

Generering av pseudoslumptal

JOHAN HÅSTAD

Datalogi, KTH

1. Inledning. På de flesta datorer behövs slumptal för många användningsområden (t.ex. för att stimulera förlopp i verkligheten). Datorer är ju deterministiska och därför kan de inte generera slumptal i ordets sanna mening. Istället tar ofta datorn ett slumpmässigt valt frö som t.ex. användaren bestämmer och sedan genererar från detta frö en lång serie av tal som kan användas som slumptal. Eftersom dessa inte är slumpmässiga brukar de kallas pseudoslumptal. En algoritm som tar ett kort slumpmässigt frö och producerar en längre sekvens av pseudoslumptal kallas en *pseudoslumptalsgenerator*. Denna uppgift går ut på att ge litet teori för sådana algoritmer och att studera ett exempel. Vi börjar med exemplet men vi måste först ge litet bakgrundsinformation.

2. Bakgrundsinformation. Låt m vara ett heltal och definiera att $a \equiv b \pmod{m}$ (utläses $a = b$ modulo m) innebära att a och b ger samma rest vid division med m . Vi har t.ex. att $7 \equiv 1 \pmod{3}$, $124 \equiv -6 \pmod{5}$ o.s.v.

Vårt exempel bygger på möjligheten att istället för att göra normal addition och multiplikation med heltal kan vi göra motsvarande operation och bara intressera oss för vilken rest svaret ger vid division med m . Dessutom behöver vi då bara veta vilken rest operanderna ger vid division med m .

Kontrollera att ovanstående påstående är riktigt.

Varje tal ger en av resterna $0, 1 \dots m - 1$ vid division med m . Vi kommer alltså att räkna med dessa tal. Vi har t.ex. följande

ekvationer

$$14 \cdot 5 \equiv 6 \pmod{16}$$

$$11 + 7 \equiv 1 \pmod{17}$$

$$4 \cdot 3 \equiv 0 \pmod{12}.$$

För ytterligare egenskaper av denna modulatoräkning, se boken av Hardy och Wright.

Nu kan vi definiera en flitigt använd och mycket studerad pseudoslumptalsgenerator.

$$\text{Frö} : (m, a, b, x_0).$$

(Ibland är m, a och b fixt och fröt är bara x_0 . Vi kommer dock att använda det stora fröt.) En följd tal $x_1, x_2 \dots$ genereras genom

$$x_i \equiv a \cdot x_{i-1} + b \pmod{m}.$$

Dessa tal används nu som slumptal. Denna generator kommer vi att kalla lineär kongruens generator och förkorta LKG.

Som ett exempel kan vi ta

$$m = 1241 \quad a = 613 \quad b = 113 \quad x_0 = 51$$

vilket ger

$$x_1 \equiv 613 \cdot 51 + 113 \equiv 31376 \equiv 25 \cdot 1241 + 351 \equiv 351 \pmod{1241}$$

$$x_2 = 613 \cdot 351 + 113 \equiv 583 \pmod{1241}$$

och serien blir

$$351, 583, 84, 724, 888, 899, 196, 1125, 983, 807, 886, 914.$$

Vid en ytlig inspektion ser denna serie ganska slumpmässig ut. Frågan är hur slumpmässig den är. Naturligtvis är den inte riktigt slumpmässig men den har några liknande egenskaper. Ungefär hälften av talen är jämna, ungefär hälften är större än $m/2$ o.s.v.

3. Statistiska test. Att räkna antalet jämna tal och se om det är ungefär hälften är ett exempel på ett statistiskt test. Man skulle vilja att pseudoslumptal uppför sig som riktiga slumptal vid de flesta och helst alla statistiska test. Tyvärr är det omöjligt att framställa pseudoslumptal som klarar alla statistiska test, förutsatt att pseudoslumptalserien är längre än det ursprungliga fröet.

Visa detta. (Fyll i detaljerna på nedanstående resonemang.)

LEDTRÅD. Anta att fröets längd skrivet binärt är N och de genererade pseudoslumptalen har sammanlagd binär längd M , där M är betydligt större än N . Kalla den aktuella algoritmen som genererar pseudoslumptal A . Vi kommer nu att definiera följande väldigt speciella statistiska test.

Är denna serie genererad av A på ett frö av längd N ?

På detta statistiska test kommer svaret alltid att bli *ja* om testet görs på en serie genererad av A . Om vi å andra sidan gör testet på riktiga slumptal är sannolikheten att svaret blir *ja* högst $2^N/2^M$ vilket är litet då M är större än N .

Den moderna definitionen av vad vi kan kräva av en pseudoslumptalgenerator utesluter test som är alltför komplicerade. Låt oss ge en informell version av denna definition:

DEFINITION. En algoritm A som genererar pseudoslumptal är *godkänd* om inget statistiskt test som går att utföra i rimlig tid uppför sig väsentligt olika på en serie tal genererade av A på ett slumpvist frö och på en serie riktiga slumptal.

För att göra den definitionen precis måste vi specificera *rimlig tid* och *väsentligt olika*. Det är inte så komplicerat men vi ger inte detaljerna här. Låt oss bara säga att rimlig tid betyder tid som är polynomiell i antalet siffror i fröt. (Om fröt har N siffror kan testet t.ex. ta N^3 tid att utföra.)

