) %% n
   }
   exp <- exp %/% 2
   base <- (base * base) %% n
  }
  if (result != 1) {
   return("Összetett")
  }
 }
 return("Prím")
}
```

Python

```python
def fermat_teszt(n, k):
if n <= 1:
return "Összetett"
if n == 2:
return "Prím"
for _ in range(k):
if n > 3:
a = random.randint(2, n - 2)
else:
a = 2
# Moduláris hatványozás
result = 1
base = a % n
exp = n - 1
while exp > 0:
if exp % 2 == 1:
result = (result * base) % n
exp //= 2
base = (base * base) % n
if result != 1:
return "Összetett"
return "Prím"
```

1. táblázat. A megoldás alapfüggvénye R-ben és Pythonban

náltuk időmérésre.

A Fermat-próba felgyorsítása érdekében az R `parallel` könyvtárát használtuk, amely lehetővé tette a párhuzamos feldolgozást. A `detectCores()` függvénnyel meghatároztuk a rendelkezésre álló processzormagok számát, majd a számok párhuzamos feldolgozásához klasztereket képeztünk.

Másik módszerünk az R-kódot lassító kódrészlet kiváltása volt C++ programozási nyelvben. Az `Rcpp` csomag lehetővé tette, hogy C++ kódot ágyazzunk be R-be, így a számításigényes moduláris hatványozást felgyorsíthattuk (2. táblázat).

A fenti módszerekkel előálló kódokat parancssor segítségével futtattuk.

A vizsgálatok révén tehát összesen nyolc, egyenként ötven elemű mérési eredményt kaptunk, $N = 100$-ra és $N = 10000$-re is négyet-négyet. Az így előálló adatállományok tartós elérhetőségét biztosítottuk [9].

A kapott eredményeket (a futási időket) a kísérleti eredmények/minták összehasonlításakor gyakran használt statisztikai próbák segítségével, illetve egyszerű arányokkal fejeztük ki.

3. Eredmények

A mért futási idők – külön-külön az $N = 100$ és az $N = 10000$ értékek által fémjelzett négyelemű csoportokban – egyszerű szemrevételezés után is igen jelentős különbséget mutattak, amit az elvégzett próbák is megerősítettek. A normalitás vizsgáló Shapiro-Wilk próba p-értékei minden esetben jelentősen az $\alpha = 0{,}05$ alatt maradtak, a Levene-próba szintén alacsony p-értékei pedig

C++

```
#include <Rcpp.h>
#include <random>
using namespace Rcpp;
// Moduláris hatványozás függvény
long long modexp(long long base, \
long long exp, long long mod) {
 long long result = 1;
 base = base % mod;
 while (exp > 0) {
  if (exp % 2 == 1) {
   result = (result * base) % mod;
  }
  exp = exp >> 1;
  base = (base * base) % mod;
 }
 return result;
}
 CharacterVector fermat_test\
 (int k, int max_n) {
     CharacterVector results(max_n);
           for (int n = 1; \
           n <= max_n; ++n) \
  {
  if (n <= 1) {
   results[n - 1] = "Összetett";
   continue;
  }
  if (n == 2) {
   results[n - 1] = "Prím";
   continue;
  }
  std::default_random_engine\
   generator;
  std::uniform_int_distribution\
  <long long> distribution(2,\
   n - 1);
  bool is_prime = true;
  for (int i = 0; i < k; ++i) {
   long long a = distribution\
   (generator);
   if (modexp(a, n - 1, n) \
   != 1) {
    is_prime = false;
    break;
   }
  }
  results[n - 1] = is_prime ? \
  "Prím" : "Összetett";
 }
 return results;
}
```

2. táblázat. A moduláris hatványozás C++ kódja

az azonos szórásokra vonatkozó nullhipotézist zárták ki. Ezzel nyilvánvalóvá vált, hogy az átlagok összehasonlítása, ill. az átlagok összehasonlítására szolgáló próbák (T-próba, Welch-próba) használata értelmetlen. Esetünkben tehát csak az egyes minták mediánjai hasonlíthatók össze. Ezek azonossága a nagy eltérés miatt fel sem merült, de azonosságukat a párosan elvégzett Wilcoxon-próbák és a négyes csoportokban elvégzett Mann-Whitney U-próbák igen alacsony p-értékei is kizárták.

