Kapitel 6

Slumptal och Simulering

6.1 Slump och Slumptal

Det finns många tillämpningar i datalogin, ska vi se i detta kapitel, där slumpen är till stor hjälp eller är nödvändig.

Utan att gå närmare in på en *filosofisk definition av slump* är det ganska överraskande att det går att framställa tal med hjälp av en dator som tycks vara slumpmässiga. Datorn är ju känd för att göra exakt vad den är satt att göra och har dessutom förmågan att upprepa sitt arbete gång på gång med samma resultat.

För att du ska få en första inblick i hur datorn genererar slumptal ska vi titta närmare på en helt ovetenskaplig metod.

Vi utgår från vad klockan råkar vara just nu – $18 : 13 : 40$ – och bildar av detta ett tal $u_0 = 0.18134$ detta tal sätter vi in i formeln

$$u_i = \text{decimaldelen av}[(\pi + u_{i-1})^5]$$

När vi beräknat uttrycket mellan hakparenteserna drar vi bort heltalet och får ett slumptal u_1. Använder vi $\pi = 3.1415927$ får vi $u_1 = 0.14243$ som första slumptal. Detta tal går sedan, avrundat, in i formeln för att bestämma nästa, som blir $u_2 = 0.97076$. Denna serie tycks kunna bli hur lång som helst (åtminstone om vi tar med många decimaler) och det verkar svårt att förutsäga vad nästa tal i serien ska bli. Kan vi kalla följden slumpmässig?

Eftersom datorer inte, utan påbyggd hårdvara, kan generera slump i egentlig mening (vad nu detta är), så kallar man dessa tal för *pseudo-slumptal*.

Här nedan ska vi nu lära oss att generera slumptal, som är mer tillförlitliga än de vi får från ”generatorn” ovan. Det stora felet med slumptalsgeneratorn ovan är, att vi inte vet efter hur lång tid, vi får tillbaka ett slumptal vi redan haft. Får vi det så har vi hamnat i en loop – eller hur? Mardrömmen är förstås om vi startar med ett slumptal som returnerar sig själv direkt. Då kan man knappast tala om slump!

UPPGIFT 6.1
Finn ett startvärde. Ta reda på ett startvärde $0 \leq u_0 < 1$ givet med fyra decimaler, som ger ett första slumptal u_1, som är identiskt med u_0 efter avrundning till fyra decimaler. Som värde på π använder vi som ovan 3.1415927. Det finns precis ett sådan tal!

6.1.1 Så fungerar de inbyggda slumptalen

Det finns två funktioner som är direkt förknippade med slumptal

- `rand` Denna funktion returnerar ett slumpmässigt heltal i intervallet [0, RAND_MAX]. `RAND_MAX` återfinns i `stdlib.h` och har ofta värdet 32767.
- `srand` För att slumptalsserierna inte ska upprepa sig krävs olika startvärden. Med hjälp av `srand` kan man ge just `rand` sådana startvärden. `srand(1234)` säkerställer att man får en annan slumptalsserie än när man startar med `srand(4321)`.
- `srand(time(0))`. För att slippa hitta på egna startvärden kan man utnyttja datorns klocka. Funktionen `time` returnerar ett `int` som motsvarar antalet sekunder som har passerat sedan `1 jan 1970 kl 00 : 00`. Detta tal, som är mycket större än vad `srand` kan ta emot transformeras ned till ett `unsigned int`, som ändå ger möjlighet till 65536 olika startvärden.

Slumptalsfunktionen `rand` kan alltså bara ge ifrån sig 32768 olika slumptal, vilket är ganska lite i sammanhanget. Detta betyder dock inte att *perioden*, längden på slumptalsserien har denna längd. I själva verket är den betydligt längre, hela $2^{32} = 4\ 294\ 967\ 298$. Vad detta innebär kommer vi till senare.

Programmet nedan visar hur de inbyggda slumptalsfunktionerna fungerar.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define randomize() srand((unsigned)time(NULL))
4  int main(void){
5    int i;
6    printf("RAND_MAX %d\n",RAND_MAX);
7    for(i=1;i<=10;i++){
8      randomize();
9      printf("%d\n",rand());
10   }
11   for(i=1;i<=10;i++)
12     printf("%d\n",rand());
13   for(i=1;i<=10;i++)
14     printf("%d\n",rand()%100);
15   for(i=1;i<=10;i++)
16     printf("%8.6f\n",(float)rand()/(RAND_MAX+1));
17 }
```

2 För att komma åt funktionerna `srand` och `rand` kan man bli tvungen att inkludera `stdlib.h`

3 Denna definition visar hur ett korrekt anrop av `srand` ska se ut. I programmet kan vi sedan istället skriva `randomize()`

6 Vi skriver ut värdet av det största slumptal som systemet kan generera. I vår test blev utskriften 32767

7–10 OBS! Detta är ett felaktigt användande av `srand`. Eftersom funktionsanropet finns inne i loopen kommer det att utföras en gång för varje varv. Vilket i sin tur betyder att alla slumptal blir lika. Om nu inte händelsevis ”tiden byter sekund” under exekveringen. Men mer än två olika slumptal kan det aldrig bli (i så fall är det dags att byta till en snabbare dator).

11-12 Så här ska man använda `srand` om man vill ha 10 slumptal. Bäst är troligtvis ett enda anrop av `srand`, i början av programmet.

13-14 Här får vi 10 slumptal i intervallet [0...99]

15-16 Klassiskt talar man om slumptal, *rektangulärfördelade*, i intervallet $[0, 1)$. Det minsta slumptalet kan alltså vara 0. Å andra sidan kan aldrig slumptalet bli 1. Med hjälp av formeln

$$\frac{\text{rand}()}{\text{RAND\_MAX} + 1}$$

ordnar vi detta.

Då man fortfarande håller på att testa ut ett program, som innehåller slumptal, vill man ogärna att slumptalen ska förändras varje gång man testkör det. Då skriver man i stället `srand(num)`, där `num` är ett godtyckligt valt heltal.

UPPGIFT 6.2

Roulett-strategi. En känd roulett-strategi: *Spela bara på rött. Om du förlorar så dubbla bara insatsen*

Du ska undersöka denna strategi genom att *simulera* en roulett. Ett rouletthjul innehåller 37 nummer, 18 svarta, 18 röda och ett grönt (nollan). Den som spelar en marker på *rött* får tillbaka den plus en marker från banken om ett rött nummer kommer upp. I annat fall är förstås insatsen förlorad.

Simuleringen startar med att spelaren satsar 1 marker (värd en krona). Spelaren dubblar sedan insatsen så länge han förlorar. Så fort han vunnit börjar han om med insatsen 1 kr. Spelet fortsätter sedan 10 000 slag eller tills spelaren inte har råd att dubbla insatsen i nästa slag.

Vilket startbelopp föreslår du att spelaren ska ha, för att ha en 50% chans att överleva de 10 000 slagen? Hur ofta går spelaren med vinst, de gånger han klarar sig? Om ett *slag* tar 90 sekunder – vilken blir då spelaren timpenning de gånger spelaren går med vinst? Programmet ska producera en tabell, som innehåller

Startbelopp	Överlever	Går vinst	Genomsnittlig vinst	Timpenning

för ett antal startkapital, med steget 10000, där varje startkapital testas 1000 gånger. Kan du förklara tabellens utseende?

6.2 Så går det till att generera slumptal

De två oftast använda metoderna för att framställa slumptal heter *linear congruential generator* och *additive number generator*

6.2.1 Linear Congruential Generator

Denna typ av generator är mycket vanlig och används i många programsystem. Vi utgår från *fröet* x_0 och med hjälp av formeln

$$x_n = (a \cdot x_{n-1} + c) \mod m$$

får vi så nästa slumptal. Valet av konstanterna a, c och m är det viktigt. Att mod betyder resten vid division vet du säkert. Denna typ av generator har många fördelar:

- stabil
- ger goda testresultat
- snabb
- kräver lite minne
- genomanalyserad

Följande kända värden på konstanter hämtar vi från litteraturen:

a	c	m	författare
7^5	0	$2^{31}-1$	Park-Miller
131	0	2^{35}	Nave
16333	25887	2^{15}	Oakenfull
3432	6789	9973	Oakenfull
171	0	30269	Wichman-Hill

Längden hos slumptalsserien, *periodlängden*, kan bestämmas med hjälp av de tre konstanterna. Till exempel då $a = 3$ eller $5(\mod 8)$, $c = 0$ och $m = 2^p$ är periodlängden 2^{p-2}.

De av Paker-Miller föreslagna konstanterna genererar samtliga tal i intervallet $1 \ldots 2147483646$ i en 'slumpmässig' ordning. Det vill säga $2^{31} - 2$ tal. Om perioden har just längden $m - 2$ kallas den a full en "full-period-multiplier". $a = 7^5 = 16807$ är det minsta talet med denna egenskap.

6.2.2 Ett program

Nedanstående program är exempel på hur hur *Linear Congruent Generator* kan implementeras. Konstanterna som använts är från *Park-Miller*. Konstanten $a = 7^5 = 16807$ hittar vi i koden, liksom $m = 2^{31} - 1 = 2147483647$.

Om vi vill åstadkomma så stora slumptal som $2^{31}-1$ måste vi ta till `long long`. Eftersom x_{n-1} ska multipliceras med 16807 så kan inte alltid resultatet av den multiplikationen rymmas i en `int`.

