Pseudoslumptalsgeneratorer

 Karel Lang 37754

 Kandidatavhandling i datavetenskap

 Handledare: Patrick Sibelius

 Fakulteten för naturvetenskaper och teknik

 Åbo Akademi

 2016

Abstrakt

En pseudoslumptalsgenerator är en algoritm vars uppgift är att skapa slumpmässiga tal. På grund av att datorer är deterministiska är det omöjligt att skapa exakt slumpmässiga tal utan att använda sig av yttre störningar och därför kallas talen pseudoslumpmässiga. Pseudoslumptalsgeneratorer skapar oftast likformigt fördelade slumpmässiga (rationella) tal mellan noll och ett (en avancerad räknemaskin brukar ofta ha denna egenskap tillgänglig). Det finns algoritmer som kan omvandla dessa tal så de följer andra fördelningar, projektionsalgoritmer, vilket kan vara nyttigt i t.ex. forskning och simulationer. I denna avhandling behandlas begreppet slumpmässighet, genereringen av pseudoslumptal med hjälp av pseudoslumptalsalgoritmer och exempel på slumpmässighetstester. Målet med texten är att få läsaren att förstå hur viktigt det är att kunna skapa slumptal och hur svårt det sist och slutligen är att förstå vad slumpmässighet i själva verket är.

Innehållsförteckning

Begrepp

1. Inledning 1

 1.1 Vad är slumpmässighet? 1

 1.2 Var behöver man slumpmässighet? 3

 1.3 Olika typer av slumpmässighet 5

 1.4 Hårdvaru- och mjukvarumetoder 7

 1.5 Slumpmässighetstester 8

2. Genereringsalgoritmer 9

 2.1 Allmänt om genereringsalgoritmer 10

 2.2 Kryptografiskt säkra pseudoslumptalsgeneratorer 11

 2.3 Middle-square metoden 12

 2.4 Linjär kongruensgenerator 13

 2.5 Blum Blum Shub 16

 2.6 Mersenne Twister 18

 2.7 Yarrow och Fortuna 20

3. Referenser 21

Begrepp

Förkortningar:

Pseudoslumptalsgenerator (psg)

Slumptalsgenerator(sg).

Frö (på engelska seed): Fröet är ett tal som används som starttillstånd för en psg. Fröet skapas ofta med hjälp av datorns klocka. Då en dator kan ge tiden med nanosekundsprecision kommer fröet att vara så gott som slumpmässig

Periodlängd (på engelska period length): En psg:s periodlängd berättar hur många tal som kan genereras före en ny period börjar.

Determinism: En världsåskådning som menar att allting är orsaksbestämt. En deterministisk algoritm ger alltid ut samma entydiga output ifall man matar in samma input.

Indeterminism: En världsåskådning som menar att allting inte är orsaksbestämt. En indeterministisk algoritm kan ge flera olika output fastän man matar in samma input.

Slumpmässighet utan sannolikhetsmått: Något som är slumpmässigt, men följer inte de regler som den abstrakta matematiskt ideala slumpmässigheten följer (sannolikhetsdistribution, frekvens av följder, osv.)

Överuppräkneligt oändlig: En oändlighet som man inte kan beräkna. Antalet irrationella tal är t.ex. överuppräkneligt oändlig när däremot antalet rationella tal är uppräkneligt oändlig.

Konfidens: Ett mått på en statistisk beräknings korrekthet. Ju högre konfidens desto säkrare kan man vara att beräkningen är korrekt. Konfidensen varierar mellan noll och ett.

AI: Artificiell intelligens

Monte Carlo-metoder: En familj av algoritmer som använder slumpmässighet för att generera önskade resultat. Det går till exempel att beräkna π om man använder en stor mängd slumptal.

Axiom: En grundsats som inte kan bevisas i det system där den existerar. Man måste helt enkelt komma överens att axiomet gäller.

1. Inledning

Slumptal är viktiga i nästan all modern forskning. I fysiken behöver man slumptal för bland annat testande av modeller genom att använda simulationer. Slumptal är också en mycket viktig del av artificiell intelligens då det finns flera olika val som är lämpliga. Slumptal behövs även i datorspel och kryptering. Fastän slumptal är mycket viktiga, är det fortfarande en utmaning att skapa slumptal snabbt och pålitligt. Datorer är mycket dåliga på att skapa slumpmässighet så man måste lita på komplexiteten av algoritmen man använder för att skapa slumptal. Ju mer komplex algoritmen är, desto bättre. Vid skapande av slumptal krävs en hel del matematiska bevis, sannolikhetslära samt avancerad talteori för att vara säker på att slumptalen algoritmen skapar verkligen är slumpmässiga. Dåliga slumptal kan vara en säkerhetsrisk och de kan också leda till att Monte Carlo-metoder ger fel resultat. Därför är det mycket viktigt att forska slumpmässighet och slumptal både från en teoretisk samt praktisk synvinkel.

1.1 Vad är slumpmässighet?

Slumpmässighet är avsaknaden av förutsägbarhet. En följd av heltal mellan noll och nio är till exempel slumpmässig om vi inte kan förutspå nästa tal i följden oberoende av hur många tal i följden vi redan känner. För att kunna förutspå nästa tal i en följd behövs det ett mönster. Således kan man också säga att slumpmässighet är avsaknaden av ett mönster.

Kan vi vara säkra på att en följd inte har något mönster? Teoretiskt sett kan vi aldrig vara helt säkra, eftersom en följds mönster kan bestå av en periodisk talserie vars periodlängd är längre än antalet tal som vi vet om följden; om vi vet *N* stycken påföljande tal i en sekvens, kan det alltid hända att sekvensen har en periodlängd som är *N*+1.

Om det inte går att bevisa att en följd saknar ett mönster, finns det istället någon metod för att bevisa att en följd följer ett mönster? Teoretiskt sett är det här också en omöjlig uppgift, eftersom en slumpmässig följd kan se ut att ha ett mönster, på grund av att det totala antalet olika mönster är oändligt. Redan med perioder kan man skapa oändligt många mönster. En talföljd kan följa mönstret 10, 110 eller 1110. En period kan bestå av *N* antal ettor och en nolla. Periodlängden kommer då att vara *N*+1. Eftersom *N* kan vara vad som helst för heltal och varje unikt *N* skapar ett unikt mönster kan man säga att det existerar ett oändligt antal mönster.

Oberoende hurdan slumpmässig talföljd man har, kommer talföljden att följa ett oändligt antal olika mönster så länge som talföljden är ändlig. Dessa mönster är naturligtvis mycket komplicerade, men de är fortfarande mönster. Betyder det här att talföljden verkligen följer alla dessa mönster och följer alltså en slumpmässig följd oändligt många mönster? Svaret är nej eftersom en talföljd borde följa ett mönster i all oändlighet, för att man skulle kunna säga att talföljden verkligen följer mönstret. En slumpmässig följd kännetecknas av att då man vet oändligt många tal i följden, finns det absolut inga mönster som följden skulle kunna sägas följa. Det är alltså omöjligt att bevisa att en följd följer något mönster ifall vi inte vet hur följden genereras, eftersom en decimalföljd på tusen ettor inte medför att nästa värde i följden är en etta. Troligtvis kommer nästa värde att vara en etta, men vi kan fortfarande inte vara helt säkra, eftersom vi inte vet hur decimalföljden genereras.