N	Módszer	Medián	Arányszámok
100	R	0,00652802	151
	Python	0,0009081	21
	R (parallel)	0,005262017	122
	R (*rcpp*)	$4{,}315376 \times 10^{-5}$	1
10000	R	0,4835695	121
	Python	0,0911599	23
	R (parallel)	0,2730645	68
	R (*rcpp*)	0,003997445	1

3. táblázat. A futási idő mediánjai (mp) és a legkisebb időre vetített arányszámok

A mérési eredmények mediánjai igen jelentős különbségeket mutatnak (3. táblázat). Jól látható, hogy a moduláris hatványozást hatékony függvénnyel kezelő Python-kód sokkal gyorsabbnak bizonyult, mint a kezdeti R-kód. A parallel programozás viszonylag kis mértékben csökkentette az R-kód futási idejét, azonban az *rcpp* csomaggal beépített C++ kódrészlet az

R-kód extrém (151-, ill. 121-szeres) gyorsulását eredményezte. Az N 100-szorosára (100-ról 10000-re) növelése a futási időt a Python-kódnál 100-szorosára, a kiinduló R-kódnál 74-szeresére, a parallel programozással támogatott R-kódnál 52-szeresére, az $rcpp$-vel támogatott R-kódnál 93-szorosára növelte. Úgy tűnik tehát, hogy nagyobb feladatnál (esetünkben) a Python-kód és az R ($rcpp$) kód előnye valamelyest kisebbnek bizonyult.

4. Konklúzió

Az R és a Python elleni programozói kritikák nem mindig veszik figyelembe a két programozási nyelv gyakran „speciális” felhasználói körét, tehát azt, hogy mindkét nyelv viszonylag könnyen elsajátítható és – ha a programok kialakításának munkaerő-szükségletét is figyelembe vesszük – hatékonyabb elemzési eszköz kutatók, oktatók, tanárok, diákok kezében, mint pl. a szoftverfejlesztésben használt programozási nyelvek. A programkódok esetleges lassúságát is ebben a kontextusban érdemes szemlélnünk. A mérési eredmények megerősítették azt a feltevésünket és egyben tapasztalatunkat, hogy a statisztikai-adatelemzési területen a funkcionalitást és a rugalmasságot tekintve talán a legjobban teljesítő R egyfajta hibrid szemléletet követve – ha az esetleg lassabb programkódelemeket szükség szerint felgyorsítjuk – hatékonyan használható erőforrás-igényesebb feladatok esetén is. A minden programozási területen rugalmas, gazdag funkcionalitású és egyszerű szintaktikájú Python-hoz hasonlóan az R mélyén ott futnak – vagy bármikor kialakíthatóak – gyors, de gyakran professzionális programozói kvalitásokat igénylő C-, C++ és Fortran-kódok, amelyek hidat képeznek a 20. század közepe óta felhalmozott, manapság talán nem eléggé tisztelt, programkódokban kijegecesedett tudással. Ez a tudás azonban pl. az R viszonylagos felhasználóbarátsága nélkül jóval kisebb körben lenne mobilizálható a mai, megváltozott, felgyorsult körülmények között.

Irodalomjegyzék

[1] *python/cpython*, 2024, original-date: 2017-02-10T19:23:51Z.
URL https://github.com/python/cpython

[2] **Crandall, R. and Pomerance, C. B.**: *Prime Numbers: A Computational Perspective*, Springer, New York, 2005, 2nd edn.

[3] **Horváth-Szováti, E.**: *A matematikatanítás eredményességét növelő módszerek a felsőoktatásban*, Dimenziók, **6** (2018), No. 6, 73–77, number: 6. doi: 10.20312/dim.2018.09.

[4] **Khadir, O. and Szalay, L.**: *Experimental results on probable primality*, Acta Universitatis Sapientiae, Mathematica, **1** (2009), No. 2, 161–168.

[5] **Knuth, D.**: *Art of Computer Programming, The: Seminumerical Algorithms, Volume 2*, Addison-Wesley Professional, Boston, 1997, 3rd edn.

[6] **Pödör, Z.**: *Adatbányászat – FIM algoritmusok*, Dimenziók, **1** (2013), No. 1, 51–56, number: 1.

[7] **Pödör, Z.**: *Az R szoftver alkalmazása az Adatbányászat tárgy oktatásában*, Dimenziók, **3** (2015), No. 3, 14–20, number: 3. doi: 10.20312/dim.2015.02.

[8] **Tóth, Z.**: *Fermat-féle prímszámpróba kódjai*, Zenodo, 2024 doi: 10.5281/zenodo.13916614.

[9] **Tóth, Z.**: *Fermat-féle prímszámpróba mérési eredményei*, Zenodo, 2024 doi: 10.5281/zenodo.13918303.

[10] **Wickham, H.**: *Advanced R, Second Edition*, Chapman and Hall/CRC, New York, 2020, 2nd edn. doi: 10.1201/9781351201315.