```
1 int rnd(int *seed){
2   long long a;
3   a=(long long)16807**seed;
4   *seed=(int)(a%2147483647);
5   return *seed;
6 }
```

Vid första anropet är parametern ett godtyckligt valt frö. I efterföljande anrop utgörs fröet (parametern) av föregående slumptal. Så här var man tvungen att göra innan det fanns 64-bits heltal.

```
1 int rnd(int *seed){
2   int seedv,seedh;
3   seedv=*seed & 0x7FFF0000;
4   seedv=16807*(seedv>>16);
5   seedh=*seed & 0x0000FFFF;
6   seedh=16807*seedh;
7   *seed=((seedv & 0x00007FFF)<<16)+seedh;
8   *seed=*seed+((seedv & 0xFFFF8000)>>15);
9   if (*seed<0) *seed=(*seed & 0x7FFFFFFF)+1;
10  return *seed;
11 }
12
```

På nätet hittade jag en funktion som ger samma resultat, men betydligt bättre kodad och dessutom snabbare.

```
1 unsigned int rnd(void){
2   unsigned int hi,lo;
3   lo=16807*(seed & 0xFFFF);
4   hi=16807*(seed >> 16);
5   lo+=(hi & 0x7FFF)<<16;
6   lo+=hi>>15;
7   if(lo>0x7FFFFFFF) lo-=0x7FFFFFFF;
8   return seed=(int)lo;
9 }
```

6.2.3 Additive Number Generator

Denna metod bygger på idén att addera två tidigare genererade slumptal för att få nästa. Vi skriver formeln

$$x_n = (x_{n-j} + x_{n-k}) \mod m$$

Om $j = 1$ $k = 2$ $m = 2^{31}$ så tar man de två föregående slumptalen adderar dem modulo m. Nu visar det sig att det hela inte fungerar speciellt bra för dessa värden på k och j. Efter experiment och teoretiska beräkningar har man visat att till exempel $k = 55$ och $j = 24$ fungerar mycket bättre. (Det finns andra par av konstanter som fungerar bra, men vi håller oss till dessa värden i detta exempel.)

Nackdelen med metoden är att vi behöver 55 startvärden! I programmet nedan har vi låtit använda `rand` för att skapa dem.

```
1 void initrnd(unsigned int y[]){
2   int k;
3   srand(time(0));
4   for(k=1;k<=55;k++)
5     y[k]=rand()*rand();
6 }
```

Funktionen `initrnd` bygger upp de 55 startvärdena i en vektor med lika många celler. Vi använder här `unsigned int` eftersom vi vill kunna addera två positiva `int` utan att få overflow.

5 För att inte startvärdena ska bli för små multiplicerar vi två slumptal från `rand`. Eftersom `rand` levererar tal ≤ 32767 kan det inte bli overflow vid multiplikationen.

Själva genereringen kan sedan ske med hjälp av en funktion liknande `rnd` nedan.

```
1 int rnd(unsigned int y[]){
2    unsigned int m=2147483648U;
3    static int j=24,k=55;
4    int s;
5    y[k]=(y[k]+y[j])%m;
6    s=y[k];
7    j--; k--;
8    if(j==0) j=55;
9    if(k==0) k=55;
10   return s;
11 }
```

2 Vi väljer här $m = 2^{31}$.

3 Genom `static` låter vi funktionen minnas värdena på `k` och `j` mellan anropen.

4 När vi nu ska generera det första slumptalet ska vi alltså använda oss av `y`[55] och `y`[24]. Vi ska addera dem mod `m`. Resultatet är vårt nya slumptal. 55 slumptal senare ska detta slumptal ingå i beräkningarna för ett nytt slumptal. Däremot ska aldrig det nuvarande `y`[55] användas igen. Så varför inte lagra det nya slumptalet i `y`[55]?

6–7 Därefter minskar vi `j` och `k` med 1. Skulle de händelsevis bli 0, så får de börja om på 55. Detta betyder att då *det andra slumptalet* ska beräknas så är `k` = 54 `j` = 23 och då ersätts `y`[54].

8 Vi skickar tillbaka det beräknade slumptalet.

Så huvudfunktionen, som genererar och skriver ut 200 slumptal.

```
1
2 int main(void){
3   unsigned int a[56],k,s;
4   initrnd(a);
5   for(k=1;k<=200;k++){
6     printf("%u",rnd(a));
7   }
8 }
```

6.3 Så går det till att testa slumptal

Kan vi då lita på våra slumptal? Den frågan har förstås ställts förr och därför har ett antal tester utformats, som försöker svara på frågan.

6.3.1 Chi-square test

Om vi kastar en vanlig tärning 1200 gånger och noterar antalet gånger, de olika antalet ögon kommit upp, kommer vi knappast att få följande resultat:

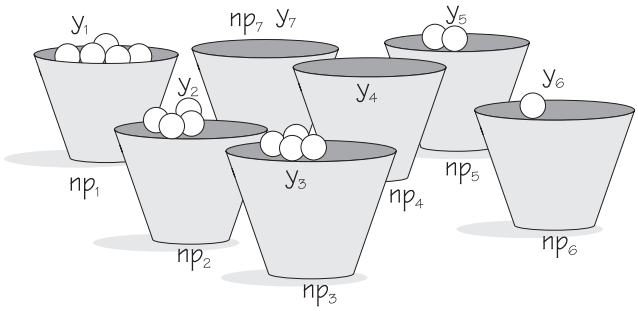
1	2	3	4	5	6
200	200	200	200	200	200

Men om resultatet i stället blir

1	2	3	4	5	6
175	210	193	215	201	206

kan vi då säga något om tärningens kondition? Är det en just tärning eller inte?

För att besvara den frågan ska vi använda oss av χ^2-test. I figuren ser du ett antal "hinkar". Ett försök kan resultera i ett av flera *utfall*. Varje utfall representeras av en "hink". När alla försöken är över vet vi hur många utfall varje "hink" har fått. Detta antal ställs nu i relation till det förväntade antalet utfall i just den hinken.



Figur 6.1: *Olika utfall hamnar i olika hinkar. Det förväntade antalet jämförs sedan med det verkliga antalet med hjälp av χ^2-test*

Först bildar vi *summan av kvadraten på skillnaderna mellan förväntat och verkligt utfall, dividerat med förväntat antal*. Eftersom vi kastar tärningen 1200 gånger och den förväntade sannolikheten är 1/6 för varje "ögonantal" är det förväntade resultat av varje $1200/6 = 200$.

Alltså

$$V = \frac{(175-200)^2}{200} + \frac{(210-200)^2}{200} + \frac{(193-200)^2}{200} + \frac{(215-200)^2}{200} + \frac{(201-200)^2}{200} + \frac{(206-200)^2}{200}$$

Detta ger $V = 5.18$. Med detta tal går vi sedan in i tabellen nedan

	p=1%	p=5%	p=25%	p=50%	p=75%	p=95%	p=99%
$\nu = 1$	0.00016	0.00393	0.1015	0.4549	1.323	3.841	6.635
$\nu = 2$	0,02010	0.1026	0.5753	1.386	2.773	5.991	9.210
$\nu = 3$	0.1148	0.3518	1.213	2.366	4.108	7.815	11.34
$\nu = 4$	0.2971	0.7107	1.923	3.357	5.385	9.488	13.28
$\nu = 5$	0.5543	1.1455	2.675	4.351	6.626	11.07	15,09
$\nu = 6$	0.8720	1.635	3.455	5.348	7.841	12.59	16.81
$\nu = 10$	2.558	3.940	6.737	9.342	12.55	18.31	23.21
$\nu = 20$	8.260	10.85	15.45	19.34	23.83	31.41	37.57
$\nu = 30$	14.95	18.49	24.48	29.34	34.80	43.77	50.89
$\nu = 50$	29.71	34.76	42.94	49.33	56.33	67.50	76.15
$\nu = 99$	69.17	77.05	89.14	98.33	108.1	123.2	134.7

ν är 1 mindre än antalet olika utfall, det vill säga 5 i vårt exempel. ν kallas *antal frihetsgrader*. $4.351 < \nu = 5.18 < 6.626$. I 75% av alla försök ger en riktig tärning ett $V < 6.626$ och i 50% av försöken ger en riktig tärning ett $V > 4.351$. Mer kan vi inte säga om detta enda försök. Det finns i alla fall inget som pekar mot att tärningen är suspekt!

För $V = 16.8$, till exempel, finns det anledning misstänka att något är konstigt eftersom det inträffar högst en gång på 100 att $\nu > 15.09$. På samma sätt, om vi hade fått $\nu = 0.5 < 0.5543$, så är det mindre än en chans 100 att vi ska få ett så lågt värde på V om vi använder en korrekt tärning. Glöm dock inte bort att "allt är möjligt". Inte förrän man gjort flera försök kan man säga något mer bestämt om tärningens kondition! En viktig tumregel för användandet av *Chi-square*-testen är att varje utfall ska ha ett förväntat värde ≥ 5. Det betyder alltså att vi måste kasta tärningen åtminstone 30 gånger (vi kastade ju 1200) för att kunna använda metoden.

UPPGIFT 6.3
Testa slumptal. Denna teknik kan vi nu använda för att testa våra olika slumptalsgeneratorer.

- Vi utgår från tärningskast och ser till att generatorn ger slumptal i intervallet $1 \ldots 6$
- Det betyder att vi har 6 möjliga utfall, vilket leder till 5 frihetsgrader (5:e raden i tabellen).
- Bestäm antalet slumptal, som ska ingå i försöket till 1200. Det förväntade antalet utfall i varje hink blir som i exemplet 200.
- Bestäm ν och använd följande straffskala

Intervall	**Straff**
$0-1\%,\ 99-100\%$	3
$1-5\%,\ 95-99\%$	2
$5-10\%,\ 90-95\%$	1

  Ju lägre poäng desto bättre generator.
- Utför 1000 försök med följande generatorer och summera straffpoängen (bör hamna runt 500). Observera att detta endast är en av flera tester som bör göras på en generator.
  - Den i inbyggda
  - Programmet med Linear Congruential Generator vars konstanter härstammar från Park-Miller.
  - Programmet med Additive Number Generator där $j = 24$ och $k = 55$.

6.3.2 Gap test

Om en serie slumptal är acceptabel i *Chi-square*-testens mening behöver inte det betyda att serien är acceptabel för det. Vad säger du om en serie tärningskast som börjar med