Om det är omöjligt att bevisa varken det ena eller det andra, vad kan man då göra? Vi kan använda oss av slumpmässighetstester. Slumpmässighetstester är statistiska hypotesprövningar där man antar att följden man testar är slumpmässig och sedan räknar ut sannolikheten att en ideal slumpmässig följd följer slumpmässighetstestets mönster på samma sätt som följden man testar. På så sätt får man ett mått på hur avvikande följden är, jämfört med den ideala slumpmässigheten. Med hjälp av avvikelsen kan man skapa en konfidens om huruvida följden möjligtvis inte är slumpmässig [1]. I kapitel 1.6 behandlar jag slumpmässighetstester och metoderna som testerna gör bruk av för att hitta mönster i sekvenser samt problem som dessa metoder skapar.

Nu när vi har definierat vad slumpmässighet är, kan vi fråga ifall ren slumpmässighet existerar. Är alltså universum deterministiskt eller indeterministiskt? Vi undersöker denna fråga genom att förklara vad kausalitet är. Kausalitet är ett begrepp som används inom vetenskapen som en självklarhet, fastän det i verkligheten är frågan om ett axiom. Om vi gör två identiska experiment, kommer resultaten att vara identiska enligt kausalitetsbegreppet. Kausalitet är alltså en direkt följd av determinism. Kausalitetsaxiomet kan varken bevisas eller motbevisas. Om två identiska mätningar av ljusets hastighet inte ger samma resultat, kan kausalitetsaxiomet inte motbevisas för att det kan hända att mätningarna inte var sen heller identiska. Därför krävs det att man skall vara säker på att mätningarna är identiska, vilket kräver mer kunskap än det är möjligt att nå. Alltså kan man endast bevisa eller motbevisa kausalitetsaxiomet om man redan vet ifall den är sann eller falsk.

Numera har man i kvantmekanikens värld stött på något som inte tycks följa kausalitetsaxiomet, på grund av att identiska mätningar har givit olika resultat. Det har diskuterats ifall experimenten har påverkats av oupptäckta variabler, vilka har lett till dessa olika resultat. Det kan mycket väl hända att sådana oupptäckta variabler existerar, men är omöjliga för oss att upptäcka.

Ifall en del av kvantmekaniken verkligen är slumpmässig skulle kausalitetsaxiomet bevisas vara osant. Då skulle alla mätningar och experiment istället handla om sannolikheter, konfidensgrader och statistik.

Således kan vi säga att vi inte vet om ren slumpmässighet existerar eller inte, men enligt konsensus innehåller kvantmekaniken rent slumpmässiga faktorer. Högst sannolikt kommer vi aldrig att kunna bevisa eller motbevisa den rena slumpmässighetens existens och således måste vi lämna frågan åt filosoferna att grubbla över. Oberoende av vilket svaret är, kommer vi fortsättningsvis att ha praktisk nytta av den teoretiska definitionen av slumpmässighet. Exempel på slumpmässighetens praktiska nytta behandlas i nästa kapitel.

1.2 Var behöver man slumpmässighet?

Händelser som är alldeles för komplicerade att förutspå brukar också kallas slumpmässiga fastän de inte är det. Ett tärningskast kan anses vara slumpmässigt, men om man vet allting om utgångstillstånden går det att beräkna vad tärningen kommer att visa när den landar. När man skapar en väderprognos använder man sig av flera olika simulationer. Dessa simulationer är förenklade versioner av verkligheten och de använder sig av slumpmässiga starttillstånd för att skapa ett antal möjliga värden. Dessa värden ger då en bra inblick i vad som möjligtvis kommer att hända i verkligheten. Ifall man har gjort hundra simulationer av den kommande temperaturen i Åbo, kan man använda sig av det högsta och minsta värdet (extremvärden) av dessa simulationer som ett utfallsrum samt använda sig av medelvärdet som den mest sannolika temperaturen.

Slumpmässighet kan också vara nyttigt när man söker efter fel i ett program. Beroende på komplexiteten av ett program kan det hända att det enda sättet att vara säker på att programmet är felfritt är att gå igenom alla olika kombinationer av input. Ofta är det omöjligt p.g.a. att antalet kombinationer kan vara astronomiskt stor. Redan ett program som tar in två 32-bitars heltal kommer att vara tvungen att gå igenom 264 olika kombinationer (alltså behöver exekvera programmet 1,85 \* 1019 gånger). Det blir mycket komplext om man försöker gå igenom alla kombinationer av input i ett program som får input av både musen och tangentbordet. Flera program använder sig också av tidpunkterna då inputen sker, vilket leder till en avsevärt större mängd kombinationer. Då är det praktiskt taget omöjligt att bevisa att programmet är felfritt. Det är även omöjligt att bestämma vilka alla kombinationer som existerar. Då kan en eller flera botar (datorprogram menade för att utföra uppgifter automatiskt) som använder sig av en indeterministisk AI vara ett mycket bra alternativ. Botens beslut fattas av en algoritm som också använder sig av en psg för att göra slumpmässiga men ändå lämpliga beslut. Dessa botar kommer inte att kunna gå igenom alla kombinationer av val, men ju mer de testar programmet desto säkrare kan man vara att programmet inte har fel.

Hur kan slumpmässig deltestning vara bättre än sekventiell deltestning? Om man har en lista av längden 1060 som innehåller endast nollor och ettor och varje plats på listan representerar en kombination av input i ett datorspel. Nollorna betyder att spelet kraschar, och ettorna betyder att spelet inte gör det. Eftersom det inte går att spara en så stor lista, är listan endast en abstrakt representation av kombinationerna. Det är omöjligt med dagens hårdvara att gå igenom hela listan, alltså kan man inte vara säker på att listan saknar nollor. Däremot kan man gå igenom t.ex. de första 109 kombinationerna för att få en viss sorts konfidens på att spelet inte kommer att krascha. Fastän de första 109 kombinationerna i listan saknade nollor, kan det ändå hända att spelet kraschar 10 % av tiden, vilket är mycket möjligt eftersom vi gått igenom listan sekventiellt och därför hittills testat endast mycket likadana kombinationer.

Oberoende av hurdant mönster vi använder för att gå igenom en del av kombinationerna, kommer samma problem ändå att existera. Ju mer komplicerat mönstret är, desto mindre är chansen att vi råkar ut för samma problem. Om vi går igenom 109 slumpmässigt valda kombinationer har vi lyckats minimera problemet eftersom ett mönster inte finns (för det är mycket osannolikt kombinationerna som valdes inte råkar ut för några kraschar, fastän kraschar kunde vara mycket vanliga. Kraschen följer ju alltid ett mönster i den abstrakta listan, vilket ren slumpmässighet inte gör).

Slumpmässighet används i spelautomater, t.ex. i alla kort- och tärningsspel. I datorspel kan slumpmässighet användas för att göra AI mindre deterministiska och lite mera lik människor. Icke spelbara figurer (eng. NPC) i en virtuell stad måste gå från en plats till en annan. För att skapa en mera trovärdig virtuell stadsmiljö skall icke spelbara figurer röra sig också trovärdigt och då kan man använda sig av slumpmässighet för att skapa figurernas promenadmönster. Filmer som använder datorgenererade bilder (eng. CGI) samt datorspel kan använda sig av slumpmässighet för att t.ex. göra texturer mera trovärdiga och realistiska.