För att belysa punkten om rimlig tid låt oss diskutera testet *Är denna serie producerad av algoritmen A på ett frö av längd N ?* Det naiva sättet att utföra detta test är att prova alla frön av längd N , generera pseudotalföljden genom att köra A och se om någon producerad serie överensstämmer med den aktuella serien. Det finns 2^N frön att prova och detta är ju inte polynomiellt och också i praktiken tar det för lång tid även för ganska små N .

Å andra sidan klarar en liten persondator ofta att generera serien snabbt som t.ex. är fallet om man använder den generator vi beskrev ovan.

4. Är LKG godkända? I denna avdelning kommer vi att visa att LKG inte är godkända. Sedan kommer vi att diskutera problem med att göra godkända generatorer.

Låt oss nu beskriva den algoritm som avslöjar att en till synes slumpmässig följd är genererad av en LKG. Lite hjälp hur man implementerar vissa operationer ges i slutet under titeln *Bra att veta*.

Given en talserie

$$x_1, x_2, x_3 \dots x_n$$

kommer vi att försöka att konstruera m , a och b så att $x_i \equiv a \cdot x_{i-1} + b \pmod{m}$. Om inga sådana m , a och b finns kommer det att visa sig under räkningarna. Om du vill kan du sluta läsa här och försöka utan de tips som följer.

Det visar sig vara bra att definiera $y_i = x_{i+1} - x_i$. Skälet är att om x_i uppfyller $x_{i+1} \equiv ax_i + b \pmod{m}$ så uppfyller y_i relationen $y_{i+1} \equiv a \cdot y_i \pmod{m}$.

Visa detta.

Därför räknar vi först ut serien y_1, y_2, \dots, y_{n-1} . Om y_1, y_2 och y_3 är de tre första talen vet vi att $y_2 \equiv ay_1 \pmod{m}$ och $y_3 \equiv a^2y_1 \pmod{m}$. Av detta följer att $y_1 \cdot y_3 \equiv y_2^2 \pmod{m}$. Således delar m talet $m_0 = y_1 \cdot y_3 - y_2^2$ (förutsatt att det inte är 0). Till att börja med kan vi gissa att $m = m_0$ och $a = a_0 = y_2/y_1 \pmod{m_0}$. (Om det finns flera möjligheter för y_2/y_1 , välj ett. Är y_2/y_1 odefinierat har vi gissat för stort m_0 . Hur detta korrigeras beskrivs under *Bra att veta*.)

Betrakta nu generatorm

$$\overline{y_i} \equiv a_0 \overline{y_{i-1}} \pmod{m_0} \text{ med } \overline{y_1} = y_1.$$

Om $\overline{y_i} = y_i$ för alla i har vi nog rätt generator. Annars kan man visa att eftersom $m|m_0$ att $a_0 \overline{y_i} \equiv ay_i \pmod{m}$, och således $\overline{y_{i+1}} = y_{i+1} \pmod{m}$. (Försök göra detta.) Detta innebär att m delar $\overline{y_{i+1}} - y_{i+1}$ om detta inte är 0. Således kan vi uppdatera vår gissning av m till största gemensamma delaren av m_0 och $\overline{y_{i+1}} - y_{i+1}$ (eftersom m delar båda talen). Konstruera en ny generator och börja om, o.s.v. Om vi kan bestämma a och m på detta sätt är det lätt att sen hitta b genom att

$$b \equiv x_2 - ax_1 \pmod{m}.$$

Fundera ut alla detaljer och implementera sedan ovanstående algoritm, eller gärna någon variant du själv hittar på.

Försök analysera hur lång tid din algoritm tar om det riktiga m har högst n siffror. (Hur många gånger kan du vara tvungen att på nytt gissa m ? Hur många operationer måste du göra för att hitta varje gissning?)

Två exempel att köra algoritmen på är följande:

```
18192 45941 42086 43501 31735 2718
41115 31870 28933 918
```

och

```
15584 34075 5151 39785 21388 35991
18143 31570 37764 15032 10300 .
```

Om du har någon intresserad kamrat kan ni byta exempel.

5. Diskussion. Nu när vi har visat att LKG inte är godkända kommer naturligtvis frågan vilka generatorer som är godkända. Några generatorer har konstruerats som antagligen är godkända, men det har ingen lyckats visa. Skälet är följande: För att visa att en generator inte är godkänd, behöver man bara visa att det finns en algoritm som går rimligt fort som skiljer på tal genererade av generatören och tal som är verkliga slumpal. För att visa att en generator är bra behöver man visa att varje statistisk test som går rimligt fort misslyckas med att skilja tal som generatören producerat från riktiga slumpal. I allmänhet att visa att vissa problem kräver lång tid att beräkna är ett mycket viktigt problem som fortfarande är öppet och det återstår mycket forskning inom detta område, som kallas *komplexitetsteori*.