```
111112222333344444...?
```

I *Gap*-testen (lucktesten), ska vi arbeta med slumptal i intervallet $[0, 1)$ och mäta avstånden (i antal slumptal räknat) mellan två slumptal, som ligger i intervallet $0 \leq \alpha < \beta < 1$. Om vi

till exempel väljer $\alpha = 0$ och $\beta = 0.1$ och genererar slumptalen

$$0.098,\ 0.346,\ 0.756,\ 0.125,\ 0.566,\ 0.083,\ 0.459,\ 0.550,\ 0.073$$

så får vi ett ”gap” på 4 (mellan 0.098 och 0.083) och ett ”gap” på 2 (mellan 0.083 och 0.073).

Gapens längd registreras sedan i en vektor v ($v[4]$ och $v[2]$ ökas med 1). När man räknat in n gap avbryts testen och vektorns innehåll bearbetats med *Chi-square*-test. I denna test är förstås inte de förväntade värdena i vektorn lika stora. Det är problemet att fastställa det förväntade antalet utfall i varje ”hink”, som är svårast i denna och nästa test.

Om vi bestämt oss för ett intervall med undre gränsen $\alpha = 0.31$ och med övre gränsen $\beta = 0.56$, betyder det att sannolikheten p att vi ska få ett slumptal, som ligger i intervallet, är $p = \beta - \alpha = 0.25$. Detta innebär alltså att sannolikheten q, att vi hamnar utanför intervallet, är $q = 0.75$

För att vi ska träffa rätt direkt, *att gapet blir 0*, är sannolikheten $p_0 = 0.25$. För att vi ska träffa rätt efter två slumptal, *att gapet blir 1*, är sannolikheten $p_1 = pq = 0.25 \cdot 0.75$

Sannolikheten för gaplängderna ges av

$$\begin{array}{llll} p = \beta - \alpha & & & \\ p_0 = p & p_1 = p(1-p) & p_2 = p(1-p)^2 & \ldots \\ \ldots & p_{t-1} = p(1-p)^{t-1} & p_t = p(1-p)^t & \end{array}$$

Vilket i sin tur leder till ett uttryck för vårt V, där n är *antalet inräknade gap*

$$V = \frac{(v_0 - np_0)^2}{np_0} + \frac{(v_1 - np_1)^2}{np_1} + \ldots + \frac{(v_t - np_t)^2}{np_t}$$

Vilket värde t får avgörs av generatorn. För en riktigt dålig slumptalsgenerator kan ju t bli hur stort som helst.

6.3.3 Kupongtest

Varje gång Kalle köper en tablettask får han slumpmässigt en bild på en av AIK's fotbollsstjärnor. Frågan är hur många askar han måste köpa för att få hela AIK-elvan i sin samling. Så skulle man kunna introducera kupongtestet.

De ursprungliga slumptalen konverteras till heltal i intervallet $[0, d-1]$. Om d är 6 kan en slumptalsserie se ut så här

$$0233152014|12300013424225|5423411420|\ldots$$

De 10 första talen ger en komplett samling. Den andra samlingen fullbordas inte förrän efter 14 tal. Självklart får dessa serier inte bli för långa för bra slumptalsgenerator.

Problemet är här som i *Gap testen* att bestämma sannolikheten för en viss längd hos serien innan samlingen är komplett. Det är så komplicerat, så vi ger formeln direkt. Sannolikheten p_r att slumptalsserien ska bli r tal lång är

$$p_r = \frac{d!}{d^r}\left\{ {r-1 \atop d-1} \right\}, \quad d \le r < t; \quad p_t = 1 - \frac{d!}{d^{t-1}}\left\{ {t-1 \atop d} \right\}$$

Uttrycket $\left\{ {a \atop b} \right\}$ syftar till *Stirlingtal av andra ordningen.* Några värden i tabellen nedan

n	$\left\{ {n \atop 0} \right\}$	$\left\{ {n \atop 1} \right\}$	$\left\{ {n \atop 2} \right\}$	$\left\{ {n \atop 3} \right\}$	$\left\{ {n \atop 4} \right\}$	$\left\{ {n \atop 5} \right\}$
0	1	0	0	0	0	0
1	0	1	0	0	0	0
2	0	1	1	0	0	0
3	0	1	3	1	0	0
4	0	1	7	6	1	0
5	0	1	15	25	10	1
6	0	1	31	90	65	15

Som ett exempel bestämmer vi sannolikheterna för olika längder r där $d = 6$. Chansen att serien ska bli komplett efter 9 tal – att man då samlat in talen $0 \ldots 5$ – är cirka 7.5%

r	p_r	p_r
6	$\frac{6!}{6^6}\left\{ {5 \atop 5} \right\}$	0.015
7	$\frac{6!}{6^7}\left\{ {6 \atop 5} \right\}$	0.039
8	$\frac{6!}{6^8}\left\{ {7 \atop 5} \right\}$	0.060
9	$\frac{6!}{6^9}\left\{ {8 \atop 5} \right\}$	0.075

UPPGIFT 6.4
Stirlingtal av andra ordningen kan beräknas rekursivt med hjälp av följande uttryck

$$\left\{ {n \atop m} \right\} = m\left\{ {n-1 \atop m} \right\} + \left\{ {n-1 \atop m-1} \right\}$$

Genom följande speciella värden

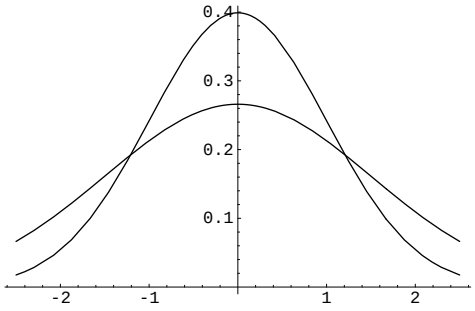
$$\left\{ {0 \atop 0} \right\} = 1 \quad \left\{ {n \atop 0} \right\} = 0\ n \neq 0 \quad \left\{ {n \atop 1} \right\} = 1 \quad \left\{ {n \atop n} \right\} = 1$$

ska du skriva en funktion `stirling(n,m)` där alltid $n \geq m$

6.4 Slumptal för olika fördelningar

Grunden för generering av slump med hjälp av en dator är de *rektangulärfördelade slumptalen*. Tal som alla har lika stor sannolikhet att "komma upp". Mycket av det man vill studera, *simulera*, har nu inte den fördelningen. Att den längsta eleven i klassen är 201 cm och den kortaste 159 cm betyder inte att de övriga 28 eleverna längd är jämnt fördelad över intervallet $[159, 201]$. Nedan ska vi bland annat se hur man kan generera 30 slumptal som väl ansluter sig till längden hos elever i en klass.

6.4.1 Normalfördelade slumptal



Figur 6.2: *I figuren ser du två normalfördelningar med olika medelvärde* μ *och standardavvikelse* σ.

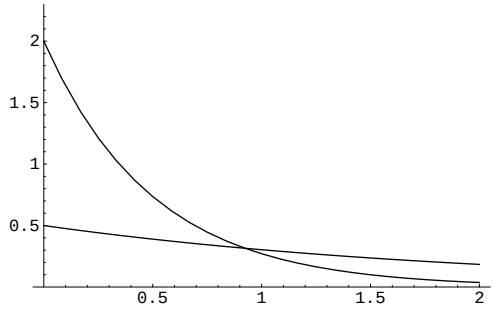
$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Vi återgår till vårt exempel med klassen och elevernas längd. Vi har anledning att tro att längden är normalfördelad. Kanske är $\mu = 176$ och $\sigma = 15$, som ofta skrivs $N(176, 15)$ (N för *normalfördelad*).

Följande schema ger oss ett normalfördelat slumptal $N(\mu, \sigma)$

- Generera två rektangulärfördelade slumptal i r_1 och r_2 i intervallet $[0, 1)$
- Med hjälp av formeln får vi $n_1 = \sqrt{-2 \ln r_1} \cos(2\pi r_2)$.
- Slumptalet n_1 tillhör $N(0, 1)$ och genom $n_2 = \mu + \sigma n_1$ får vi ett slumptal som tillhör $N(\mu, \sigma)$

6.4.2 Exponentialfördelade slumptal



Figur 6.3:

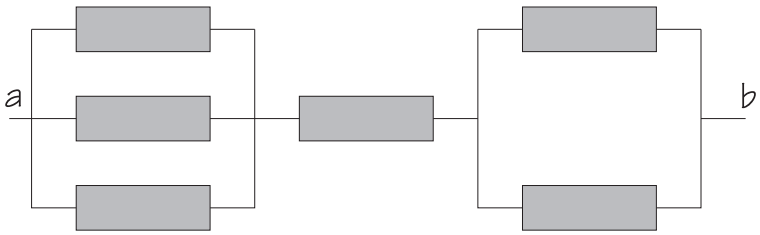
Exponentialfördelningen förekommer ofta när man talar om livslängden hos glödlampor eller tiden det tar att bli behandlad av doktorn. Nedan ser du frekvensfunktionen för *exponentialfördelningen*.

$$f(x) = \lambda e^{-\lambda x} \; x \geq 0$$

Så här genererar du exponentialfördelade slumptal på datorn:

- Generera ett rektangulärfördelat slumptal r_1, som tillhör intervallet $[0, 1)$.
- Bestäm $e_1 = -\ln r_1/\lambda$

Medelvärdet är $\mu = 1/\lambda$ och standardavvikelsen är $\sigma = 1/\lambda$



Figur 6.4: *Hur länge kan någon ström överhuvud taget flyta mellan a och b?*

UPPGIFT 6.5
Elkrets. I figuren ser du en ”elkrets”. Strömmen kan flyta mellan a och b, så länge det finns en väg mellan punkterna med idel hela komponenter. Alla komponenter har en medellivslängd på 1000 timmar. Livslängden är exponentialfördelad.

Skriv ett litet enkelt program som bestämmer ”elkretsens” livslängd genom 100 simuleringar. Bortse från att detta problem ganska enkelt kan lösas på teoretisk väg.

6.4.3 Poissonfördelade slumptal

Man vet att det kommer 10 kunder i timmen till affären. Hur stor är då sannolikheten att det kommer en kund nästa minut? Svaret på den frågan kan besvaras med hjälp av *poissonfördelningen*.

$$f(x) = \frac{e^{-\lambda}\lambda^x}{x!} \quad x = 0, 1, 2, \ldots$$

Medelvärdet $\mu = \lambda$ och standardavvikelsen $\sigma = \sqrt{\lambda}$ Svaret på vår lilla fråga från inledningen blir alltså om vi räknar om $\lambda = 10/60$, en $\frac{1}{6}$ kund/minut.

$$f(1) = \frac{e^{-1/6}(1/6)^1}{1!} = 0.14$$

Det är alltså 14% chans att det kommer en kund nästa minut. Så här beräknar du ett poissonslumptal, då du känner λ. Det vill säga hur många händelser som kommer att inträffa nästa tidsenhet.

Med datorns hjälp genererar vi *poissonfördelade slumptal* genom följande algoritm.

1 Sätt $i = 0$ och $s = 0$
2 Generera ett rektangulärfördelat slumptal r_i
3 Beräkna $p_i = \frac{-\ln r_i}{\lambda}$
4 Öka s med p_i
5 Öka i med 1
6 Utför punkterna 2 till 5 så länge $s < 1$
7 $i - 1$ är det önskade *poissonfördelade* slumptalet.

6.4.4 Debugger'n i Code::Blocks

En debugger är ett ytterst viktigt hjälpmedel vid programutveckling. Här följer en introduktion till hur debuggern fungerar i *Code::Blocks*.

- Ett tråkigt problem, som kostade mig en del tid, är att den path i vilket projektet ligger inte får innehålla något mellanslag!
- Debuggern fungerar endast under **projekt** i Code::Blocks.
- Starta med att ladda ner *Debugger testkod* från hemsidan och skapa ett projekt. Byt ut den automatiskt genererade `main`-filen mot `main`-filen i `zip`-filen. Lägg in `talen.txt` i samma map.
- Gå nu in under *Meny→Project→ Project Bulid Options* och bocka för *Produce debugging symbols [-g]*.
- Du ska nu ha ett program med 43 rader på skärmen.
- Klicka till höger om radnumret 29, så ska en röd cirkel uppenbarar sig. Vi har satt in en *breakpoint*. När exekveringen når hit kommer programmet att stanna.
- Exekveringen startas genom *Meny→Debug→Start*, eller genom att trycka på F8.
- Vi ser nu att programmet nått till den röda cirkeln genom att där finns en gul triangel.
- Välj nu *Watches* under *Meny→Debug→Debuggig Windows*. Ett fönster, *Watches*, öppnas nu till vänster på skärmen.
- Klicka på plustecknet framför *Local variables*. Vi ser nu vilka värden de tre lokala variablerna har. `fil` är ju en pekare till en struct, som ska lagra info om den fil vi så småningom ska öppna.
- Programmet innehåller två globala variabler `tab` och `tal`. Högerklicka när muspekaren befinner sig över *wathches*-fönstret och välj *Add watch*. Skriv in `tab` och upprepa detta för `tal`. Vi har nu kontroll över samtliga variabler som kommer att påverkas i de närmaste satserna.
- Klicka på plustecknet framför `tab` och vi får se en räcka med 20 nollor, vilket motsvarar innehållet i denna array. Vi ser också att även `tal` innehåller enbart 0:or. Av detta kan vi sluta oss till att nollställningen av de två arrayerna är helt onödig.
- När det gäller de två loopvariablerna `i` och `j`, har de ett värde som motsvarar `INT_MAX` eller möjligtvis ett slumpmässigt innehåll.
- Tryck nu F7 ett antal gånger. Du ser nu hur programmet exekveras sats för sats. Betrakta värdena hos variablerna `i` och `j`.
- Det känns tråkigt att stega sig igenom alla dessa satser och du vill därför hoppa direkt till rad 34. Klicka in dig någonstans i koden på den raden och tryck F4.
- Vid nästa F7 kommer filen att öppnas. Därefter följer en ny dubbelloop. För varje varv i den inre loop kommer ett tal att läsas in till `tab`. Håll utkik på hur elementen i denna array förändras och att även elementen i `tal` förändras.
- När du är övertygad om vad som händer, klickar du in dig på rad 40 och trycker F4. I nästa steg ska programmet anropa funktionen `solve(0)`. För att följa exekveringen

in i denna funktion tryck *Shift* F7 och vi befinner oss i en rekursiv funktion.

- Med hjälp av F7, nästa sats, och F4, för att hoppa framåt. Kan man nu följa vad som händer i `solve`.
- Genom *Call stack* under *Meny→Debug→Debuggig Windows* kan du se hur många anrop som just nu ligger på stacken och anropens parametrar.
- Under *Meny→Debug* ser du vilka olika funktioner som står till buds. Man avbryter en debug-session genom att klicka på *Meny→Debug→Stop debugger*
- Forska vidare! Ta bland annat reda på hur *breakpoints*-fungerar. Läs mer på till exempel
  `wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks}`

UPPGIFT 6.6

Epidemin. I en liten mellansvensk stad bor 10 000 personer. En vacker(?) dag blir, under en resa till en av av grannstäderna, en av stadens invånare smittat av *snuva*.

Eftersom *inkubationstiden* är endast ett dygn kommer vår man att under nästa dag riskera att smitta sina medmänniskor i den lilla staden.

Genom ett program ska du nu simulera hur smittan sprider sig i staden. Följande fakta gäller.

- Det finns 10 000 människor i staden
- Varje person, smittad eller inte, träffar varje dag M (ett jämnt tal) andra personer från staden. *Vilka* de möter är, lite orealistiskt, helt slumpmässigt. För varje person plockar vi bara fram M/2 slumpmässiga personer, vilket förstås kan leda till att en del personer kommer att träffa fler än M personer, någon gång även sig själv!
- Sannolikheten att en sjuk person ska överföra smitta till en frisk person, vid ett möte dem emellan är $0 \leq P \leq 1$
- Sjukdomen varar i D dagar och man är smittbärare under hela perioden.
- Den som haft snuvan blir immun och kan alltså inte smittas igen. Ingen i staden har haft sjukdomen innan epidemin bryter ut.

Programmet ska presentera följande data:

- Efter hur många dagar staden var smittfri.
- Hur många procent av stadens invånare som blev sjuka.
- Det maximala antalet personer som var sjuka samtidigt och vilken dag det inträffade.

Programmet har tre indata, M, P och D som alla ska ges genom `#define`-rader i början av koden, för att enkelt kunna ändras. Testa programmet med $M = 6$, $P = 0.2$ och $D = 4$

UPPGIFT 6.7

Fotbollsallsvenskan. I denna uppgift ska vi försöka simulera utgången av fotbollsallsvenskan säsongen 1978, trots att den är färdigspelad för 30 år sedan. I tabellen nedan får vi reda på hur det hela slutade.

Öster	26	15	8	3	46-20	38
Malmö	26	12	8	6	29-15	32
Göteborg	26	13	5	8	39-29	31
Kalmar	26	11	9	6	35-30	31
Djurgården	26	10	10	6	50-32	30
Elfsborg	26	10	9	7	44-37	29
AIK	26	10	7	9	31-35	27
Halmstad	26	7	11	8	24-29	25
Hammarby	26	9	5	12	32-38	23
Landskrona	26	6	10	10	28-38	22
Norrköping	26	7	7	12	33-39	21
Åtvidaberg	26	9	1	16	31-42	19
Örebro	26	5	8	13	31-45	18
Västerås	26	6	6	14	20-44	18

För den som är ovan vid att läsa lagsportstabeller meddelas följande. De 7 kolumnerna i tabellen har från vänster följande betydelse:

- Antal spelade matcher
- Antal vunna matcher
- Antal oavgjorda matcher
- Antal förlorade matcher
- Antal gjorda mål
- Antal insläppta mål
- Antal poäng där seger ger två poäng och oavgjort en poäng. (1978 fick man bra två poäng för seger)

På textfilen `LAGNAMN.DAT` finns lagrat uppgifter om de 14 fotbollsklubbarna som spelade allsvensk fotboll säsongen 1978. För varje klubb finns två rader. På första raden finns namnet och på andra raden, två tal – *antalet gjorda mål* och *antalet insläppta mål* totalt under serien.

På filen `FOTBOLL.DAT` finns lagrat samtliga 182 matchresultat från serien på formen:

- Hemmalagsnummer
- Bortalagsnummer
- Hemmalagets gjorda mål
- Bortalagets gjorda mål

Lagnumren bestäms av platsen i namnfilen. En av de 182 raderna i filen kan ha följande utseende:

```
3 5 2 0
```

vilket betyder att lag nummer 3 spelade hemma och vann mot lag nummer 5 med $2 - 0$. Resultatet behöver vi inte för den här simuleringen.

Uppgiften AIK, till exempel, har under säsongen gjort 31 mål, vilket kan utläsas i tabellen ovan. Eftersom en match varar i 90 minuter och laget har spelat 26 matcher är sannolikheten för att laget ska göra mål under en viss minut 31/2340. Att Öster ska göra mål under en minut blir genom samma beräkning 46/2340. När de två lagen möts i en 90 minuter lång match är sannolikheten större att Öster gör fler mål än att AIK gör det – eller hur? Att ”spela” en match innebär alltså att stega sig igenom matchen – minut för minut – och ta reda på om lagen gjorde mål under den minuten.

Uppgiften går nu ut på att simulera alla 182 matcherna och presentera en sluttabell, sorterad efter följande regler som tillämpas i tur och ordning.

- Av två lag kommer det lag före som har flest poäng.
- Om två lag har samma poäng kommer det lag före som har bästa målskillnaden – gjorda mål minus insläppta mål.
- Om två lag har samma poäng och samtidigt samma målskillnad kommer det lag före som har flest gjorda mål.
- Om lagen inte går att skilja åt genom dessa tre regler får slumpen avgöra vilken inbördes ordning de ska ha.

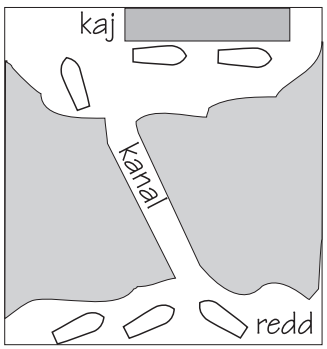
Extra: Det är möjligt att, på ett lämpligt sätt, även ta med de båda lagens *antal insläppta mål* i simuleringen. Ett starkt försvar, få insläppta mål, är det förstås svårare att göra mål på.

UPPGIFT 6.8

Den lilla hamnen. I denna uppgift gäller det att simulera *ett dygns* lossning och lastning vid en liten hamn någonstans i södra Sverige.

I figuren ser vi kartan över en hamn med inlopp. Hamnen har två kajplatser vilket betyder att två fartyg kan *lastas* eller *lossas* samtidigt. För att nå fram till en kajplats måste fartyget från *redden* passera en liten *kanal*. I denna kanal, som det tar 10 minuter att passera, är det *omöjligt* för två fartyg att mötas. Två fartyg får inte ens finnas i kanalen samtidigt, även om de är på väg åt samma håll. Däremot får det, förutom fartygen vid kajplatserna, plats ett fartyg inne i viken.

Uppgiften går ut på att simulera ett dygn vid denna hamn. Vi antar att lastning och lossning sker oavbrutet dygnet runt. Simuleringen ska vara *tidsstyrd*, det vill säga programmet stegar fram *en tidsenhet* i taget tills ett dygn har gått. Vår tidsenhet blir 1 minut. För simuleringen behöver vi två olika typer av slumptal, *poissonfördelade* och *normalfördelade*. De första



Figur 6.5: *En karta över den lilla hamnen, där just nu båda kajplatserna är upptagna av var sin båt.*

för att ta reda på om ett fartyg anlänt till redden och den senare för att ta reda på hur lång tid det tar för ett fartyg att lastas och lossas.

Olika händelser Följande händelser kan inträffa i hamnen:

- Fartyg anländer till redden
- Fartyg lämnar redden och går in i kanalen
- Fartyg går ut ur kanalen och väntar på kajplats
- Fartyg går ut ur kanalen och direkt till kajplats
- Fartyg går från väntan i viken till kajplats
- Fartyg lämnar kajplats och väntar på att kanalen ska bli fri
- Fartyg lämnar kajplats och går direkt in i kanalen
- Fartyg lämnar kanalen vid redden

Normalfördelning Bekanta dig med de *normalfördelade slumptalen* genom att skriva en enkel funktion, som då den anropas returnerar ett normalfördelat slumptal med medelvärdet $m = 60$ minuter och standardavvikelsen $s = 20$ minuter.

Låt sedan funktionen i ett program generera och skriva ut 100 sådana slumptal. Som en test av funktionen kan programmet gärna få bestämma medelvärdet av de 100 talen. Ett värde som förstås ska ligga nära 60.

Poissonfördelning Gör dig på liknande sätt bekant med de *poissonfördelade slumptalen.* Du ska nu skriva en funktion som då den anropas anger hur många båtar som anländer till redden under denna minut. Parametern λ ska vara $1/30$, det vill säga det kommer en båt var 30 minut.

Testa funktionen genom att i ett program anropa den 1200 gånger. Summan av de returnerade talen ska ligga nära 40.

Uppgiften Skriv ett program som simulerar *ett dygn* i hamnen efter de förutsättningar som givits ovan. I medeltal anländer en båt till redden var 30:e minut. Lastning och lossning tar i medeltal 60 minuter med en standardavvikelse på 20 minuter.

Alla förflyttningar, utom färden genom kanalen, antas ske på ”nolltid”.

Programmet ska skrivas så att det dirigerar båtarna optimalt, vilket är samma sak som att minimera summan *ledig kajtid*. Programmet ska presentera följande data efter simuleringen:

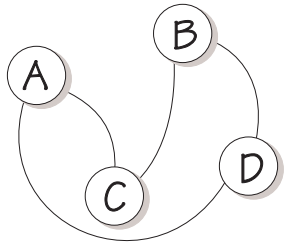
- Antal betjänade båtar under dygnet
- Max antal båtar vid redden
- Total väntetid för båtarna
- Total tid med ledig kajplats

Tips Programmet kommer att bestå av en huvudloop. Varje gång den genomlöps har en minut passerat. De tider som programmet hanterar ska vara i heltal. Då ett fartyg når kajplatsen kan, lite orealistiskt, lossnings- och lastningstiden bestämmas ”i förtid”

6.5 Grafer

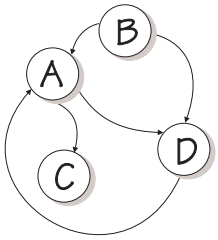
Grafteori är ett stort och expansivt område inom matematiken med mängder av tillämpningar inom datalogin. Först några definitioner (eftersom litteratur på svenska inom området är begränsad och det därför inte alltid finns vedertagna svenska namn på begrepp inom området anger vi också det engelska namnet)

- **graf.** *graph* En mängd V av *noder* (eller *hörn*, eng. *vertices*) och en mängd E av *bågar* (eller *kanter*, eng *edges*) sådan att ändpunkterna på varje båge i E finns i V. En graf skrivs $G = (V, E)$. Ett exempel: $G = (\{a, b, c, d\}, \{(a, c), (b, d), (b, c), (a, d)\})$ (se figur 6.6)



Figur 6.6:

- **riktad graf.** *digraf*, en graf där varje båge är riktad. (se figur 6.7)
- **väg.** *path*, en vandring i grafen där alla noder är olika, utom möjligtvis att den första och den sista, som kan vara samma nod. I en *sluten väg, closed path*, är alltid den första och sista noden densamma. Synonym: *cykel, cycle*
- **förbindelsematris.** *adjacency matrix* har lika många rader och kolumner som antalet noder i grafen. Talet 1 i rad i och kolumn j betyder att det går en båge mellan



Figur 6.7:

nod i och nod j. Talet 0 betyder att i samma position i matrisen betyder att ingen båge finns mellan dessa noder. Förbindelsematrisen för grafen i figur 6.6

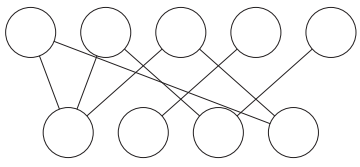
	A	**B**	**C**	**D**
A	0	0	1	1
B	0	0	1	1
C	1	1	0	0
D	1	1	0	0

Motsvarande matris för den riktade grafen i 6.7

	A	**B**	**C**	**D**
A	0	0	1	1
B	1	0	0	1
C	0	0	0	0
D	1	0	0	0

Observera alltså att förbindelsematrisen för en graf är *symmetrisk* till skillnad från en matris för en riktad graf.

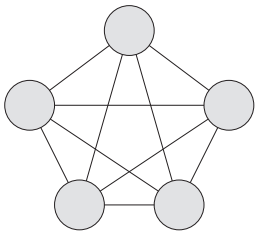
- **sammanhängande graf.** *connected graph* är en graf där det finns en väg mellan alla par av noder. Grafen i figur 6.6 är sammanhängande.
- **brygga.** *bridge*, är en båge, sådan om den tas bort så är inte längre grafen sammanhängande.
- **tvådelad graf.** *bipartite graph*. Här är mängden av noder V uppdelad i två delmängder V_1 och V_2. Varje båge i grafen går mellan en nod i V_1 och en i V_2. Se figur 6.8.



Figur 6.8:

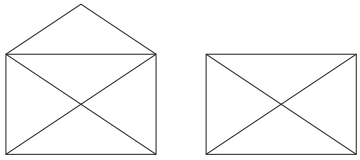
Den övre raden av noder V_1, kan till exempel vara personer. Den undre V_2 lediga jobb. Varje båge betyder då ett jobb sökt av en person. En naturlig fråga i sammanhanget är då om de sökande kan fördelas så att alla tjänster blir tillsatta.

- **kromatiskt tal.** *cromatic number*. Det minsta antalet färger med vilka noderna kan färgläggas, så att två intilliggande noder alltid har olika färger. För grafen i figur 6.6 är det kromatiska talet 2. För tvådelade grafer är det kromatiska talet alltid 2. Man skriver $\chi(G) = 2$.
- **intilliggande noder.** *adjacent vertices*. Två noder som sammanbinds av en båge.
- **komplett graf.** *complete graph*, en graf där varje par av noder har en gemensam båge. Grafen i figur 6.9 är komplett, med sina 10 bågar. En komplett graf med n noder har $n(n-1)/2$ bågar. Förbindelsematrisen innehåller alltså idel 1:or. Vilket är det kromatiska talet för en komplett graf med n noder?
- **gradtal.** *degree*. Givet en nod v, säger vi att $deg(v)$ är antalet bågar som är anslutna till denna nod v. I en komplett graf med n noder har varje nod v $deg(v) = n - 1$. *valens* är en synonym, vars koppling till kemin man kan förstå.



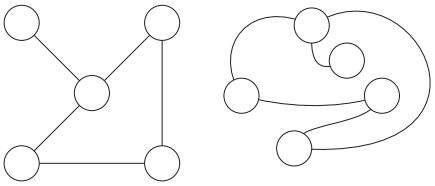
Figur 6.9:

- **komponent.** *component* är en delmängd V_1 av V, sådan att denna delgraf är sammanhängande. V_1 kan inte göras större utan att delgrafen då inte längre är sammanhängande. $\beta_0(G)$ betecknar antalet komponenter i en graf. Alla graferna i figurerna ovan har $\beta_0(G) = 1$.
- **avstånd.** *distance*. Givet två noder v och w den kortaste vägen *d(v,w)* mellan dessa noder.
- **diameter.** *diameter*. Det största avståndet *d(v,w)* som finns i en sammanhängande graf.
- **bågfärgning.** *edge coloring* att tilldela bågarna färger, så att två bågar som är anslutna till samma nod olika färger.
- **Euler tur.** *Euler tour* En sluten väg som går genom samtliga grafens bågar precis en gång. Vilka av dessa figurer kan man rita utan att lyfta pennan?
- **skog.** *forest*, en graf utan cykler. Om skogen är sammanhängande är det förstås ett träd *tree*.
- **rundtur.** *girth* antalet noder i grafens kortaste cykel.
- **Hamilton tur.** *Hamilton cycle* En sluten väg som besöker samtliga grafens noder precis en gång.



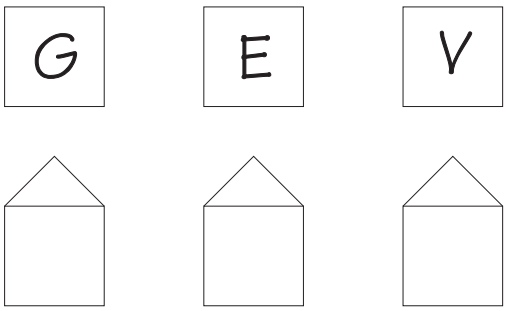
Figur 6.10:

- **isomorfi.** *isomorphic* Två grafer med samma matematiska struktur. Är de två graferna i figur 6.11 isomorfa?



Figur 6.11:

- **loop.** *loop* en båge som förenar en nod med sig själv. Ganska ointressant för oss.
- **granne.** *neighbor* Alla noder som är intilliggande en given nod v.
- **ordning.** *order* antal noder i grafen G, skrivs $|V_G|$.
- **plan graf.** *planar graph* kan ritas i planet utan att några bågar skär varandra. Ett klassiskt problem: *Kan man dra ledningar från gasverket (G), elverket (E) och vattenverket (V) till vart och ett av de tre husen utan att någon av de nio ledningarna korsar någon annan?*



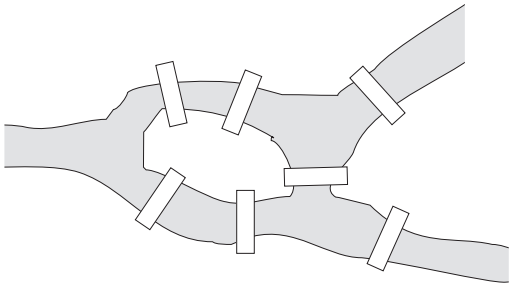
Figur 6.12:

- **delgraf.** *subgraph* är en graf H vars noder och bågar alla tillhör en given graf G
- **viktad graf.** *weighted graph* en graf där där varje båge tilldelats ett tal som kallas för vikten eller kostnaden. Spelar en stor roll i datalogin. I motsvarande förbindelsematris

placerar vi in kostnaden istället för 1. Om man använder 0 för att ange avsaknaden av båge mellan två noder, så utesluter man förstås därmed kostnaden 0.

Grafer ingår i en mängd tillämpningar. Här är några:

- **Programmering** Födesscheman. Noderna är programsteg och riktade bågar visar flödet. Eller noderna är funktioner och bågarna berättar vilka som anropar vilka.
- **Sociala organisationer** Sociala nätverk. Noderna är personer och bågarna representerar relationer.
- **Trafik** Vägnät. Noderna är vägkorsningar och bågarna är vägar. Eller skapa enkelriktade gatunät. Noderna är gatukorsningar och de riktade bågarna är gator.
- **Sociologi** Hierarkisk dominans. Noderna är individer och bågarna visar vem som rapporterar till vem.
- **Ekologi** Näringskedjor. Noderna är arter och bågarna anger vilka arter som äter vilka.
- **Optimering** Schemaläggning. Noderna är aktiviteter och bågarna konflikter mellan aktiviteter.
- **Släktforskning** Familjeträd. Noder är familjemedlemmar och bågar är föräldraskap.
- **Sannolikhetslära** Markov kedjor. Noderna är tillstånd och bågarna är övergångar från ett tillstånd till ett annat.
- **Datalogi** Datornätverk. Noderna är servrar och bågarna är länkar mellan servrar.
- **Optimering** Transporter. Noderna är fabriker och affärer och bågarna är transportvägar.
- **Postutdelning** Chinese Postman Problem. Noderna är vägkorsningar och bågarna är vägar.
- **kartografi** Att måla kartor. Noderna är länder och bågarna är gränser.
- **Elektriska kopplingar** Undvika kortslutning. Noderna är komponenter och bågarna är trådar.
- **VLSI datorchips** Minimera antalet lager. Noderna är komponeneter och bågarna är anslutningar.
- **Informationshantering** Binära sökträd. Noderna är poster och bågarna är beslut.
- **Nätverksoptimering** Billigaste uppspännande trädet. Bågarna är viktade.
- **Personaladministration**. Rätt man på rätt plats. Noderna är personer och arbetsuppgifter. Bågarna är kapacitet.



Figur 6.13: *Det var så den började – grafteorin – med sju broar i Königsberg 1736. Är det möjligt att gå över de sju broarna utan att gå över samma bro mer än en gång. Du får starta var du vill!*

UPPGIFT 6.9

Tennisklubben I min tennisklubb finns 100 spelare. Vi är alla rankade från den bäste med rangordning 1 till den sämste som har rangordning 100. Förra året spelades 500 matcher i klubben.

Själv ligger jag på 99:e plats, men jag slog faktiskt 32:an, som i sin tur slog 85:an, som slog 44:an som slog 1:an – en *kedja* med bara **fyra** länkar till toppen!

Det finns flera som är intresserade att få reda på hur lång deras kedja till toppen är och du ska därför skriva ett program som tar emot uppgift om rankingnumret $2 \leq n \leq 100$ för en viss spelare och som beräknar hur många länkar den kortaste kedjan till toppen har.

För 44:an som slog 1:an, i en hård match, är kedjans längd 1 och för mig med nr 99 alltså 4. För att göra det hela lite enklare lovar jag att alla de 100 spelarna har en kedja som är ≤ 10 lång.

All matcherna finns på filen `tennis.dat`. Filen består av 500 rader. Varje rad utgör en match. Det första av de två talen på raden anger rankingnumret för den spelare som vann matchen och det andra talet vem som förlorade. Ett körningsexempel