Slumpmässighet används också i flera vetenskapsgrenar: kaosteori, kryptografi, spelteori, informationsteori och kvantmekanik för att bara nämna några. Man kan säga att det moderna samhället behöver metoder för att skapa slumpmässiga tal för att fungera.

1.3 Olika typer av slumpmässighet

Vi har hittills endast talat om slumpmässighet som en helhet, utan att gå in på hurdana typer av slumpmässighet det finns. Slumpmässighet kan delas i två subklasser, slumpmässighet med- eller utan sannolikhetsmått. Slumpmässighet med sannolikhetsmått innehåller all slumpmässighet som beter sig som matematiskt sätt ideal slumpmässighet (slumpmässighet som saknar mönster). All annan slumpmässighet hör till slumpmässighet utan sannolikhetsmått. Eftersom vi inte ens vet om ideal slumpmässighet existerar används dessa definitioner som riktlinjer och målet är att använda en slumptalsgenerator som är så lik den ideala slumpmässigheten som möjligt. En människa däremot är ett bra exempel på en slumptalsgenerator som skapar slumpmässighet utan sannolikhetsmått.

Varje slumpmässighet som har ett sannolikhetsmått har också en specifik sannolikhetsfördelning. Man använder sig av sannolikhetsfördelningar för att ytterligare indela slumpmässighet. Sannolikhetsfördelningar är indelade i diskreta- och kontinuerliga fördelningar. I diskreta fördelningar är antalet element i utfallsrummet (utfallsrummets kardinaltal) ändligt när däremot kontinuerliga fördelningar har uppräkneligt eller överuppräkneligt oändligt med element. Ett tärningskast har en diskret sannolikhetsfördelning eftersom utfallsrummets kardinaltal är 6. Tiden det tar för en radioaktiv atom för att sönderfalla har kontinuerlig sannolikhetsfördelning eftersom utfallsrummet är oändligt (och i detta fall överuppräkneligt oändlig). I följande stycken ges några specifika exempel på sannolikhetsfördelningar.

Den vanligaste sannolikhetsfördelningen är den likformiga sannolikhetsfördelningen. Likformiga sannolikhetsfördelningen menar att sannolikheten för varje enskilt värde i utfallsrummet är samma. En ideal tärning har en diskret likformig sannolikhetsfördelning eftersom man vill att chansen att tärningen landar på varje sida är den samma samt att antalet sidor är ändlig. I räknemaskiner finns det ofta en funktion som skapar ett slumpmässigt reellt tal mellan 0 och 1. Eftersom antalet reella tal mellan 0 och 1 är överuppräkneligt oändlig kan man säga att funktionen har en kontinuerlig likformig sannolikhetsfördelning, fastän kardinaltalet på funktionens utfallsrum är fortfarande ändligt. Det går inte att skapa en ideal kontinuerlig sannolikhetsfördelning för då skulle man kräva en oändlig precision på den processen som skapar fördelningen. Föreställ dig en räknemaskin som skulle klara av att skapa slumptal som följer en kontinuerlig likformig sannolikhetsfördelning mellan 0 och 1. Antalet decimaler som krävs för att representera ett sådant tal skulle ju vara oändlig.

En annan vanlig sannolikhetsfördelning är binomialfördelningen som är en diskret sannolikhetsfördelning. När man slingar en slant hundra gånger, kommer antalet kronor man får att följa binomialfördelningen. När man skapar en binomialfördelning behöver man veta antalet gånger något skall göras samt chansen att det som skall göras också lyckas. I exemplet som just nämndes skulle antalet gånger som något skall göras vara hundra och sannolikheten att det som skall göras lyckas 50 %. I den diskreta likformiga sannolikhetsfördelningen krävs det endast utfallsrummets kardinaltal för att skapa slumptalen (till exempel 6 för att det skulle fungera som en vanlig tärning).

När antalet gånger som något skall göras blir mycket stort, kan man använda sig av den normala fördelningen som binomialfördelningen konvergerar till. Observera att normalfördelningen är en kontinuerlig fördelning. För att skapa en normalfördelning behöver man veta vad väntevärdet och standardavvikelsen är. Om man vill approximera en binomialfördelning med hjälp av normalfördelningen kan man sätta väntevärdet som *N*\**P* och standardavvikelsen som

(*N*\**P*\*(1- *P*))0,5 där *N* är antalet gånger man gör någonting och *P* är chansen att det man gör lyckas. Ju större *N* är, desto bättre blir approximationen.

Det finns många saker i naturen som följer normalfördelningen, som t.ex. det tidigare nämnda radioaktiva atomens sönderfall. Det intressanta med normalfördelningen är att eftersom binomialfördelningen konvergerar till den är utfallsrummets minimum och maximum oändliga. En slumptalsgenerator som skapar normalfördelade tal kan alltså ge vilket som helst reellt tal. Sannolikheten att få tal längre ifrån väntevärdet minskar exponentiellt, men det existerar inga gränser för hur stort eller hur litet tal man kan få (jämför med den kontinuerliga likformiga sannolikhetsfördelningen där det finns ett bestämt största och minsta tal som kan genereras).

1.4 Hårdvaru- och mjukvarumetoder

Vi har definierat slumpmässighet, gått igenom exempel på olika typer av slumpmässighet och givit exempel på dess användbarhet, sedan ställer vi frågan om hur slumpmässighet skapas. Det finns två olika metoder för att skapa slumptal, hårdvaru- och mjukvarumetoder. Dessa metoder är mycket olika och används därför för olika ändamål.

Hårdvarumetoderna är bättre för att skapa ideal slumpmässighet för att de använder sig inte av deterministiska algoritmer. Hårdvarumetoder får sitt namn från att deras implementation kräver alltid någon sorts sensor. Sensorerna använder sig av yttre störningar för att generera slumptal, såsom sönderfall och fluktuationer i elektriska laddningar. Fastän hårdvarugenererade slumptal saknar ett mönster samt en period, är de problematiska gällande deras sannolikhetsmått. Det finns ingen metod för att få perfekt likformiga slumptal från en sensor utan det kommer alltid att finnas någon sorts systematisk avvikelse. Det svåra i hårdvarumetoder är alltså processen att använda sig av den mycket ideala slumpmässigheten för att skapa slumpmässighet med ett sannolikhetsmått (oftast likformig sannolikhet). Hårdvarumetoderna är oftast också långsamma, speciellt när man måste göra vidare beräkningar för att få slumptal utan vinklingar. Hårdvarumetoderna kallas för slumptalsgeneratorer. På grund av den nästan rena slumpmässigheten används sg främst för uppgifter där det är mycket viktigt att ingen kan dechiffrera talen och på så sätt gissa sig till vilka tal som kommer att genereras. Hasardspel, kryptering samt slumpmässiga urval är exempel på situationer då det lönar sig att använda sg.

Mjukvarumetoder använder sig av funktioner för att generera talföljder som imiterar ren slumpmässighet. Eftersom en funktion är deterministisk, behöver mjukvarumetoderna ett frö, något som ofta skapas med en slumptalsgenerator, för att inte alltid ge samma följd av ”slumptal”. Noterbart är att eftersom mjukvarumetoderna är deterministiska kan man inte säga att talen som genereras är slumptal. Istället används benämningen pseudoslumpmässighet. Mjukvarumetoderna kallas därför för pseudoslumptalsgeneratorer. Psg:n är mycket snabba på att skapa tal jämfört med sg:n och är därför lämpliga för uppgifter där ett stort antal slumptal är ett av kraven. Eftersom psg:n är avsedda för att imitera ren slumpmässighet kan de ibland klara av slumpmässighetstester bättre än sg:n [2]. Det här stämmer på grund av att sg:ns råa data kan innehålla något slags systematisk avvikelse, vilket psg:n är designade att inte vara. I kapitel 2 går vi igenom ett par psg-algoritmer samt diskuterar frågan om huruvida det är möjligt att skapa en psg som saknar period.

