Vaja 1 – Praštevila

1. Splošna predstavitev problema

Praštevila so naravna števila, ki imajo natanko dva delitelja, število ena in samega sebe. Prav zaradi te definicije števila 1 ne štejemo med praštevila, saj ta dveh deliteljev nima, ker je deljivo zgolj samo s sabo. Praštevila so zelo pomembna v teoriji števil in sicer se lahko vsako neničelno naravno število *n* faktorizira v produkt praštevil, oziroma v produkt potenc različnih praštevil. Celo več, ta faktorizacija je lahko izvedena na en sam možen način, če seveda ne upoštevamo različnega možnega vrstnega reda posameznih členov v produktu.
To lastnost praštevil so ugotovili že v starem Egiptu, z njimi pa so se veliko ukvarjali v stari Grčiji, še posebej aleksandrijski matematik Evklid (okoli 300 pr.n.š.), ki ga bolj poznamo kot očeta geometrije.
Praštevila pa niso bila pomembna samo v starih civilizacijah, ampak so enako ali pa še bolj pomembna danes v računalništvu, še posebej pri RSA kriptografiji, ki temelji na tem, da je zelo težko faktorizirati velika števila.
Naša prva vaja bo tako poiskati *n* mestno praštevilo, pri čemer se bomo lahko omejili na število decimalnih mest ali pa na število bitov, ki jih imamo na voljo za zapis števila. Za generiranje bomo uporabili dve metodi, naivno in pa Miller-Rabinov psevdo test praštevil, ki se uporablja pri generiranju velikih praštevil, se pravi prav takih, ki jih uporabljamo pri RSA kodiranju.
Da bomo z generiranjem praštevil lahko začeli, bomo potrebovali dober generator naključnih števil, ki ga bomo prav tako morali napisati sami, saj različni programski jeziki in sistemi uporabljajo različne pristope. Pri tem se bomo osredotočili na linearne kongruentne generatorje, ki so še vedno najpogosteje uporabljeni psevdonaključni generatorji, kadar nimamo na voljo posebnih strojnih generatorjev. Zaradi tega bomo najprej pogledali, kako generirati splošno naključno naravno število, nadaljevali pa bomo s postopki za generiranje praštevil.

2. Pomoč pri implementaciji

V nadaljevanju bomo najprej pogledali metodo za generiranje velikih naključnih števil, ki jih bomo nato uporabili pri generiranju praštevil, nato pa bomo pogledali še obe metodi za generiranje praštevil.

Linearni kongruentni generatorji (LCG)

Da bomo lažje začeli najprej ponovimo, kaj pomeni izraz kongruenten, ki je del imena generatorjev naključnih števil, ki nas zanimajo. Velja, da sta dve števili kongruentni, če je ostanek pri deljenju teh dveh števil z istim deliteljem enak. Ta relacija je tudi razvidna iz rekurzivne enačbe, ki definira LCG. Enačba je naslednja:

$$RN_{i+1} = aRN_i + b \pmod{m},$$

pri čemer so a, b in m številske konstante, ki definirajo generator. Naključno število RN_{i+1} se izračuna neposredno iz svojega predhodnika RN_i. Število RN_0 imenujemo seme in ga lahko poda uporabnik, ali pa ga fiksiramo znotraj generatorja. Linearne kongruentne generatorje označujemo z oznako $LCG(m, a, b, RN_0)$ in jih poznamo zelo veliko. Preden lahko generator v praksi uporabimo, mora prestati veliko testov, s katerimi pa se ne bomo ukvarjali, ampak bomo kot naš generator uporabili kar preizkušen generator, z imenom Super–Duper. Ta je definiran z naslednjimi konstantami: $LCG(2^{32}, 69069, 0, 1)$.

Naivna metoda za iskanje praštevil

Naivna metoda za iskanje praštevil, je eliminacijska metoda, ki izhaja iz dejstva, da je praštevilo deljivo samo z dvema vrednostma, z vrednostjo ena in sama s sabo. Za poljubno n– mestno naključno število r tako v testu preverimo vsa števila, ki bi lahko bila potencialni delitelji. Takšnih kandidatov je relativno malo, saj pridejo v poštev zgolj števila manjša od $\sqrt{r}$. V kolikor najdemo na ta način še kakšnega delitelja, število n ni praštevilo, zato generiramo novega kandidata. Naivna metoda za generiranje praštevil je prikazana v izpisu 1:

```
function NAIVNA()
 begin
      generiraj naključno n-mestno število r;
      if r je sod then
         r := r+1;

      while praštevilo ni najdeno do
         g := sqrt(r);
         j := 3;
         while r/j ni celo število and j<=g do
            j := j+2;
         end
         if j>g then
            return r;

         r := r+2;
      end
end
```

Izpis 1: Iskanje praštevil z naivno metodo

Iz izpisa vidimo, da naivna metoda izkorišča dejstvo, da imamo samo eno sodo praštevilo, tako da praštevila išče zgolj med lihimi števili. Prav tako pa so lihi tudi potencialni delitelji, tako da se kar najbolj zmanjša število potrebnih operacij. Kljub temu pa metoda ni primerna za generiranje velikih praštevil, kakršna so potrebna v kriptografiji. V takih primerih uporabljamo hitrejši pristop, kakršen je recimo Miller–Rabinov test.