```
Vilken spelare? 99
Kortaste kedjan till toppen består av 4 steg
```

Annan datafil kan användas vid rättningen, dock alltid med 500 matcher och med en kedja som är ≤ 10 för varje spelare.

UPPGIFT 6.10

Bordsplacering För att få en fest så trevlig som möjlig bestämde värdinnan att alla bjudna damer skulle poängsätta de inbjudna herrarna med en *trivselpoäng* i intervallet $1 \ldots 10$. Här ser vi ett exempel på resultatet

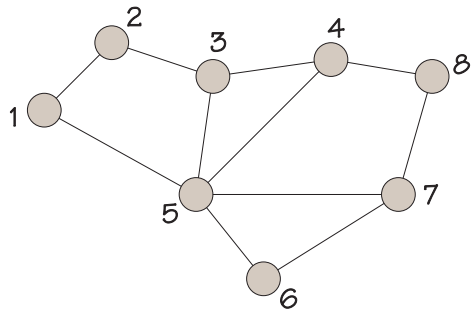
	Adam	Bertil	Curt	David	Erik
Anna	6	4	8	6	9
Britt	7	5	5	4	8
Cecilia	5	6	7	7	10
Doris	4	8	9	5	9
Eva	8	5	7	4	10

Denna tabell användes sedan för att bestämma vilka par som skulle sitta tillsammans runt middagsbordet. Med paren *Anna - Curt* (8), *Britt - Adam* (7), *Cecilia - David* (7), *Doris - Bertil* (8) och *Eva - Erik* (10) fick man den *högsta totala trivselpoängen*

Skriv ett program som från filen `party.dat` läser in poängtabellen och utifrån denna bildar de par som ger den högsta möjliga totala trivselpoängen. Filen inleds med en rad som innehåller antalet par $n, 2 \leq n \leq 10$ (5 i vårt exempel). Därefter följer n rader med n tal i varje.

Utskrift från programmet