1.5 Slumpmässighetstester

Slumpmässighetstester är nyttiga för att mäta kvalitén på en slumptalsgenerators genererade tal. Ingen slumpmässighetstest är tillräcklig för att garantera att en slumptalsgenerator skapar rent slumpmässiga tal. Som det nämndes tidigare i kapitel 1.1, är det omöjligt att bevisa slumpmässigheten av en följd och därmed också slumpmässigheten av en slumptalsgenerator. Slumpmässighetstester är istället menade för att motbevisa slumpmässigheten av slumptalsgeneratorer. En enskild slumpmässighetstest räcker inte, utan man brukar använda flera slumpmässighetstester för att försöka hitta något mönster i genererade slumptalen. Slumpmässighetstester brukar oftast vara avsedda för likformigt fördelad slumpmässighet eftersom en likformig fördelning är lätt att skapa, enkel att göra statistiska beräkningar med och mycket ofta använd. Testerna brukar oftast använda sig av bit-representationen av slumptal. Då får man en följd av ettor och nollor av vilka man relativt enkelt kan göra tester på.

Då man använder ett slumpmässighetstest på en slumptalsgenerator finns det totalt fyra stycken olika utfall. Det kan finnas situationer när slumpmässighetstest ger fel resultat, eftersom de inte är absolut säkra tester. I två av de fyra stycken utfallen har slumpmässighetstesten avgjort rätt, de andra två fallen har slumpmässighetstestet gjort fel avgörande. Då ett slumpmässighetstest förkastar slumptalsgeneratorn fastän slumptalsgeneratorn är slumpmässig kallas för typ-1-fel. Om ett slumpmässighetstest godkänner en slumptalsgenerator fastän slumptalsgeneratorn inte är slumpmässig är det ett typ-2-fel.

Enkla slumpmässighetstest är viktiga fastän de kan kringgås mycket enkelt av en människa. Några exempel på enkla slumpmässighetstester är frekvenstestet(1), frekvenstestet inom ett block(2) och bitväxlingstestet(3).

Frekvenstestet handlar om att välja ett stickprov och sedan beräkna hur många av bitarna är ettor. Med avancerade statistiska beräkningar går det att skapa ett värde *p* som är mellan ett och noll. Så länge som p är större än ett visst värde *k*, kan man säga att slumptalsgeneratorn klarade av frekvenstestet. Värdet *k* är konfidensen att slumpmässighetstesten inte gjort ett typ-1-fel. Då *k* är noll, kommer ingen följd att förkastas, alltså är chansen för typ-1-fel också noll. Vanligtvis brukar *k* vara 0,05, 0,01 eller 0,001 beroende på hur säker man vill vara att man inte har förkastat en slumptalsgenerator i onödan. Frekvenstestet inom ett block handlar om att dela stickprovet i mindre delar och sedan beräkna *p* från resultaten. Observera att dessa båda tester är mycket enkla att kringgå. Frekvenstestet förkastar inte en följd av *N* stycken ettor följda av *N* stycken nollor. Frekvenstestet inom ett block förkastar inte en följd av alternerande ettor och nollor (101010...). I bitväxlingstestet ser man på hur många gånger ettor blir nollor och vise versa. Bitväxlingstestet förkastar inte en följd av alternerande ettor och nollor av längden två (110011001100…). Den sist nämnda följden klarar av alla de tre tester som nämndes(1-3).

Ju mera olika slumpmässighetstester man använder desto bättre resultat får man. Samtidigt kommer antalet falskt positiva resultat att öka. Om man testar med två oberoende test med *k* = 0,05 kommer sannolikheten för åtminstone ett typ-1-fel att vara 9,75 % ifall man testar en slumpmässig följd. Därför är det viktigt att göra fler tester ifall något slumpmässighetstest förkastar stickprovet, eftersom resultatet kunde ha varit fel.

Det finns ett antal slumpmässighetstester som hör till en samling som kallas för *diehard*-testerna [3]. Dessa tester är designade för att vara svåra för psg:n att klara av. Ett av testen är *Birthday spacings*-testet där man använder slumptal för att välja punkter i ett stort intervall och sedan räknar ut *p* för testet med hjälp av punkternas mellanrum. Testerna är mycket komplicerade och därför är det mycket svårt att designa en algoritm som avsiktligt skulle generera en följd som klarar av alla *diehard*-testen.

2. Genereringsalgoritmer

Som det redan nämndes är det omöjligt att skapa slumptal med en algoritm, men istället går det att skapa tal som är fördelade som om de var slumpmässiga, dessa tal kallas för pseudoslumptal. I detta kapitel går vi igenom några kända psg:n, hur de fungerar och deras för- och nackdelar. Märk väl att när vi diskuterar genereringsalgoritmer talar vi endast om sådana vars uppgift är att genera slumptal. De flesta genereringsalgoritmer som behandlas använder sig av heltalsdivision och moduloräkning. Tecknet ”/” representerar heltalsdivision och ”%” representerar moduloräkning (15 % 8 = 7 eftersom resten av 15/8 är 7).

2.1 Allmänt om genereringsalgoritmer

En genereringsalgoritm vars uppgift är att skapa slumptal måste kunna skapa olika tal oberoende av när algoritmen används. Vi vill alltså att samma algoritm skall ge olika tal när man exekverar den flera gånger. Eftersom en algoritm alltid är deterministiskt måste den använda sig av någon yttre faktor, alltså ett frö. Fröet måste vara slumpmässigt för att man skall kunna få bra data från genereringsalgoritmen. Ett sätt att skapa fröet är att använda sig av datorns egen klocka. Eftersom klockan är mycket precis (nutida Intels processorer har en klocka som uppdateras 100 miljoner gånger per sekund) är det mer eller mindre omöjligt att exekvera program med samma noggrannhet, och därför kan man säga att några av klockans minst signifikanta siffror (de sista i följden) är mer eller mindre slumpmässiga.

Fröet är algoritmens tillstånd. Ifall algoritmen på något sätt har fått samma frö som tidigare kommer algoritmen också att skapa exakt samma slumptal eftersom ursprungstillståndet varit det samma. Varje gång genereringsalgoritmen skapar ett slumptal förändras tillståndet. Det här sker oftast så, att slumptalet som algoritmen genererat blir algoritmens nya tillstånd. Därför bestämmer antalet bitar som tillståndet använder direkt den maximala periodlängden på algoritmens pseudoslumpmässiga talföljd. Om algoritmens tillstånd använder sig av 32 bitar måste den börja upprepa sig senast efter att den skapat 232 slumptal. Algoritmen behöver inte börja upprepa sig helt från början, utan det kan också hända att det skapas en kortare följd som sedan upprepas i all oändlighet.