Generiranje praštevil s pomočjo Miller–Rabinovega testa

Generiranje praštevil s pomočjo Miller–Rabinovega testa poteka podobno kot pri naivni metodi. S pomočjo generatorja naključnih števil zgenerimo *n*–mestno liho število *r*, za katerega z Miller–Rabinovim testom preverimo, ali je praštevilo. Če test pokaže, da gre za sestavljeno število, zgeneriramo novo naključno število. Zaradi hitrosti delovanja tudi tukaj novega števila ne pridobimo s pomočjo generatorja naključnih števil, ampak enostavno prištejemo trenutnemu naključnemu številu vrednost 2. Miller–Rabinov test ni eksakten, ampak nam pove, ali je testirano število praštevilo zgolj z določeno zanesljivostjo. Ta zanesljivost je odvisna od prametra *s*, ki ga skupaj s številom, ki ga testiramo, podamo kot vhodni parameter. Psevdokod Miller–Rabinovega testa je prikazan v izpisu 2.

```
function MILLER_RABIN(r, s)
 begin
   if r  <= 3 then return true;
   if r je sod then return false;
   Poišči takšna k in d, da velja:  d2^k=r−1

   for j:=1 to s do
     begin
      a := RANDOM(2, r-2);
      x := POW(a, d) mod r;
      if x ≠ 1 then
        for i:=0 to k-1 do
          begin
            if x = r-1 do break;
            x = POW(x,2) mod r;
          end

        if x ≠ r-1 then
          return false;
     end
   return true;
 end
```

Izpis 2: Psevdokod Miller–Rabinovega testa

Kot je iz izpisa 2 razvidno, Miller–Rabinov test kliče funkcijo RANDOM(*a*, *b*) in POW(*p*, *q*). Prva vrne naključno število z intervala (*a*, *b*), druga pa vrednost p^q. Funkcija MILLER_RABIN vrne vrednost *true*, če je *r* praštevilo, sicer vrne vrednost *false*. Funkcija RANDOM(*a*, *b*) je zelo preprosta in je prikazana v izpisu 3.

```
function RANDOM(a, b)
 begin
   return a + LCG() mod (b-a+1);
 end
```

Izpis 3: Psevdokod funkcije RANDOM

Vidimo lahko, da funkcija RANDOM uporablja linearni kongruentni generator, ki generira števila z intervala [0, *m*-1]. Ta je v našem primeru [0, 2^{32}-1]. Če bi števila enostavno generirali tako dolgo, dokler ne najdemo tistega z intervala [*a*, *b*], bi to enostavno trajalo predolgo, tako pa si pomagamo z ostankom pri deljenju, ki nam ta

proces občutno skrajša. Pomembni števili v Miller-Rabinovem testu pa sta tudi števili *d* in *k*, ki ju določimo tako da je izpolnjena enačba $d2^k = r - 1$. To naredimo s kodo, prikazano v izpisu 4.

```
                        .
                        .
                        .
d := r-1;
k := 0;
while d sod do
  begin
    d := d/2;
    k := k+1;
  end
                        .
                        .
                        .
```

Izpis 4: Reševanje enačbe $d2^k = r - 1$

Pri algoritmu moramo paziti predvsem na velika števila pri enačbi $a^b \bmod n$.
Uporabiti moramo modulsko potenciranje iz izpisa 5.

```
function MODULAR-EXPONENTIATION(a, b, n)
  begin
    d ← 1
    bodi ⟨bj, bj-1,..., b0⟩ dvojiška, j-mestna predstavitev števila b
    for i ← j downto 0
        begin
            d ← (d · d) mod n
            if bi = 1 do d ← (d · a) mod n
        end
    return d
  end
```

Izpis 5: Modulsko potenciranje ($a^b \bmod n$)

Ker je Miller–Rabinov test veliko hitrejši od naivnega pristopa, je primeren za generiranje velikih praštevil pri čemer mislimo na 100 mestna števila, ki jih potrebujemo pri kriptografiji.

3. Zahteve naloge

Implementirati je potrebno aplikacijo za tvorbo praštevil. Pri tem je potrebno implementirati obe metodi, tako naivno kot Miller–Rabinovo. Uporabnik naj ima možnost vnesti, koliko mestno praštevilo želi in to v obliki zahtevanega števila cifer praštevila ali števila bitov, potrebnih za zapis števila. Pri tem se omejimo na deset mest v desetiškem zapisu, oziroma na 32 bitne vrednosti. Poleg generiranja števil naj program omogoča tudi testiranje, ali je vneseno število praštevilo ali ne, z obema metodama.