Vad menar jag då med att det finns generatorer som antagligen är godkända. Jo, man har konstruerat generatorer som om de inte är godkända så går något mycket studerat beräkningsproblem att lösa betydligt effektivare än alla har lyckats med. T.ex. har man visat att det finns godkända generatorer om faktorisering av stora tal är svårt. Men det är inget bevis.

Försök konstruera en generator som är effektiv och verkar godkänd. Lättast är att producera tal på något iterativt sätt som vi gjorde med LKG. Använd någon mer komplicerad funktion än att

bara multiplicera med en konstant och lägga till en annan konstant. Använd gärna modulär räkning men försök att göra något annorlunda.

Ta reda på vilken pseudoslumptalsgenerator din dator använder och försök att visa att den inte är godkänd. Ofta använder den modulär räkning. Försök göra något liknande det vi gjorde med LKG. Gissa först m och hitta sedan de övriga konstanterna som ingår.

Bra att veta. I algoritmen behövs två trick. Nämligen att beräkna största gemensamma delare av två tal och att givet y_1, y_2 och m beräkna $y_1/y_2 \pmod{m}$. Båda görs med Euklides' algoritm.

Låt oss börja med största gemensamma delare. Största gemensamma delare av två tal a och b betecknas med (a, b) och är det största tal som delar både a och b . Idén bakom algoritmen är att om ett tal delar a och b så delar det också $a - k \cdot b$ för alla heltal k . Algoritmen beskrivs nu kanske enklast med ett exempel.

Låt oss ta talen 534 och 114. Anta att d delar dessa två tal, då delar det också

$$534 - 4 \cdot 114 = 78$$

och också

$$114 - 78 = 36$$

och också

$$78 - 2 \cdot 36 = 6$$

$$36 - 6 \cdot 6 = 0.$$

Nu slutar algoritmen eftersom vi fick talet 0. Det sista talet 6 kontrolleras lätt vara svaret.

Låt oss beskriva hur man beräknar $y_1/y_2 \pmod{m}$. Först måste vi definiera vad detta betyder. Låt oss säga att c är det tal så att

$c \cdot y_2 \equiv y_1 \pmod{m}$. Om det finns flera c så välj ett godtyckligt, finns det inget sådant tal är y_1/y_2 odefinierat.

Beräkna nu y_1/y_2 på följande sätt.

Beräkna först (y_1, y_2) . Om detta är d , använd att

$$y_1/y_2 = (y_1/d)/(y_2/d).$$

Vi kan således anta att $(y_1, y_2) = 1$. Om nu $(y_2, m) > 1$ är y_1/y_2 odefinierat (*visa detta*). Om $(y_2, m) = 1$, beräkna $e \equiv 1/y_2 \pmod{m}$ med Euklides algoritm. Sedan är svaret $y_1 \cdot e \pmod{m}$.

Vi ger ett exempel på hur man beräknar $1/53 \pmod{91}$. Utför Euklides algoritm på 53 och 91

$$91 - 53 = 38$$

$$53 - 38 = 15$$

$$38 - 2 \cdot 15 = 8$$

$$15 - 8 = 7$$

$$8 - 7 = 1.$$

Låt oss nu använda ekvationerna baklänges.

$$\begin{aligned} 1 &= 8 - 7 = 8 - (15 - 8) = 2 \cdot 8 - 15 \\ &= 2 \cdot (38 - 2 \cdot 15) - 15 = 2 \cdot 38 - 5 \cdot 15 \\ &= 2 \cdot 38 - 5 \cdot (53 - 38) = 7 \cdot 38 - 5 \cdot 53 = 7 \cdot 91 - 12 \cdot 53. \end{aligned}$$

Ur detta följer att

$$12 \cdot 53 \equiv -1 \pmod{91}$$

och

$$(-12) \cdot 53 \equiv 1 \pmod{91}$$

och således

$$79 \cdot 53 \equiv 1 \pmod{91}$$

d.v.s.

$$\frac{1}{53} \equiv 79 \pmod{91}.$$

Ibland i vår algoritm kan vi råka ut för tal y_1 och y_2 så att $y_1/y_2 \pmod{m_0}$ inte existerar beroende på att vår gissning m_0 av m är för stor. Om $(y_1, y_2) = 1$ beror detta på att $(y_2, m_0) > 1$ och vi byter nu ut m_0 mot $\frac{m_0}{(m_0, y)}$.

Vi kan vara tvungna att göra detta flera gånger men sedan återstår den största faktor av m_0 för vilken y_1/y_2 existerar.

Visa detta.

Litteratur

För en bok som behandlar elementär talteori Hardy, G.H. & Wright, E.M., *An Introduction to the theory of numbers*. Fifth edition, Oxford Univ. Press, Oxford 1979.

En utförlig diskussion av linjär kongruens generatorer finns i Knuth, D.E., *The Art of Computer Programming, Vol. II*. Semi-numerical Algorithms, Addison Wesley 1981.

Den moderna definitionen av slumpstal finns i Blum, M., Micali, S., How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13, s 850–864.

Algoritmen att LKG inte är godkänd är delvis tagen ur Plumstead, J.B., Inferring a sequence generated by a linear congruence. *Proceedings of 23rd IEEE Symposium on Foundations of computer Science*, s 153–159.