Man kan ställa sig frågan huruvida en genereringsalgoritm alltid måste ha periodicitet. I själva verket finns det sätt att skapa genereringsalgoritmer som genererar tal utan periodicitet. Ett enkelt sätt är att approximera irrationella tal. Irrationella tal är tal vars decimalföljd saknar periodicitet. Om man lätt kan räkna ut decimaltal för något irrationellt tal, har man skapat en algoritm som genererar tal utan periodicitet. Då kommer algoritmens tillstånd att vara antalet decimaltal man hittills genererat. Ifall man ytterligare vill att talen skall vara slumpmässiga blir det svårt. Definitionsmässigt är de genererade decimaltalen inte slumpmässiga eftersom de följer en viss regel; det irrationella talets decimalföljd. Alltså måste man först med hjälp av slumpmässighetstester se till att decimalföljden är tillräckligt slumpmässig för ändamålet.

Det existerar tal vars decimalföljd på vilken som helst talbas innehåller alla ändliga talföljder, och då dessa talföljder vidare har samma sannolikhet att förekomma i en delföljd av decimalföljden som en följd av idealt likformigt slumpmässiga tal. Dessa tal kallas för normala tal. Man tror att t.ex. *e* och π är normala tal, men det har man inte ännu bevisat. Märk väl att normalitet av ett irrationellt tal inte betyder att talets decimaltal skulle vara slumpmässiga. Till exempel har champernownes konstant bevisats vara normal fastän den följer ett mycket enkelt mönster (champernownes konstant för det decimala talsystemet är 0,123456789101112131415...). Definitionsmässigt är detta omöjligt eftersom det alltid måste finnas en algoritm för att skapa dessa normala tal.

Att räkna ut det *n*: te decimaltalet av ett irrationellt tal är mycket jobbigt. Ju större *N* är desto längre tar det att räkna ut talet. Det krävs även mycket minne av en dator för att göra de stora räkneoperationerna. Alla genereringsalgoritmer som skapar slumptal med en periodicitet har en konstant komplexitet, samt behöver de endast spara tillståndet i minnet. Det tog 18,8 dagar för Ron Watkins att beräkna tio triljoner decimaler av kvadratroten ur två med en dator som använde sig av 141 gigabytes minne och 24 trådar (eng. threads) [4]. Datorn klarade alltså av att skapa ca 6,2 miljoner decimaltal per sekund, vilket motsvarar ca 320 000 stycken 64-bitars tal per sekund. Jämför det med att kunna skapa nästan 100 miljoner 64-bitars slumptal per sekund med endast en tråd och mycket litet krav på minnets storlek. Fastän en psg utan periodicitet har vissa fördelar kommer dessa att utklassas av nackdelarna och därför använder man sig endast av genereringsalgoritmer med periodicitet.

2.2 Kryptografiskt säkra pseudoslumptalsgeneratorer

Alla psg:n kan indelas i två skilda delar, de som är kryptografiskt säkra, och de som inte är det. Målet med en kryptografiskt säker psg är att det skall vara mycket svårt att knäcka den. De kan användas för att kryptera information som är inte menat för allmänheten. Kryptografiskt säkra psg:n är oftast långsamma samt mycket komplicerade. De är alltså inte lämpliga för t.ex. Monte Carlo-metoder.

Lägg märke till att en psg kan vara kryptografiskt säker fastän den inte klarar av slumpmässighetstester. Perfekt slumpmässighet är naturligtvis absolut kryptografiskt säker, men det betyder inte att en psg:s kryptografiska säkerhet mäts med hur nära perfekt slumpmässighet dess output är. En psg vid namn Mersenne Twister, som behandlas i kapitel 2.6, klarar av många slumpmässighetstester fastän det är mycket enkelt att knäcka den.

2.3 Middle-square-metoden

Middle-square-metoden är ett av de äldsta psg:n. Den skapades år 1949 av John von Neumann. Algoritmen tar först som input fröet, sedan kvadreras fröet och man väljer den mellersta delen av resultatet som både output och som det nya tillståndet. Om fröet är 42 kommer outputen och det nya tillståndet att bli 76 eftersom 42 kvadrerat blir 1764 och därnäst blir outputen och tillståndet 77 eftersom 76 kvadrerat blir 5776. Denna metod kan användas för hur stora tal som helst.

Middle-square-metoden har många problem med periodiciteten. Om man använder talet 60 som fröet kommer algoritmen alltid att fastna vid 60 eftersom 60 kvadrerat blir 3600. Samma gäller för 0, 10, och 50. Alla input kommer alltid att fastna på en mycket liten periodicitet jämfört med hur stora tal det handlar om. För 8-bitars tal krävs det endast 36 iterationer för att genereringsalgoritmen för alla 256 frön skall hamna i ett tillstånd där den redan varit. Talet 173 kommer att skapa 35 unika tal med hjälp av 8-bitars-versionen av algoritmen före den genererar ett tal den redan har genererat (inkluderande starttillståndet). Märk väl att talet 173 kommer att så att säga fastna på talet 0 ((02/16) % 256 = 0), alltså är talets slutliga periodlängd endast 1. För alla 8-bitars tal är den längsta periodlängden endast två och det får man med tillstånden 33 och 68. För ett 8-bitars tillstånd är den teoretiskt största periodlängden 256, alltså använder algoritmen mycket litet av den maximala periodlängden, åtminstone när det gäller 8-bitars frön. Om algoritmen använder sig av 16-bitars frön kommer algoritmen att hamna i ett tidigare tillstånd om senast 346 iterationer, vilket man får med fröet 61203. Största periodlängden för 16-bitars frön är 111, vilket man får med tillståndet 31473. Jämfört med 16-bitars maximum, en periodicitet på 65536, är algoritmen här också mycket slösaktig fastän en periodlängd på 111 är mycket bättre än en periodlängd på två.

Algoritmen skapar mycket dåliga periodlängder jämfört med antalet bitar som används. Finns det något som man ändå kan vinna genom att använda middle-square-metoden för att skapa pseudoslumptal? Jo, i själva verket är algoritmen mycket snabb. Det enda vi behöver räkna ut för att skapa nästa tillstånd då vårt nuvarande frö är *N* är (*N 2*/2*B*/2) % 2*B* där *B* står för antalet bitar. Algoritmen använder sig av endast en multiplikation, en division och en moduloräkning för att skapa nästa tillstånd. När vi använder oss av potenser av två går det att göra både divisionen och moduloräkningen till enkla bitvisa operationer. Divisionen kan bytas till högerskift eftersom vi inte bryr oss om vad resten blir då vi dividerar. Då kan man skriva algoritmen på java på följande sätt:

( *N* 2 >> (*B*/2) ) % 2*B*. Moduloräkningen kan bytas till en and-operation eftersom %(2*B*) är ekvivalent med &(2*B* – 1) (där & är den bitvisa AND-operationen). Nu använder algoritmen endast av en multiplikation och två bitvisa operationer. Notera att eftersom 0 ≤ *N* < 2*B* så kräver beräkningen av *N* i kvadrat dubbelt antal bitar, alltså krävs det 64-bitar för att använda sig av 32-bitars frön. Det finns sätt att kringgå detta, men de kommer också att göra algoritmen långsammare. Fastän algoritmen är mycket snabb, hjälper det inte eftersom den slösar både på periodlängder och bitar.