```
Högsta möjliga trivselpoäng är 40
```



Figur 6.14: *Detta hus innehåller 8 rum. Musen startar alltid sina vandringar från rum 1.*

UPPGIFT 6.11

Musvandring För att ta reda på om en mus verkligen förflyttar sig slumpmässigt mellan olika rum i ett hus, önskar man ett program där man med hjälp av slumpen kan uppskatta hur många slumpmässiga förflyttningar som krävs för att komma från ett rum till ett annat givet.

I figur 6.14 ser vi ett hus, i form av en graf, med 8 rum. Musen startar alltid i rum 1 och har här två val, till rum 2 eller 5. Eftersom det är viktigt att vandringen verkligen är slumpmässig är sannolikheten 0.5 för vart och ett av dessa. På samma sätt har varje rum sannolikheten

0.2 att bli valt, då musen befinner sig i rum 5, eftersom musen lika väl kan återvända till rum 1.

Skriv ett program som tar reda på det förväntade antalet förflyttningar, genom att simulera vandringen 10000 gånger och presentera ett medelvärde med två decimaler. Filen `hus.txt` inleds med ett tal n, som anger hur många dörrar det finns. Därefter följer n rader med två tal på varje, numren på rummen mellan vilka det finns en dörr. Rummen är numrerade från 1 (där musen alltid startar) upp till som högst 10.

