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 kriptografiji RSA, ki temelji na tem, da je zelo težko faktorizirati velika števila.
Naša 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 metodo in Miller-Rabinov psevdo test praštevil, ki se uporablja pri generiranju velikih praštevil, se pravi prav takih, ki jih uporabljamo pri kodiranju RSA.
Da bomo z generiranjem praštevil lahko začeli, bomo potrebovali dober generator naključnih števil, ki ga bomo napisali sami, saj različni programski jeziki in sistemi uporabljajo različne pristope (za primerjavo preverite sledeči vir: https://www.random.org/analysis/#visual). Pri tej vaji se bomo osredotočili na linearne kongruentne generatorje, ki so še vedno zelo razširjeni 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:

$$R_{i+1} = aR_i + b \pmod{m},$$

pri čemer so *a*, *b* in *m* številske konstante, ki definirajo generator. Naključno število RN_{i+1} se izračuna neposredno iz svojega predhodnika R_i. Število R_0 imenujemo seme in ga lahko poda uporabnik, ali pa ga fiksiramo znotraj generatorja. Linearne

kongruentne generatorje označujemo z oznako $LCG(m, a, b, R_0)$ 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: $LCG(2^{32}, 69069, 0, 1)$.

Ponavadi želimo, da nam LCG vrne *n*-bitna števila oz. števila v intervalu med a in b, zato zapišimo funkcijo RANDOM(*a*, *b*), ki vrne naključno število z intervala [*a*, *b*] in je prikazana v izpisu 2.

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

Izpis 1: 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.

Naivna metoda za iskanje praštevil

Naivna metoda za iskanje praštevil izhaja iz dejstva, da je praštevilo deljivo samo z dvema vrednostma, z vrednostjo ena in sama s sabo. Za poljubno *n*-mestno naključno potencialno praštevilo *p* tako v testu preverimo vsa števila, ki bi lahko bila potencialni delitelji. Takšnih kandidatov je relativno malo, to so zgolj števila manjša od $\sqrt{p}$. V kolikor najdemo na ta način še kakšnega delitelja, število *p* ni praštevilo, zato generiramo novega kandidata. Naivna metoda za generiranje praštevil je prikazana v izpisu 2:

```
function NAIVNA(n)
 begin
      generiraj naključno n-mestno število p;
      if p je sod then //praštevilo ne more biti sodo
         p := p + 1;

      while true do
         j := 3; //potencialni delitelj števila p
         while j<= √p  do
            if p/j je celo število
              break;
            j := j+2;
         end
         if j> √p  then //potencialnega delitelja od p nismo našli
            return p;

         p := p+2; // naslednje liho število
      end
end
```

Izpis 2: 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 z Miller–Rabinovim testom

Generiranje praštevil z Miller–Rabinovim testom poteka podobno kot pri naivni metodi. Z generatorjem naključnih števil zgeneriramo *n*–mestno liho število *p* (potencialno praštevilo), za katerega z Miller–Rabinovim testom z določeno zanesljivostjo 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 z generatorjem 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 parametra *s*, ki ga skupaj s številom, ki ga testiramo, podamo kot vhodni parameter. Psevdokod Miller–Rabinovega testa je prikazan v izpisu 3. Funkcija MILLER_RABIN pove, če je *p* praštevilo z določeno verjetnostjo ali zagotovo sestavljeno število.

```
function MILLER_RABIN_TEST(p, s)
begin
   if p <= 3 then return PRAŠTEVILO;
   if p je sod then return SESTAVLJENO_ŠTEVILO;

   Poišči takšna k in d, da velja:  d 2^k = p − 1

   for j:=1 to s do
   begin
      a := RANDOM(2, p-2);
      x := a^d  mod p;  // iskanje dokaza za sestavljeno število

      if x = 1 then
        continue;     // verjetno praštevilo

      // če ∃i: a^(d2^i) ≡ −1 (mod p)  potem je p verjetno praštevilo
      for i:=0 to k-1 do
      begin
        if x = p-1 then
          break;
        x := x^2  mod p;
      end

      if x ≠ p-1 then
        return SESTAVLJENO_ŠTEVILO;
   end
   return VERJETNO_PRAŠTEVILO;
end
```

Izpis 3: Psevdokod Miller–Rabinovega testa

Pomembni števili v Miller-Rabinovem testu pa sta tudi števili d in k, ki ju določimo tako, da je izpolnjena enačba $d\,2^k = p - 1$. To naredimo s kodo, prikazano v izpisu 4.

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

Izpis 4: Reševanje enačbe $d\,2^k = p - 1$

Pri algoritmu moramo paziti predvsem na velika števila pri enačbi $a^b \bmod n$, saj lahko število a^b preseže število bitov za predstavitev števil. Zato moramo uporabiti modulsko potenciranje iz izpisa 5.

```
function MODULAR-EXPONENTIATION(a, b, n)
  begin
    d ← 1
    bodi ⟨b_j, b_{j-1},..., b_0⟩ dvojiška, j-mestna predstavitev števila b
    for i ← j downto 0
        begin
            d ← (d · d) mod n
            if b_i = 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 z menijem oziroma grafičnim vmesnikom za tvorbo praštevil. Pri tem je potrebno implementirati obe metodi, tako naivno kot Miller–Rabinovo. Uporabnik naj ima možnost v aplikaciji vnesti, koliko mestno praštevilo želi in to v obliki zahtevanega števila bitov, potrebnih za zapis števila. Torej je 5-bitno število v bitnem zapisu v območju med 10000 in 11111. Pri tem se omejimo do 32 bitnih vrednosti. Poleg generiranja števil naj program omogoča tudi testiranje, ali je vneseno število praštevilo ali ne, z obema metodama. Uporabnik naj ima možnost določiti parameter *s* (za Miller-Rabinov test).
Pri tej nalogi nas zanima tudi, koliko je tvorba praštevil hitrejša z Miller-Rabinovim testom, zato izmerite čas trajanja tvorbe *n*-bitnih praštevil z naivno metodo in z Miller-Rabinovim testom. Narišite graf časovne zahtevnosti za obe metodi pri *n* v območju med 4 in 32. Glede na to, da je čas izvajanja Miller-Rabinovega testa odvisen od parametra *s*, narišite še graf časovne zahtevnosti tvorbe 32-bitnega praštevila z Miller-Rabinovim testom glede na parameter *s* v območju med 1 in 20.