Även om middle-square-metoden är mycket dålig på att skapa slumptal, bestämde jag mig för att analysera dess 32-bitars version. Jag använde java för att skapa programmet. Grundidén med programmet var att spara all information om de tal man beräknat. På detta sätt behövde man inte räkna skilt iterationsmängden för alla 4,2 miljarder tal. Det fanns ett antal problem som behövde lösas, såsom hur man borde göra för att behandla periodiciteten rätt, det faktum att java inte stöder positiva heltal (eng. unsigned integers) och problemet med minnet då man försökte använda sig av en lista innehållande 4,2 miljarder tal. Det tog en halv timme att gå igenom alla 32-bitars tal och det totala antalet iterationer som krävdes ifall man skulle ha gått igenom talen utan en lista var 116 triljoner.

Resultaten av undersökningen var som förväntat. Största periodlängden var 5594 där ett av talen i perioden var 4195603965. Största antalet iterationer utan upprepning var 87177 (87178:e iterationen skulle ha gett ett tal som talföljden redan innehållit) som kan skapas med fröet 358236979. Oberoende om talen som middle-square-metoden skapar skulle klara av slumpmässighetstester, kan vi säga att middle-square-metoden bevisligen är en mycket dålig psg och endast klarar av kravet på hastighet om man utgår från nutida krav på psg:n.

2.4 Linjär kongruensgenerator

En linjär kongruensgenerator (eng. Linear congruential generator) är ett mycket enkelt och snabbt sätt att skapa pseudoslumptal. Jämfört med middle-square-metoden är linjära kongruensgeneratorer snabbare och deras periodlängd är mycket större. Vissa linjära kongruensgeneratorer kan till och med ha en periodlängd som omfattar alla värden inom bitlängden.

En linjär kongruensgenerator kräver först ett frö *N* som input, sedan räknar den ut (*a*\**N* + *b*) % *m*. Resultatet av beräkningen blir algoritmens output samt nya tillstånd. För att algoritmen skall vara en linjär kongruensgenerator skall *m* vara ett heltal som är större än noll, oftast 232 eller 264. Både *a* och *b* måste vara heltal och mindre än *m*. För att algoritmen skall fungera rätt skall *a* vara större än noll och *b* vara större än eller lika med noll. Det rekommenderas att *a* skall vara ett stort tal så att algoritmens output kommer att se mera slumpmässig ut. Ifall *a* och *b* är ett kommer algoritmen endast att ge tal som stegvis blir ett större än fröet. Observera att ifall a är ett och b är noll kommer algoritmen alltid att ge fröet som output. Notera också att algoritmens största möjliga periodlängd är *m*.

Man har kunnat bevisa att endast om följande villkor gäller [5], kommer algoritmen att ha en periodlängd på *m*: största gemensamma delaren för *b* och *m* är 1, (*a –* 1) är delbar med alla *m*:s primtalsfaktorer, ifall *m* är delbar med fyra är (*a –* 1) också delbar med fyra. För en godtycklig 2*n*, där *n* är ett heltal, är den enda primtalsfaktorn två. För alla potenser av två där exponenten är två eller större kommer talet att vara delbart med fyra. För alla potenser av två kommer den största gemensamma delaren med ett annat tal alltid att vara ett då det andra talet är udda. Så här kan vi göra en version av villkoren då m är en tvåpotens: *b* skall vara udda och (*m –* 1) skall vara delbar med både två och fyra, vilket kan förkortas till endast fyra. Om vi väljer att *m* är 216 så kan vi välja *a* att är 201 och *b* att är 16171. Då är vår linjära kongruensgenerator definierad som (201\**N* + 16171) % 216 och eftersom den uppfyller de tidigare nämnda kraven vet vi att dess periodlängd är 216.

Ett specialfall av linjära kongruensgeneratorer är den multiplikativa kongruensgeneratorn. Då är *b* noll och således får fröet inte vara noll och generatorns periodlängd kommer alltid att vara mindre än *m*. Fastän denna version är sämre på nästan alla sätt än en version där *b* är olika noll, har den fått en implementation i C++11 [6] eftersom algoritmen är så enkel att förstå.

Den linjära kongruensgeneratorn är lika snabb som middle-square-metoden och dess periodlängd mycket är bättre. Men den har fortfarande ett liknande problem som middle-square-metodens *N* 2 där man hamnar räkna ut en produkt som blir större än tillåtna antalet bitar. Kan en 64-bit dator använda sig av en 64-bitars implementation av linjära kongruensgeneratorn? Jo, det finns ett sätt där algoritmen inte överflödar, men det kommer att göra algoritmen en aning långsammare och den kommer inte att kunna generera exakt 64-bitars pseudoslumptal utan högst 63-bitars tal. Beräkningen av (*a*\**N* + *b*) % *m* skall alltså inte ha några delsteg där vi skulle räkna med tal större än *m*. Eftersom värden *a*, *b* och *m* är konstanta kan vi göra beräkningar på dem utan att göra algoritmen långsammare. På detta sätt inför vi två nya konstanter: *q* som är samma som *m*/*a* och *r* som är samma som *m* % *a*. För att algoritmen inte skall överflöda måste vi välja *m* och *a* på så sätt att *r* < *q* stämmer. Då går det att skriva (*a* \* *N*) % *m* som *a* \* (*N* % *q*) – *r* \* (*N* / *q*) + *m* \* *T* där *T* är 1 ifall

*a* \* (*N* % *q*) – *r* \* (*N* / *q*) blir negativt och 0 ifall samma blir positivt eller lika med noll. Vi kallar värdet av hela beräkningen för *d*. Observera att vi måste använda oss av heltal som kan bli både positiva och negativa (eng. signed integers) eftersom positiva heltal inte kan bli negativa och således kommer en *N*-bitars implementation att kunna skapa högst (*N –* 1)-bitar stora pseudoslumptal. Det enda som måste göras är att beräkna (*d* + *b*) % *m*. Både *d* och *b* är mindre än *m* samt större än eller lika med noll, alltså måste vi endast lägga märke till de fall där *d* + *b* blir större än eller lika med *m*. Det här kan göras enkelt genom att jämföra *m – b* med *d*. Ifall m *– b* är större än *d* skapas inga problem och vi kan beräkna det slutliga värdet med (*d* + *b*) % *m*, om *m – b* är lika med *d* blir det nya tillståndet 0 och om m *– b* är mindre än *d* beräknar vi tillståndet med (*d* – *m* + *b*) % *m* och på detta sätt undviks möjligheter till överflöd.

Samma metod kan inte användas i middle-square-algoritmen eftersom algoritmen kräver de mellersta bitarna. Däremot har jag undersökt en implementation för middle-square-algoritmen där man använder sig av högst 4\**B*/3 (där *B* är antalet bitar) istället för 2\**B* när man beräknar ett tillstånd. Fastän det fanns en liknande metod för middle-square-algoritmen, skulle den bli betydligt långsammare eftersom *r* och *q* måste beräknas på nytt för varje iteration. Lägg märke till att kravet att *r* skall vara större än *q* också kommer att orsaka ett problem.