```
Till vilket rum: 8
I medeltal behövs 15.81 förflyttningar
```

Även om du inte kommer att få exakt detta svar ska det ligga ganska nära. Du bör använda `srand`.

6.5.1 Datstrukturen Graf

Den abstrakta datastrukturen för Graf bör ha följande operationer:

- Sätta in en nod
- Sätta in en båge
- Ta bort en nod
- Ta bort en båge
- Söker upp en nod i grafen
- Bestämmer grannarna till en given nod

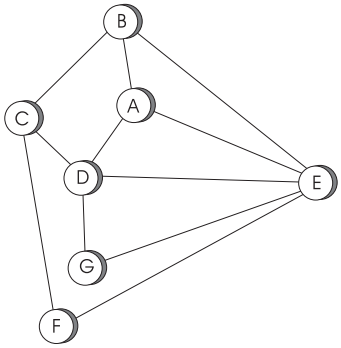
I många uppgifter i den här kursen ska man hantera en graf. Grafen ges genom en textfil. På varje rad i filen finns ofta två nodnummer, vilket innebär att det finns en båge mellan dessa noder. Ibland är ordningen viktig, i så kallade riktade grafer. Ibland finns ett tredje tal på raden som anger en vikt för bågen, till exempel en längd eller kostnad. Det finns flera idéer till datastrukturer för grafer. Här följer tre stycken, som vi exemplifierar med denna graf

Den enklaste metoden, men kanske också den mest resurskrävande är att lagra noderna i i en array $n \times 2$ eller $2 \times n$

A	A	A	B	B	C	C	D	D	E	E
B	D	E	C	E	D	F	E	G	F	G

Varje gång man söker en förbindelse, måste man söka igenom arrayen (listan). Är grafen liten innebär denna datastruktur inga problem. Är grafen riktat kan man bestämma att bågen utgår från den övre noden. Är grafen viktad kan man använda array till dimensionen $n \times 1$ eller $1 \times n$ där man lagrar vikterna på samma index som motsvarande bågar.

Nästa förslag kallas *adjacent matrix* (jag kan inte komma på något bättre svenskt ord än *förbindelsematris*). Matrisen har lika många rader och kolumner som antalet noder. Talet



Figur 6.15:

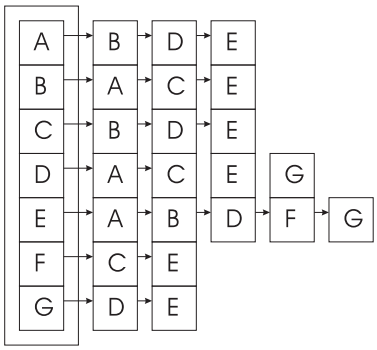
1 anger att de finns en båge mellan noderna med motsvarande rad- och kolumnindex. Att ta reda på om det finns en båge mellan två angivna noder kan förstås inte göras snabbare. Nackdelen är att då antalet noder är stort blir tar matrisen stort minnesutrymme. Har grafen dessutom få bågar blir matrisen gles, många nollor.

Men även om det finns 1000 noder i grafen och varje båge tilldelas en `int` kommer strukturen inte att ”kosta” mer än 4 MB. I våra problem handlar det snarare om 50 än 1000 noder.

I en oriktad graf blir matrisen symmetrisk, varje båge ger upphov till två 1:or. Är grafen däremot riktad sätter man bara ut en 1:a. Till exempel kan man låta radindex ange från vilken nod bågen utgår. Är grafen viktad ersätter man 1:orna med aktuell vikt. Ett litet problem uppstår om vikten kan vara 0.

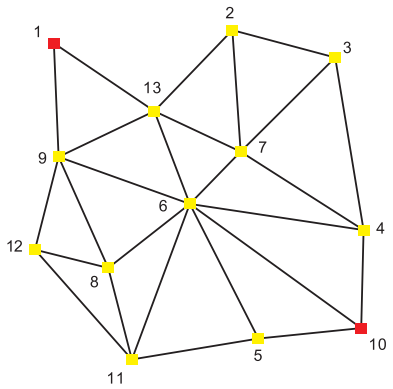
	A	B	C	D	E	F	G
A	0	1	0	1	1	0	0
B	1	0	1	0	1	0	0
C	0	1	0	1	0	1	0
D	1	0	1	0	1	0	1
E	1	1	0	1	0	1	1
F	0	0	1	0	1	0	0
G	0	0	0	1	1	0	0

Den sista idén består av en array innehållande pekare till en länkad lista som innehåller de noder som är förbundna med motsvarande nod i arrayen. Om det handlar om en oriktad graf består är antalet objekt i de länkade listorna två gånger antalet bågar. Vid en riktad graf överensstämmer antalet objekt med antalet bågar. Vid en viktad grafer innehåller lämpligast det länkade objektet aktuell vikt. Strukturen, som kan verka väl ambitiös i denna kurs, tar mindre plats än förbindelsematrisen, mer är inte lika snabb. När man väl skrivit koden är den förstås både snabbare och bättre än den inledande strukturen.



Figur 6.16:

6.5.2 En semestertur – Ett löst problem



Figur 6.17:

Adam har just vaknat upp på ett hotell i staden 1 och har bestämt sig för att göra en semestertripp till staden 10. På vägen dit måste han övernatta på hotellen i städerna han kommer till. Möjliga vägar mellan städerna framgår av kartan. Resan mellan två städer är en dagsetapp. Han har tagit reda på vad det kostar med ett enkelrum på aktuella hotell, som framgår av tabellen nedan

Stad	1	2	3	4	5	6	7	8	9	10	11	12	13
Pris	250	310	280	310	200	1200	950	830	350	425	275	290	340

På filen `karta.txt` finns alla data. Filen inleds med ett tal n, som anger antalet städer $n \leq 100$. Därefter följer n rader med ett tal på varje, priset för hotellrum i motsvarande stad. Därefter följer ett tal m som anger antalet vägar på kartan. Filen avslutas så med m rader med två tal på varje, numren på två städer mellan vilken det finns en väg.

Skriv ett program som tar reda på den billigaste resan (i hotellkostnader räknat) mellan stad 1 och stad 10. Observera att hotellkostnaderna för stad 1 och stad 10 ska inkluderas.

Ett lösningsförslag