Den linjära kongruensgeneratorn har stora problem gällande slumpmässighetstester. Ett bra exempel är den så kallade spektrala testen (eng. spectral test) som använder sig av en representation av slumptal i olika dimensioner för att hitta olika mönster. Dessa mönster är oftast ett antal ränder som är parallella med varandra. I nästa sida finns en bild på output av den linjära kongruensgeneratorn (vänstra bilden) som vi definierade tidigare samt en bild av output från javas psg (högra bilden). Algoritmen körs 215 gånger, hälften av periodlängden, och en punkt definieras på följande sätt: om *t* är algoritmens output kommer *x* att vara *t* / 256 och *y* att vara *t* % 256. På så sätt kommer varje punkt att användas exakt en gång då man går igenom hela periodlängden. Eftersom vi inte kan se några mönster i outputen om hela bilden var blå, har bilden endast 215 output och som frö användes 15677. Man ser tydligt att algoritmen skapar något sorts mönster jämfört med den högra bilden som innehåller tal av javas egna psg.



Linjära kongruensgeneratorer är alltså mycket snabba och de har mycket bra periodlängder men man kan lätt finna mönster på dem då man gör spektrala tester. Algoritmen är inte lämplig för varken kryptering eller för Monte Carlo-metoder, men den kan användas i datorspel där algoritmens hastighet är mycket viktig och algoritmens slumpmässighet inte behöver vara perfekt.

2.5 Blum Blum Shub

I Blum L., Blum M. och Shub M:s artikel *A Simple Unpredictable Pseudo-Random Number Generator* May 1986, sidorna 364-383 definieras psg:n Blum Blum Shub. Blum Blum Shub-algoritmen är mycket lik middle-square-algoritmen. Båda metoderna handlar om att kvadrera ett tillstånd men Blum Blum Shub har krav som middle-square-algoritmen inte kräver. Modulot *M* måste vara en produkt av två primtal, *p* och *q*, och båda primtalens rest då man delar med fyra måste vara tre (*p* % 4 = *q* % 4 = 3) för att algoritmen skall fungera på rätt sätt. För att få den längsta möjliga periodlängden skall gcd(φ (*p* – 1), φ (*q* – 1)) vara så liten som möjligt, där gcd är talens största gemensamma delare och φ är Eulers fi-funktion. Vi kallar denna beräkning för B(*M*), där

*M* = *p*\**q* och *p* och *q* fyller de tidigare nämnda kraven. Om fröet är *X* skapas algoritmens nästa tillstånd med att beräkna *X* 2 % *M*. Lägg märke till att algoritmens output inte kommer att vara samma som tillståndet eftersom man då enkelt skulle hitta delar där *X* 2 inte blir större än *M*. Om *M* är 77 och *X* är 2, kommer tillståndet att bli 4 och därnäst 16 och först vid 16\*16 kommer vi att behöva använda moduloräkningen. Detta är betydligt lättare att märka då *M* är stort. Istället kommer tillståndets minst signifikanta bit, eller ett antal minst signifikanta bitar att vara algoritmens output. Algoritmen använder alltså en mycket liten del av det den beräknat och är därför mycket långsam.

Periodlängden på algoritmen är dålig fastän man har valt *p* och *q* på så sätt att B(*M*) blir 2, vilket är det minsta möjliga värdet på B(*M*) (ifall varken *p* eller *q* är 2). Om *p* är 7 och *q* är 11 kommer *M* att vara 77 och den största periodlängden blir endast 4 fastän *B*(*M*) i detta fall är 2. Om vi gör *p* och *q* större så att *p* är 719 och *q* är 727 kommer B(*M*) fortfarande att vara 2 och vi får som största periodlängd 19690. Jämfört med storleken på *M* (vilket är 522713) använder algoritmen bara 3,8 % av den maximala periodlängden. Då B(*M*) är stor blir den största periodlängden snabbt mycket liten. Om *p* är 7151 och *q* är 7283 kommer B(*M*) att bli 300 och största periodlängden blir endast 60. Om vi byter *q* till 7247 är B(*M*) än en gång 2 och största periodlängden blir större än eller lika med 108660, vilket är endast 0,2 % av den största möjliga periodlängden för *M* (vilket är 51823297).

Finns det någon orsak att använda Blum Blum Shub när den är både mycket långsam och har relativt små periodlängder? Man har kunnat bevisa att knäckande av algoritmen är lika svårt som att lösa kvadratiska resten av ett tal (försöket att hitta något heltal *x* mindre än *m*, då man valt *a* så att

*x*2 % *m* = *a*), vilket i sin tur tros vara lika svårt som att faktorisera primtal (vars svårighet som t.ex. RSA-krypteringen bygger på). Man har kunnat bevisa att för ett modulo *M* kan man välja

O(log log *M*), där O är ordot, minst signifikanta bitar utan att mista algoritmens svårighet [7]. Lägg märke till att O(1) = O(1000) och på samma sätt är O(log log *M*) samma somO(1000 \*log log *M*) alltså kan man inte använda sig av beviset på något konkret sätt. Fastän algoritmens svårighet verkar vara den samma som faktoriseringen av primtal, går det inte att säga att algoritmen är lika svår som att faktorisera *M*. Reduceringen av algoritmen är mycket ordentlig och det går därför inte att säga något om säkerheten av en Blum Blum Shub-algoritm som använder sig av ett *M* som är av storleken 3000-bitar [8]. Då det krävs flera tusen bitars *M* för att vara säker på att det är mycket svårt att knäcka algoritmen med en Turingmaskin, kommer algoritmen att generera pseudoslumptal mycket långsamt.

Blum Blum Shub-algoritmen är långsam, den har även i bästa fall en relativt dålig periodlängd och dess svårighet konkret sett betyder mycket litet eftersom det krävs mycket stora tal för att få algoritmen att bli bevisligen svår att knäcka. Således har algoritmen ingen praktisk nytta.

2.6 Mersenne Twister

I Matsumoto M. och Nishimura T:s artikel *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator* 1998, sidorna 1-25 beskrivs Mersenne Twistern. Mersenne Twistern har fått sitt namn från det att den använder ett mersenneprimtal. Den mest kända versionen av algoritmen använder mersennetalet 219937 –1. Algoritmen har den högsta möjliga periodlängden (en periodlängd på 219937 –1) men den behöver också lite mera än 19937 bitar för att spara tillståndet. Algoritmen är mycket svårare att beskriva jämfört med de tidigare psg:n som avhandlingen behandlat. Algoritmen handlar om att göra ett antal bitvisa operationer på en lista som innehåller 624 32-bitars positiva heltal. När programmet exekveras används fröet för att beräkna alla noder på listan. *N*: te elementet i listan beräknas med hjälp av ett antal bitvisa operationer på föregående element gånger ett specifikt värde. Första elementet i listan kan inte använda sig av något tidigare värde utan den använder sig av fröet. När listan är initierad kan man börja skapa pseudoslumptal. Algoritmen går sekventiellt igenom varje nod på listan och gör ett antal bitvisa operationer för att skapa dess output. Algoritmen kommer inte att ändra på elementens värden, utan går i stället till det nästa elementet i listan då den skall skapa nästa pseudoslumptal. Då algoritmen har nått den sista noden på listan gör den en liknande beräkning som när den skapade listan och ändrar värdet på alla element i listan. När denna beräkning är klar kommer algoritmen att fortsätta generera pseudoslumptal sekventiellt genom att börja från den första noden i listan. På detta sätt är algoritmen mycket snabb förutom vid varje 624:e användning, då den måste räkna om alla 624 element i listan för att kunna skapa fler pseudoslumptal. Denna beräkning kallas för twist.