```
1  #include <stdio.h>
2  #include <limits.h>
3  #define ANTAL 101
4
5  int main(void){
6    int karta[ANTAL][ANTAL]={0},kostnad[ANTAL],ejvarit[ANTAL]={0};
7    int nhotell,nvagar,mintot=INT_MAX;
8    int start=1, mal=100;
9    init(&nhotell,kostnad,&nvagar,karta);
10   ejvarit[start]=1;
11   solve(start,mal,kostnad[start],&mintot,nhotell,
12         karta,kostnad,ejvarit);
13   printf("Minkostnad %d\n",mintot);
14 }
```

H Här är `main`. Vi har ambitionen att klara oss utan globala variabler. Därför blir parameterlistorna ganska långa.

3 Vi skriver ett program som klarar upp till 100 städer.

6 Vi behöver en matris `KARTA` för att lagra *förbindelsematrisen.* och en vektor som håller reda på priserna i de olika städerna. Dessutom behöver vi en vektor `ejvarit`, med flaggor som ska hålla reda på vilka städer vi besökt. Det kan ju aldrig vara någon mening med att besöka samma stad mer än en gång.

7 `nhotell` och `nvagar` håller reda på hur många städer och vägar som ingår i problemet. `imintot` håller reda på den hittills lägsta kostnaden man funnit.

8 `start` och `mal` håller reda på var resan startar respektive målet för resan.

9 Inläsning av data sker genom funktionen `init`

10 Innan resan startar markerar vi att vi varit i stad 1.

11-12 Nu anropar vi den rekursiva funktion som ska lösa problemet, med inte mindre än åtta aktuella parametrar: Där resan startar, Målet för resan, Kostnaden så här långt, Den minsta kostnaden vi hittills funnit för en resa till målet, Antalet städer (eller hotell), Förbindelsematrisen, Vektorn med priserna och Vektorn som ska hålla reda på vilka städer vi besökt. De fem sista parametrarna plus `mal`, kunde lika väl ha deklarerats globalt och skulle då inte behövas i parameterlistan.

13 När vi återvänder hit vet vi resultatet.

```
1  void init(int *nhotell,int kostnad[],
2            int *nvagar,int karta[][ANTAL]){
3    FILE *fil;
4    int i,fran,till;
5    fil=fopen("karta2.txt","rt");
6    fscanf(fil,"%d",nhotell);
7    for(i=1;i<=*nhotell;i++)
8      fscanf(fil,"%d",&kostnad[i]);
9    fscanf(fil,"%d",nvagar);
10   for(i=0;i<*nvagar ;i++){
11     fscanf(fil,"%d %d",&fran,&till);
12     karta[fran][till]=1;
13     karta[till][fran]=1;
14   }
15   fclose(fil);
16 }
```

5-8 Vi öppnar filen och läser in antalet städer och priserna för de olika hotellen. Eftersom städerna är numrerade `1...13` kan det kännas bra att inte använda index `0`.

9 Vi läser in antalet vägar och i en loop läser vi in två tal (numren på två städer) i taget, som vi använder som index för att markera i förbindelsematrisen att det finns en väg mellan dessa två städer. Observera att markeringen ska ske på två ställen.

```
1  void solve(int nu,int mal,int tot,int *mintot,int nhotell,
2             int karta[][ANTAL],int kostnad[],int ejvarit[]){
3    int i;
4    if(nu==mal){
5      if(tot<*mintot)
6        *mintot=tot;
7    }
8    else{
9      for(i=1;i<=nhotell;i++)
10       if(karta[nu][i]==1 && ejvarit[i]==0){
11         ejvarit[i]=1;
12         solve(i,mal,tot+kostnad[i],mintot,nhotell,
13               karta,kostnad,ejvarit);
14         ejvarit[i]=0;
15       }
16   }
17 }
```

H Vi kommer nu till programmets hjärta. En förhållandevis kort kod.

4-7 Först testar vi om vi nått målet. Parametern `nu` håller reda på i vilken stad vi befinner oss. Om den är samma som målet, `mal`, är vi framme. Vi kommer fram många gånger genom olika vägval. Varje gång vi kommer fram testar vi om denna väg är billigare än vi tidigare noteringar. Om så är fallet uppdaterar `mintot`.

9-15 Om vi inte är framme ska resan fortsätta.

9-10 För att hitta en ny stad testar vi om det finns en väg mellan denna `i` och den stad vi nu befinner oss i `nu`. Dessutom får vi inte hitta talet 1 i `ejvarit`, för i så fall har vi redan varit i denna stad.

11 Hamnar vi här har vi hittat en stad till vilken vi kan resa. Vi markera nu att vi kommer dit.

12-13 Vi anropar nu `solve` med nya parametrar. För det första är `i` nu den stad vi befinner oss i. Kostnaden ökar med hotellkostnaderna för aktuell stad. I övrigt skickar vi bara parametrarna vidare.

14 När vi återvänder måste vi boka av staden igen genom att slå av flaggan. Detta är förstås mycket viktigt!

Vidare

Vad händer då om vi ökar antalet städer till 100. Självklart får vi ett större rekursionsträd. Kanske för stort för att orka vänta på svaret. Följande lilla tillägg kan dramatiskt minska exekevringstiden:

```
1         if(tot+kostnad[i]<*mintot)
2           solve(nr+1,i,mal,tot+kostnad[i],mintot,
3                 nhotell,karta,kostnad,ejvarit);
```

Om den kostnad, det innebär att åka till den staden, blir större än den minsta kostnad vi hittills hittat finns det ju ingen anledning att resa till den staden. Vi klipper i trädet.

Hittar programmet snabbt ett ”bra” värde kan det bli många klipp och exekveringstiden förkortas betydligt.

Nu vet Adam vad det kostar men han har ingen aning om vilken väg han ska ta. Detta löser vi genom en vektor `vagen`. För varje nu delresa vi företar oss lägger vi in aktuell stad i vektorn. När vi hittar en ny resa som är billigare koperar vi `vagen` till en annan `bastavagen`, som man sedan skriver ut till slut. Dessutom behöver man nu en parameter som håller reda på vilken etapp under resan man befinner sig på, som index till `vagen`. Tycker man det är trevligt får man utöka parameterlistan med `vagen` och `bastavagen`, annars fungerar det utmärkt med globala variabler.