Jämfört med tidigare psg:n som har behandlats, är Mersenne Twisterns periodlängd massiv. Den klarar därför av att ge exakta samma output två gånger i ett sträck, vilket t.ex. linjära kongruensgeneratorer inte klarar av eftersom deras tillstånd är samma som deras output. Så länge som tillståndet är den samma kommer outputen att vara den samma. I Mersenne Twisterns fall är det möjligt få exakt samma värde två eller flera gånger, eftersom tillståndet är mycket större än outputen. Om Mersenne Twisterns output skulle vara ett 19937-bitars heltal skulle algoritmen bevisligen inte mera kunna skapa två identiska på varandra följande output.

Algoritmen klarar av många slumpmässighetstester (även diehard-tester) och är därför lämplig för att användas i Monte Carlo-metoder. Däremot är algoritmen inte lämplig för kryptering, eftersom man efter 624 påföljande pseudoslumptal kan beräkna algoritmens exakta tillstånd. Hur kan det här vara möjligt? När algoritmen använder sig av listans noder för att skapa dess output gör den följande operationer.

*x* = *lista*[*nod*]

*x* = *x* ^ (*x* >> 11)

*x* = *x* ^ ((*x* << 7) & 2636928640)

*x* = *x* ^ ((*x* << 15) & 4022730752)

*x* = *x* ^ (*x* >> 18)

*return x*

Här är ^ bitvisa XOR-operationen, & bitvisa AND-operationen, << är vänsterskift och >> högerskift. Vi kallar beräkningen för funktion *F*(*x*). Dessa beräkningar på *x* ser arbiträra ut, men de är valda på ett sådant sätt att funktionen blir bijektiv. För två olika input kommer funktionen aldrig att ge samma output och antalet olika input är den samma som antalet olika output. Det finns några input som kommer att ge som output inputen (dessa värden är 0, 2, 16, 18, 64, 66, 80 och 82). Eftersom funktion är bijektiv är det möjligt att finna inversen av funktionen. Då man gått igenom 624 påföljande pseudoslumptal vet man också deras ursprungliga värden tack vare inversen av funktionen. Med hjälp av dessa tal går det nu att beräkna exakta värden på listan efter twisten. Om vi vet den senare hälften av värden på listan före twisten och tidigare hälften av värden på listan efter twisten kan man använda sig av värden före twisten för att beräkna de återstående värden i listan efter twisten. När hela listan efter en twist-operation är beräknad, har vi löst tillståndet på Mersenne Twistern. Om man vet vilken Mersenne Twister-algoritm som används kan man oberoende av vilket frö som används med endast 624 påföljande generade pseudoslumptal räkna ut exakta tillståndet.

Fastän Mersenne Twistern klarar av många slumpmässighetstester är den inte lämplig för kryptering, eftersom det är relativt lätt att beräkna tillståndet. Det låter som en paradox att något kan vara mycket slumpmässigt men lätt att knäcka. Mersenne Twistern är lätt att knäcka endast då man vet hur algoritmen ser ut men; så länge som man inte vet hur algoritmen ser ut kommer det att vara mycket svårare att knäcka den. För att en psg skall kunna användas för kryptering skall det inte finnas någon enkel metod för att knäcka algoritmens tillstånd fastän man vet exakt hur algoritmen ser ut. Tidigare nämnda Blum Blum Shub-metoden har bevisats uppfylla kravet men som det förklarades, är beviset av mycket liten praktisk nytta. Middle-square-metoden och linjära kongruensgeneratorn har båda samma problem som Mersenne Twistern; genast när man vet hur algoritmen ser ut kan man med senaste genererade pseudoslumptalet förutsäga alla fortsatta tal.

2.7 Yarrow och Fortuna

I Kelsey J, Schneier B och Ferguson N:s artikel *Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator* 1999, sidorna 1 till 14 förklaras hur Yarrow-algoritmen fungerar. Algoritmen är avsedd för att skapa kryptografiskt säkra pseudoslumptal. Jämfört med alla andra psg:n som avhandlingen behandlat är Yarrow mycket avancerad. Det finns många olika versioner av Yarrow-algoritmer. Algoritmen använder både en kryptografisk envägs hash-funktion och ett blockchiffer för att generera kryptografiskt säkra pseudoslumptal. En hash-funktion tar som input data av vad som helst för längd som den sedan med hjälp av bland annat bitvisa operationer gör till output av en viss storlek. Däremot tar ett blockchiffer som input data av storleken *N* bitar som den sedan på liknande sätt som hash-funktionen gör till en output av storleken *N*. Skillnaden mellan hash-funktionen och blockchiffret är att blockchiffret måste vara bijektivt. Hash-funktionen får inte vara bijektiv och det är omöjligt för den att ens vara det då inputen är av ett större antal bitar än vad outputen skall vara.

I Yarrow-160-algoritmen används SHA-1 som hash-funktionen och Tripple DES som blockchiffret. Dess output är av storleken 64-bitar. Algoritmen fungerar mycket olikt jämfört med tidigare nämnda psg:n. Algoritmen kan delas i fyra beståndsdelar: En entropiackumulator (eng. entropy accumulator), *reseed mechanism* (en mekanism för att byta fröet), genereringsmekanism (eng. generation mechanism) och *reseed control* (en kontroll för att byta fröet).

Eftersom algoritmen är designad för att vara kryptografiskt säker har den inget starttillstånd som användaren kan ge. Därför kräver den någon sorts slumptalsgenerator som kan skapa riktiga slumptal. Det finns alltså inget sätt att exekvera algoritmen så att den skulle generera exakt samma räcka av pseudoslumptal som tidigare. Således saknar algoritmen också periodicitet. Algoritmen har en entropiackumulator vars uppgift är att samla riktiga slumptal och sedan sätta dem i två skilda *pools* (en mängd av resurser som är färdiga att användas). Slumptalen som entropiackumulatorn sparar används i algoritmen som nycklar. Genereringsmekanismen, som består av hash-funktionen och blockchiffret, använder nycklar för att generera pseudoslumptal. En nyckel används ett obestämt antal gånger av genereringsmekanismen. Först då *reseed control* anser att det behövs en ny nyckel, kommer *reseed mechanism* att välja en ny nyckel åt genereringsmekanismen från någondera av de två *pools*. Detta komplicerade system gör det mycket svårt att knäcka algoritmen.

Numera är Yarrow-160 inte mera lämplig för att skapa kryptografiskt säkra pseudoslumptal eftersom SHA-1-algoritmen bevisligen kan knäckas [9]. Kryptografiskt säkra psg:n Fortuna, skapad av Schneier och Ferguson, är en uppdaterad version av Yarrow. Fortuna använder SHA-256-algoritmen så det är mycket svårare att knäcka den jämfört med Yarrow-algoritmen.

3. Referenser

[1] Rukhin A. et. al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.* 1-4 April 2010

[2] Rukhin A. et. al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.* 1-2 April 2010

[3] Marsaglia G. Et al. *Some difficult-to-pass tests of randomness*, 2002

[4] http://www.numberworld.org/digits/Sqrt(2)/

[5] Donald E. Knuth, Art of Computer Programming, Volume 2: Seminumerical Algorithms

[6] http://www.cplusplus.com/reference/random/minstd\_rand0/

[7] Alexi W. et al. *RSA and Rabin functions: certain parts are as hard as the whole.* April 1988

[8] Koblitz N. et al. *Another look at ”Provable security”. II.* July 2006

[9] Stevens M. et al. *The first collision for full SHA-1*